Machine Learning on Web Documents

by

Lawrence Kai Shih

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

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2004

©Massachusetts Institute of Technology, 2004

Author .....
Department of Electrical Engineering and Computer Science
November 23, 2003

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

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

Machine Learning on Web Documents

by

Lawrence Kai Shih

Submitted to the Department of Electrical Engineering and Computer Science
on November 23, 2003, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

The Web is a tremendous source of information: so tremendous that it becomes difficult for human beings to select meaningful information without support. We discuss tools that help people deal with web information, by, for example, blocking advertisements, recommending interesting news, and automatically sorting and compiling documents. We adapt and create machine learning algorithms for use with the Web's distinctive structures: large-scale, noisy, varied data with potentially rich, human-oriented features.

We adapt two standard classification algorithms, the slow but powerful support vector machine and the fast but inaccurate Naive Bayes, to make them more effective for the Web. The support vector machine, which cannot currently handle the large amount of Web data potentially available, is sped up by "bundling" the classifier inputs to reduce the input size. The Naive Bayes classifier is improved through a series of three techniques aimed at fixing some of the severe, inaccurate assumptions Naive Bayes makes.

Classification can also be improved by exploiting the Web's rich, human-oriented structure, including the visual layout of links on a page and the URL of a document. These "tree-shaped features" are placed in a Bayesian mutation model and learning is accomplished with a fast, online learning algorithm for the model.

These new methods are applied to a personalized news recommendation tool, "the Daily You." The results of a 176 person user-study of news preferences indicate that the new Web-centric techniques out-perform classifiers that use traditional text algorithms and features. We also show that our methods produce an automated ad-blocker that performs as well as a hand-coded commercial ad-blocker.

Thesis Supervisor: David R. Karger
Title: Associate Professor

Acknowledgements

My thesis begins, as I did, with my parents, who have always supported my dream of achieving a PhD. My mom taught me, by example, to be self-sufficient, practical and efficient; without those attributes, I would doubtless be graduating years from now. My dad's intellectual curiosity, his joy in exploring and learning, is also a part of me; without his influence, I would probably not be in graduate school in the first place. My admiration of my parents only grows with time.

My committee was full of terrific professors who pushed me to grow by challenging me to improve in areas that are not necessarily my strengths. David Karger, my advisor, got me to think about broader, theoretical issues; about how I could formalize and sharpen my intuitive ideas. Leslie Kaelbling taught me to pay attention to scientific details: those items that transform good ideas into a good, clear, scientific argument. Randy Davis taught me to improve the organization, flow and accessibility of the thesis. It makes me happy to think about the talent and ability that was focused on reviewing my thesis.

My roommates at Edgerton were a constant source of companionship and fun. Along with my original roommates, Tengo Saengudomlert and Yu-Han Chang, we had many great times–we ate well, played well, and competed in almost everything. To their credit, they let me pretend foosball was a "sport" so I could be good at something. Anyways, they were terrific and fun roommates. I also had good times with many other Edgerton residents: Pei-Lin Hsiung, Eden Miller, Fumei Lam, Erik Deutsch, Ayres Fan, and George Lee.

The AI Lab was also a great working environment, where my collaborators were also my close friends. Jason Rennie, Jaime Teevan, and Yu-Han Chang all co-authored papers with me; but we've also shared drinks, food, and laughs often. I also regularly ate lunch with a group of people that included Jaime Teevan, Christine Alvarado, Mike Oltmans, and Metin Sezgin. The AI Lab had many special events like Graduate Student Lunch, the Girl Scout Benefit, and the AI Olympics (to editorialize, I think some of the special culture of the AI Lab is diminishing with our merger into CSAIL).

Finally, I'd like to acknowledge those people who have stuck with me through the years: Andy Rudnik has been a close friend for as long as I can remember; Joshua Uy, my best friend all the way back in elementary school; Kristen Nimelli Truong and Hajime Inoue who have kept in contact since college; Rudy Ruggles and Eric Droukas from my days at Ernst & Young; and Sarah Rhee, a close friend through graduate school.

Chapter 1

Introduction

People spend an enormous and increasing amount of time using the Web. According to the October 2003 Nielsen/NetRatings estimates[1], Americans now average 8 hours at home and 18 hours at work surfing the Web. Some significant fraction of that time is spent in navigation overhead—clicking links, opening new Web pages, locating content, avoiding and closing advertisements, and so forth. This work focuses on applications, like ad-blocking or content recommendation, that save users time and aggravation when surfing the Web.

This work attempts to build applications that learn appropriate behaviors, rather than receiving hand-written rules. Traditional ad-blocking involves periodic human effort; an engineer writes new code and rules ("an ad is an image 250 by 100 pixels") to block the evolving advertisements that appear. Our approach stresses that the machine *learn* rules to distinguish between ads and content, since such approaches are minimally intrusive to users. Similarly, we emphasize learning what content interests a user, rather than having a user manually specify those interests.

A general and well-studied framework for such learning is the field of classification. Classification algorithms take a "training" set of labeled documents (these documents are ads; those documents are content), and tries to find a relationship between the labels and a set of features, such as the words in the documents. For example, one algorithm might find the relationship that documents with the word "Viagra" are always advertisements, but that documents containing the word "baseball" are not. The algorithm, upon seeing a new "test" document with the word "Viagra" would label it an advertisement. The algorithm would then be judged by its *accuracy*, or the fraction of test documents that it labeled correctly. Classification, and common classification algorithms, are covered in more detail in the background, Chapter 2.

This work traces out a path from standard classification techniques to a working news recommendation system. In between are several improvements to existing classification techniques that make them better suited for the Web application.

Consider a Web application like ad-blocking. The Web typically generates large amounts of training data, and usually, more training data (images labeled as advertisements or not) leads to better performance. However, more training data (even

[1] `http://nielsen-netratings.com`

moderate amounts) can lead to situations in which standard classifiers can not complete in a span of weeks. To build a working Web application based on classification, one needs to find fast, scalable, accurate classification algorithms.

We modified two standard classification algorithms, the fast but inaccurate Naive Bayes, and the slow but powerful support vector machine, to make them more effective for Web usage.

The Naive Bayes classifier is improved through the application of three techniques aimed at fixing some of the severe, inaccurate assumptions Naive Bayes makes without slowing down the classifier significantly. Empirically, this improves Naive Bayes's accuracy dramatically, even approaching state of the art on some standard text classification tasks.

Working from the opposite direction, we take a slow but accurate classification algorithm and speed it up. Our method, known as bundling, combines training points together to form a smaller number of representative points. The smaller number of representative points gives the slow classifier enough information to write good rules, but takes a fraction of the time that training over all the original documents would.

While the size of the Web makes algorithmic scalability a challenge, the Web's rich, human-oriented structure offers features that may usefully augment standard document text features. Web search engines became dramatically better with algorithms like PageRank [7] that observed that the Web's interconnections contained information about what pages might be interesting to a Web searcher. Similarly, in Chapter 5 we note that it is a Web editor's job to make Web sites that provide a consistent interface that provides contextual clues to the Web site's readers. We discuss how portions of a Web site's structure, like the URL and the table layout, can be viewed as a tree. That structure might help a computer algorithm understand relatedness. We make the observation that items near one another in a tree often share similar characteristics, and then create a formal, probabilistic model that reflects that observation. We discuss algorithms that classify over the model and are appropriate for real-world classification problems.

In Chapter 6 we discuss the application of the algorithms and features described in Chapter 5. We describe one way to build an ad-blocker by learning new rules based on the URL of the links. We show that our ad-blocking techniques performs comparably to a commercial ad-blocker, even when trained by a simple heuristic requiring no human input. We also describe a user study of 179 people who hand-labeled news recommendation stories. We compare various techniques for predicting the way users empirically labeled the data. The tree-based features coupled with the tree-based algorithms performed significantly better than existing algorithms applied to the text of the target document.

We use the various classifiers together in creating our filtering/recommendation system for Web news, the *Daily You*[2]. The Daily You generates news recommendations based on past clicked-upon stories. Chapter 7 discusses the real-world aspects of engineering such a system.

[2] `available for public use at http://falcon.ai.mit.edu`

1.1 Tackling the Poor Assumptions of Naive Bayes for Text Classification

We have already discussed our desire to create real-world Web applications using classification. In Chapter 3, we take the so-called "punching-bag of classifiers" [36] and try to improve its performance on text problems so that it is competitive with state of the art classification algorithms.

Naive Bayes' severe assumptions make it scalable but often inaccurate. We identify some of the problems that Naive Bayes suffers from, and propose simple, heuristic solutions to them. We address systemic issues with Naive Bayes classifiers and problems arising from the fact that text is not generated according to a multinomial model. Our experiments indicate that correcting these problems results in a fast algorithm that is competitive with state of the art text classification algorithms such as the support vector machine.

One such problem is that when there are an unequal number of training points in each class, Naive Bayes produces a biased decision boundary that leads to low classification accuracy. We propose a "complement class" method that boosts the absolute amount of training data per class while also making the ratios of training data in different classes more equal.

We also show that applying some pre-processing steps, commonly used in information retrieval (a variant of TFIDF, or term-frequency inverse document frequency), can significantly boost Naive Bayes' accuracy on text classification problems. We give some empirical analysis of the data that helps to explain why the transforms might be useful. For instance, we show how the transforms convert the data such that the transformed data matches our target multinomial model better than the original data did.

We borrow tools commonly used to analyze other classifiers to show that the standard Naive Bayes classifier often generates weight vectors of different magnitudes across classes. This has the effect of making some classes "dominate" the others; that is, they have a disproportionate influence on the classifier's decisions. We propose this effect is partially due to Naive Bayes' independence assumption and suggest that normalizing those weights—equalizing the magnitude of each classes' weight vector—helps reduce the problem.

Modifying Naive Bayes in these ways results in a classifier that no longer has a generative interpretation. However, since our concern is classification, we find the improved accuracy a valuable trade-off. Our new classifier approaches the state-of-the-art accuracy of the support vector machine (SVM) on several text corpora. We report results on the Reuters, 20 News, and Industry Sector data sets. Moreover, our new classifier is faster and easier to implement than the SVM.

1.2 Statistics-Based Bundling

In Chapter 3, we described work on making a fast, inaccurate classifier more accurate in our overarching goal of finding fast, accurate classifiers suitable for Web appli-

cations. In Chapter 4, we take an accurate but slow classifier, the support vector machine (SVM), and speed it up (Chapter 4). Our method, known as bundling, tries to summarize sets of training points into a condensed set of representative points. Having fewer points speeds up the SVM algorithm, and we argue that enough information is retained that the SVM still produces a good classification boundary.

Suppose one wanted to perform classification on a problem with only two classes. One standard approach, used when each data point is represented by a "feature vector" in some high-dimensional space, is to assume that a plane can be used to divide the classes. Classifiers that assume a plane can be used to separate two classes are called "linear classifiers."

Perhaps the simplest linear classifier is known as Rocchio [47]. Rocchio simply takes the centroids (means) of each class, and uses the perpendicular bisector of the centroids as the separating plane. Rocchio is a fast, but somewhat inaccurate, linear classifier.

The SVM has been proven an excellent general-purpose classifier, performing well across a variety of domains. On many head-to-head classification tasks, it achieves high accuracy (e.g. Yang's comparison of various algorithms on the text domain [59]). This sort of general purpose, high-accuracy algorithm comes at the cost of scalability: the standard SVM implementations, SvmFu and SvmLight, can take a very long time to complete. When applied to real-world data discussed in Chapter 4, some tests took more than a week to complete.

Rocchio can be thought of as replacing a set of points with a single representative point—the centroid—before finding the separating plane. Bundling tries to replace smaller clusters of points with their representative centroids, and then to apply the SVM to the representative set.

Our bundling work views the SVM and Rocchio as two classifiers along a continuum that moves from slow but accurate to fast and less accurate. By sending different numbers of representative points (a sort of "partial Rocchio") to a SVM (ranging from all the points to one per class), one can control the amount of speed-up the SVM realizes.

Bundling consists of two pre-processing steps: partitioning the data and combining the data. Partitioning the data is the process of selecting which points should be bundled together. The step needs to be fast but also do a reasonable job of grouping together similar points. We suggest two possible partition algorithms: a random partition and a partition based on Rocchio which tries to group together nearby points.

The next step, combining the data, takes each set of the partitioned data and "bundles" them into the centroid of that bundle.

We compared the bundled-SVM technique to several other existing speed-up techniques like subsampling, bagging, and feature selection. On several standard text corpora, including Reuters, Ohsumed, 20-News and Industry Sector, the bundled-SVM technique performs consistently well at speeding up classification and usually outperforms the other techniques. In addition we suggest a more general framework for applying bundling to other domains.

1.3 Using Pre-Existing Hierarchies as Features for Machine Learning

In Chapter 5 we observe that the Web's rich, human-oriented structure produces features that may be more effective than text classification for certain Web applications.

As the field of information retrieval moved from flat text to the Web, researchers noticed that search performance could benefit greatly by paying attention to the link structure of the Web. Work such as PageRank observed that multiple links to a page indicated a quality page [7]. Rather than trying to understand a page by looking at the text alone, PageRank sought to use the structure of the Web to estimate the answers to more human-oriented questions like "what is high quality?" Later extensions of PageRank also dealt with personalized versions of the algorithm [24].

In a similar vein, we want to exploit structures in the Web that might help us answer, for example, a human-oriented question like "what is interesting?" We observe that many Web-sites work hard to find and highlight stories of interest to their readership. A Web editor tries to drop semantic clues to orient readers to context and meaning. It is these clues we want to exploit for answering questions like "is this interesting to the user?" or "is this an advertisement?"

Most people intuit that the top-center "headline" portion of most news pages is the story the editors think is of the widest possible interest at that moment in time. We want our classification algorithm to leverage the editor's work in choosing and placing stories; it may be easier to decipher the editor's clues than to imitate the editor and understand the content of a document based on its text alone.

We observe that many editors use the visual placement of a link in a page to provide clues about the document. The visual placement, which depends on HTML table elements, can be parsed into a tree. We suggest classification algorithms that follow our intuitions about how position in a hierarchical tree might relate to our classification problems. For example, we might intuit that the top-center of a news page contains a collection of links that will be interesting to the average user of the Web site.

To implement the idea of classification using tree-based features, we solved several problems. We developed a classification model that allows us to train and make predictions using a tree-based feature. That model is a formalization of the idea that "things near one another in a tree are more likely to share a class." We use the concept of a mutation, a low-probability event that flips the class of a child from its parents, to convert our intuitions about trees into a Bayes-net over which we can perform standard inference procedures.

On the algorithmic front, we adapt Bayes-net learning to our application domain. Our problem is "online" in that a user will be expected to interleave labelings of items (interesting Web pages) with queries (requests for more interesting Web pages). We show how to update our classifier *incrementally* when new examples are labeled, taking much less time than reevaluating the entire Bayes-net. The update time is proportional to the depth of the new training point in the taxonomy and independent of the total size of the taxonomy or number of training examples. Similarly, we give

an algorithm for querying the class of a new testing point in time proportional to its depth in the tree, and another algorithm for dealing with real-world precision issues.

1.4 Empirical Results of Tree Learning

We follow our more theoretical description of tree-based features and algorithms with empirical results that support their use in Web applications (Chapter 6).

First we discuss ad-blocking, an example we have used to motivate much of our work. Ad-blocking is a difficult problem because advertisements change over time, in response to new advertising campaigns or new ad-blocking technology. When new advertisements arrive, a static ad-blocker's performance suffers; typically engineers must code a new set of rules, and users must periodically download those rules.

As we have mentioned, our approach is to have the computer learn sets of rules; such an approach means less work for the engineers and fewer rule updates for the users. In order to require no human training, we use a slow, but reasonably accurate, heuristic to label links on a page as either advertisements or not. Our tree-learning technique then tries to generalize from the labeled links to produce rules for ad-blocking which are applied to the links on previously unseen pages. We show that such a system can have commercial-level accuracy, without requiring updates on the part of either engineers or users.

Second we discuss recommendation systems, another example of the type of time and attention saving applications we are interested in building. We performed a 176-person user study which asked users to indicate what stories were interesting to them on several day's worth of the *New York Times*. We used a variety of algorithms and features, including the tree-algorithms and tree-features we have mentioned, to predict each user's interest in stories, given examples of other stories labeled as interesting or not.

The results of our user-study and classification test showed that our tree-learning algorithm coupled with the URL feature performed better than other combinations of algorithms (including the SVM) and features (like the full-text of each story). Our results helped us build the Daily You, a publicly-usable recommendation system discussed next.

1.5 The Daily You News Recommendation System

We put our algorithms and insights into use in our application, the Daily You. Like a good assistant, it strives to save your time by anticipating your interests (defined here as predicting the links a user will click on), then scanning the Web and saving the content you might have clipped yourself. In Chapter 7 we describe the work it takes to build and evaluate a real system for news recommendations.

All of these issues arise because we want the system to be stable and responding to requests in real-time. We describe the building of such a system, which includes frequent caching of information, prioritization of activities and other tradeoffs to make the system really work.

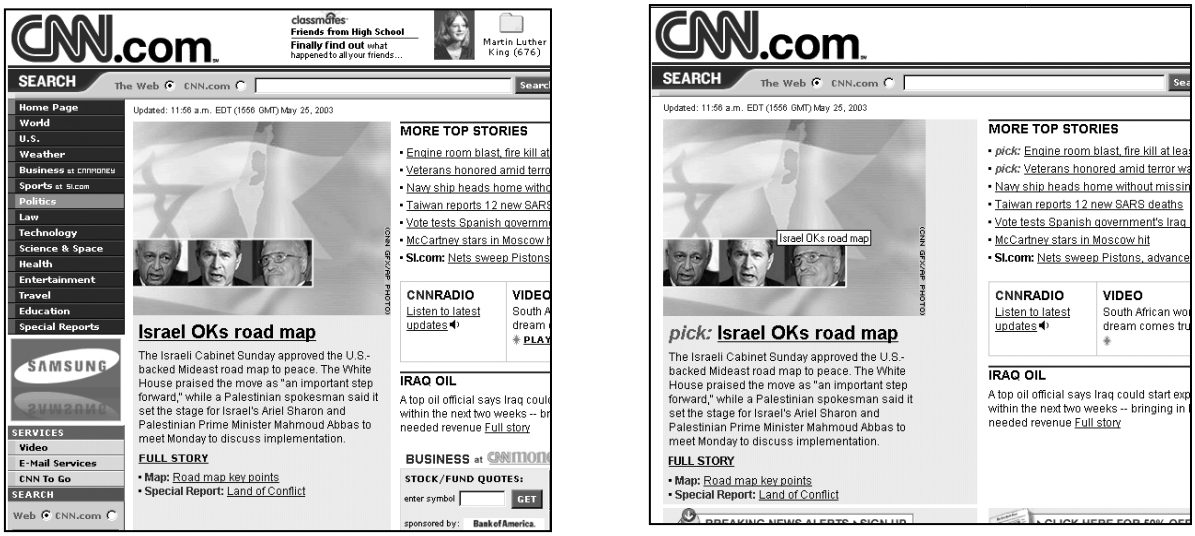


Figure 1-1: Screen-shots from an original CNN page (left) and the same page viewed through the Daily You (right). Notice the Daily You's version removes the advertisements, some of the navigation boxes, and also writes the word "pick" near recommended news articles.

A screen-shot of the Daily You is shown in Figure 1-1, with the original version of a page on the left and the Daily You's version of the same page on the right. The image on the right has less visual clutter on the page. The ad-blocking system removes the advertisements (for instance the advertisement at the top, to the right of the CNN logo); and the recommendation system adds in the work "pick" next to recommended links (the headline of the day). Other systems within the Daily You are also described. One system, the portal, invisibly tracks a user's clicked links and sends the information to the recommendation system. Another system, the page filter, removes certain templated content like the navigation bar on the left side of the page. We removed the navigation bars because from a brief user-study, it appears that most people were interested in the dynamic contents of a page.

1.5.1 Related Work

Our application resembles that of other prominent Web news services. Most existing work only operates on pre-set sites, while ours allows the users to specify arbitrary target sites. The large commercial services aggregate pages together ([3]), but require pre-set sites and do not make user-specific recommendations. Some research applications like NewsSeer.com and NewsDude [4] do make recommendations, but only from pre-set news sources. NewsDude specifically uses text-based classifiers both to select good articles, and to remove articles that seem overly redundant with already seen articles. An earlier attempt at user-profiling, called Syskill & Webert [42], also used a Bayesian model of text to predict interesting Web pages. Newsblaster [2] and a sim-

[3] `http://my.yahoo.com`

ilar service at Google ([4]) scan multiple pre-set sites and summarize similar articles. Newsblaster uses complex natural language parsing routines to combine articles from multiple sites into one summary article.

Other Web applications allow the user to select their own set of news sources, like the Montage system [1]. Rather than focusing on new information, Montage is more like an "automated book-marks builder." It watches the user to identify pages they visit frequently, and creates collections of Web pages that let the user get to those pages more quickly. It uses traditional text-classification algorithms (SVMs) to break these book-marks into coherent topic categories. In contrast, our system aims to recommend Web pages that are new but that we believe will be interesting to the user.

RSS is a system for publishing news information automatically in a format that is easy for machines to understand [5]. In practice, major news sites do not publish their content in machine-understandable formats, because such publication is at odds with their major revenue stream of advertising. The Daily You tries to learn some of the information that an RSS feed might make explicit, if an RSS feed were available.

Another type of related work is wrapper induction [33]. In wrapper induction, the goal is to identify the part of a Web page containing a specific piece of information. An example goal would be to find a specific pattern, like a stock's quote on some financial Web page, day after day. In contrast, we try to generalize from a specific item to other similar items (for example, breaking news about the same company as the stock quote).

There are several potential and existing ways to perform news recommendations, briefly detailed below.

Some approaches can broadly be viewed as using explicit or implicit "voting" information to inform recommendations. Collaborative filtering statistically finds similarities between a given user and other users in the system [51]. The typical method is to identify a user with a vector of 1's (clicked) and 0's (not-clicked) for each possible recommendation item. By taking a dot product (or other similarity measure) with other user's interest vectors, they identify people or groups of people who are similar. The system then recommends items the similar group has chosen that the user has not. Collaborative filtering is one of the most popular methods for recommendations but isn't as useful for the news service. Collaborative filtering needs a large user base, which the Daily You does not have, and it recommends commonly-viewed items which tend to be things that are "not news" (like events that everyone already knows about).

There is also a more implicit method for measuring user votes, the sub-field of link analysis, where the algorithm scores a page by its position in the topology of the Web. Common examples include PageRank, which is a part of the Google search engine [7], and Kleinberg's hubs and authorities algorithm [30]. The PageRank algorithm calculates the probability a "random surfer" on the Web, who randomly picks out-links on a page, will end up on a given page. The higher the probability, the

[4] `http://news.google.com`

[5] `http://www.w3.org/2001/10/glance/doc/howto`

more recommended that page becomes. The hubs and authorities algorithm gives a higher "hub" score to pages that point to lots of good "authorities" and gives high "authority" scores to pages pointed to by good "hubs." These are not appropriate for the recommendation problem because they are not personalized to the user; as with most search engines, the results are not based on past interests on the part of the user.

Chapter 2

Background

2.1 Classification

Classification is a problem where a learner is given several labeled training examples and is then asked to label several formerly unseen test examples. This is done regularly by humans: a child might be told the color of some blocks, then asked to identify the color of other objects. A friend might list a series of movies they enjoy, and one might then suggest other movies she will enjoy.

Within computer science, classification has also been applied to a variety of domains. For example, Amazon's[1] familiar recommendation system predicts what products you might want to buy given past behavior. Other systems receive pictures of handwritten numbers and learn to read new hand-written numbers [35]. UCI maintains a large database of machine learning problems [2] ranging from breast-cancer detection to guessing whether solar flares will occur.

Throughout this thesis, we use $\vec{x} = \{x_1 \dots x_n\}$ to represent a vector of n training points, and $\vec{y} = \{y_1 \dots y_n\}$ to represent its labels. Each training point x_i typically has a set of j features $\vec{x_i} = \{x_{i1}, x_{i2}, \dots, x_{ij}\}$ that may or may not help the learner predict labels. For example, a child's blocks might have features like shape, color, and weight; eventually they will learn to disregard the shape and weight when deciding what color label the block receives.

One common way to visualize the problem is to think of each of the n points in a j dimensional space, where j is the number of features. Then, one simple way to classify new test elements is known as the k-nearest neighbors classifier: you predict a label based on the majority-label of the k nearest training points [59]. This can be slow—one doesn't compare a fire hydrant to every block from child-hood in order to deduce that the hydrant is red. The goal is to generalize from those initial examples and to make new classifications based on that generalization.

One simple generalization is to take a plane that cleaves through this j dimensional space. These are known as "linear classifiers." If there are only two possible labels -1 and 1, then one side of this plane will receive a -1 label and everything else will

[1] `http://amazon.com`

[2] `http://www.ics.uci.edu/ mlearn/MLRepository.html`

receive a 1. This approach is seen in many common classifiers like Naive Bayes [15], the perceptron [5], and the linear support vector machine [8]. The only difference between these classifiers is the method in which they find that separating hyperplane.

Sometimes a more complicated shape than a simple plane is needed to generalize. The most common method for generalizing such shapes is to use so-called kernel methods (see Burges [8] for a tutorial) which project the problem into a higher-dimensional space, and create a linear separator in that space. When that plane is returned to a lower dimensional space, it takes on a more complex form than a simple plane.

2.1.1 Text Classification

Text classification has many properties that have made it one of the most common classification domains. Document text is easily and broadly available to researchers, and covers a broad range of topics. One standard text dataset labels the corpora according to news subject [34]; another according to keywords used to describe medical documents [22].

Text problems are among the largest standard classifier problems. Text is generated during the normal course of work, so the number of training points (documents) is often larger than other problems. The Ohsumed text dataset has about 180,000 documents; most non-text problems have several orders of magnitude less data. For example, most of the UCI datasets have only hundreds of instances because the information is difficult to collect. The breast cancer database, for example, is limited to the number of patients a particular doctor interviewed. Text also has orders of magnitude more features than many standard problems. There are tens of thousands of words even if one ignores word ordering, and all of them are potentially features. In most other domains, features are fairly limited because every new feature must be measured; most of the UCI datasets have fewer than a hundred features.

Besides having extremely large data sets, text classification has a tendency to be used in user-oriented applications, meaning that speed is more important. There are natural applications of text learning to Internet data (for example news recommendations [4]) and the application, including the recommendations, needs to complete in reasonable amount of time.

This is part of what makes text classification so interesting: it is larger than most problems, but also needs to complete in a practical amount of time. One nearly standard way of reducing the size of the problem is to assume that word ordering is irrelevant. This reduces the problem from having a large number of word-ordering relationships to a merely huge number of words. The standard representation, which we use, is called "bag-of-words", in which every document x_i is composed of features $x_{i1}, x_{i2}, \ldots, x_{iV}$ such that x_{ij} is the number of times word j appears in x_i, and V is the total number of words (vocabulary) of all the documents.

2.1.2 Standard Classification Algorithms

In this section, we describe several common classification algorithms that will be used and compared in this thesis.

Rocchio

Several times in this thesis, we mention the Rocchio classification algorithm [47]. For completeness we describe it in full detail here. Consider a binary classification problem. Simply put, Rocchio selects a decision boundary (plane) that is perpendicular to a vector connecting two class centroids. Let $\{\vec{x}_{+1}, \ldots, \vec{x}_{+l_+}\}$ and $\{\vec{x}_{-1}, \ldots, \vec{x}_{-l_-}\}$ be sets of training data for the positive and negative classes, respectively. Let $\vec{c}_1 = \frac{1}{l_+} \sum_i \vec{x}_{+i}$ and $\vec{c}_2 = \frac{1}{l_-} \sum_i \vec{x}_{-i}$ be the centroids for the positive and negative classes, respectively. Then, we define the Rocchio score of an example $\vec{x}$ as

$$\text{RocchioScore}(\vec{x}) = \vec{x} \cdot (\vec{c}_1 - \vec{c}_2). \tag{2.1}$$

One selects a threshold value, b, which may be used to make the decision boundary closer to the positive or negative class centroid. Then, an example is labeled according to the sign of the score minus the threshold value,

$$l_{\text{Rocchio}}(\vec{x}) = \text{sign}\left(\text{RocchioScore}(\vec{x}) - b\right). \tag{2.2}$$

Multinomial Naive Bayes

The Naive Bayes classifier has a long history. One of the earlier descriptions is given by Duda and Hart [14]. It is commonly used because it is fast, easy to implement and relatively effective. Domingos and Pazzani [13] discuss the independence assumption and why Naive Bayes is able to perform better than expected for classification even though it makes such a poor assumption. McCallum and Nigam [37] posits the multinomial as a Naive Bayes-like model for text classification and shows improved performance compared to the multi-variate Bernoulli model due to the incorporation of frequency information. It is the multinomial version, which we will call "multinomial Naive Bayes" or MNB for short, that we discuss and analyze in this paper.

The distribution of words in a document can be modeled as a multinomial. A document is treated as a sequence of words and it is assumed that each word position is generated independently of every other. For classification, we assume that there are a fixed number of classes, $c \in \{1, 2, \ldots, m\}$, each with a fixed set of multinomial parameters. The parameter vector for a class c is $\vec{\theta}_c = \{\theta_{c1}, \theta_{c2}, \ldots, \theta_{cn}\}$, where n is the size of the vocabulary, $\sum_i \theta_{ci} = 1$ and θ_{ci} is the probability that word i occurs in class c. The likelihood of a document is a product of the probabilities of the words that appear in the document,

$$p(x_i|\vec{\theta}_c) = \frac{\left(\sum_j x_{ij}\right)}{\prod_j x_{ij}} \prod_j (\theta_{ci})^{x_{ij}}, \tag{2.3}$$

where x_{ij} is the frequency count of word j in document x_i. By assigning a prior distribution over the set of classes, $p(\vec{\theta}_c)$, we can arrive at the minimum-error classification rule [14] which selects the class with the largest posterior probability,

$$l(x_i) = \text{argmax}_c \left[\log p(\vec{\theta}_c) + \sum_j x_{ij} \log \theta_{cj} \right], \tag{2.4}$$

$$= \text{argmax}_c \left[b_c + \sum_j x_{ij} w_{cj} \right], \tag{2.5}$$

where b_c is the threshold term and w_{cj} is the class c weight for word j. These values are natural parameters for the decision boundary. This is especially easy to see in the binary classification case where the boundary is defined by setting the differences between the positive and negative class parameters equal to zero,

$$(b_+ - b_-) + \sum_j x_{ij} \left(w_{+i} - w_{-i} \right) = 0.$$

The form of this equation (i.e. linear) is identical to the decision boundary learned by the (linear) support vector machine, logistic regression, linear least squares and the perceptron. Naive Bayes' relatively poor performance results directly from how it chooses the b_c and w_{ci}.

Parameter Estimation

For the problem of classification, the number of classes is known, but the parameters of each class are not. Thus, the parameters for the classes must be estimated. We do this by selecting a Dirichlet prior and taking the expectation of the parameter with respect to the posterior. Instead of going into details, we refer the reader to section 2 of Heckerman [21]. Suffice it to say, we arrive at a simple form for the estimate of the multinomial parameter, which involves the number of times word j appears in class c documents (N_{cj}), divided by the total number of class c word occurrences (N_c). For word j, the prior adds in α_j imagined occurrences so that the estimate is a smoothed version of the maximum likelihood estimate,

$$\hat{\theta}_{cj} = \frac{N_{cj} + \alpha_j}{N_c + \alpha}, \tag{2.6}$$

where α denotes the sum of the α_i. Technically, this allows a different imagined count for each word, but we follow common practice by using the same count across words.

The prior class probabilities, $p(\theta_c)$, are estimated in a similar way. However, they tend to be overpowered by the multiplied effect of the per-word probabilities that Naive Bayes produces, and have little effect on the decision boundary.

Substituting our estimates for the true parameters in equation 2.4, we get the

multinomial Naive Bayes classifier,

$$l_{MNB}(x_i) = \text{argmax}_c \left[\log \hat{p}(\theta_c) + \sum_j x_{ij} \log \frac{N_{cj} + \alpha_i}{N_c + \alpha} \right],$$

where $\hat{p}(\theta_c)$ is the class prior estimate. The weights for the decision boundary defined by this classifier are the log parameter estimates,

$$\hat{w}_{cj} = \log \hat{\theta}_{cj}. \tag{2.7}$$

We will refer back to this fact during our discussion of skewed data bias in section 3.2.1.

The support vector machine

The support vector machine (SVM) is a classifier that finds a maximal margin separating hyper-plane between two classes of data [56]. In an intuitive sense, the SVM tries to place a block of wood such that everything on one side of the block is in one class and everything on the other side of the block is in the other class (the equation constraints). The optimization problem is to find the widest possible block of wood represented in the equations by $\|w\|^2$. The idea is a wider block of wood is a better separator of the two class points.

One common variant is to "soften" the boundary; rather than completely ruling out points on the wrong side of the wooden block, it simply punishes the "score" for every point that is on the wrong side of the block. The $C \sum_i \xi$ term in the equations tells the optimization how much to punish points on the wrong side of the block (at C=infinity, the constraint is absolute; as C decreases, the algorithm is more accepting of points on the wrong side of the wooden block).

An SVM is trained via the following optimization problem:

$$\hat{w} = \text{argmin}_w \frac{1}{2}\|w\|^2 + C \sum_i \xi_i,$$

with constraints

$$\begin{aligned} y_i(x_i \cdot w + b) &\geq 1 - \xi_i \quad \forall i, \\ \xi_i &\geq 0 \quad \forall i, \end{aligned}$$

where each d_i is a document vector, y_i is the label ($+1$ or -1) for d_i and $\hat{w}$ is the vector of weights that defines the optimal separating hyper plane. This form of the optimization is called the "primal." By incorporating the inequality constraints via

Lagrange multipliers, we arrive at the "dual" form of the problem,

$$\hat{w} = \text{argmax}_w \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (d_i \cdot d_j)$$

subject to

$$0 \leq \alpha_i \leq C \quad \forall i$$
$$\sum_i \alpha_i y_i = 0$$

Given optimized values for the α_i, the optimal separating hyper plane is

$$\hat{w} = \sum_i \alpha_i y_i x_i.$$

For more information about the SVM, see Burges' tutorial [8], Cristianini and Shawe-Taylor's book [11], and Vapnik's book [57].

2.1.3 Evaluating Classifiers

In order to compare classifiers against one another, typically the experimenter will split all the labeled into two disjoint sets, the training and testing sets. The training set is used to create a set of rules about how to classify points, and the testing set is then classified with those rules.

Those rules produce guesses for the test data that are compared with the actual labels. One common method of evaluating classifiers, *accuracy*, is simply seeing the percentage of the test data where the guesses equaled the actual labels.

For some domains like information retrieval, a different evaluation method is more common, which emphasizes the ranking a classifier produces. The most common way of evaluating a system is by using precision-recall curves. There are two sets of documents: a set of m documents that are actually relevant (as judged by a human) and a set of n documents that the system "retrieves", i.e., believes are relevant. The more these two sets match, the better, and this is what precision-recall tries to describe. Let o be the number of documents that the human and computer agree are accurate. Then precision is defined as o/n and recall is defined as o/m.

If the computer were perfect, and retrieved exactly the same set of documents as the human-labeled relevant set, then both precision and recall would equal 1 ($o = n = m$). In the normal case, the computer-retrieved and human-relevant sets will be different. Precision then is generally at odds with recall. For example, if the computer simply labels everything as relevant, then the computer will have high recall (it found all the relevant documents) but low precision (but it found too many irrelevant documents, too). Conversely, if the computer only makes one very good guess, it will receive high precision (everything the system retrieved was relevant) and low recall (only a small fraction of the relevant documents were found).

The typical way to show precision-recall graphically is to vary the number of

documents retrieved (retrieve the top document, then the top two, then the top n...) and for each value of n plot precision against recall. To reduce precision-recall to a single number, often people employ precision-recall breakeven, which is when $o/n = o/m$. Mathematically, that corresponds to when $n = m$ (when the set of retrieved documents is equal in size to the set of relevant documents).

Chapter 3

Tackling the Poor Assumptions of Naive Bayes for Text Classification

3.1 Introduction

Given our desire to build Web applications that function in real time, and can handle the huge amounts of data available on the Web, one natural approach is to take a fast but inaccurate classifier and improve it. We aim to improve Naive Bayes, which is often used as a baseline for text classification tasks because it is fast, easy to implement, and has a clear set of assumptions about the way data is generated. Background on the Naive Bayes classifier can be found in Chapter 2. This chapter is the expanded version of joint work with Jason Rennie, Jaime Teevan and David Karger [45].

Naive Bayes is often used as a baseline for text classification tasks because it is fast and easy to implement. Its "naive" assumptions make such efficiency possible but also adversely affect its performance. We identify some of the problems that Naive Bayes suffers from, and propose simple, heuristic solutions to them. We first address systemic issues with Naive Bayes, which are problems that arise out of inherent problems with the way Naive Bayes produces a classification boundary. Then we address problems that arise from the fact that text is not generated according to a multinomial model. Our experiments indicate that correcting these problems results in a fast algorithm that is competitive with state-of-the-art text classification algorithms such as the support vector machine.

Naive Bayes has been denigrated as "the punching bag of classifiers" [36], and has earned the dubious distinction of placing near last in numerous head-to-head classification papers [59, 26, 16]. Still, it is frequently used for text classification because of its speed and ease of implementation. Algorithms that yield lower error tend to be slower and more complex. In this chapter, we investigate the reasons for Naive Bayes' poor performance. For each problem, we propose a simple heuristic solution. In some cases, we treat Naive Bayes as a linear classifier and find ways to improve the weights that it learns for the decision boundary. In others, we try to better match the distribution of text with the distribution assumed by Naive Bayes.

In doing so, we fix many of the problems without making Naive Bayes significantly slower or more difficult to implement.

In Section 3.2, we discuss some systemic problems with the classifier. One such problem is that when there are more training examples in one class than another, Naive Bayes chooses weights for the decision boundary poorly. This is due to an under-studied bias effect that causes the weights for the class with fewer examples to be more extreme than they should be. For example, after one flip of a coin (without smoothing), the probability of heads looks like either 100% or 0%. We solve this problem by introducing a "complement class" formulation of Naive Bayes that evens out the amount of training data per class.

Another systemic problem with Naive Bayes is that it does not account for dependencies between words. As a result, when multiple words are inter-dependent, they will receive their own independent weights (as opposed to perhaps a single weight for an entire phrase). This leads to double counting for words like "San" and "Francisco." If one class has stronger word dependencies, the sum of weights for that class will tend to be much larger. To keep classes with more weight from dominating, we normalize the classification weights.

In addition to other problems, multinomial Naive Bayes does not model text well. It uses the multinomial model for term occurrence, which is a poor model of text. In Section 3.3 we suggest several transforms to make text work better within the multinomial model. These transforms are common in information retrieval, so we present arguments why they should work well with Naive Bayes. For example, we show that a simple transform allows us to emulate the power law, a distribution that better matches real term frequency distributions. We also discuss two other pre-processing steps that incorporate knowledge of how text documents are usually created. Empirically, the use of these transforms significantly boosts classification accuracy.

Modifying Naive Bayes in these ways results in a classifier that no longer has a generative interpretation. However, since our concern is classification, we find the improved accuracy a valuable trade-off. Our new classifier approaches the state-of-the-art accuracy of the support vector machine (SVM) on several text corpora. Moreover, our new classifier is faster and easier to implement than the SVM and other modern-day classifiers.

3.2 Correcting Systemic Errors

Naive Bayes has many systemic errors—byproducts of the algorithm that cause an inappropriate favoring of one class over the other. In this section, we will discuss two errors that cause Naive Bayes to perform poorly. What we call "skewed data bias" and "weight magnitude errors" have not been discussed heavily in the literature, but both cause Naive Bayes to routinely make misclassifications. We discuss how these errors cause misclassifications and propose solutions to mitigate or eliminate them.

Coin 1 $\theta = 0.6$	Coin 2 $\theta = 0.4$	Coin 3 $\theta = .8$	p(event)	$\hat{\theta}_1$	$\hat{\theta}_2$	$\hat{\theta}_3$	NB Label for H
H	T	.8H, .2T	.36	1	0	.8	Coin 1
H	H	.8H, .2T	.24	1	1	.8	Coin 1 or 2
T	T	.8H, .2T	.24	0	0	.8	Coin 3
T	H	.8H, .2T	.16	0	1	.8	Coin 2

Figure 3-1: Shown is a simple classification example. There are three coins. Each has a binomial distribution with probability of heads $\theta = 0.6$, $\theta = 0.4$, and $\theta = .8$ respectively. The first two coins are flipped once each, and the third coin is flipped a million times. Even though the probability for heads is greatest for the third coin, it only appears as the most probable coin 24% of the time, versus 60% of the time for coin 1.

3.2.1 Skewed Data Bias

In this section, we show that *skewed data*—more training examples in one class than another—can cause the decision boundary weights to be biased. This causes the classifier to perform poorly and unwittingly prefer one class over the other. We show the reason for the bias and propose a fix that alleviates the problem.

A simple example shows that a small amount of training data can skew probabilities towards extremes. Suppose we have three coins, and we flip them a varying number of times. In reality, the first coin has a 60% chance of landing heads, the second coin a 40% chance of landing heads, and the third coin has an 80% chance of landing heads. We flip the first two coins once each, and the third one a million times. Let's say the most likely outcome (with 36% probability) has the first coin coming up heads, the second one tails, and the third one consistently shows heads 80% of the time after a million flips.

Someone randomly takes one of the three coins, flips it, and announces it is a head. Which coin might it be? One might suppose the first coin, since the evidence shows it has a 100% chance of landing heads—perhaps it is a two-headed coin. Or one might suppose the first two coins were fair, and that randomly one of the two landed heads. Figure 3-1 shows that even when coin 3 has the highest probability of heads, it is chosen less often than coin 2, even though coin 2 is half as likely to show a head.

The problem is that empirical probabilities are extreme–either one or zero. The problem isn't solved by smoothing; light smoothing does not change the example, and heavy smoothing causes other problems as the empirical evidence is ignored in favor of the smoothing effects.

Our problem is caused by the small amounts of training data per class. Were coins 1 and 2 to receive additional flips, coin 3 would seem increasingly likely when presented with a head. One idea might be to replicate some of the examples ("pretend" that coin 1 was in fact flipped as many times as the other coins). Replicating examples does not solve this problem, because it leaves the empirical probabilities of coins 1

Coin 1 $\theta = 0.6$	Coin 2 $\theta = 0.4$	Coin 3 $\theta = .8$	p(event)	$\hat{\theta'}_1$	$\hat{\theta'}_2$	$\hat{\theta'}_3$	Label for H
H	T	.8H, .2T	.36	.8-	.8+	.5	Coin 3
H	H	.8H, .2T	.24	.8+	.8+	1	Coin 1 or 2
T	T	.8H, .2T	.24	.8-	.8-	0	Coin 3
T	H	.8H, .2T	.16	.8+	.8-	.5	Coin 3

Figure 3-2: Shown is the same classification example, when classification is done using the complement class formulation, with the same set-up as Figure 3-1. CCNB uses everything outside a class to produce an estimate for $\hat{\theta'}$, then chooses the class which has the lowest estimate. The symbol ".8+" means that the estimate is slightly higher than .8; and ".8-" is slightly lower than .8. In this case, coin 3 is properly chosen 76% of the time, versus only 24% of the time when using the standard formulation.

and 2 unchanged at either 0% or 100%.

Rather, our proposed solution is to use the other classes to supplement the existing empirical data. Note this is only applicable to multi-class problems, or problems involving more than two classes (different coins in our example). We introduce a "complement" version of Naive Bayes, called Complement Class Naive Bayes, or CCNB for short. In CCNB we use everything *outside* the class to build a model. When our classification problem involves multiple classes, we can reformulate our classification rule so the amount of training data we use for the parameter estimates is more even between the classes. This yields less extreme weight estimates and improved classification accuracy.

In Figure 3-2 we show how the complement class formulation changes our three coin example. Rather than calculating $\hat{\theta}$, the probability that a given coin has heads, we calculate $\hat{\theta'}$, the probability that the other coins come up heads. We guess the coin with the lowest $\hat{\theta'}$ (the one whose complements are least likely to show heads), to be the most likely to flip a heads.

In estimating weights for regular MNB, we use equation 2.6. This only uses training data from a single class, c. CCNB estimates parameters using data from all classes except c,

$$\hat{\theta}_{\tilde{c}j} = \frac{N_{\tilde{c}j} + \alpha_j}{N_{\tilde{c}} + \alpha}, \tag{3.1}$$

where $N_{\tilde{c}j}$ is the number of times word j occurred in documents in classes other than c and $N_{\tilde{c}}$ is the total number of word occurrences in classes other than c; α_j and α are smoothing parameters. As before, the weight estimate is $\hat{w}_{\tilde{c}j} = \log \hat{\theta}_{\tilde{c}j}$ and the classification rule is

$$l_{\text{CCNB}}(d) = \text{argmax}_c \left[\log p(\vec{\theta}_c) - \sum_j x_{ij} \log \frac{N_{\tilde{c}j} + \alpha_j}{N_{\tilde{c}} + \alpha} \right].$$

	MNB	OVA-NB	CCNB
Industry Sector	.577 (.008)	.651 (.007)	.890 (.004)
20 Newsgroups	.847 (.006)	.853 (.005)	.857 (.004)

Figure 3-3: Experiments comparing a variety of ways to handle multi-class problems which are described in the text: multinomial Naive Bayes (MNB); one-versus-all Naive Bayes (OVA-NB); and complement class Naive Bayes (CCNB). Industry Sector and 20 News, the two multi-class problems studied in our work, are reported in terms of accuracy, with standard deviations in parentheses. CCNB generally performs better than both MNB and OVA-NB, which are more standard formulations.

The negative sign represents the fact that we want to assign to class c documents that *poorly* match the complement parameter estimates. We think these estimates will be more effective because each uses a larger amount of training data per estimate, which will lessen the bias in the weight estimates.

CCNB is related to the one-versus-all technique that is commonly used in multi-label classification, where each example may have more than one label. The one-versus-all technique builds a classifier for each class that subtracts the probability something is outside the class ("all"–more precisely "all but one") from the probability something is inside the class ("one"). Berger [3] and Zhang and Oles [61] have found that one-versus-all MNB works better than regular MNB. The classification rule for one-versus-all MNB is

$$l_{\text{ova}}(d) = \text{argmax}_c \left[\log p(\vec{\theta}_c) + \left(\sum_j x_{ij} \log \frac{N_{cj} + \alpha_j}{N_c + \alpha} - \sum_j x_{ij} \log \frac{N_{\tilde{c}j} + \alpha_j}{N_{\tilde{c}} + \alpha} \right) \right]. \quad (3.2)$$

This is a combination of the regular and complement classification rules, where the first summation comes from the regular rules and the second summation comes from the complement rules. We attribute the improvement with one-versus-all to the use of the complement weights. We find that the complement version of MNB performs better than one-versus-all and regular MNB since it eliminates the biased regular MNB weights.

Figure 3-3 compares multinomial Naive Bayes, one-versus-all Naive Bayes and complement class Naive Bayes on two multi-class data sets. For all the Naive Bayes experiments, we used our own Java code and set $\alpha = 1$. See the Appendix at the end of the thesis for a description of, and pre-processing for, the Industry Sector and 20 Newsgroups data sets. The differences between the various formulations were all statistically significant ($p < 0.05$) according to Student's T-test above 95% for both Industry Sector and 20 Newsgroups.

3.2.2 Independence Errors

In the last section, we discussed how uneven training sizes could cause Naive Bayes to bias its weight vectors. In this section, we discuss how the independence assumption can erroneously cause Naive Bayes to produce classifiers that are overly biased towards one class.

When the magnitude of Naive Bayes' weight vector $\vec{w}_c$ is larger in one class than the others, the larger-magnitude class may be preferred. For Naive Bayes, differences in weight magnitudes are not a deliberate attempt to create greater influence for one class. Instead, as we will show, the weight differences are partially an artifact of applying the independence assumption to dependent data. In other words, Naive Bayes gives out more influence to the classes that most violate the independence assumption. The following simple example illustrates this effect.

Consider the problem of distinguishing between documents that discuss the Yankees and those that discuss Red Sox. Let's assume that "Yankees" appears in Yankees documents about as often as "Red Sox" appears in Red Sox documents (as one might expect). Let's also assume that it's rare to see the words "Red" and "Sox" except when they are together. Thus the three terms "Red", "Sox" and "Yankees" all have the same weight after training. Then, each time a test document has an occurrence of "Red Sox," Multinomial Naive Bayes will double count—it will add in the weight for "Red" and the weight for "Sox." Since "Red Sox" and "Yankees" occur equally in their respective classes, a single occurrence of "Red Sox" will contribute twice the weight as an occurrence of "Yankees." Hence, the summed contributions of the classification weights may be larger for one class than another—this will cause MNB to prefer one class incorrectly. For example, if a document has five occurrences of "Yankees" and three of "Red Sox," MNB will erroneously label the document as "Red Sox" rather than "Yankees."

A baseball analogy would have the Red Sox receiving 6 outs per inning as compared to the Yankees' 3 outs. This would (hopefully) bias most games towards the Red Sox—they would simply receive more chances to score. Our solution, technically expressed as weight normalization, would be to give the Red Sox shorter bats–in other words normalize the overall batting abilities of each team, even if one team had received more outs.

We correct for this by normalizing the weight vectors. In other words, instead of calculating $\hat{w}_{cj} = \log \hat{\theta}_{cj}$, we calculate

$$\hat{w}_{cj} = \frac{\log \hat{\theta}_{cj}}{\sum_k \log \hat{\theta}_{ck}}. \tag{3.3}$$

One could also reduce the problem by optimizing the threshold terms, b_c. Webb and Pazzani give a method for doing this by calculating per-class weights based on identified violations of the Naive Bayes classifier [58].

Since we are manipulating the weight vector directly, we can no longer make use of the model-based aspects of Naive Bayes. Thus, common model-based techniques to incorporate unlabeled data and uncover latent classes, such as EM, are no longer

	MNB	WCCNB
Industry Sector	0.577 (.008)	0.890 (.004)
20 Newsgroups	0.847 (.006)	0.859 (.005)
Reuters (micro)	0.739	0.782
Reuters (macro)	0.270	0.548

Figure 3-4: Experiments comparing multinomial Naive Bayes (MNB) to weight normalized Complement Class Naive Bayes(WCCNB) over several datasets (left column). Industry Sector and 20 News are reported in terms of accuracy; Reuters in terms of precision-recall breakeven. WCCNB generally performs better than MNB. Standard deviations are listed in parenthesis for Industry Sector and 20 Newsgroups. The differences between the classifiers are statistically significant (as outlined in the Appendix) on the Industry Sector and 20 Newsgroups data sets.

applicable. This is an unfortunate trade-off for improved classification performance.

3.2.3 Bias Correction Experiments

We ran classification experiments to validate our techniques. Figure 3-4 gives classification performance on three text data sets. We used different metrics for the different test sets according to convention in past experiments done by the community. We report accuracy for 20 Newsgroups and Industry Sector and precision-recall breakeven for Reuters. As mentioned before, we use a smoothing parameter of $\alpha = 1$. Details on the pre-processing and experimental set up for the experiments is found in the Appendix at the end of the thesis.

Reuters is a collection of binary-class problems, but the complement class formulation we explained only works on multiple-class problems. Therefore, we applied an analog of the complement class formulation to the Reuters problem. In Reuters, there is a one-to-many relationship between a document and the classes; so the standard way for running the experiment is to build a classifier that compares documents inside a class against a classifier built on all the documents. In the complement class formulation, WCCNB, we build a classifier that compares all the documents outside of a class, against a classifier built on all the documents. This only contributed a small amount to higher precision-recall scores, with the weight-normalization helping more.

WCCNB showed marked improvement on all data sets. The improvement was greatest on data sets where training data quantity varied between classes (Reuters and Industry Sector). The greatly improved Reuters macro P-R breakeven score shows that much of the improvement can be attributed to better performance on classes with few training examples (as explained in the Appendix, macro P-R scores are heavily dependent on classes with only a few training examples).

WCCNB shows a small but significant improvement on 20 Newsgroups even though the distribution of training examples is even across classes (though the number of word occurrences varies somewhat).

Our multinomial Naive Bayes (MNB) results are similar to those of others. Our 20

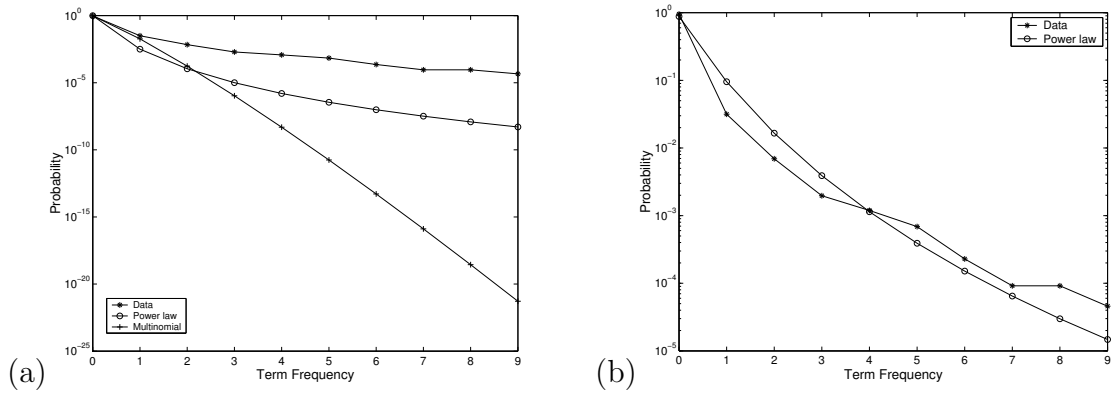


Figure 3-5: Shown are various term frequency probability distributions. The data has a much heavier tail than predicted by the multinomial model. A power law distribution ($p(x_{ij}) \propto (d + x_{ij})^{\log\theta}$) matches the data much more closely. Figure (b) compares the data with the best fit power law distribution, and Figure (a) compares it when $d = 1$.

Newsgroups result exactly matches that reported by McCallum and Nigam [37] (85%). The differences in Ghani's [18] results (64.5%) on Industry Sector are likely due to his use of feature selection. Zhang and Ole's [61] result on Industry Sector (84.8%) is significantly different due to their optimization of the smoothing parameter, which, depending on the method used, can also remove redundant features. Our micro and macro scores on Reuters differ from Yang and Liu [59] (79.6%, 38.9%), likely due to their use of feature selection, and a different scoring metric (the F1 metric).

3.3 Modeling Text Better

So far we have discussed issues that arise when using a Naive Bayes classifier in many applications. We are using a multinomial to model text; in this section we look at steps to better align the model and the data. In this section we describe three transforms that make multinomial Naive Bayes a more effective text classifier. These transforms are known in the information retrieval community; our aims in this section are to show why these transforms might work well coupled with Naive Bayes, and to give empirical evidence that they do improve Naive Bayes' performance.

There are three transforms we discuss. One transform affects frequencies—empirically, term frequency distributions have a much heavier tail (e.g. are more likely to exhibit for high frequencies) than the multinomial model predicts. We also transform based on document frequency, since we don't want common terms to dominate in classification, and based on length, so that long documents do not dominate during training. By transforming the data to be better suited for use with a multinomial model, we find significant improvement in performance over using MNB without the transforms.

3.3.1 Transforming Term Frequency

In order to understand whether MNB would do a good job classifying text, we investigated if the term distributions of text actually followed a multinomial distribution. We found that the distributions had a much heavier tail than predicted by the multinomial model, instead appearing like a power-law distribution. We show how these term distributions that look like power-law distributions can be transformed to look more multinomial.

To measure how well the multinomial model fits the term distribution of text, we compared the empirical distribution of a term to the maximum likelihood multinomial for that term. For visualization purposes, we took a set of words with approximately the same occurrence rate and created a histogram of their term frequencies in a set of documents with similar length. These term frequency rates and those predicted by the best fit multinomial model are plotted in Figure 3-5 on a log scale. This figure shows that the empirical term distribution is very different from the term distribution a multinomial model would predict. The empirical distribution has a much heavier tail, meaning that high frequencies are more probable than the multinomial model predicts. The probability of multiple occurrences of a term is much higher than expected for the best fit multinomial. The multinomial model assumes that whether a term occurs again in a document is independent of whether it has already occurred, while clearly from Figure 3-5(a), this is an inaccurate assumption. For example, the multinomial model predicts that the chance of seeing an average word occur nine times in a document is $p(f_i = 9) = 10^{-21.28}$—so low that such an event would be unexpected even in a collection of all news stories that have ever been written. In reality the chance is $p(f_i = 9) = 10^{-4.34}$—very rare in a single document, but not unexpected in a collection of 10,000 documents. This behavior, also called "burstiness", has been observed by others [9, 29].

While Church and Gale [9] and Katz [29] develop sophisticated models to deal with term burstiness, we find that even a simple heavy tailed distribution, the power law distribution, can better model the text and motivate a simple transformation to the features of the multinomial model. Figure 3-5(b) shows an example of the empirical distribution, alongside a power law distribution, $p(f_i) \propto (d + f_i)^{\log \theta}$, where d has been chosen to closely match the text distribution. This means the probability is also proportional to $\theta^{\log(d+f_i)}$. This shares a form that is very similar to the multinomial model, in which the probability is proportional to θ^{f_i}. Thus, using the multinomial model we can generate probabilities that are proportional to a class of power law distributions via a simple transform, $f'_i = \log(d + f_i)$. More precisely, if f_i has a power law distribution, then

$$f'_i = \log(1 + f_i)$$

has a multinomial distribution.

has the advantages of being an identity transform for zero and one counts, while, as we would like, pushing down larger counts. The transform results in a much more realistic handling of text without giving up the advantages of MNB. While setting

$d = 1$ does not match the data as well as an optimized d, it still results in a distribution that is much closer to the empirical distribution than the best fit multinomial, as can be seen in Figure 3-5(a).

3.3.2 Transforming By Document Frequency

Another useful transformation is to discount terms that occur in many documents. Such common words are unlikely to be related to the class of a document, but random variation creates an apparent fictitious correlation. This simply adds noise to the parameter estimates and hence the classification weights. Since these words appear often, they can hold sway over a classification decision even if the weight difference between classes is small. It would be advantageous to down-weight these words.

For example, suppose that the word "I" appears frequently in many documents of all classes. Just by chance, it will probably occur in one class more than the other; and this will lead Naive Bayes to draw chance correlations about the number of times "I" appears in each class (when there is no real correlation). Conversely, if the word "baseball" only appears a few times in one class (sports), we might believe it has a greater significance when present. Heuristically, we want a transform that decreases the strength of words that occur in many documents; while increasing the strength of words that only occur in a few.

To correct for this, a common heuristic transform in the information retrieval (IR) community is to discount terms by their document frequency [28]. This is known as "inverse document frequency." One such transform is

$$f'_i = f_i \log \frac{\sum_j 1}{\sum_j \delta_{ij}},$$

where δ_{ij} is 1 if word i occurs in document j, 0 otherwise, and the sum is over all document indices [49]. Rare words are given increased term frequencies; common words are given less weight. Its common use in IR led us to try it for classification, and we also found it to improve performance.

3.3.3 Transforming Based on Length

Documents have strong word inter-dependencies. After a word first appears in a document, it is more likely to appear again. Since MNB assumes occurrence independence, long documents can negatively effect parameter estimates. We normalize word counts to avoid this problem. Figure 3-6 shows empirical term frequency distributions for documents of different lengths. It is not surprising that longer documents have larger probabilities for larger term frequencies, but what is worth noting is that the jump for larger term frequencies is larger than the jump in average document length. Documents in the 80-160 word group are, on average, twice as long as those in the 0-80 word group, yet the chance of a word occurring five times in the 80-160 word group is larger than a word occurring twice in the 0-80 word group. This would not be the case if text were multinomial.

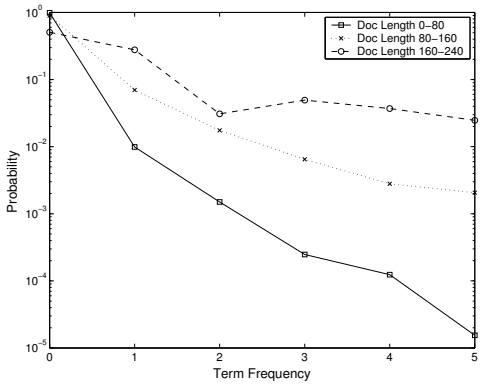


Figure 3-6: Plotted are average term frequencies for words in three classes of Reuters-21578 documents—short documents, medium length documents and long documents. Terms in longer documents have heavier tails.

We deal with this by again turning to a transform that is common in the IR community, but not seen with Naive Bayes. We discount the influence of long documents by transforming the term frequencies according to

$$f'_i = \frac{f_i}{\sqrt{\sum_k (f_k)^2}},$$

yielding a length 1 term frequency vector for each document. The transform is common within the IR community because the probability of generating a document within a model is compared across documents; in such a case one does not want short documents dominating merely because they have fewer words. For classification, however, because comparisons are made across classes, and not across documents, the benefit of such normalization is more subtle, especially as the multinomial model accounts for length very naturally [36]. The transform keeps any single document from dominating the parameter estimates.

The three transforms convert integer frequencies into floating point numbers. Some of the transforms, like the log of the term frequencies, can still be understood in generative model terms. The new model of word distributions becomes a power law rather than a multinomial. Others, like term frequency, are harder to understand on generative grounds. Therefore, the trade-offs for this transform approach is the loss of a clean generative model (which was already lost in our other modifications to the algorithm); the benefits seem to be improved text classification.

In tests, we found the frequency transform to be the most useful, followed by the length normalization transform. The document frequency transform seemed to be of less import. Results for these tests are in figure 3-7. Full results comparing the combined transforms are in the next section.

	WCCNB	F-WCCNB	L-WCCNB	I-WCCNB
Industry Sector	0.890 (.004)	0.914 (.002)	0.904 (.003)	.813 (.004)
20 Newsgroups	0.859 (.005)	0.867 (.004)	0.851 (.004)	.848 (.004)

Figure 3-7: Experiments comparing weight-normalized complement class Naive Bayes (WCCNB) with various transforms: the frequency transform (F-WBCCNB), the length transform (L-WCCNB) and the inverse document transform (I-WCCNB). Accuracy is reported and standard deviations are shown in parenthesis. We found the transform that increased accuracy the most was the frequency transform, followed by the length transform. While the inverse document transform sometimes hurt accuracy alone, when combined with the other transforms, as shown in the next section, it benefited classification accuracy. The Industry Sector results are all statistically significant, but some of the 20 Newsgroups results are not.

	MNB	WCCNB	TCCNB	SVM
Industry Sector	0.577 (.008)	0.890 (.004)	0.919 (.004)	0.928 (.003)
20 Newsgroups	0.847 (.006)	.859 (.005)	0.868 (.005)	0.865 (.003)
Reuters (micro)	0.739	0.782	0.844	0.887
Reuters (macro)	0.270	0.548	0.650	0.694

Figure 3-8: Experiments comparing multinomial Naive Bayes (MNB) to Transformed Complement Class Naive Bayes (TCCNB) to a support vector machine (SVM) over several datasets (left column). Note that our linear-time TCCNB's performance is substantially better than MNB, and comes close to the SVM's performance. Industry Sector and 20 News are reported in terms of accuracy; Reuters in terms of precision-recall breakeven. Standard deviations are reported in parenthesis. All of the Industry Sector and 20 Newsgroups data are statistically significant, with the exception of the TCCNB and SVM results for 20 Newsgroups.

3.3.4 Experiments

We have described a set of transforms for term frequencies. Each of these tries to resolve a different problem with the modeling assumptions of Naive Bayes. When we apply the transforms in the sequence described, and in conjunction with CCNB and weight normalization schemes from before, we find a significant improvement in text classification performance over MNB. Figure 3-8 shows classification accuracy for Industry Sector and 20 Newsgroups and precision-recall breakeven for Reuters. TCCNB is the application of the described transforms on top of the CCNB classifier. We show results on the Support Vector Machine (SVM) for comparison [8]. We used the transformations described in Section 3.3 for the SVM since they improved classification performance.

We discussed how our results compared to other results in section 3.2.3. Our support vector machine results are similar to others'. Our Industry Sector result matches that reported by Zhang and Oles [61] (93.6%). The difference in Godbole et al.'s [19] result (89.7%) on 20 Newsgroups is due to their use of a different multi-

class schema. Our micro and macro scores on Reuters differ from Yang and Liu [59] (86.0%, 52.5%), likely due to their use of feature selection, and a different scoring metric (F1). The larger difference in macro results is due to the sensitivity of macro calculations, which weights the smallest classes most heavily.

3.4 Conclusion

- Let $\vec{x} = (\vec{x}_1, \ldots, \vec{x}_n)$ be a set of documents;
- Let x_{ij} be the count of word j in document i.
- TCCNB($\vec{x}$)
  - Pre-process data with transforms
    replace x_{ij} with $\log(x_{ij} + 1)$ (TF transform Section 3.3.1 †)
    replace x_{ij} with $x_{ij} \log \frac{\sum_k 1}{\sum_k \delta_{ik}}$ (IDF transform Section 3.3.2 †)
    replace x_{ij} with $\frac{x_{ij}}{\sqrt{\sum_k (x_{kj})^2}}$ (length normalization Section 3.3.3 †)
  - Use the complement class formulation (Section 3.2.1)
    $\hat{\theta}_{cj} = \frac{\sum_{j:y_j \neq c} x_{ij} + \alpha_i}{\sum_{j:y_j \neq c} \sum_k x_{kj} + \alpha}$
  - Weight normalization (Section 3.2.2)
    $w_{cj} = \log \hat{\theta}_{cj}$
    $w_{cj} = \frac{w_{cj}}{\sum_i w_{cj}}$
  - Let $t = (t_1, \ldots, t_n)$ be a test document; let t_i be the count of word i in the document.
  - Label the document according to

$$l(t) = \arg\min_c \sum_i t_i w_{cj} \tag{3.4}$$

Figure 3-9: Our Naive Bayes procedure. Assignments are over all possible index values. Steps denoted with † are not part of the CCNB classifier.

We have described several techniques, summarized in Figure 3-9, that attack deficiencies in the application of the Naive Bayes classifier to text data. The complement variant solves the problem of uneven training data. Normalization of the classification weights improves Naive Bayes' handling of word occurrence dependencies. Also, a series of transforms from the information retrieval community improves Naive Bayes' text classification performance. In particular, one transform converts text, which is closely modeled by a power law, to look more like a multinomial. These modifications better align Naive Bayes with the realities of bag-of-words textual data and, as we have shown empirically, significantly improves its performance on a number of data

sets. This modified Naive Bayes is a fast, east-to-implement, near state-of-the-art text classification algorithm.

The cost of making these somewhat heuristic fixes is that the modified Naive Bayes no longer has a generative interpretation, and that the modifications are aimed particularly at text classification. While some of the steps can be thought of in generative terms, like the log transform, other steps, like weight normalization, have no generative analog. The output of Naive Bayes is no longer a probability, so algorithms like Expectation Maximization can no longer work with our modifications.

Chapter 4

Statistics-Based Data Reduction for Text

4.1 Introduction

In the last chapter, we discussed ways to improve a fast but inaccurate classifier, Naive Bayes, with the goal of finding algorithms suitable for use with Web applications. In this chapter, we discuss how a slow but accurate classifier, the Support Vector Machine (SVM), can be made faster. This chapter is the expanded version of joint work with Jason Rennie, Yu-Han Chang and David Karger [52] [54].

There is a great need to find fast, effective algorithms for text classification. A KDD panel headed by Domingos [12] discussed the tradeoff of speed and accuracy in the context of very large (e.g. one-million record) databases. Most highly accurate text classifiers take a disproportionately large time to handle a large number of training examples. These classifiers may become impractical when faced with large datasets, like the Ohsumed data set with more than 170,000 training examples [22].

The SVM is one such highly accurate, general classifier (see the background chapter 2.1.2 for more information). The SVM has consistently outperformed other algorithms [59, 25, 16, 44]. However, the SVM's ability to classify well over many domains means it can take a long period to train. Joachims [27] estimates a training time between $O(n^{1.7})$ and $O(n^{2.1})$ for n training examples. In our own experiments with the Ohsumed data set, a standard implementation of the SVM [46] did not complete training within a week.

In contrast, a fast algorithm like Rocchio [47] can complete training on the same data set in around an hour. Rocchio is a simple, fast classifier that finds the plane exactly between the centroids of the two classes (more technically, the perpendicular bisector of the centroids of the two classes). By throwing away some information—the exact location of each point—Rocchio comes up with a fast decision boundary that has reasonably high accuracy. More information on Rocchio can be found in Chapter 2.

Bundling forms a continuum of classifiers between the SVM and Rocchio, allowing one to choose a time-appropriate classifier for a given task. Bundling averages sets

of points by class into representative points and sends them to the SVM. At the fast "Rocchio" end, bundling combines all the points by class. When sent to the SVM, this method produces the same decision boundary as Rocchio would (the SVM, applied to two points, also finds the perpendicular bisector decision plane). At the slow "SVM" end, bundling does not combine any points at all, and acts identically to the SVM. In between, however, bundling combines points such that the algorithm runs faster than the SVM, and probably more accurately than Rocchio.

There are standard alternatives to bundling, which also reduce the inputs to the SVM. Subsampling, for example, reduces the data set by removing training points. Conceptually, bundling is combining information from the original points together, whereas subsampling is cutting information. Suppose they were both used to reduce the original data to one point per class—intuitively, the one subsampled point would not produce a very good decision boundary when compared to the decision boundary a known algorithm like Rocchio would.

In the next section, we describe more fully the alternative data reduction techniques. Section three gives a description of the bundling technique applied to text problems, and Section four follows with a more formal and general approach to bundling. Section five contains experimental results on four text data sets that shows how bundling can be effective in speeding up the SVM, without sacrificing significant accuracy.

4.2 Existing Data Reduction Techniques

Given the current state-of-the-art theory on SVMs, we argue that current implementations will always take at least $\Theta(n^2)$ time on most text classification problems. As mentioned in the background, the SVM tries to optimize a value function that is based on the "dual form" equations (see the SVM background 2.1.2). Part of that equation shows two summations over all support vectors (all items in the summations that are not equal to zero); merely computing the function the SVM tries to optimize takes time proportional to the square of the number of support vectors.

We argue that the number of support vectors in text problems is proportional to the number of training documents, which in turn shows that current theories on SVMs lead to an $O(n^2)$ bound. Merely computing the function we want to optimize requires time proportional to the square of the number of support vectors. So we argue that the number of support vectors are proportional to the number of training documents (in the text domain).

'Most new documents contain at least one new word (feature); perhaps a proper name (the author's name, for instance), a typo, or a new concept. Recalling that support vectors help define the SVM's decision boundary, every document that contains a unique word (feature) must be a support vector. The SVM will try to correlate every feature with one class or the other, and in order to make that correlation, the SVM needs to create a support vector from each document with a unique word. As long as some fraction of documents contain a new word, the $O(n^2)$ bound holds.

In the context of the SVM's slow training times, we discuss existing techniques

for reducing data, including subsampling, feature selection, bagging and squashing. Note that all of these algorithms reduce the data, meaning any of these (along with bundling) could be used in conjunction with any algorithm, not just the SVM. As mentioned before, throughout the thesis, we have adopted the notation that an individual data point is x_{ij}, where i indexes the training point, and j indexes the feature. One way to visualize the training points is to imagine a large matrix with the training points representing rows and the features representing columns.

Subsampling is probably the simplest method for reducing the data; one just removes training points. This may be thought of as removing rows from the i by j matrix of training points discussed above. Often when people refer to subsampling they mean removing data randomly; this procedure is probably the fastest and easiest to implement of the data reduction techniques.

Given a classification algorithm that is super-linear, another potential solution is bagging [6]. Bagging partitions the original data set and learns a classifier on each partition. A test document is labeled by a majority vote of the classifiers. This technique makes training faster (since each classifier on a partition has fewer examples), but slows down testing since it evaluates multiple classifiers for each test example. Hence the overall training and testing time does not always decrease.

We can also speed up classifiers by reducing the number of features. While subsampling was removing rows from the matrix of training points, feature selection removes columns. Feature selection has been the focus of much work [60] [39]. Note that all classifiers already perform a type of feature selection: if the classifier sees a feature as irrelevant, it simply ignores that feature by zeroing out a weight corresponding to that feature. Thus, an important contribution of feature selection is to have a fast pre-processing step that reduces the overall training time.

Our general method of using statistics to compress data is similar in nature to squashing, a method first proposed in Dumouchel et al. [17]. The squashing method involves three steps, which mirror bundling's steps: partitioning the data, computing statistics over the data, and the generation of data. However, squashing does each of these steps in a different way and produces a different output from bundling.

The original squashing algorithm has a large dependency on the number of features in the dataset. The second stage of the algorithm, computing moment statistics over the data, requires time proportional to the number of features to the fourth power. This time comes from their computations of the means, minima, maxima, along with all the second, third and fourth moments, as well as the marginal fifth moment. This is appropriate for a small constant number of features, but inappropriate for text classification. In text classification, the features are distinct words, and the number of features is typically larger than the number of training examples (and grows with more examples). When the number of features is large, the squashing algorithm is slower than many of the classification techniques squashing is attempting to speed up.

In order to satisfy all of the squashing algorithm's statistical constraints, the output of squashing requires that weights be attached to each point. The traditional classification problem (described fully in the background, 2.1) does not accept weighted points as a portion of the input set, and this means squashing is not immediately

applicable to classification problems.

A later paper addresses squashing's problem of putting weighted points into a classification algorithm by proposing a modification of the SVM to accept weighted input points [41]. Their approach was useful in that it modified the SVM's optimization routine to take advantage of the weights produced by squashing.

Our general approach differs from the squashing approach in being a practical method for large numbers of features, especially when applied to text classification. Specifically, it operates in linear time with respect to the number of features (and training examples) and it produces an output which is usable by standard classification techniques without modification.

4.3 Text Bundling

Bundling, parametrized by the bundle size, generates a continuum of classifiers between the SVM and Rocchio. On the fast end of the continuum, Rocchio uses only the mean statistics of the data and thus achieves fast running time. Rocchio only requires one input point per class, namely the mean. Training and testing for Rocchio are clearly linear time. On the other end of the continuum, the SVM achieves high accuracy but requires a prohibitively long training time on large data sets. The SVM uses each individual document to train its classifier so it trains slower but has higher accuracy.

In order to achieve both high accuracy and fast training time, we explore the space of classifiers between Rocchio and the SVM. The bundled SVM is designed to combine the speed of Rocchio and the accuracy of the SVM by forming a new input data set that is smaller than the original data set but larger than the one mean (centroid) per point used by Rocchio. The bundle-size parameter, s, determines the size of the new input data set, and we will sometimes refer to the bundled SVM with parameter $s = k$ as a bundled SVM ($s = k$). The bundled SVM combines documents in the original data set together to form new training documents, each composed of s original documents of the same class. This preserves the proportion of documents per class between the original data set and the bundled set.

The bundled SVM ($s = 1$) performs no combinations and behaves identically to the SVM in both accuracy and training time. Training time is approximately cn^2, where n is the number of training examples, and accuracy is generally high. When each class is reduced to one point, denoted $s = \max$, the bundled SVM gives the SVM one point per class as input, namely the combination of all documents in a class. In the binary classification case, the SVM receives two vectors, one for each class, and the SVM will find the widest separating hyper-plane between the two vectors, which happens to be the perpendicular bisector, the same as Rocchio. Training time becomes linear, cn, but accuracy is generally lower than in the $s = 1$ case.

In between these extreme values of the bundle-size $s = 1$ and $s = \max$, the training time is somewhere between cn and cn^2, and we expect the accuracy to be somewhere between the accuracies of the original SVM and Rocchio. It is clear that training time can be improved over the usual SVM as we decrease the size of the input data

procedure Randomized Bundling [1] Let n be the number of documents. Let m be the chosen partition size (we assume n/m is an integer). Randomly partition the set of documents x into m equal-sized partitions $P_1, \ldots, P_m$. Compute $x'_{ij} = s_j(P_i)$, where $\vec{x}'_i$ is the i^{th} reduced (mean) data point and x'_{ij} is the count of word j. $x' = \{\vec{x}'_1, \ldots, \vec{x}'_m\}$ is the reduced data set.

set by increasing the bundle size s. Accuracy probably improves over Rocchio as we decrease the bundle size s because combining documents into bundles preserves more information about the distribution of the documents than the mean alone. Each bundle provides some additional information about the underlying structure of the distribution of document vectors within each class.

As a particularly noteworthy example of the tradeoff between accuracy and speed within this continuum, we can choose a bundle size of $s = \sqrt{n/m}$, where m is the number of classes given to us in the classification problem. Our approach proceeds by concatenating documents together on a per-class basis until we are left with $\sqrt{n/m}$ documents. We are thus left with at most $\sqrt{nm}$ documents. Using this set as our input data, the SVM training time is cn, when m, the number of classes, is fixed.

The remaining question is determining how to partition the points. We present two algorithms, randomized bundling and Rocchio bundling. In randomized bundling, we simply partition points randomly (see Figure 4.3 for pseudo-code). This takes very little time: one pass over the n training points—the same as subsampling.

Randomized bundling puts together random points, so it poorly preserves data point locations. Ultimately we would like to preserve as much location information as possible by bundling nearby points. We might try doing a bottom-up clustering where we combine elements closest to one another; but simply computing all pairs of distances is too time consuming. An alternate, faster clustering algorithm is k-means, an algorithm that iteratively attracts points to k centroids. Empirically this did not perform favorably: it raised the pre-processing time without yielding significantly better results than the randomized bundling. Next, we describe an algorithm that preserves more location information than random, but runs faster than the two clustering algorithms.

It is difficult to do any fast clustering while considering all dimensions of the data. If we can project the points onto one important dimension, then we can cluster as fast as we can sort. Rocchio bundling projects points onto a vector and then partitions points that are near one another in the projected space. For binary classification, that vector is the one that connects the two class centroids. For multi-class problems, we choose a vector for each class that connects the class centroid with the centroid of all the other classes' data.

Let $\vec{c}$ be the centroid of one class, and $\vec{c}'$ the other centroid. Let $\vec{x}_i$ be the data point. Our score is the projection of $\vec{x}_i$ on to $\vec{c}' - \vec{c}$:

$$\text{RocchioScore}(\vec{x}_i) = \vec{x}_i \cdot (\vec{c} - \vec{c}'), \tag{4.1}$$

By sorting documents by their score, then bundling consecutive sorted documents (the number of documents is controlled by the bundle-size parameter, s), we combine sim-

procedure Rocchio Bundling [1] Let n be the number of documents. Let m be the chosen partition size (we assume n/m is an integer). Sort the document indices $\{1, \ldots, n\}$ according to RocchioScore($\vec{x}_i$). Let $r_1, \ldots, r_n$ be the sorted indices. Partition the documents according to the sorted indices. Let $P_i = \{x_{r_{(i-1)m+1}}, \ldots, x_{r_{im}}\}$ be the i^{th} partition. Compute $x'_{ij} = s_j(P_i)$, where x'_i is the i^{th} reduced (mean) data point and x'_{ij} is the count of word j. $x' = \{\vec{x}'_1, \ldots, \vec{x}'_m\}$ is the reduced data set.

ilar (by Rocchio score) documents. This algorithm is $O(n \log n$. The pre-processing time for Rocchio bundling can be made as fast as $O(n \log m)$, because a full sort is not necessary: the documents within each partition can remain unsorted. Such a partial sort can be done recursively with known algorithms: briefly, one does a binary recursive split around the original set of scored documents. The standard sort is not much slower and is easier to implement using standard sorting libraries.

This provides a quick way to decide which points fall within a bundle. Further details on the algorithm are provided in the Rocchio bundling pseudo code (Figure 4.3).

4.4 General Bundling Algorithm

In the previous section, we described a method for bundling on text (or other domains where the mean statistic is important). In this section, we generalize and formalize bundling to other domains.

One way to understand a large body of data is to summarize them with statistics, which are functions over the data. Let $\vec{x} = \{x_1, x_2, \ldots, x_n\} \in \mathbb{R}$ be a set of data. Then a statistic, $s(\vec{x})$, is a function that maps the data to a single value in $\mathbb{R}$. For example the mean statistic is simply the average value of the data points.

Subsampling does not preserve the entire set of data, rather it preserves all statistics on a subset of the data. In contrast, we propose to preserve certain statistics of *all* the data. For example, in text we preserve the mean statistic.

4.4.1 The General Bundling Algorithm

We can think of the tradeoffs between speed and accuracy in an information sense: the less raw information we retain, generally the faster the classifier will run and the less accurate the results. Each data reduction technique operates by retaining some information and removing other information. By carefully selecting our statistics for a domain, we can optimize the information we retain.

Bundling preserves a set of k user-chosen statistics, $\vec{s} = (s_1, \ldots, s_k)$, where s_i is a function that maps a set of data to a single value. Bundling imposes a global constraint as follows.

global constraint Let $\vec{x}$ be a set of data. Let $\vec{x}'$ be a reduced set of training data, the "bundled" set. Bundling requires that $s_i(\vec{x}) = s_i(\vec{x}')\ \forall i$.

There are many possible reduced data sets, $\vec{x}'$, that can satisfy this constraint. But, we don't *only* want to preserve the global statistics. We also want to preserve

additional information about the distribution. To get a reduced data set that satisfies the global constraint, we could generate several random points and then choose the remaining points to preserve the statistics. This does not retain any information about our data except for the chosen statistics. We can retain some of the information besides the statistics by grouping together sets of points and preserving the statistics locally:

local constraint Assume that the data points, $\vec{x}$, and the reduced data, $\vec{x}'$ are partitioned into the same number of sets. Let $\vec{y}_{(j)}$ be the data points from the j^{th} partition of $\vec{x}$. Let $\vec{y}'_{(j)}$ be the data points from the j^{th} partition of $\vec{x}'$. Bundling requires that $s_i(\vec{y}_{(j)}) = s_i(\vec{y}'_{(j)})\ \forall i, j$.

The bundling algorithm's local constraint is to maintain the same statistics between subsets of the training data.

The focus on statistics also usually implies that the bundled data will not have any examples in common with the original data. This is a necessary consequence of our wish to fully preserve certain statistics rather than the precise examples in the original training set. The bundling algorithm ensures that certain global statistics are maintained, while also maintaining a relationship between certain partitions of the data in the original and bundled training sets.

Here we describe how bundling might be used in domains where more statistics need to be preserved or where statistics other than the sample mean are important. This section is not relevant to text, but may be of interest to a wider audience interested in applying it to different domains.

For single, simple statistics like maximum, minimum or means of each feature can be solved in a straightforward fashion like the text example. If each local bundle has the maximum of each feature, then the global maximum for each feature will also be conserved.

One can also bundle with two or more statistics simultaneously, though only in special cases. Instead of bundling a partition to one point, we bundle to two or more. One can preserve the minimum and maximum statistics by creating two points, one of which contains the minimum value for each feature, the other containing the maximum. One can preserve mean and variance statistics by converting each partition into two points that have the same mean and variance statistics as the partition. Both of these examples simultaneously satisfy the local and global constraints.

4.5 Results

4.5.1 Data Sets and Experimental Setup

Our experiments try to quantify the relationship between speed and accuracy for five different data reduction techniques at varying levels of speed-ups. In order to perform these comparisons, we made a test bed as broad and fair as possible. We compared the various reduction techniques on SvmFu, a fast, publicly available SVM implementation [46]. We coded each pre-processing step in C++, and compared the

	20 News	IS	Reuters	Ohsumed
Train Size	12,000	4797	7,700	179,215
Test Size	7,982	4,822	3,019	49,145
Features	62,060	55,194	18,621	266,901
SVM time	6464	2268	408	?
& accuracy	86.5%	92.8%	88.7%	?

Table 4.1: This table summarizes the four text datasets used in our experiments. SVM timing and accuracy are based on SVMFu; question marks (?) indicate SVM runs that did not finish within a week. The Reuters accuracy is based on precision-recall break even; 20 news and Industry Sector are based on multi-class accuracy; and Ohsumed on binary accuracy. Features refer to the number of features found in the training set.

total reported preprocessing, training and testing time reported by the UNIX time command, for user process time. We use a fast, relatively un-used machine (1GB RAM, 1.6GHz PIII processor) to perform all the experiments.

We use four well-known text data sets: 20 Newsgroups [37, 55, 3], Industry Sector [18], Reuters-21578 [59, 25, 50], and Ohsumed[22]. The data sets are summarized in Table 4.1. According to past convention, we reported accuracy for 20 Newsgroups, Industry Sector and Ohsumed along with precision-recall for Reuters. Please see the Appendix at the end of the thesis for further details on the data sets and how we pre-processed and recorded results and statistical significance results for them.

We chose to base our tests on the Support Vector Machine (SVM), a highly accurate, but slower (super-linear) algorithm for classification. In many text-classification tasks, it has consistently outperformed other algorithms [59, 25]. The SVM takes the positive and negative training points, and tries to place a hyper-plane between them that optimizes a tradeoff between classification error and margin width. It is more sensitive to the exact location of the training points than algorithms that simply use the feature's means. For more information about the SVM, see the Burges [8] tutorial and our background section on SVMs (Chapter 2.1.2).

For our experiments, we use the preprocessing transforms described in the "transforms" section of the Naive Bayes chapter (Chapter 3.3). The SvmFu package is used for running experiments [46]. Exact command line parameters for the SvmFu are listed in the Appendix at the end of the thesis. We produce multi-class labels by training a one-versus-all SVM for each class and assigning the label of the most confident SVM. We use the linear kernel for the SVM since after applying our transforms from Chapter3 (which we did), the linear kernel performs as well as non-linear kernels in text classification [59].

Besides our bundling technique, we implemented the subsampling, bagging and feature selection techniques described in the section on other data reduction techniques. We did not implement squashing which seemed impractical in light of its time bounds (f^4 where f is the number of features; slower than the SVM's bounds), and impracticality (the output is weighted points, which the standard SVMs do not

accept as input).

For subsampling we selected points randomly by class; this was the fastest to code and also had the fastest running times. We parameterized subsampling similarly to bundling's parameters. We indicated a partition size ($s = 1 \ldots s = \max$), and one element from each partition was picked to go to the reduced data set. Like bundling, this preserves the proportion of original documents in the subsampled document class. When $s = 1$, the SVM runs as normal; when $s = \max$, one random point from each class is sent to the SVM.

The general bagging procedure again begins by partitioning the data. Each set of s points from a class would be sent to an SVM classifier (implying n/s partitions) if there were n points in the original class. For example, if there were 150 points in each class, and $s = 3$, then bagging would generate three separate classifiers that each had 50 points per class. The three classifiers would each produce a classification on a given test point, and the majority labeling of the three classifiers would be the final label. In the example above, bagging would take at least three times longer than subsampling, which would only train on one of the three classifiers that bagging used. Consistent with our other notation, $s = \max$ is the value of s at which bagging went the fastest, and $s = 1$ (i.e. one partition containing all the original points) is equivalent to the SVM.

We use one of the best performing feature selection algorithms proposed in Mladenic [39], which ranks features according to $|p(f_i|+) - p(f_i|-)|$, where $p(f_i|c)$ is the empirical frequency of f_i in class c training documents. Each feature was scored according to these probability heuristics. The features were then sorted by its score, and the ones with the lowest scores were removed. The parameterization again ranges from $s = 1$ to $s = \max$. To keep consistent with our other notation, s means that $s - 1$ features were removed; this allows $s = 1$ to mean the regular SVM is run (i.e. no features are removed) and $s = \max$ means only one feature is retained.

The remaining two algorithms, Randomized bundling and Rocchio bundling are the focus of this paper. The s parameter applies to all of the algorithms mentioned; so $s = 1$ is always the original SVM, corresponding to sending the SVM a maximum of information, and $s = \max$ is always the "fastest" algorithm, corresponding to sending the SVM the minimum information.

4.5.2 Experimental Results

In this section, we analyze the results from our empirical work found in Table 4.2 and in Figure 4-1 showing the exact tradeoffs between speed and accuracy on the various datasets. We discuss how each of the five speed-up techniques worked.

Our results on Ohsumed explain how different data reduction techniques work on truly large datasets. Neither bagging nor feature selection were useful on the dataset. No bagging runs completed within the allotted time (8 hours per run) and feature selection required reducing the feature set to 50 features, which yielded very poor results.

In general, feature selection was a disappointment on both speed and accuracy grounds. The actual feature selection algorithm is empirically a small factor slower

	20 News		Industry Sector	
	Slowest Results			
	Time	Accuracy	Time	Accuracy
Bagging	4051 (5.68)	.843 (.004)	1849 (1.55)	.863 (.006)
Feature Selection	5870 (8.23)	.853 (.002)	2186 (1.03)	.896 (.005)
Subsample	2025 (2.84)	.842 (.004)	926 (.774)	.858 (.005)
Random Bundling	2613 (1.31)	**.862** (.005)	1205 (.552)	.909 (.004)
Rocchio Bundling	2657 (1.01)	**.864** (.002)	1244 (.573)	**.914** (.004)
	Fastest Results			
	Time	Accuracy	Time	Accuracy
Bagging	2795 (4.81)	**.812** (.005)	1590 (.249)	.173 (.006)
Feature Selection	4601 (4.71)	.649 (.004)	1738 (1.81)	.407 (.003)
Subsample	22 (.241)	.261 (.02)	59 (.142)	.170 (.005)
Random Bundling	117 (.08)	.730 (.007)	177 (.064)	**.9** (.005)
Rocchio Bundling	173 (.078)	.730 (.007)	248 (.064)	**.9** (.005)

	Reuters Mod-Apte		Ohsumed	
	Slowest Results			
	Time	Accuracy	Time	Accuracy
Bagging	346	.886	?	?
Feature Selection	507	.884	11010*	.647
Subsample	173	.859	39340	.808
Random Bundling	390	.863	25100	.830
Rocchio Bundling	404	.882	26710	.804
	Fastest Results			
	Time	Accuracy	Time	Accuracy
Bagging	295	.878	?	?
Feature Selection	167	.423	11010*	.647
Subsample	9.6	.213	13	.603
Random Bundling	105	.603	74	**.731**
Rocchio Bundling	129	.603	134	**.731**

Table 4.2: This table summarizes results for various text corpora (columns) against various data reduction algorithms (rows). Results are expressed as empirical time-accuracy pairs, with standard deviations in parenthesis. Boldface items have statistically higher accuracy than the other methods at the 95% confidence range, using statistical significance tests specified in the Appendix. The rows headlined by "Slowest Results" show the time and accuracy for the algorithms resulting in the slowest classification times (not including the base-case where no pre-processing algorithms are used); the other rows ("Fastest Results") represent the fastest classification times (i.e. subsampling down to one point per class). The maximally bundled data is functionally similar to the Rocchio algorithm. Question marks (bagging) indicate runs that did not complete in the allotted time. Only one run of feature selection completed within the allotted time (*).

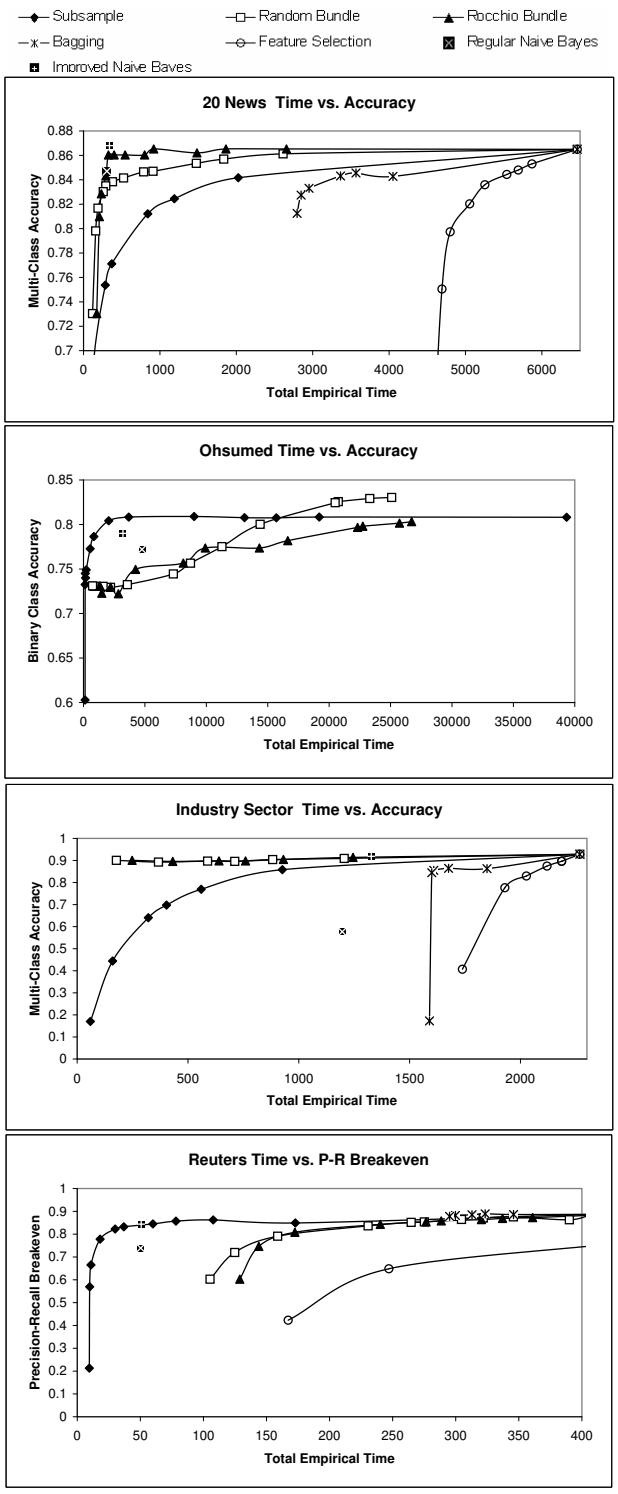


Figure 4-1: Speed against accuracy plotted for the four data sets, and the five data reduction techniques included in this study. The key is at the top of the figures

than the other algorithms: one must perform calculations for every feature, and the number of features tends to be larger than the number of data-points. This effect is relatively minor. More importantly, the speedups in training times for our SVM were relatively minor. This might be due to the previously mentioned effect where SVMs will be have quadratically in the number of training points, which under feature selection, is not changed. In our tests, reducing the number of training points sped up the algorithm more than reducing the number of features.

Our results on Ohsumed help explain why feature selection does so poorly in our empirical results. At the fast, inaccurate end, we chose the top 50 features. However, those 50 features were distributed among 170,000 training points, so many documents ended up consisting of a single non-zero feature. If a document's single feature was a somewhat common word, it tended to appear at least once in both the class and its opposite class. So common words (features) resulted in situations where any algorithm would have a difficult time understanding which class a feature was correlated with. If the document's single feature was somewhat rare, and only appeared in one class, it generally could only help classify a small percentage of the test documents. So when a large amount of feature selection was used, classification accuracy became near random. When we chose more features for Ohsumed (even 100 total features) we found that the SVM takes much longer than our eight hour time limit.

Bagging generally achieved higher accuracy than feature selection, but could not operate very quickly. As discussed before, splitting the training set into more bags reduced the training time, but increased the test time. The total training time versus the number of bags was shaped like a "U": early speed increases were due to less training time, later speed decreases were due to more testing time. This means that bagging could only speed up an algorithm to a certain point; this point was often slower than the speed-ups experienced with some of the other algorithms. On the Ohsumed database, there were no bagging sizes that came close to completion within our eight hour time limit.

Bagging yielded good accuracies for Reuters, and acceptable accuracies for 20 news and industry sector. Bagging's accuracy was tied to subsampling's accuracy: the sharp drop off visible in the Industry Sector graph was due to the fact that subsampling too much yielded extremely low (nearly random) accuracies; and combining a series of almost random classifiers through voting does not really help create a better classifier.

Subsampling worked overall better than expected by the authors. Relative to the other algorithms, the process of randomly selecting documents is fast. For a given reduced training set size, subsampling's pre-processing time ran faster than any of the other data reduction algorithms. More importantly, the data points that it produces were more sparse (more zero entries) than bundling, and hence an algorithm like the SVM will run faster with subsampled points than with denser bundled points. For example, on Ohsumed we could run the SVM with 18,000 subsampled points, but with only 12,000 bundled points.

Subsampling also led to surprisingly accurate classification on the two binary problems, Reuters and Ohsumed. On Reuters, it appeared that a small number

of documents could adequately define a class; so for a given amount of time, subsampling would often perform better than the other algorithms. On Ohsumed, the accuracy seemed to level off in certain ranges, performing worse than bundling for higher amounts of time, but better than bundling for intermediate amounts of time. Subsampling did not work well on the multi-class problems. We believe this is because the multi-class problems were more difficult and required more training points to generate a good classifier. The drop off in accuracy begins from the start and keeps falling (as opposed to bundling where it might flatten off for some ranges of s).

In all of our datasets, subsampling has a steep drop off in accuracy; eventually at 1 point per class, it will intuitively do poorly. The difficulty with subsampling is knowing when that drop off will occur. One might get lucky, like with Reuters, where we found the heavy drop off doesn't occur until you remove 19 of every 20 documents. Or one might get unlucky, like with 20 News, where removing every other document causes an immediate drop in accuracy.

Certainly the most consistently good algorithms, and arguably the best performing algorithms were the two bundling algorithms. They had the best performances for 20 news and industry sector, and alternated the lead on Ohsumed with subsampling. For most datasets, they had the highest scores at the slow/high accuracy end (bundling pairs of points generally performs as well as the unbundled SVM; for Ohsumed, combining every 14 points into 1); and also did not drop as sharply as subsampling on the fast/low accuracy end. As mentioned before, full bundling combined with the SVM acts like the Rocchio classifier.

The Rocchio and random bundling methods have different strengths and weaknesses. As Table 4.2 shows, with minimal amounts of bundling (two points per bundle), Rocchio usually outperforms random and most other algorithms. At the other end of the spectrum, Rocchio has very few choices. For example, when bundling to one point Rocchio has no choices—it bundles identically to random. However, Rocchio takes more time to complete. Thus, Rocchio works well when bundling to more points, but suffers from higher preprocessing times when fewer points are retained.

We also compared all of these algorithms to both multinomial Naive Bayes and the modified Naive Bayes presented in the previous chapter 3. These points are shown on the graph as, respectively, an "X" and a "+". While multinomial Naive Bayes did not yield good results, our modified Naive Bayes did; in 20 News, Reuters and Industry Sector, the modified Naive Bayes was along the same line as the best performing algorithm for those data sets. For the text data sets, at least, the modified Naive Bayes performs well. The only drawback is that modified Naive Bayes is "tuned" for text and may not work well on other domains, and that its time is not adjustable. On the other hand, since Naive Bayes is a linear time algorithm, its times should generally be fast (an optimization focused on speed and written in a faster language than our java-based algorithm would show even better performance characteristics).

Subsampling and bundling seem to give the best results of the data reduction techniques, so in Figures 4-2 and 4.3 we studied them further.

Figure 4-2 shows that, for a given number of points, bundling usually performs better than subsampling (true in all cases with the exception of Ohsumed). This verifies our earlier argument that given an equal number of points, then combining

	Subsampling		Rocchio Bundling	
Training Size	time	accuracy	time	accuracy
6000	2025	0.84	2657	0.87
4000	1189	0.82	1864	0.87
3000	841	0.81	1485	0.86
1500	369	0.77	919	0.87
1200	285	0.75	799	0.86
600	140	0.70	545	0.86
300	71	0.62	400	0.86
150	39	0.51	326	0.86
120	35	0.49	292	0.84
60	26	0.38	233	0.83
40	24	0.33	207	0.81
20	22	0.26	173	0.73

Table 4.3: Shown is a comparison of subsampling and Rocchio bundling for reducing to various numbers of training points. For a given number of points, Rocchio bundling takes longer (due to denser vectors to dot-product) but has statistically significantly better accuracy. Results are shown as accuracy on the 20 Newsgroups data set.

points, like bundling, is to be preferred to cutting points, like subsampling. On Industry Sector and Reuters, the two bundling techniques perform approximately the same; while on Ohsumed, random bundling performs better and on 20 Newsgroups, Rocchio bundling performs better.

Figure 4.3 shows the empirical times and accuracies for a given number of points on the 20 Newsgroups data set. Note that, for an equal number of training points, Rocchio bundling always takes longer than subsampling. This is because Rocchio has a longer pre-processing time, and because each vector of the bundled points is more dense than the vectors of the subsampled points. When there are many points, the time differs by around 30%; in this case, the $O(n^2)$ nature of the SVM probably dominates. For smaller numbers of points, the effects of having a longer pre-processing time dominates and with one point per class, Rocchio bundling takes about 8 times longer than Subsampling.

In conclusion, we have shown a variety of methods for dealing with the problem of large data sets. From our empirical results, we have a few recommendations. When training time does not matter, or for sufficiently small data sets, the SVM untouched works very well; it produces the highest accuracies, though it may take a long time to complete. When an extremely low training time matters, then subsampling is the fastest of the techniques tested; though the accuracy can drop to near-random. In between, when scalability is an issue in the context of accuracy, both the modified Naive Bayes and the bundled SVMs have advantages according to the situation. On one hand, the bundled SVM has the advantage of being a general purpose classifier, so it would be used on large problems that are not textual in nature. The bundled SVM is also flexible in the sense that one can choose a point along the continuum

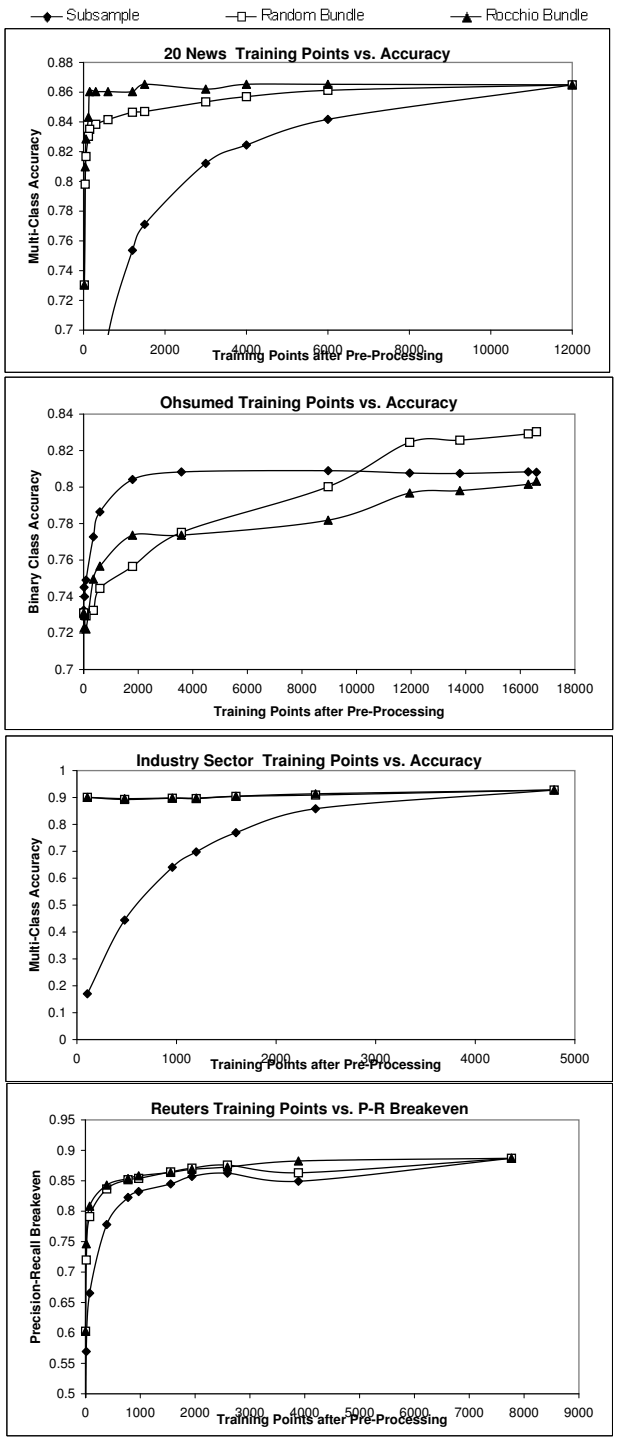


Figure 4-2: Number of points is plotted against accuracy plotted for the four data sets, and the three data reduction techniques that reduce the number of points: subsampling, random bundling and rocchio bundling. The key is at the top of the figures.

that is time-appropriate to the application. On the other hand, the modified Naive Bayes performs well (particularly on text), often achieving a good trade-off between time and accuracy.

Chapter 5

Using Pre-Existing Trees as Features for Machine Learning

5.1 Introduction

In this previous two chapters, we discussed modifying existing algorithms in order to make fast, scalable classifiers primarily for learning text on the Web. In this and the next chapter, we explore features and algorithms that exploit the Web's rich, human-oriented structure towards building learning Web applications. This chapter is the expanded version of work done with David Karger [53].

Suppose someone was given a Web page, like the one featured in Figure 5-1, and asked to recommend a link to a Chinese friend. Suppose they could not read any Chinese, had never visited the particular site [1], and generally could not decipher any of the contents of the page. They might reasonably recommend the story in the "top headline" area of the page. In fact, they might suggest that same "top headline" area as a candidate recommendation for next week, page unseen, even though no one even knows the exact content that will reside there. Without knowing any specifics of the Web page, most people can make good guesses about certain semantic attributes of that page.

This is not accidental; Web editors spend considerable effort trying to drop easily-understood human clues for the readership's benefit. We attempt to exploit human-oriented clues, like the spatial clues that allow one to understand where the "top headline" resides, for the benefit of recommendation systems and ad blocking.

As we attempt to transition from the domain of flat-text files (as discussed in the previous chapters) towards the rich, semantic world of the Web, there are many new, human-oriented features we can exploit. These features have the ability to make hard problems in the domain of text much easier. For example, as the world of information retrieval came to the Web, work like PageRank [7] noticed that the link structure of the Web contains lots of information about quality. The work proceeded from the observation that links into a page indicated some type of interest; and that observation rooted a series of algorithms that improved search engines dramatically.

[1] `http://dailynews.sina.com`

Figure 5-1: Without understanding Chinese or having seen this exact layout before, most humans could guess what portion represented the "top headlines."

Similarly, our work proceeds from the observation that Web editors want to provide consistent clues towards the contents and context of a page. It may be difficult to build correlations between the Chinese characters displayed on the Chinese news site and a particular user's interest; but luckily, the Web site's editor is already filtering and implicitly recommending interesting stories by their placement of stories on the page.

There is a certain logic that goes into most Web sites, and we want to codify that into algorithmic terms a computer can understand. Referring to Figure 5-2, a reasonable guess is that the sets of items in each box are somehow related. We might not know what the elements inside box A are (advertisements? indices? minor news stories?); but if your Chinese friend told you the first element in box A was an advertisement, one might (correctly) assume everything else in box A was an advertisement too. Similarly, if your friend indicated that B-1 was interesting, one might guess that B-2 was interesting as well, even if you weren't fully sure what made B-2 interesting.

In Figure 5-3, we show how a computer algorithm might understand the earlier stated intuition that items visually clustered might have similar properties—for example, that box A might contain a series of advertisements. On the upper right-hand side we display snippets of the HTML that describes the site; various visually distinct areas from the Web page are encoded into continuous blocks of HTML. The HTML forms a tree, shown by indentations. That tree is abstracted on the bottom left. Suppose someone labels that node A-1 is an advertisement (white), while node B-1 is content (black). The tree on the bottom right shows a potential generalization of that information in which everything in box A is considered an a advertisement. The work described here tries to formalize and build algorithms that might automatically make generalizations like these.

Our discussion of tree learning algorithms spans two chapters. In this chapter, we describe the algorithms that facilitate learning relationship like the ones described in our advertisement example above. In the next chapter, we describe empirical results for experiments done on the advertisement-blocking and recommendation domains.

5.1.1 Related Work

In discussing related work, it is worth devoting some time to work that appears related but is not. Much work has been done on the problem of *placing* an item in a tree. Both McCallum et al. [38] and Koller and Sahami [31] tackle the problem of classifying text documents into a hierarchy using words in the documents as features. In their problems an item's place is the *output* based on other features, while we use item position as a predetermined *input* feature for classification. McCallum uses a method called shrinkage that generates a classifier at each node in the tree, along with a weight for that classifier. New documents are placed within the tree by finding the node which maximizes the weighted sum of the classifiers between the root and each leaf. Koller's work creates a classifier at each node in the tree which indicates which child to move towards. The document moves downward from the root, greedily selecting the most likely child, until it finally ends at a leaf node. In effect, McCallum

Figure 5-2: Web sites have a certain logic to them that is easily intuited by most Web surfers; everything in box A probably has something in common—for example, they might all be advertisements. Given that B-1 is interesting, it intuitively feels more likely that B-2 would be interesting than something in A. We can use the spatial layout to make educated guesses about content without being able to decipher anything specific about the contents themselves.

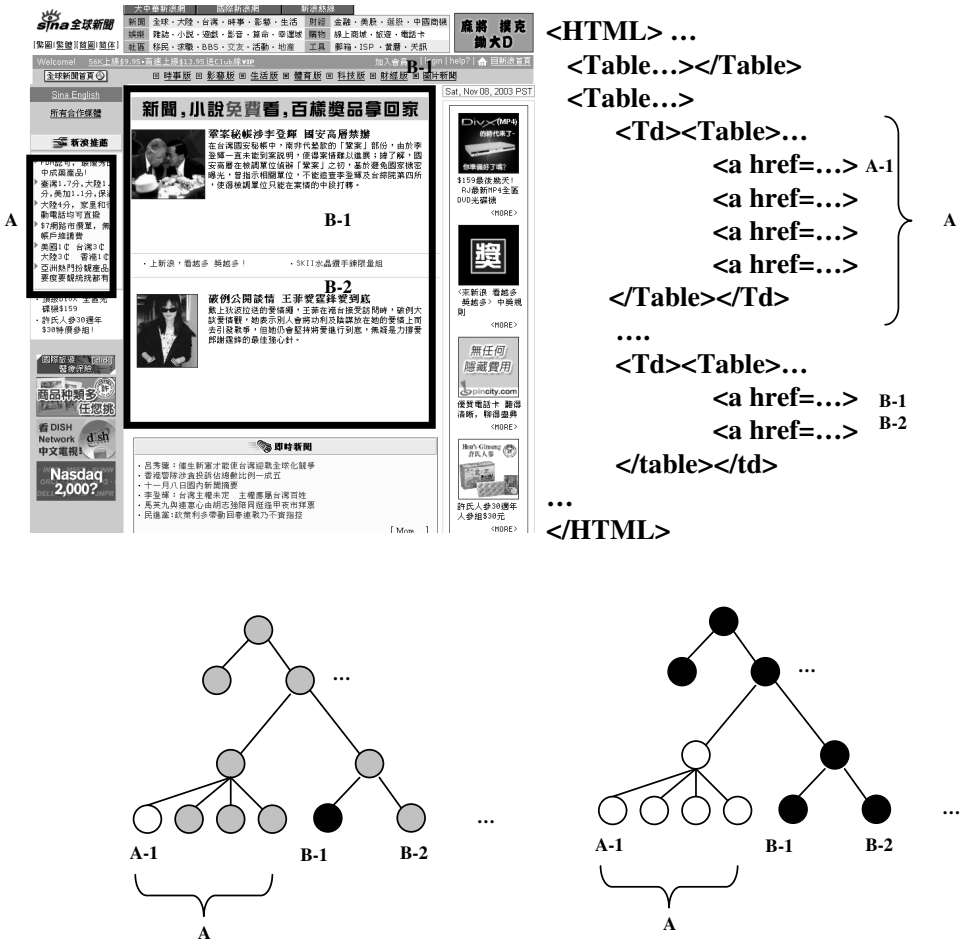


Figure 5-3: Shown is an abstraction of Web problem to the domain of "tree learning." The top-left shows the original Web-site. The top-right shows that visual portions of the page are collected in chunks of HTML, which are indented to show the HTML's tree structure. The bottom left shows the abstracted tree, receiving a partial labeling of the page: white ("advertisement"), black ("content") and grey ("unknown") nodes. The bottom right shows one potential generalization of the tree which suggests everything in box A is an advertisement.

is maximizing a score with respect to a weighted sum of classifiers along a path through the tree; and Koller is making a series of local decisions to choose between children along a path through the tree.

Both McCallum's and Koller's work assume a greater amount of information resides at each node. They assume that each node has a related textual document or documents that can be used to build local classifiers that might inform the global classification task. Our approach abstracts away the exact contents of the node, and only relies on the node's position in a tree for classification.

Natural language work also involves trees, often in describing some grammar. Collins and Duffy [10] develops a special kernel for use in comparing multiple trees; however, this does not appear relevant to our problem of using a single specific tree.

There is also truly related work. Haussler [20] is perhaps the first to propose using position in a preexisting tree as a feature for classification. He proposes a more restricted model (requiring that the concept to be learned form a conjunction of subtrees) and performs a VC-dimension analysis. Among other differences with our work, Haussler's model does not capture classes that are *complements* of subtrees. In the terminology we will introduce shortly, he only allows for forward mutations, but do not allow for backward mutations.

There is also work that extends the work presented here. Derryberry [43] extends the basic algorithms in a variety of ways and provides an analysis of situations in which these algorithms do not appear effective.

5.1.2 Model-Based versus Discriminant Classifiers

One might wonder why investigate a model-based classifier when a good discriminant classifier, like the SVM, might be able to work equally well or better. Ng and Jordan [40] noted that "the prevailing consensus seems to be that discriminative classifiers are almost always to be preferred to generative ones."

Ng also provides some of the reasons we have taken the model-based approach. Model-based classifiers can theoretically and empirically outperform discriminant classifiers when there is not much training data available [40]. Discriminant classifiers are general, and can choose from many potential solutions when there is only a small amount of training data. On the other hand, model-based classifiers can use domain knowledge as a substitute for training data, when the amount of real training data is small. For example, returning to our advertising example, there is no way to suggest to the SVM that most links on a page are not advertisements. Such suggestions are possible in probabilistic models, where priors can pre-dispose the classifier one way or another. In our own experiments detailed in the following chapter, we empirically found that our model-based classifier could out-perform the SVM.

As discussed in the previous chapters on scalability, SVMs are not always fast. The time it takes to search through all possible solutions may take longer than the more constrained search over a model's possible solutions. Empirically on the recommendation problems, a standard SVM implementation took more than three hours to learn from a relatively small amount of training data, while our algorithm took seconds. This is impractical for the type of Web applications we are trying to build.

There are also philosophical reasons to prefer a model-based classifier; it clarifies and sharpens one's assumptions about a domain. Those assumptions can be empirically tested or discussed, all to the benefit of understanding the domain better.

5.2 Models for Classification in a Tree

Returning to Figure 5-3, the bottom pictures of trees presented a partially labeled tree (left) and one that had been generalized such that only box A (white) was labeled as advertisements (right). Our basic assumption, one that we want to codify, is that the coloration of the tree will prefer consistent labels (coloration) in similar areas of the tree.

One way to codify that intuition is to make it more likely for a parent to match than to mis-match its parent's label. Then the most likely coloration of the tree will be the one that has the fewest mis-matches (given certain pre-specified labels). We use the phrase "mutations" to mean a change in labels between a parent and a child node; and later we will give a formal description of these mutations that gives rise to a Bayesian network.

We mentioned earlier that it might be useful to lend prior knowledge to the model in order, for instance, to suggest that advertisements were less likely than content. Thus, our model has two types of mutations: one "forward" mutation rate that expresses the probability that a parent will be black (content) and its child white (ad); and a "backward" mutation rate that a parent will be white and its child black. Changing the relative rates of those mutations allows one to influence the model towards or against a particular label.

In the next sections, we formally discuss our model, which takes the form of a Bayes-net. One can run standard inference procedures on the Bayes-net to generate probabilistic answers (give that node A is an ad, what is the probability that node B is also an ad?) We also suggest an *incremental* algorithm that allows adding evidence nodes or doing certain inference problems in time proportional to the tree's height. This is followed by a method that solves the real-world problem of precision in probabilistic computations.

5.2.1 A Model of Classes Correlated to a Hierarchy

Our model consists of a Bayes-net that has a structure identical to the tree that we are studying. Our assumption that the class to be learned is "correlated" to the tree is captured in a model based on mutations. Although the items to be classified are leaves in the tree, we extend the notion of class label to the internal nodes of the tree. The class of an internal node is chosen to represent the class that leaves below that node *tend* to have. Tendency is the appropriate word here: we wish to capture cases where a substantial majority, rather than all, descendants have the class. For example, in Figure 5-3, most of the nodes might not be advertisements, so they are colored black; but a mutation occurs that leads one branch of the tree to be entirely advertisements (white).

To formalize this model in a generative fashion, we declare that some class-tendency is chosen at the tree root, and that this tendency is then propagated down the tree, with mutations happening probabilistically on the way. Formally, let N_i denote the class of node i (class tendency if i is an internal node). Then if N_i is N_j's parent, we declare that:

$$p(N_j = 1|N_i = 0) = \theta; 0 < \theta < \frac{1}{2} \text{ and} \tag{5.1}$$

$$p(N_j = 0|N_i = 1) = \phi; 0 < \phi < \frac{1}{2} \ . \tag{5.2}$$

Thus the probability of a "forward mutation"—shifting from a tendency for class 0 to a tendency for class 1— is θ, while the probability of a "backward mutation" is ϕ. Both parameters have a probability less than $\frac{1}{2}$, which means that mutations are a less common event than non-mutations. Propagating this rule down to the leaves of the tree defines classes for the tree leaves.

The ratio of forward to backward mutations induces a behavior on large-depth leaves. The probability that a deep leaf has a given characteristic converges towards a expression in terms of the mutation rates, $\theta/(\phi + \theta)$. This also suggests a natural prior distribution for the root's class-tendency. If we set $p(N_1 = 1) = \theta/(\phi + \theta)$ then any node in the tree will, prior to any evidence, have a tendency towards class 1 with probability $\theta/(\phi + \theta)$.

In standard Bayesian terms, *evidence nodes* are those nodes whose class is given to us. Our goal is to predict the class of node i given the evidence nodes: $p(N_i = 1|\text{evidence})$. We simply see if the probability of a class is greater or less than a half, given the evidence. If $p(N_i = 1|\text{evidence}) \geq \frac{1}{2}$ then we predict it is in class 1; otherwise we predict it does not. Of course, other thresholds may be preferable, for example, to minimize the number of false positives. Or, in a ranking problem, we may sort based on this probability measure.

At this point, we have converted the original, intuitive problem into a formally specified Bayes-net. To summarize the steps in creating a Bayes net out of the given hierarchy:

- set conditional probabilities for all edges in tree for node i with child j,
  $p(N_j = 1 \mid N_i = 0) = \theta$
  $p(N_j = 0 \mid N_i = 1) = \phi$
- set the root prior: $p(N_0 = 1) = \theta/(\phi + \theta)$
- assert the classes of evidence nodes.

Note that more complex models, such as mutation probabilities being a function of node location, are also possible.

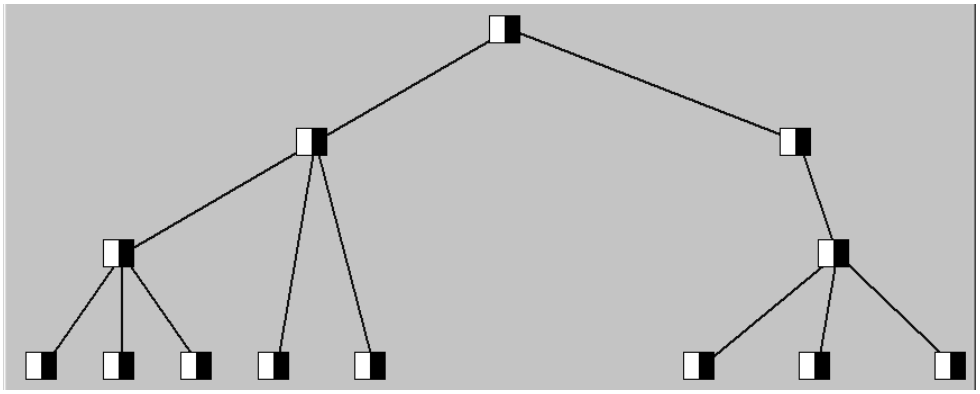


Figure 5-4: Shown is a sample tree with no evidence nodes, and equal forward and backward mutation rates. At each square, we show the probability of the node given the evidence, given that the node is white and black, by proportionally shading the nodes white and black. Since there is no evidence, the probability of the evidence at each node is split evenly at 50% white and 50% black.

5.2.2 Intuition Behind Model

In this section, we show some of the consequences of the model, and argue that the model fits with our intuition of how trees are correlated with classes.

Figures 5-4 through 5-8 display a set of increasingly complex situations in which new evidence nodes are added to a tree. Figure 5-4 shows a tree without any evidence nodes, which has no bias towards one label or the other; this is graphically represented by boxes that are half shaded black and half shaded white. Adding a white evidence node (Figure 5-5) creates a bias towards white. Following our intuition, the bias is greater nearer the evidence node and decreases with the distance from the node. Adding a black evidence node (Figure 5-6) induces the sub-trees to be biased towards white, other sub-trees biased black, and other sub-trees which seem unbiased. Adding a further black evidence node (Figure 5-7) turns most of the tree biased towards black, with the exception of the left sub-tree which remains white. Figure 5-8 shows the same example with the mutation rate turned up; this corresponds to a model in which the labels are not thought to correlate with the tree, and as a result most of the non-evidence nodes have a high degree of uncertainty.

In the next section, we discuss a fast algorithm for computing the conditional probabilities $p(N_i = j|evidence)$.

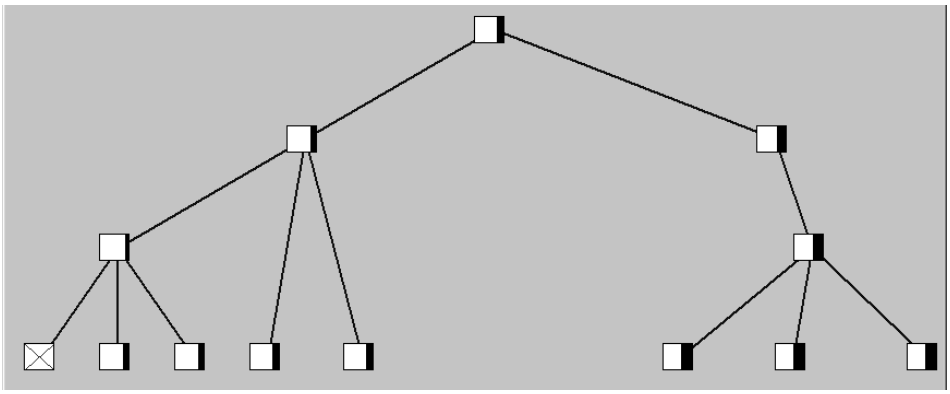


Figure 5-5: Shown is the same tree with one evidence node, the far left leaf, which is marked white. The nodes nearest to that evidence node have the highest probability of being white as well; as the nodes increase their distance from the evidence, the probability of being white decreases, but is always greater than the probability of being black. Again the shading is done in terms of the probability of the node given the evidence.

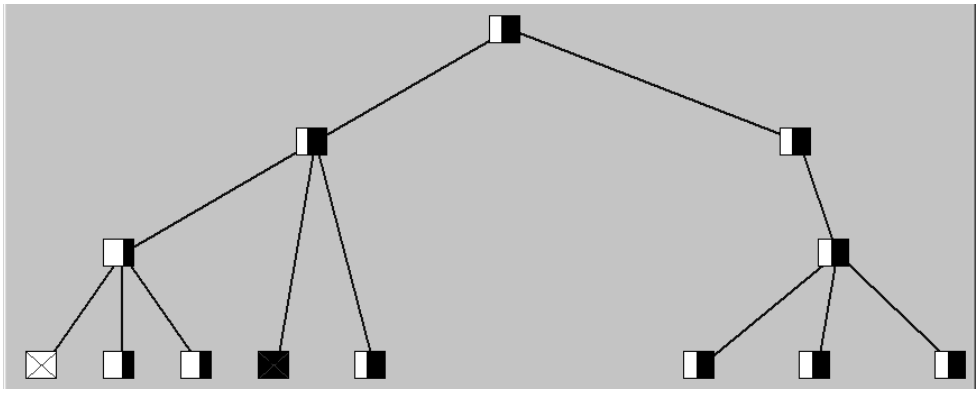


Figure 5-6: A black evidence node is added to the fourth leaf from the left. The nodes in the immediate vicinity of that black node favor black; and the nodes in the immediate vicinity of the white node favor white. The nodes far from both (right branches) are unsure about what color to favor.

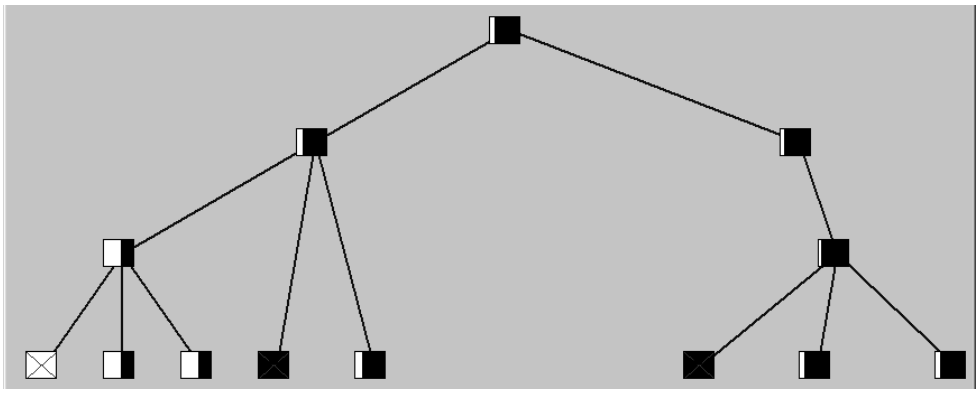


Figure 5-7: Another evidence node, black, is added to the third node from the right. This induces most of the tree to strongly favor a black label, with the exception of the four nodes on the far left.

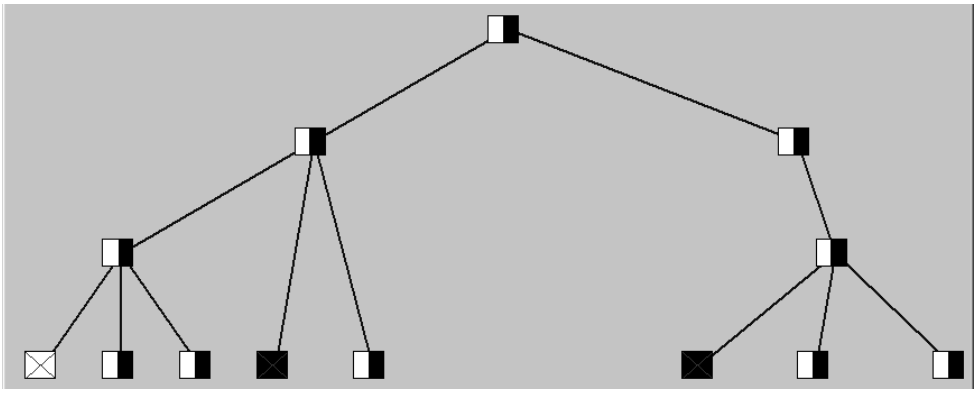


Figure 5-8: The same tree is displayed with a much higher mutation rate. The higher mutation rate means that the certainty of each node becomes much lower, almost random. This reflects the case when the model does not believe the tree is correlated with the labels, and thus it suggests a high degree of uncertainty for the non-evidence nodes.

5.3 Fast Classification in Trees

As discussed above, given a set E of evidence nodes, our goal for a particular test node i is to compute the posterior distribution $p(N_i = k \mid E) = p(N_i = k \wedge E)/p(E)$. We can use this calculation to classify new nodes using $\text{argmax}_k p(N_i = k \mid E)$, the probability of having a characteristic given the evidence. Alternatively we can use the posterior probabilities to rank nodes and select some top set of them as "best candidates."

There are many exact algorithms for solving this problem on trees, all doing fundamentally the same thing but with qualitatively different flavors. Russell and Norvig [48] describe backward chaining (a linear-time algorithm for computing the posterior distribution for one node) and forward chaining (a linear time algorithm for computing the posterior distribution at all nodes simultaneously). Our goal is to memoize some intermediate probabilities so that updates—new pieces of evidence—can be done quickly.

The specifics of our intended application make these algorithms inappropriate in their original forms. Our application is an *online* learning problem, in which *updates* (the arrival of new evidence labels on nodes) are interleaved with *queries* (evaluations of the posterior distribution at unlabelled nodes). For example, a user may mark several new Web pages as interesting/uninteresting, and then immediately ask for some new recommendations. An algorithm such as backward chaining that takes linear time to resolve a query, or an algorithm such as forward chaining that must be re-run from scratch whenever new evidence arrives, will have unreasonably high algorithmic complexity in our application. We instead devise an algorithm that supports fast updates and queries to the Bayes net.

The observation behind our online algorithm is that the addition of a single evidence node only makes incremental changes in the tree's probabilities. That is, we can memoize certain probabilities at every node in the tree, and the addition of a single evidence node will only adjust a small number of those probabilities (specifically, the probabilities that lie between the evidence node and the root).

We start by describing the calculation of the evidence probability $p(E)$. Computing $p(E \wedge N_i = k)$ can be done the same way (using our fast update algorithm), and from these two quantities we can compute $p(N_i = k \mid E)$. Let E_i denote the evidence in the subtree rooted at node i. Then standard Bayes net inference asserts that for node i,

$$p(E_i \mid N_i = k) = \prod_{j \in \text{children}(i)} \left(\sum_{\ell \in 0,1} p(E_j \mid N_j = \ell) p(N_j = \ell \mid N_i = k) \right). \tag{5.3}$$

This expression says that the probabilities $p(E_i \mid N_i = k)$ can be written in terms of i's children's probabilities. The product is because in a Bayes-net, the children are (by definition) independent from one another given that the parent's value is known. Inside the product is a summation which sums out the probability over the possible values of the children, which is the expression indexed by ℓ.

We simplify notation by writing

$$p_{ik} \equiv p(E_i|N_i = k).$$

At the leaves, the starting value of the probability p_{ik} is easy to compute. If N_i is an evidence node with value k, then:

$$p(E_i|N_i = k) = 1$$

and

$$p(E_i|N_i \neq k) = 0.$$

These are simple leaf starting conditions that say a node's value matches its evidence (if it is an evidence node). If the leaf is not an evidence node, then the starting values are:

$$p(E_i|N_i = 1) = 1$$

and

$$p(E_i|N_i = 0) = 1.$$

.

Recalling also our mutation parameters θ and ϕ we can rewrite the recursive equation, 5.3.

$$p_{i0} = \prod_{j \in \text{children(i)}} (p_{j0}(1-\theta) + p_{j1}\theta) \tag{5.4}$$

$$p_{i1} = \prod_{j \in \text{children(i)}} (p_{j0}\phi + p_{j1}(1-\phi)). \tag{5.5}$$

If we recursively move from the leaves to the root applying this definition, at the root N_1 we will have calculated p_{10} and p_{11}—the probability of all the evidence conditioned on the root's class-tendency. We then incorporate the root prior and the probability of all the evidence E_1:

$$p(E) = p(E_1) = p_{10}p(N_1 = 0) + p_{11}p(N_1 = 1) \tag{5.6}$$

$$= \frac{p_{10}\phi}{\phi + \theta} + \frac{p_{11}\theta}{\phi + \theta} \tag{5.7}$$

The above calculation can be applied recursively to compute $p(E)$ in linear time (in this size of the tree). We now modify the approach in order to quickly update $P(E)$ when a new evidence node is observed. To do so, we memoize the quantities p_{ik} computed at each node in the above recursion. We can then observe that when a new evidence node e is introduced, the only values that change are those on the path from the node to the root. As we work up from e to the root, all inputs needed to compute the changed memos at a node are already memoized at the children of that node.

More precisely, let g range over nodes along the path from e to node 1 (the root)

and let j denote g's child *on that path*. Then the algorithm moves from leaf to root with the following update rule:

$$p'_{g0} = p_{g0}(p'_{j0}(1-\theta) + p'_{j1}\theta)/(p_{j0}(1-\theta) + p_{j1}\theta) \tag{5.8}$$

$$p'_{g1} = p_{g1}(p'_{j0}\phi + p'_{j1}(1-\phi))/(p_{j0}\phi + p_{j1}(1-\phi). \tag{5.9}$$

This says you can update the probability in the tree by multiplying in the "new" contribution of N_j on N_g, and dividing out the "old" contribution of N_j on N_g.

The above calculation incorporated evidence from a new labeled node e, determining $p(E \wedge N_e)$ in time proportional to the depth of node e, independent of the size of the tree or data. The same calculation can be applied to an *unlabeled* node j to compute its posterior probability $p(N_j \mid E) = p(E \wedge N_j)/p(E)$. The numerator $p(E \wedge N_j = k)$ can be computed using the update rule (as if $N_j = k$ were evidence) while the denominator is already memoized.

In many human-designed trees, the depth of the tree has a fixed maximum number. For example, the hierarchy of all organisms only has ten fixed levels (kingdom, phylum ... , species); new organisms create more branches, but not more depth. In such cases, our algorithm updates in a fixed time, regardless of how many other items have already been added to the tree.

5.3.1 Numerical Precision

In many probabilistic applications, including this one, there is a problem manipulating and storing the raw probabilities the algorithm generates. For example, the probability of all of the aggregated examples monotonically decreases as the number of examples increases. In real-world applications, these numbers can easily become lower than the precision offered by floating point numbers.

The Naive Bayes classifier handles precision by calculating *log* probabilities. The log function has the advantage of being monotonically decreasing with the original number, but declining at a much slower rate. While one hundred examples might induce precision problems in the regular space, it would take e^{100} examples for precision problems to occur in the log space.

In the case of Naive Bayes, the log probability is easy to compute, by simply converting the original equation

$$p(x_i|\vec{\theta}_c) = p(\vec{\theta}_c) \prod_j (\theta_{ci})^{x_{ij}}$$

in a straightforward manner into log space:

$$p(x_i|\vec{\theta}_c) = \text{argmax}_c \left[\log p(\vec{\theta}_c) + \sum_j x_{ij} \log \theta_{cj} \right].$$

In the case of our tree learning algorithm, the log probabilities are not as easy to compute. The algorithm we earlier derived (equation 5.9) updated relevant probabilities by traveling from the new evidence nodes to the root. That computation is re-written here:

$$p'_{g0} = p_{g0}(p'_{j0}(1-\theta) + p'_{j1}\theta)/(p_{j0}(1-\theta) + p_{j1}\theta). \tag{5.10}$$

This computation contains summations (unlike naive Bayes which is entirely products), and those summations remain inside the logs:

$$\log p'_{g0} = \log p_{g0} + \log(p'_{j0}(1-\theta) + p'_{j1}\theta) - \log(p_{j0}(1-\theta) + p_{j1}\theta).$$

This does not solve the precision problem because $p_{j0}(1-\theta) + p_{j1}\theta$ must still be calculated outside of log-space. We can only work with a limited set of numbers in log-space—those that could have been computed in earlier steps: $\log p_{j0}$, $\log p_{j1}$ (along with their primed versions), θ and ϕ (along with their logs).

We can rewrite the difficult part of the computation:

$$\begin{aligned}&\log(p_{j0}(1-\theta) + p_{j1}\theta)\\ =&\log(e^{\log p_{j0}(1-\theta)} + e^{\log p_{j1}\theta}).\end{aligned}$$

We can factor this expression such that we retain the most significant part of the computation. With no loss of generality, assume that $\log p_{j0}(1-\theta) \geq \log p_{j1}\theta$. We factor the larger portion out:

$$\begin{aligned}&= \log(e^{\log p_{j0}(1-\theta)}(1 + e^{\log p_{j1}\theta - \log p_{j0}(1-\theta)}))\\ &= \log(p_{j0}(1-\theta)) + \log(1 + e^{log p_{j1}\theta - \log p_{j0}(1-\theta)})\\ &= \underbrace{\log p_{j0} + \log(1-\theta)}_{\text{known in log space}} + \underbrace{\log(1 + e^{\log p_{j1}\theta - \log p_{j0}(1-\theta)})}_{\text{approximated numerically}}.\end{aligned}$$

We have access to the two leftmost expressions, $\log p_{j0}$ and $\log(1-\theta)$, as they are already in log space. Precision is retained for those expressions which are the most significant in the overall calculation. The rightmost expression, $\log(1+e\ldots)$ contains the *ratio* of $\log p_{j1}\theta$ and $\log p_{j0}(1-\theta)$. The expression must be computed outside of log space. If the ratio is very small, then computer precision will set the rightmost expression to zero. This causes a very small loss in precision, but the significant part of the computation (the two leftmost expressions) are retained.

To summarize, these are the original recursive equation:

$$\begin{aligned}p'_{g0} &= p_{g0}(p'_{j0}(1-\theta) + p'_{j1}\theta)/(p_{j0}(1-\theta) + p_{j1}\theta);\\ p'_{g1} &= p_{g1}(p'_{j0}\phi + p'_{j1}(1-\phi))/(p_{j0}\phi + p_{j1}(1-\phi).\end{aligned}$$

We replace this with equations for the log of both sides:

$$\log p'_{g0} = \log p_{g0} + f(p'_{j0}, p'_{j1}, \theta) - f(p_{j0}, p_{j1}, \theta);$$
$$\log p'_{g1} = \log p_{g1} + f(p'_{j0}, p'_{j1}, \phi) - f(p_{j0}, p_{j1}, \phi).$$

where

$f(x, y, z) =$

if $x(1-z) > yz$ return $\log x + \log(1-z) + \log(1 + e^{\log yz - \log x(1-z)})$

otherwise return $\log y + \log z + \log(1 + e^{\log x(1-z) - \log yz})$.

Chapter 6

Ad Blocking and Recommendation Results

In the previous chapter, we described details of a model-based classifier that could learn correlations between certain tree-structured features and learning problems correlated with those features. In this chapter, we apply our classifier to two real-world Web domains that we have discussed throughout the thesis: ad blocking and recommendations. Ad-blocking is the problem of automatically identifying and removing advertisements from a Web-site, and recommendations is the problem of finding and presenting interesting information on a user-specific basis. This chapter is the expanded version of work done with David Karger [53].

6.1 Tree Structured Features for the Web

In this section, we describe in more detail two tree-structured features, URLs and HTML-tables, and why they might help for our ad-blocking and recommendation domains.

6.1.1 URL Features

The World Wide Web Consortium argues that document URLs should be opaque (`http://www.w3.org/Axioms.html#opaque`). On this Web page, Tim Berners-Lee writes his axiom of opaque URIs: "... you should not look at the contents of the URI string to gain other information...".

In contrast to those style guidelines, most URLs nowadays have human-oriented meanings that are useful for recommendation problems. Indeed, the guideline's URL contain semantics including authorship (w3.org), that the page is written in HTML, and that the topic relates to an "Axiom about Opaqueness." As the document's URL demonstrates (somewhat ironically), URLs are more than simply pointers: authors and editors assign important meanings to URLs. They do this to make internal organization simpler (authorship rights, self-categorization), and sometimes to make that organization scheme clear to readers. Readers often make inferences from URLs,

which is why browsers and search engines usually display URLs along with the text description of a link. We can infer from a URL that a document serves a particular function (a specific Web directory might always serve ads); or relates to a topic ('business' stories might be under one directory); or has a certain authorship. In short, similar documents (as defined by the site's authors) often reside under similar URLs. A good URL structure provides helpful contextual clues for the reader. Note that URLs, which usually start with "http://" are distinct from the Web site structure (the graph of links within a Web site).

The URLs themselves are extremely good features for learning techniques. First, they are easy to extract and relatively stable. Each URL maps uniquely to a document (at a given time), and any fetchable document must have a URL. In contrast, other Web features like anchor text, alt tags, and image sizes, are optional and not unique to a document. URLs can be obfuscated, but such schemes require effort and knowledge beyond simple HTML. Second, URLs have an intuitive and simple mapping to certain classification problems. For example, we argue and give empirical evidence in Section 6.2 that the URL is highly correlated with whether a link is an advertisement or not. Most advertisement clicks are tracked through a small number of programs; these programs are usually contained in subtrees of the URL tree, like `http://doubleclick.net` or `http://nytimes.com/adx/...` . Third, URLs can be parsed without downloading the target document, which makes them fast to read. This is a necessary condition for real-time classification tasks like ad-blocking.

To convert a URL into a tree-shape, we tokenized the URL by the characters `/`, `?` and `&`. The `/` is a standard delimiter for directories that was continued into Web directories; `?` and `&` are standard delimiters for passing variables into a script. The left-most item (`http:`) becomes the root node of the tree. Successive tokens in the URL (i.e. `nytimes.com`) become the children of the previous token.

6.1.2 Table Features

Similarly, the visual layout of a page is typically organized to help a user understand how to use a site. This layout tends to be templated—most pages will retain a 'look and feel' even though the underlying content might be dynamic. For example, different articles on one particular topic might appear in the same place on the page day after day. The page layout is usually controlled by HTML table tags, corresponding to rectangular groupings of text, images and links. Often, one table along the side or top of a page will contain much of the site's navigation. A site might use tables to group together articles by importance (the headline news section of a news-magazine), by subject, or chronologically (newest items typically at the top). Like the URL, this page layout can be used to eliminate certain content (such as the banners at the top of the page); or to focus on other content (the headlines, or the sports section). Like the URL feature, tables make good features for machine learning. For a page to display properly in browsers, the tags have to be in a specific form specified by the World Wide Web Consortium; this also makes the table feature easy to extract. In the next section, we give the example of a Chinese Web site that might be understood even barring understanding the specifics of the content on the site.

Figure 6-1: Shown is how the URL can be de-constructed into a tree that contains some human-oriented meanings.

To convert the HTML table structure into a tree-shape, we used a hand-written perl program that extracted the HTML table tags (`<table>` and `<td>`). The root of the tree is the entire page's HTML. The children of a node are the next lower level of table elements.

6.2 Ad Blocking

Throughout this thesis, we have discussed our desire to build real-world Web applications that save users time and effort, such as an ad-blocker. In this section, we give experimental evidence that our tree learning algorithm can yield performance that matches a hand written, commercial ad blocker.

Conventional wisdom says blocking ads is as simple as blocking certain fixed aspect ratios on incoming images. This is simplistic, and belies the fact that entire companies have formed around the problem of ad-blocking—and ad-creation. Advertisers constantly change the form of advertisements: they are becoming larger, shaped more like content images, or even come in the form of text or colored tables. A static ad-blocker goes out of date in so-called "Internet time."

The conventional way to adapt to advertising changes is time-intensive for humans. Engineers write new sets of rules that match the advertisements but not the page's content. The rules might include combinations of any number of factors: height, width, or size of an advertisement; the domain serving the advertisement; or certain word patterns. These rules are not easy to write, as they must be general enough to accommodate next week's advertisements but specific enough that they don't accidentally block next week's content. After coding and testing the new rules, the software is released and users have to periodically update their systems with new rules. All of these steps take time, for both the engineers and the users.

Our general goal is to save human time by building a system that finds its own rules, eliminating the need for human input whatsoever. Our goal is to see whether an entirely self-directed computer can achieve accuracies comparable to teams of humans working on the identical problem. If so, our ad-blocker can learn new rules nightly, adapting daily to new ad forms.

Our work is similar in spirit to the AdEater system which also learns ad-blocking rules [32]. AdEater has a 97% accuracy (the only metric reported) on their ad blocking experiments, though it requires 3,300 hand labeled images to achieve that accuracy. AdEater uses humans to perform the relatively simple task of labeling images as ads or not; the machine learning takes on the more difficult task of discovering rules. However, AdEater suffers in the same way that most ad-blocking programs do: it takes considerable human effort and time to produce an update; hence its accuracy decreases as new ad forms surface.

We want to label our data for training our classifier. One alternative is to use human labeled data, the way AdEater does. However, this is time consuming and eliminates much of the benefit of having the computer learn ad-blocking rules. Instead of using high-quality human data, we use a heuristic called *Redirect*. Redirect labels ads if a link goes to one site then redirects to another place. The redirect heuristic

Figure 6-2: A Table can be de-constructed into a tree that contains some human-oriented meanings.

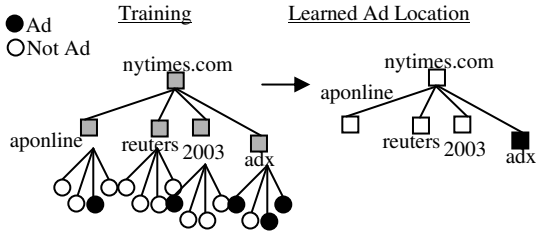


Figure 6-3: (Left) A portion of the parsed New York times hierarchy, as applied to the ad blocking problem. Notice that a portion of the real news (APOnline) is incorrectly labeled as an ad. (Right) Shown is the algorithm's maximum likelihood guess of the color of internal nodes. Despite some mis-labelings of training data, the algorithm quickly identifies the "Adx" sub-directory as filled with ads.

makes sense because it captures the normal process that advertisers use: tracking the click, then sending the user to the advertised site. Notice that this is much more of a "content" based heuristic than image sizes which are "form" heuristics; ads can take any shape and size, but most current advertisements incorporate some type of tracking mechanism.

As we shall see, the Redirect rule is about as accurate as AdEater at identifying ads. But there is a big barrier to using Redirect itself as an ad blocker: to decide whether to block an ad, Redirect must fetch it first. This generates a significant overhead in additional connections and downloads—one which in today's network environment makes the Redirect heuristic too slow to use in real time ad blocking. However, we will show that our tree-based learner can predict a redirect without actually trying it—this gives us an approximation to the redirect heuristic that *can* be used for real-time ad blocking.

In practice, click through tracking requires back-end infrastructure like databases and CGI scripts. Therefore, click-through ads tend to be located together under a small number of URL directories per site (i.e. under xyz.com/adserver). It is rare for a site to have advertisements and content in the same leaves of the URL tree. Therefore, we use our tree learning algorithms to associate existing URLs with an 'ad' or 'not ad' label provided by the Redirect heuristic. As a new page is loaded, the learning model predicts whether new URLs are 'ads' or 'not.' A big advantage of the redirect heuristic is that training examples can be provided *automatically* by an off-line algorithm that, when the user is doing other things, visits sites and takes the time to check and follow redirects. In other words, we can train without any human input. Figure 6-3 shows how our algorithm might find ads on the New York Times Web site. One drawback to this approach is that each Web site must be trained on a per-site basis; more standard rules like "an ad is a 300 x 250 image" can apply to many Web sites simultaneously.

What follows is an experiment that compares the performance of our ad-blocker to a commercial system. We compared two control ad-blocking 'trainers' with two ad-blocking 'learners.'

6.2.1 Experimental Setup

In order to label the advertisements on a given page, we downloaded the page and saved the HTML into a cached file. The cached file was necessary so that every technique would be viewing exactly the same Web page. We went through the cached file locating links on the page. We used perl to find all links on a page, with a regular expression that matched `<a href= ... /a>`. This corresponds to a standard HTML definition of links on pages.

From the list of links we found on the cached page, we only used links that contained images. This is because, at the time of our experiments, the commercial ad-blocker (WebWasher) only blocked image-based advertisements. To detect images embedded within links, we only took the subset of links that contained the text `<img` which is a standard HTML definition for displaying images. For a given page, we call this subset of the links on a page *image-links*. While we solely train on image-based

links, later we talk about how such rules might apply to text-based links as well.

Given this list of image-links, we used four rules to label whether a link contained an advertisement or not. The four systems are listed below:

• *WebWasher* is a commercial product with handwritten rules that uses many features like the dimensions of the ad, the URL and the text within an image. It is in use by four million users. WebWasher can be downloaded free for educational and personal use[1]. We used the WebWasher Linux version, which was installed as a proxy. The WebWasher proxy reads in the requested Web page, erases the advertisements, then forwards the modified page to the Web client.

In order to have WebWasher label a page, we ran our cached file through the WebWasher application. This removed all the image-based links it deemed advertisements. Any link that was in our original list of image-links, but was not present in the WebWasher-processed page, was deemed an advertisement by our WebWasher rule.

• *Redirect* is the simple heuristic mentioned above that monitors third-party redirects. As we mentioned, it is a somewhat noisy heuristic, meaning its accuracy is less than human's; but it can run in the background without human input.

The first step is to write a simple perl script to check for third-party redirects. The script goes through the list of image-links and tries to fetch each link using the program lynx[2]. Lynx downloads the resulting pages, expressed as HTML (rather than graphically), which allows the program to check for certain conditions that might serve as redirects. Redirects are defined in the program as anything that has the phrase "Location:" at the top of the page (a standard way to redirect pages quickly in HTML). The text after the "Location:" phrase contains the target (redirect) site; the text is parsed to arrive at the top-level domain. Other forms are redirects are described below.

The domain was defined as anything after "http://" but before the next "/" (i.e. in "http://nytimes.com/adx/...", the domain is "nytimes.com"). We extracted the top-level domain from the domain. The top-level domain is only the right-most portion of the domain, the two strings separated by a ".". For example, "sports.yahoo.com" and "finance.yahoo.com" both share the same top level domain, which is "yahoo.com". If the domain of the redirect differed from that of the originally cached page, then Redirect labeled the image-link as an advertisement.

There were instances of links and redirects that the lynx program could not understand like some javascript or flash-based links. Lynx would return either a blank page or the phrase "no content." Heuristically, we found many of these links (but not all) were advertisements, and we had Redirect label empty or "no content" pages as advertisements.

• *Learn-WW* is our tree-based URL algorithm, *trained on WebWasher's output.* The URL provided the tree-structure as discussed in Section 6.1.1, and WebWasher provided the labeling for the leaf nodes. The tree-algorithm was then run to label new, unseen links as either advertisements or not according to our tree-algorithm as

[1] `http://www.Webwasher.com/client/download/private_use/index.html`

[2] `http://lynx.org`

specified in Chapter 5.

We had mentioned before that some links contained javascript which were not understandable by lynx. While Redirect and WebWasher might not understand javascript, the links were read into the tree-learner as "javascript: ..." As described, our tree-algorithm would view the various javascript-based links as siblings of one another, and classify them with the same labels. Empirically, this tended to have good results.

• *Learn-RD* is our tree-based URL algorithm, *trained on Redirect*. The method for training Learn-RD is identical to that of Learn-WW, except that the RD (third-party redirect) heuristic was used in place of WebWasher to label the leaf nodes.

As we have mentioned before, RD alone is a good heuristic, but cannot operate in real-time. Our hope is to train a classifier like Learn-RD off-line periodically with RD, allowing for a fast ad-blocker that has been trained by the RD heuristic.

6.2.2 Testing and Training Data

In the previous sub-section we described a variety of algorithms designed to label whether the links on a page contained advertisements or not. In this section, we describe how we acquired training and testing data.

We generated a dataset from Media Metrix's largest 25 Web properties as of January 2002[3]. Empirical evidence shows the average user spends all their time on a small number of the largest sites. We felt that blocking ads of the largest 25 Web properties would be both representative and beneficial to many, if not most, users.

We crawled through a given Web site, randomly picking eleven pages linked from the front page that shared the top-level domain with the front page. We checked whether the top-level domain of the target and front page matched with the method described in the Redirect heuristic from the previous sub-section.

The links on those eleven pages, along with the links on the front page, were divided randomly into a six-page training and a six-page test set. Each Web site went through random training and testing twice. WebWasher and Redirect classified each linked image in the training group, and those classifications, along with the link URL, were used to train Learn-WW and Learn-RD respectively.

Next, all four classifiers were applied to the linked images on the test group. If all four classifiers agreed that an image was either an ad or all agreed it was not an ad, they were all deemed to have classified the image correctly. Spot-checks suggested that agreement between all methods almost always lead to the correct prediction. If one technique disagreed with the others, a human was used to judge whether the image was really an ad or not.

6.2.3 Experimental Results

In total, 2696 images were classified, and all four classifiers ended up with an average classification accuracy across all sites of within a quarter percent of 93.25% (see Ta-

[3] `http://www.jmm.com/xp/jmm/press/MediaMetrixTop50.xml`

Table 6.1: shown are the Web blocking accuracies for several Web sites, along with standard deviation information for the top 25 Web sites.

	Top 25 Sites	weather	look smart	euniverse
Web-Wash	.935 (.136)	.907	.857	.517
Learn-WW	.931 (.118)	.860	1	.517
Redirect	.933 (.08)	.842	.857	1
Learn-RD	.934 (.08)	.837	1	1

ble 6.1). The first column contains average error rates for the 25 sites, with standard deviations in parenthesis. Standard deviations are based on site-to-site comparisons. The overall false negative rates (labeling an ad as content) and false positive rates (labeling content as ads) were approximately 26% and 1% respectively, and that was fairly consistent between all the classifiers. Note that the mistakes were biased towards false negatives, which means the classifiers let through some ads, but rarely blocked content. This is probably the appropriate behavior for an ad-blocker.

Table 6.1 also shows data for specific sites. On various sites, trainers beat learners (weather.com); and learners beat their trainers (looksmart.com). Thus learners are not exact imitations of their trainers, but on average end up with the same accuracy rates.

The standard deviations of the four classifiers are relatively large because errors tend to cluster around a few Web sites. WebWasher, for example, does poorly marking ads on euniverse.com, which uses different dimensions for its ad images than many sites. Redirect does poorly on portals and internal ads. For example, Redirect would not label a New York Times advertisement for its own newspaper as an advertisement.

It is possible for the learners to generalize to better performance than the trainers. For example, all of the New York Times advertisements are in a few URL directories. Thus, a few incorrect examples from Redirect (internal ads that don't redirect) are ignored in favor of the larger number of correct examples; the learner correctly "overrides" Redirect. Conversely, the learners can also generalize incorrectly; if the trainers were a little more wrong than right, the learners could end up generalizing in the wrong direction and consistently mis-labeling the links. Such problems happened on the weather.com Web site.

The Redirect heuristic worked well; its accuracy was not different by a statistically significant factor from the commercial ad-blocking program. Redirect's simplicity suggests that, for now, that it is correctly understanding the mechanisms by which most sites serve up ads (i.e. through third party redirects). That mechanism can certainly change, but such changes would be harder for advertisers to make than simply changing the advertisement sizes, which can foil WebWasher at times.

The URL seemed to be a good feature for ad-blocking. Both WebWasher's and Redirect's rules seemed to be expressible in many cases by the single URL feature. Had the URL been a poor feature, one would expect the tree-learning based ad-blockers to have much lower accuracies.

A spot-check of random sites (using a random link generator like Mangle[4]) suggests that the URL feature works even better on smaller sites than larger sites, because smaller sites tend to use third party advertisers whose URLs are almost entirely used to serve advertisements, like doubleclick.com[5]. For a service like Overture, which is owned by Yahoo, the advertisements appear in a subdirectory of Yahoo,[6].

We wish to make a few points about our results. First, our Learn-RD algorithm achieved performance comparable to a commercial ad-blocker, without needing complex, hand-written rules. In fact our system performs better in some respects. Most ad-blocking systems do not remove text-based ads (ads, for example, placed within search engine results), while both Redirect and our learner trained on Redirect acts no differently for image-based ads and text-based ads. Second, we argue that our ad-blocking classifier will adapt in the long term better than a static version of WebWasher, since it can update rules nightly without any human input. Of course, in the adversarial world of advertisements, it is probable that this system, too, would be defeated if it became widespread: an advertiser could deliberately obfuscate their URLs. Third, we point out that unlike the black and white "redirect" heuristic, our learner gives pages a range of "blackness" scores. It can thus be tuned to tradeoff false positives and false negatives depending on the user's preference.

6.3 Recommendation Experiments

In the previous section on Ad blocking, we showed that machine learning on tree-based features could match or outperform hand-coded rules (WebWasher) and an inefficient heuristic (Redirect). In this section, we demonstrate that tree-based learning can also outperform classification algorithms based on traditional textual (and other) features. As we mentioned in the introduction, it is our goal to build real-world Web systems that save time and effort on behalf of users. One of those systems was a recommendation system designed to find interesting links customized to individual users. In this section, we discuss experiments to judge the effectiveness of such a system.

Recommendation problems are typically difficult, especially for domains that are greatly subjective (like news, movies, and music). There are a few common problems that span all recommendation systems. One problem is that most systems have no real semantic knowledge of most of these domains. They do not understand the news, have never bought a movie ticket, and have a terrible ear for music. This puts them at a terrible disadvantage; for the most part, their recommendations are not based on an understanding of their domain.

Artificial intelligence, to date, has a hard time understanding things that would be useful in a human context. On the other hand, that is precisely the job of all the human editors. It may be difficult for a computer, or even an unknowledgeable human, to distinguish between college and professional football articles—but sports

[4] `http://www.mangle.ca`
[5] `http://doubleclick.com`
[6] `/partner/.../overture`

editors can differentiate between the two easily. It is the job of such a sports editor to place a web page within the site, at a URL, and within the page layout. Our learning, then, strives to use all this information crafted by editors to help learning progress.

Humor is typically hard for a computer to understand, and yet one correct recommendation is for a humorous article titled "Food pantry gets 3,600 confiscated eggs." It would be difficult to teach a computer that the text of this article was humorous, but the system has quickly learned that these sorts of articles reside near other clicked articles in a certain place on the boston.com web site. In fact, the URL that it has keyed upon would not be intuitive to humans (under boston.com/news/daily/).

Overall the two tree algorithms performed well, even against the Support Vector Machine which is commonly thought of as one of the best general purpose learning algorithms. In the next sub-section (6.3.1) we describe the set-up of the user study, followed by a sub-section (6.3.2) on the algorithms we tried and their advantages and disadvantages. That is followed with empirical results and analysis of our algorithms (6.3.3).

6.3.1 User-Study Set-Up

We asked study participants, totalling 176 users, to indicate any news articles they would normally click on. Users were asked to click on links for 7 front pages of data. The pages were visually unaltered replicas of pages downloaded from the Web, consisting of 5 consecutive days worth of the *New York Times* (September 15th through 19th, 2003). The pages and user study are available from our server [7]. Based on the submitted email addresses of the participants, the users encompassed a broad range of people extending all the way from the United States to Australia.

In order to encourage a large number of participants, we made the experiments easy to complete and gave financial incentives for doing the experiments. The experiment was entirely conducted in one session on the Web, by using the cached news stories mentioned above.

Some users did not complete the study, and a small number of others did not click on any links; these were discarded from the study and are not considered part of the 176-user sample set. Note that according to our statistical methodology (see Appendix), this does not change the significance of our results.

Each click on a link within the user study would place a check-mark in a checkbox corresponding to that link, and when the user submitted the page, a list of all the clicked (and un-clicked) links was noted on the server. The clicked (and un-clicked) links were used to generate positive and negative examples with a variety of data sources (next subsection) which were then made available to various learning algorithms for prediction of unseen clicks.

We chose the *New York Times* as one of the most popular news sites in the world, meaning that the content of those pages was broadly focused. We believe it is representative of news sites that many users read.

[7] `http://e-biz.mit.edu/data3`

Data Sources

For each link on each page, we collected a variety of data for use in our algorithms, to the extent practicable. For each link, we have a copy of the link's URL, the anchor text of the link, the position of that link in the table structure of the page, and the full text of the article. In total there were 1105 links across the five pages[8].

There are some difficulties with collecting the data for every link, particularly the textual data.

For example, not every link has anchor text inside it, since some links are only images. Therefore, when available, we took the "alt" tags to stand for the text. Even so, there are several image-only links that have no available anchor text. In total, approximately 1% of the links did not have anchor text we could parse within them. We will refer to data based on the anchor text in the link as "Anchor."

It is even more difficult to fetch all of the pages behind the links. For example, many sites (including the ones we chose) only allow registered users, which generally means that the fetching agent needs to understand and respond to cookies. Harder still is following the many types of redirects and translating properly between absolute and relative links. Also many pages (again, including the pages we chose), have javascript links that require a javascript interpreter to understand. We used lynx as a browser, and wrote special routines to follow redirects, to automatically log in to the site via cookies, and to translate between absolute and relative links. While not perfect, we believe we gathered as many of the target pages as was reasonably possible. We did not follow javascript-based links because we were not aware of any reasonable way to write a javascript parser within lynx.

There is also a problem of rights for visiting Web sites. Many Web sites (including those we chose) do not allow spiders on the pages underneath the home page. Also, standard etiquette says you may not download more than one page every five seconds. We followed the second rule but not the first since the first makes it impossible to grab the full text of articles. One occasional consequence of having a slight delay between downloading Web pages is that sometimes the pages disappear before downloading. Though we tried to get as much of the pages as possible, around 5% of them were not download-able for the reasons given above. We will refer to the data based on the full text of the fetched document as "Doc."

The URL of the link is comparatively easy to read, as it is a pre-requisite to having a usable link (and a pre-requisite to being able to download any pages, for both our robot and for a user). The table structure the links sits in is slightly more difficult, because it requires parsing the page into table elements, but every link has a position in the table structure. The table features and the URL were extracted directly from existing code from the Daily You (the application based on these empirical tests, described more fully in Chapter 7) without modification. Both of these datasets were complete: i.e. for every link a person could click on, there is a corresponding URL "URL" and table element "Table" that link sat within.

A drawback to both the table and URL features is they are almost always site-

[8]The raw recommendation data and features are available at http://www.ai.mit.edu/ kai/recommendation_dataset.txt

specific. That is, learning a certain subdirectory or visual block of the *Times* Web-site does not help us learn anything about other news Web-sites.

To summarize, there were five basic features used:

Anchor: the anchor text within the link.

Doc: the full text of the words in the linked document

Url: the URL in a form that retains its path through the tree

Table: the location of the link in the table, in a form that retains its path through a tree.

All: a vector of all the features. (See Chapter 2 for the standard method, which we used, of converting textual features into vectors).

6.3.2 Recommendation Algorithms

We tried several algorithms with various datasets labeled with each user's click data. That is, for each of the 1105 data points, 176 different combinations of positive and minus labelings was found based on the user's empirical news selection.

We used two "general purpose" classifiers, Naive Bayes ("NB") and the Support Vector Machine ("SVM"), across all four of the feature sets, plus our tree learner ("TL") on the two tree-structure features. Naive Bayes is detailed in Chapter 3, the SVM is discussed in the background and in Chapter 4 and the tree learner is discussed in Chapter5.

We chose the SVM because it is considered to be one of the best general purpose, discriminant classifiers [25]. We used a standard, fast, publicly available implementation called SVMFu [46].

In order to give the various non-tree algorithms (SVM, Naive Bayes) a chance to learn based on the URL and Table tree-features, we took each node from the tree and translated it into a 'word' that contained information about the path to that node. For example, the URL `http://nytimes.com/business` was tokenized into three 'words': `http://`, `http://nytimes.com`, and `http://nytimes.com/business`. We chose this "nested" tokenization instead of the obvious splitting up of the URL so that the representation would still convey the exact position of a node in the tree, giving the non-tree algorithms the opportunity to generalize in the same way that our tree-algorithms might.

Test Environment Parameters

The algorithms we used required specific parameter settings that we describe here so the experiments can be replicated. We chose Naive Bayes because it is a fast, common model-based classifier whose model does not work well with tree structures. Specifically, NB assumes independence between features whereas a tree implies certain strict dependencies. For NB we used an α smoothing parameter of 1 [14].

The SVM required the setting of a "C" parameter that measures the outlier penalty. We used a smaller portion of the test set (2 pages) to determine an optimal C parameter for the four different feature sets. We tried C values of 1, 3, 5 and 10. Across the four feature types (Link, Doc, Url, Table), changing C did not

substantially change the results, so we fixed a C parameter of 1. Other parameters included setting the number of cache rows to 3000 (a switch that changed speed but not accuracy) and setting the kernel to linear. Both the kernel and the data were represented as floating point numbers.

Overall we tried a wide variety of algorithms on a cross-validated set of the *New York Times* data. We used 4 training and 1 test document (that contained multiple links to other documents) per user to compile our final results, across all of the 176 users. In total, this corresponds to 182,325 classified links per experiment (1105 *Times* links x 176 users).

We tried a total of 10 different algorithms corresponding to mixtures of different algorithms and features:

Support Vector Machine (C=1): SVM-Anchor SVM-Doc SVM-Url SVM-Table SVM-All

Naive Bayes ($\alpha = 1$): NB-Anchor NB-Doc NB-Url NB-Table NB-All

Tree Learning ($\theta = .2$ $\phi = .05$):TL-Url TL-Table

For example, NB-Url means we used Naive Bayes on the URL feature.

6.3.3 Recommendation Results

Our recommendation results are from the five-page New York Times data set. We used 5-fold cross validation, across each of the pages. For each of the 176 users, each page was used once for testing and four times for training. We used the statistical significance methodology found in the Appendix as described in Hollander and Wolfe [23]. Briefly, we used a non-parametric statistical test which compared two algorithms. If one algorithm consistently outperformed the other across much of the 176-user sample, then the difference between the algorithms was considered significant.

On average, users selected nine stories a day out of an average of 221 possibilities each day (that is, for each front page presented, they picked nine links from that page). Each of the classifiers produced a ranked list of one page recommendations for each user and each page (one page holds one day of stories). Results are written in terms of the number of correct recommendations across all users and pages within the top recommendation, top three recommendations, top five recommendations, and top ten recommendations. Complete results are shown in Table 6.2. A perfect classifier (second row) would have achieved 857 correct recommendations when producing one recommendation per day per user (176 users times 5 days; a small number of users did not click on any recommendations for some days). A random classifier, on the other hand, would have done a statistically significantly worse job than any of the trained classifiers.

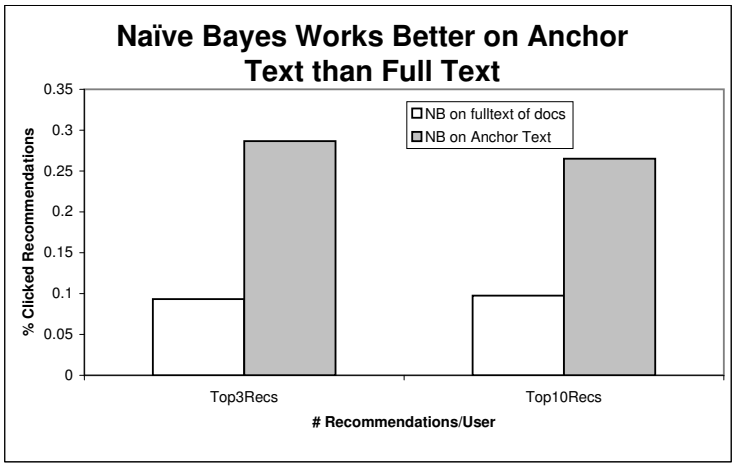
Overall the best performing algorithm was the tree learning algorithm applied to the URL feature, which had the best scores for three out of the four categories. The second best algorithm was a tie between the SVM applied to the URL feature and the tree learning algorithm applied to the table feature. Each of these performed well in different categories.

Classifier	Top Rec	Top 3 Recs	Top 5 Recs	Top 10 Recs
Random	29	104	162	330
Perfect	857	2488	3899	6093
NB-Doc	81	232	342	594
NB-Anchor	302	713	1021	1615
NB-Table	72	180	278	576
NB-Url	80	319	507	1036
NB-All	121	271	372	464
SVM-Doc	59	156	257	520
SVM-Anchor	220	614	913	1438
SVM-Table	177	472	662	934
SVM-Url	308	839	1268	1953
SVM-All	203	542	786	1311
TL-URL	385	**979**	**1388**	**2149**
TL-Table	**401**	900	1176	1512

Table 6.2: Summary of all the classifiers and features on the New York Times datasets. The numbers represent the number of clicked recommendations each classifier would get, if they (columns) recommended the top, top three, top five, and top ten highest scoring links. Bolded items have higher accuracies than non-bolded items for a given column (statistical significance methodology is reported in the Appendix).

Figure 6-4 shows a comparison between two of the text features (Anchor and Doc) against two algorithms (SVM and NB). Results are drawn as the percentage of perfect (as defined in the table) that each classifier reached for, respectively 3 and 10 recommendations. In both cases, the anchor text provides a much better feature than the fetched document's text. This may be because the anchor text is supposed to contain a summary of the document, and it is easier for the classifier to understand these summaries than the documents themselves. This is a positive result for text classifiers because the anchor text is much easier to fetch than the target document's text (which, as mentioned before, can be slow and can be difficult technically to download). The differences are all statistically significant.

Figure 6-5 shows the difference between the various algorithms on the tree structured features. As mentioned previously, we "tree-ified" the features for the benefit of NB and the SVM so each algorithm would be aware of the position in the tree. The tree algorithm does the best job of taking advantage of the tree-structured features. This may be partially due to its domain knowledge of the problem, coupled with a relatively small amount of training data. In those situations, model-based classifiers often perform better than discriminative classifiers [40]. This is particularly true on the table features, which tend to be much flatter than the URL: the depth of the table tree fluctuates less than the depth of the URL tree. The next best performing algorithm is the SVM, which generalizes well considering it has no domain knowledge. The worst classifier on tree-structured features is NB. We believe this is because NB assumes independence between features while a tree actually generates highly depen-



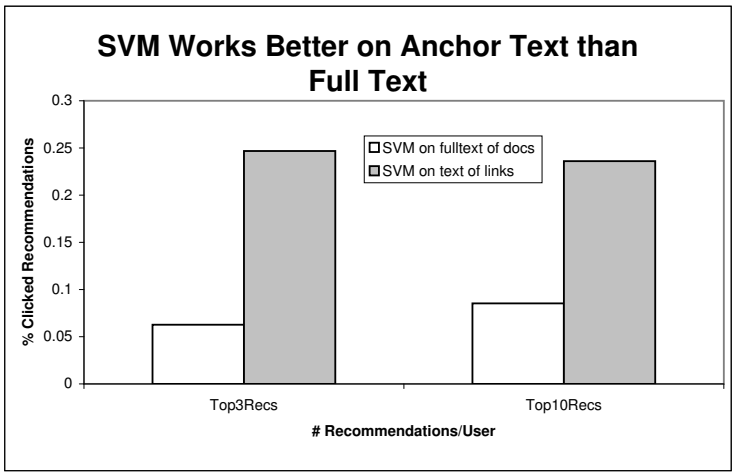


Figure 6-4: A comparison of using the SVM and NB classifiers on text features. The bars are measured in terms of the percentage of clicked recommendations each algorithm made when respectively suggesting their top three and top ten best scoring links. The graph shows that the anchor text is a better feature than the document text for our recommendation problem.

dent features (a given child always has the same parent). The differences shown are all statistically significant according to the methodology described in the Appendix.

When all the feature classes were combined together (SVM-All and NB-All), classifier performance degraded when compared to the single best feature class (respectively SVM-Url and NB-Anchor). In the case of naive Bayes, the addition of the tree-structured features probably hurt the classifier. As mentioned before, tree-structures are in conflict with naive Bayes' model, and the addition of those features probably causes erroneous weightings of those features (see Chapter 3 for a description of why Naive Bayes does poorly on features that are dependent). In the case of the SVM, given enough information it should be able to combine all the features fairly well. Unfortunately, with the small amounts of training data available, the extra features probably caused it to over-fit and generalize improperly.

Figure 6-6 shows the performance using the best feature with the various algorithms on our data sets. Naive Bayes needs to use the text-based features (since it is unsuitable for the tree-based features), and as discussed before, the best text-based feature was the anchor text. Both the SVM and the tree-learner did best on the URL feature, suggesting that the URL feature is a good feature for use on recommendation problems like the one we posed. The tree-learner slightly outperformed the SVM on the URL feature, and all the differences were statistically significant.

Figures 6-7 and 6-8 show scatter plots with the rank of one classifier's recommendations against the rank of the second classifier's recommendations. The figures show that the way recommendations are made through various features (Figure 6-7) have only small amounts of correlation; but that the SVM and tree-learning applied to the same features (Figure 6-8) have similarities in many regions.

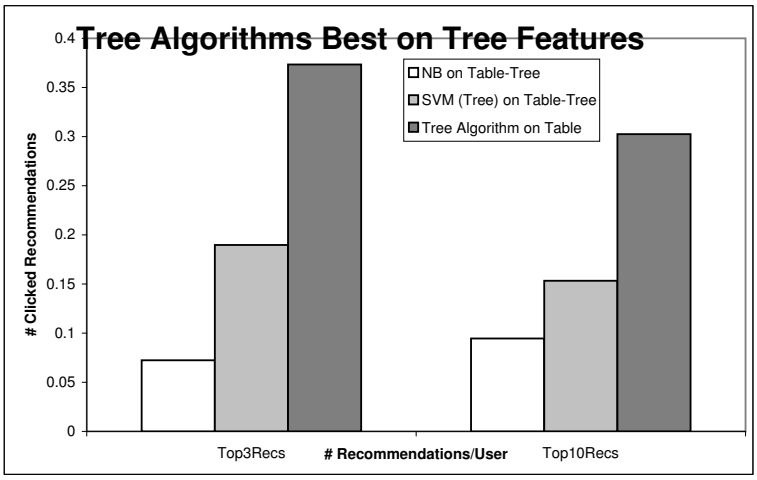
6.4 Conclusions

We began with the observation that the proper choice of features can have a significant impact on the performance of classification algorithms. Since Web site authors have an incentive to organize their materials, for the sake of both the author and their audience, we hypothesized that the URL of documents and the physical placement of elements on a page could provide clues into the Web site's fundamental organization.

To take advantage of this observation, we noted that URLs and table structure can both be viewed as trees, and this facilitated certain machine learning algorithms. These learning techniques try to find correlations between certain properties (i.e. the document being an ad) with the document's location in either the URL or table tree.

We argued that our new features and learning would produce fast, good classification schemes. We gave empirical results on two Web applications: an ad-blocker and a recommendation system. We showed that the ad-blocker could achieve commercial-grade accuracy without requiring any human inputs. The recommendation results showed that our tree-learning approach outperformed conventional techniques and features on real world news recommendation data.

Our tree-based algorithms exhibit a well-recognized tradeoff between specificity and accuracy. The text-based classifiers we compared our work to are very general:



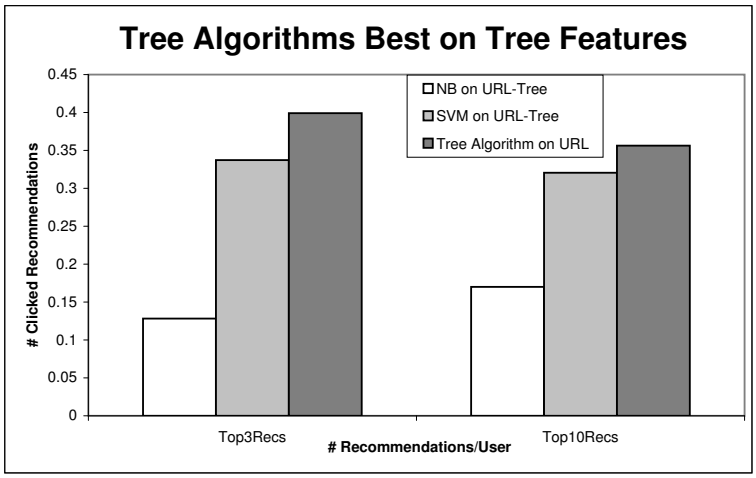


Figure 6-5: These graphs compare various algorithms on the tree structured features. Naive Bayes, as expected, does poorly as its independence assumption is wrong. The SVM does well on the URL feature but not as well on the table feature, while the tree algorithm seems to take advantage of the tree structure.

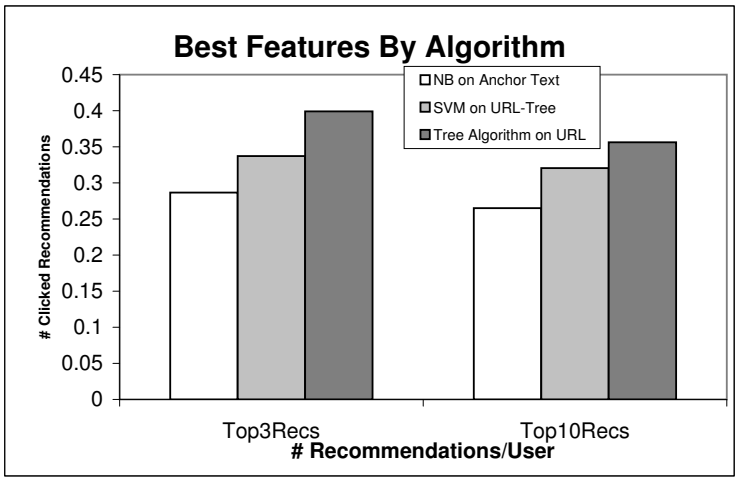


Figure 6-6: These graphs show the best features for each algorithm. Naive Bayes works best on the anchor text, while the SVM and tree learner performs best on the URL feature. The differences between the algorithms are statistically significant.

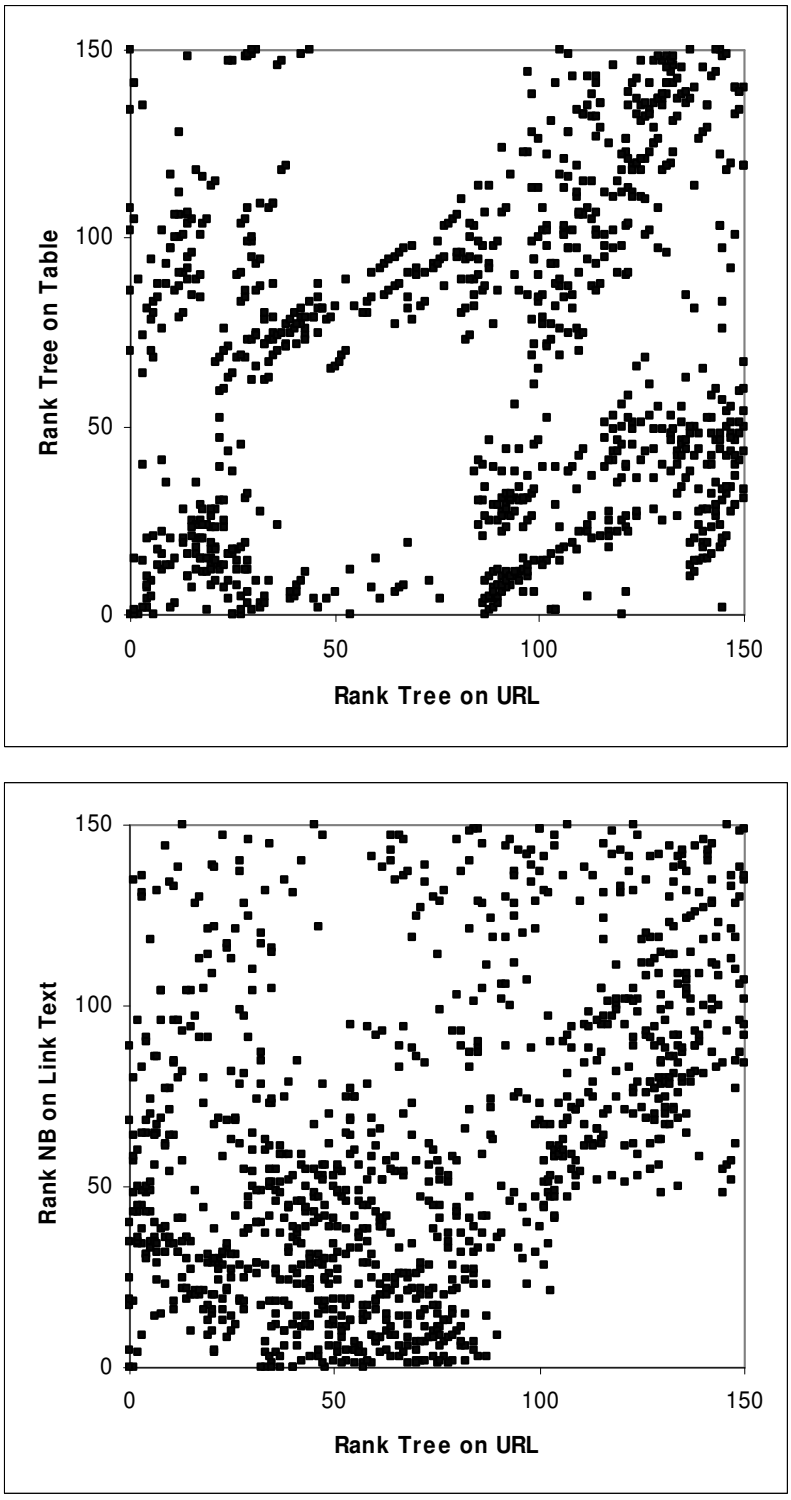


Figure 6-7: Scatter plots show how various algorithms are correlated with or against one another. (Top) The tree on the URL and the tree on the Table features exhibit some correlation (points near the x-y axis); (Bottom) Naive Bayes on the link text shows little correlation with the tree learner, though the white space in the lower right indicates NB did not rank anything highly that the tree learner ranked low.

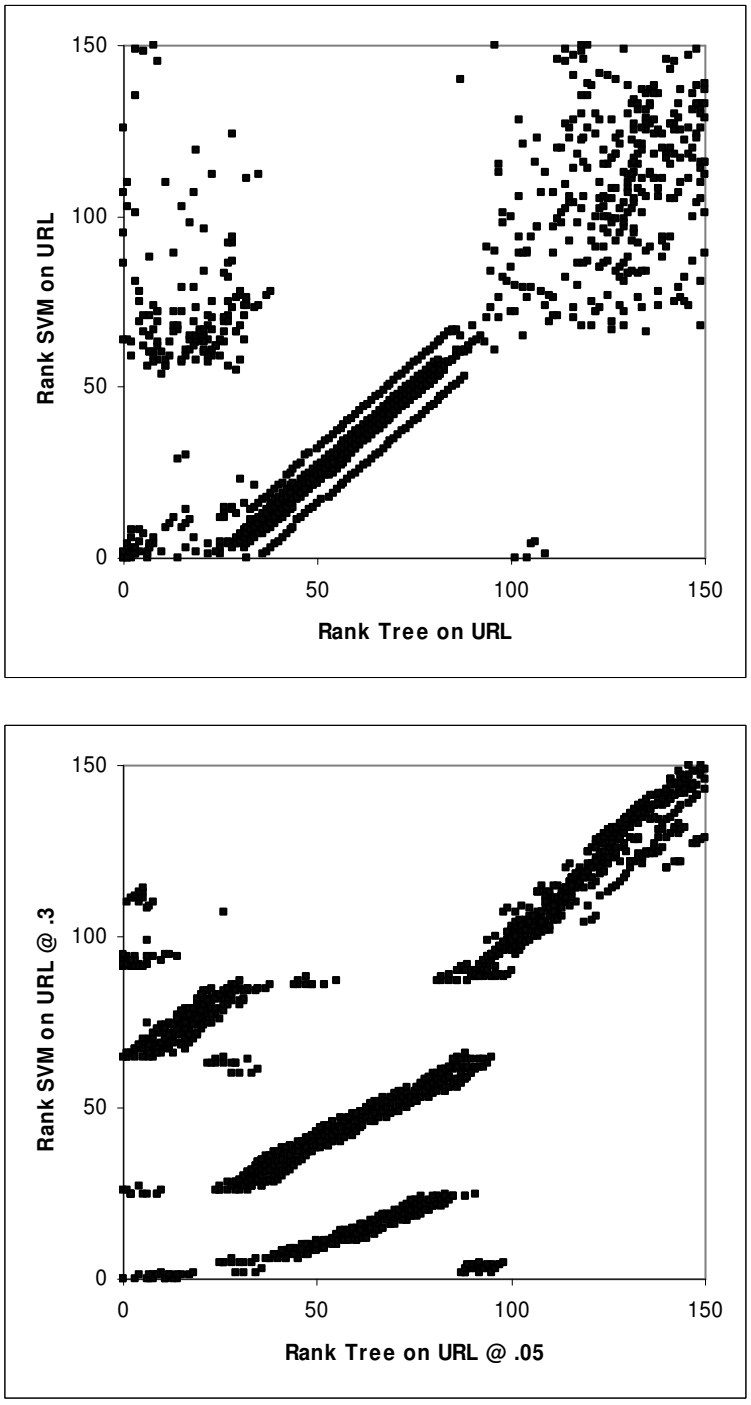


Figure 6-8: Scatter plots show how various algorithms are correlated with or against one another. (Top) The tree algorithm on the URL is generally in agreement with the SVM, but many results are shifted (diagonal element). (Bottom) The tree algorithm with two different parameter settings shows strong agreement with the negative choices along with some translated rank features (diagonal lines in the lower left quadrant).

the set of words that characterizes documents "interesting" to a given user is not (very) specific to any particular site. Our URL classifier is site specific—what it learns about the URLs of ads on one site will not generalize to other sites. And our table-based classifier is even more specific, as it focuses on the layout of a specific page.

Chapter 7

The Daily You Web Application

One application of this work is to create a real, usable news recommendation service. In this section, we describe the components of such a system, which is currently in public use[1]. This Chapter represents joint work with David Karger.

7.1 Introduction

As indicated in the thesis' introduction, The Daily You is a web application that clips "interesting" news articles for a user. Like a good assistant, it strives to save the user's time by anticipating the user's interests, then scanning the Web and finding new related content.

Two things make this particular domain more difficult than generic recommendation problems; and our system, algorithms, and empirical work revolves around addressing recommendations within those special difficulties.

To recommend news while it is still new(s), the system must keep itself aware of what is new and what is changing—and then to process that information and intelligently select items for the particular user. Classifying news is a time-sensitive problem, unlike many other popular recommendation systems (Amazon[2] or NetFlix[3], for example). Thus, an approach of doing recommendations as a nightly or weekly background process is not practicable for our system.

Most recommendation systems have the advantage of implicit domain knowledge, while The Daily You does not. The *New York Times* knows the subject, author, editor, and date for every story it publishes. The Daily You can not directly access any of this information, so it must use a combination of speedy heuristics and algorithms to pull out content oriented towards the user. It is a challenge for a computer to even do simple human tasks—distinguishing advertisements from menus from content—and harder yet to recognize elements within the content like authorship, title, date, and subject. Most news recommendation research has dealt with known content—which requires hand-written data extraction. Our goal is to build a system in which

[1] `http://falcon.ai.mit.edu`
[2] `http://amazon.com`
[3] `http://netflix.com`

the user doesn't need to put in any extra effort to enjoy recommendations from an arbitrary favorite news site.

Data from real-world applications is always noisy. A particular user might not click on a story because she was simply not present, or had already seen that story through a different news source, was too busy or was simply not in the mood. This means the data-stream has a random component and the data looks extremely inconsistent with itself. For example, one problem that many discriminant classifiers (like the SVM) have with real news data is that data from opposite classes *often* lie near one-another. It is common for two links to go to the exact same destination, but most users will only click on one.

While some news recommendation applications already exist, we provide a different method for understanding user interests that we feel produces better recommendations (i.e. produces more recommendations that matches a user's interests), in a smaller amount of time, than existing applications.

Section 7.2 gives a summary of The Daily You's architecture. The steps between a theoretical algorithm and a working application are not always easy, and this section details some of the trade-offs between response time, content freshness, ease-of-use, customization and computational resources. The section details some design choices in building a news application that works in real time for a set of users. The section also contains several screen-shots of the working application.

One of the major components of The Daily You is to take the large amount of information present on a web-page and to reduce or eliminate the irrelevant portions while highlighting the relevant portions. The ad-blocking portion tries to remove elements that distract your attention (Section 6.2). The page-filter further removes templated information that the user has probably already seen (Section 7.2.2) and the recommendation system (Section 7.2.3), which applies the previous chapter's machine learning contributions. Experimental results for our recommendation system can be found in Section 6.3, where we show results of a variety of algorithms on a two-hundred person user study done to compare various learning algorithms on real user-generated data.

7.2 Architecture

In this section, we detail the architecture of The Daily You. The system needs to be simultaneously fast, functional, and easy to use; this requires prioritizing certain operations along with making tradeoffs between different resources like memory and processor time. Figure 7-1 shows many of the system's components, how they interact, and their priorities relative to one another. The following components are detailed below: the web proxy, the page filtering system, the recommendation system, and finally the output options. Figure 7-2 shows a visual example of how the system de-clutters an original CNN page and inserts recommendations.

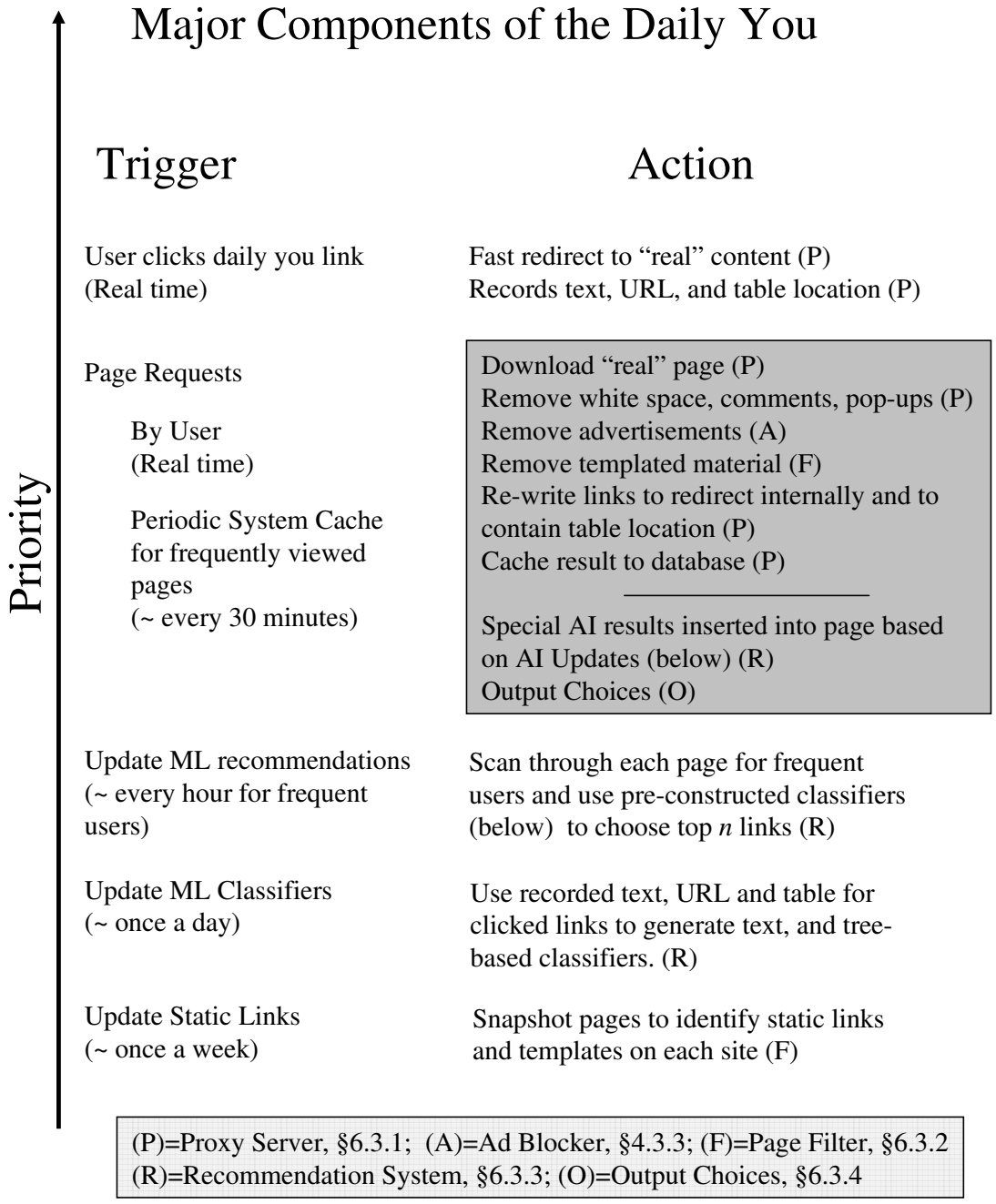


Figure 7-1: Different elements of The Daily You receive different priorities according to the utility they provide a user.

Page Requests of the Daily You

Remove the visual clutter, focus on the recommendations

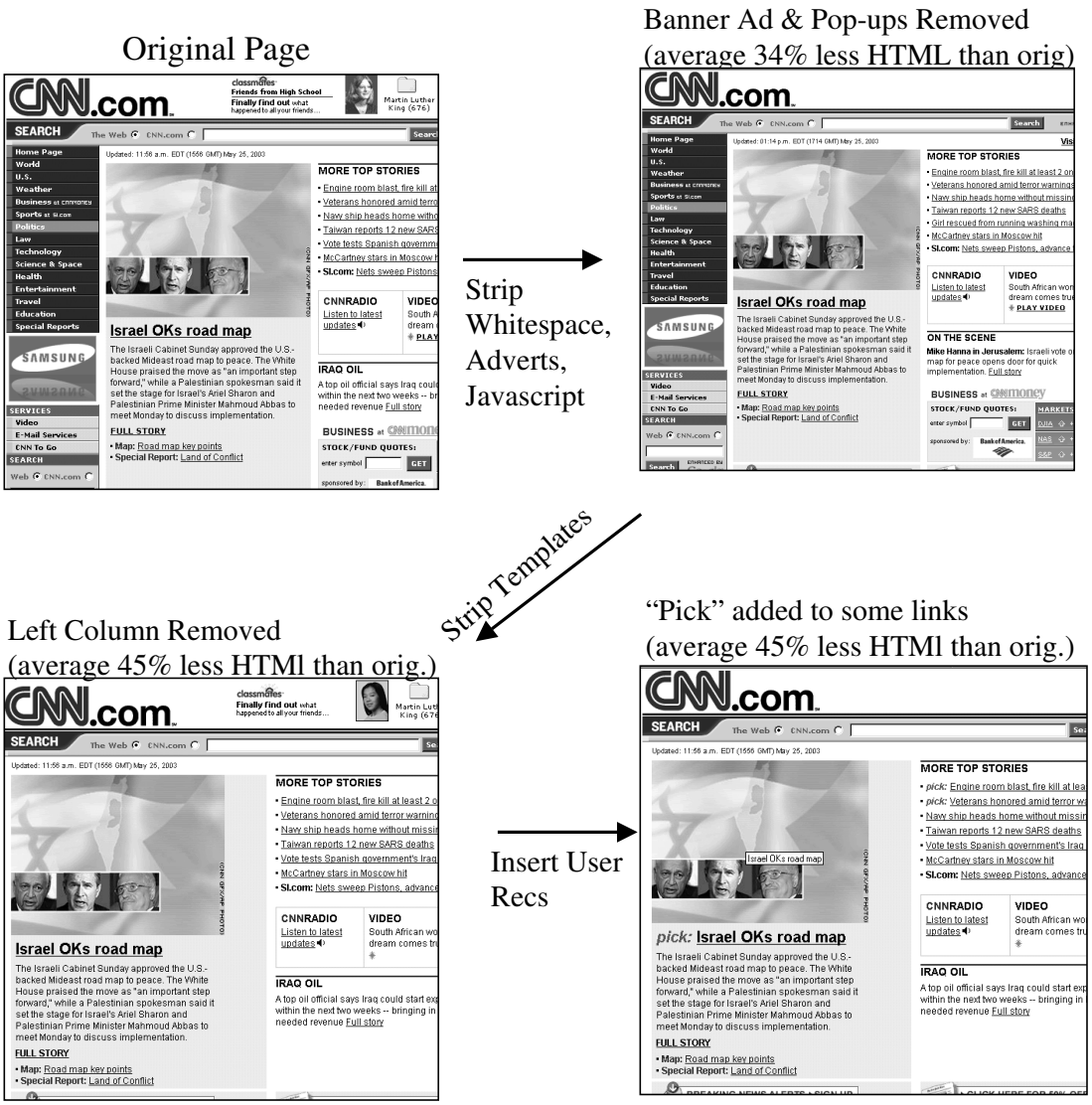


Figure 7-2: The CNN home-page after being altered by the various components of The Daily You

7.2.1 Web Proxy Server

The proxy server takes care of several important, but relatively mundane, tasks. Its chief goal is to provide necessary infrastructure for the higher-level components while being as unobtrusive to the user as possible.

When a user views a page through a proxy server, it should appear and function just like the original page. At the same time, the proxy server is silently collecting information about the user's clicks.

The Daily You's proxy server is custom-built and differs from the most common proxies in several ways. First, The Daily You records far more information about the clicked-on links than a normal proxy, which only records the URL. In addition to the URL, our proxy records the words in the anchor text of the link, and which table-location the link was found in. This is done through a step which re-writes each link to secretly contain this additional table and text information.

Second, The Daily You does not require any browser setting changes to record the user's actions. Most widely-distributed proxies, like Squid [4] require the user to make some changes to the browser settings. Instead, our proxy manipulates the page in such a way that it looks like the original page but is actually being hosted on The Daily You servers. For example, each link actually points back to The Daily You, which tracks the click and quickly redirects to the original destination. This code is similar to the way an advertising link-tracker works, and involves some subtle detailed issues to catch different ways of formatting links. Many of the ideas in this code come from CGI Proxy [5].

Third, because the server is custom-made, it can also work efficiently within our needs. For example, not all information goes through the proxy: images and style panes are sent directly from the server. This speeds up image downloads and saves the server processing time.

Forth, the server downloads HTML with lynx (an open-source HTML browser), modifies the incoming HTML in minor ways to make the HTML smaller. For example, it removes extraneous white-space and comments from the HTML, which shortens the length of each file, on average 19%, according to our study of several sites (see Table 7.1). It also removes javascript commands which cause pop-up advertisements and flash-based images. This shortening of the HTML has a noticeable effect on performance. Since all the subsequent steps require a scan through the HTML file (whether to find links that are ads, or to find links to recommend), a shorter file can be parsed more quickly. This in turn means the page is delivered to the user more quickly and requires less memory and processor time of the server.

Like other proxy servers, The Daily You caches frequently requested pages. Approximately every thirty minutes, popular pages are downloaded, parsed, and saved to a mysql database for faster access times when a user requests those pages.

[4] `http://devel.squid-cache.org`
[5] `http://www.jmarshall.com/tools/cgiproxy`

7.2.2 Page Filtering System

This section describes The Daily You's efforts to automatically crop unwanted visual material from a web site. This automatic, real-time cropping produces a page with much less visual clutter (Figure 7-3). Like the ad blocker, this component was used to remove distracting and unwanted content automatically from the web pages. In the context of the recommendation system, it is used to quickly filter out a large number of links.

We began with a small user study of six users, who were asked to "crop" their favorite web pages to show which portions of the pages they would like to keep and remove. Users cropped the pages so they could see more content and less clutter. Cropping was through a user interface that allowed the removal of entire tables or table sub-elements.

The results, which were fairly consistent across both various sites and users, were that people removed the banners across the top, the navigation bars across either the sides or the top, and the copyright and corporate information across the bottom. While users would sometimes remove additional table elements, like specific sections of the page they were uninterested in, the general trend was to remove the static elements. In other words, users tended to remove recurring and unchanging content—HTML templates. These templates allow the web site to have a consistent design and layout, leaving the web master to focus on the more dynamic and changing portion of the web site.

Clearly the web site designers put some effort into incorporating templated information; why would users consistently want to remove it? One answer, partially articulated in the Montage paper [1], is due to the differing needs of new and repeat visitors. New users might use the banner and the navigation bars to get a sense of the overall contents of the site. If they had specific interests, they might scan the navigation bar to see if such a section existed. If so, they might jump over to that section. In contrast, repeat visitors would already know the rough organization of the site; if their entire interest was focused on one section, that section (as opposed to the front page) would be their favorite site.

Our original intention was to learn, in a method similar to the ad-blocking section, which table elements in a web site were unwanted. This proved difficult for a few reasons. First, it is a user interface challenge to figure out what portion of the page the user has chosen to look at (in contrast, for link recommendation, it is easy to figure out what the user has clicked on). Second there are issues, though infrequent, with table structures changing either by site redesign, or by the sudden addition of advertisements placed in advertisements (this is a problem for the table-based recommendation system as well). Third, one would want to strip off table elements conservatively, since removing wanted information is worse than keeping unwanted information. We decided to use a heuristic to avoid these problems.

The page filter tries to remove the static portions of the page, echoing the results from our user study. We recursively chop the page up into table elements (roughly, the horizontal and vertical rectangular areas that make up most web-pages). Each element is assigned a heuristic score that measures how static it is. Elements above a

Figure 7-3: A portion of the *New York Times* is shown (left) normally and (right) after ad-blocking and page filtering. The ad-blocking portion removes the images in the top right and left, and the page filtering removes the entire left-side static navigation pane. Our user study shows that most users prefer the less cluttered versions of web pages, like those provided by The Daily You.

threshold are removed. The remaining elements are glued back together to create a stripped down page. The visual effect is that certain rectangular blocks are "whited out" from the original page. This preserves the dynamic content on the page, and retains a similar look-and-feel from the original. Figure 7-3 shows a typical example of how the page filter de-clutters a page. The end result is that a heuristic does a fairly good job of "filtering" the unwanted table elements in a page. That process is good enough for our purposes.

Our heuristic simply looks at the number of static links (defined in the next paragraph) in each table element. If there are three or more static links, and no dynamic links, then the table element is removed. The heuristic generally tries to cut out the worst offenders (generally long lists of static index links) and leave in some borderline cases (banners, search bars, and so forth).

We find static links by running a weekly process. Once a week, an automatic process runs that reads all the web pages that The Daily You monitors (currently, 240 pages are monitored, which consist of pages that our 28 registered users have entered into the system). The links on each page are recorded. If the same URLs are seen two weeks later, they are deemed static.

At this point, the components have automatically converted arbitrary source pages into a less cluttered version of the same page. From our user study, most users prefer the stripped down look. While it is difficult to measure clutter and aesthetic appeal, we make some quantitative comparisons that illustrate how much the original pages are being reduced.

Table 7.1 uses some simple metrics to see what is being removed after each component is completed. The columns represent, in order, the original HTML, then the cumulative effect of the various components (the web proxy, which simply removes whitespace; the ad blocker; and the page filter).

The first row compares the total megabytes of HTML incorporated in the 240

Table 7.1: Effects of various components on HTML size, image size, and link count

	Orig.	Web Proxy	Ad Block	Page Filter
HTML (size)	8.6M	7.0M	5.7M	4.7M
HTML (% dec.)	0%	19%	34%	45%
Images (size)	800K	800K	617K	539K
Images (% dec.)	0%	0%	23%	33%
# of links	20K	20K	18K	14K
links (% dec.)	0%	0%	10%	30%

pages being monitored by The Daily You. The second row shows the percentage decrease in the HTML. Besides the purely aesthetic advantages, a reduction in HTML corresponds to a reduction in download latency. Modem users, for example, would find pages downloading almost twice as quickly by using The Daily You. The simple act of removing whitespace, part of the web proxy's functionality, decreases the HTML size significantly. The ad blocker removes about 1.3M of HTML over the 240 pages, which implies that a high proportion of HTML on a page is actually devoted to advertisements (just the href tags). About 1M of HTML is removed by the page filter which implies that a lot of information sits in static templates over the assortment of web sites.

The third and fourth rows shows the aggregate size of images that are downloaded through the Internet Explorer browser. Like the HTML, reducing the numbers of downloaded images will also decrease download times. Roughly, the size of an image correlates with how large and distracting it is. We only used 20 random web sites from the original 240 for this calculation. This is simply because it is inconvenient to total the size of images from 240 web sites. The web proxy, which duplicates the look of the original site, does not remove any images. The ad blocker removes 23% of the images—which reflects another decrease in download times. Interestingly, the page filter also removes images, which turn out to be small navigation pictures, like arrows or graphical section headings. The bulk of the remaining size comes from hi-resolution photos of news events.

The final two rows measure the number of links found on the pages. This metric shows how the first three components can help out the link recommendation system (the next sub-section), by automatically removing candidate links. Comparing the ad block numbers, blocking 10% of the links corresponds to reducing the image size by 23%. In other words, there are a relatively small number of advertisements, but each one takes a disproportionate amount of attention.

One interesting extension of our finding, that web pages have a large amount of templated HTML, is in the area of HTML compression. The idea would be to split (as we have) a document's information into static and dynamic components. The main page would retain all the static information, except the page would be marked as not expiring. Tables full of dynamic information would be replaced by iframes (floating, border-less, positionable frames that are now standard HTML), whose contents would expire immediately. The iframes would update as needed, but the surrounding static HTML would only be downloaded once. In this way, the page would have the same

content as the original, but refreshes would only need to send the dynamic content.

7.2.3 Recommendation System

The recommendation system takes a list of past click data from the web proxy, and uses it to predict the user's future clicks. This system implements our idea that if a user has clicked on a link, they are more likely to click on links with similar URLs or in nearby table elements. For this paper, our focus is on single-user learning, as opposed to collaborative filtering [51].

Our recommender starts by equally weighting the URL and table learners, which produces a ranked list of probabilities that a link will be clicked upon. In practice, portions of this list may result in ties (for example, when the user prefers college football stories, which all share a common URL "father" and reside in the same place in the table tree). Therefore, we use the tree features to compile a top ten list of candidate links, and use a Naive Bayes classifier [36] on the text of the link to produce a top three list. This sometimes has the advantage of fine tuning the coarser results from the tree classifiers; for example, it might choose the link which contained your alma mater from a list of college-sports related news.

As already discussed, the classifier is fairly fast because it does not need to download all the links on a page. All of the information is contained on the source page, including the table structure, link URLs and the text inside the link.

There are certainly cases when these human-oriented features are either obfuscated (the URL is deliberately misleading) or uninformative (the page always displays the newest stories first, but you only care about the sports articles). Machine learning already addresses this issue: if there is little or no correlation between a URL and a user's click stream, our system simply ignores the URL feature. In other words, we let the learning sort through what is and is not relevant.

Another problem in most recommendation systems is that they do not receive enough training data to make useful recommendations. A portion of this is due to user-interfaces that require the user to expend extra effort in comparison to what they normally would do. For instance, movie recommendation systems expect you to log on and rate all the movies you have seen. News recommendation systems expect you to rate each news article after you have viewed it. Newsdude [4] reads articles verbally to users, who hit a button when they are tired of hearing the story.

In contrast, The Daily You's interface is simply that of a normal browsing session (with out the advertisements and clutter). If anything, the browsing experience is more pleasant than normal. The system invisibly tracks clicks, like most proxies[6], and stores the information. Thus, because the system is providing utility to the user, we expect to receive a larger amount of data than in other systems. More data almost always leads to a higher quality recommendations.

[6] i.e. `http://devel.squid-cache.org`

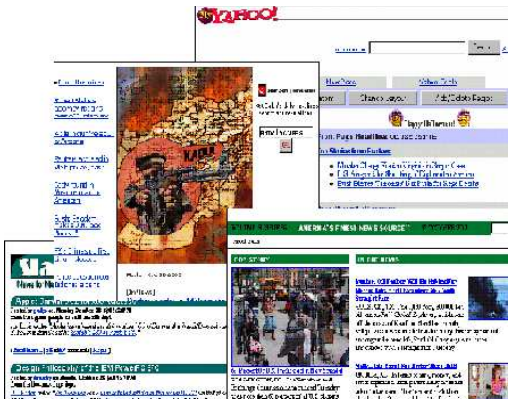


Figure 7-4: The Daily You's emphasis on flexibility allows one to grab any page, including popular sites like Salon, Slashdot and the Onion. It can even function as a meta-agent, grabbing information from other portal sites like My Yahoo.

7.2.4 Output Choices

Various combinations of the components can be used with simple front ends in order to produce a variety of different useful applications. All of these applications can be used through a normal browser interface.

The first such application is a web-based ad blocker. As the user navigates through the web, all the pages are retrieved through the web proxy. The web proxy re-writes all the links so they point to the web proxy version of the original link. Essentially, every page is now fetched through the web proxy, rather than directly from the source. This application was inspired by an excellent web application called CGI proxy [7].

With a little back-end programming, The Daily You can also associate a user name with a set of commonly viewed pages. Each chosen page can be fed through the web proxy, ad-blocker, and page filter. This results in a set of pages looking like the New York Times example in Figure 7-3. Next, all the pages are concatenated into one long, vertical page with navigation buttons to move jump between the chosen pages. This lets users have arbitrary often-viewed content sitting on one handy page, much like the Montage system [1]. This saves the user time both in terms of download time (Table 7.1) and in terms of browsing (a user's favorite sites are aggregated on one page). For the end-user, "installation" of this product simply involves picking a user name and typing in a list of commonly-viewed URLs.

In effect, by aggregating arbitrary user-selected sources pages together, The Daily You serves as a true "portal" to the rest of the Internet. In contrast, most common commercial portals only allow users to choose from a small number of content sites. Figure 7-4 shows some popular pages aggregated together and used by Daily You users; these pages are not typically available on commercial portal sites.

The Daily You can also take the user pages above and run the recommendation component. Figure 7-5 shows a summary page of one of the author's Daily You pages.

[7] `http://www.jmarshall.com/tools/cgiproxy/`

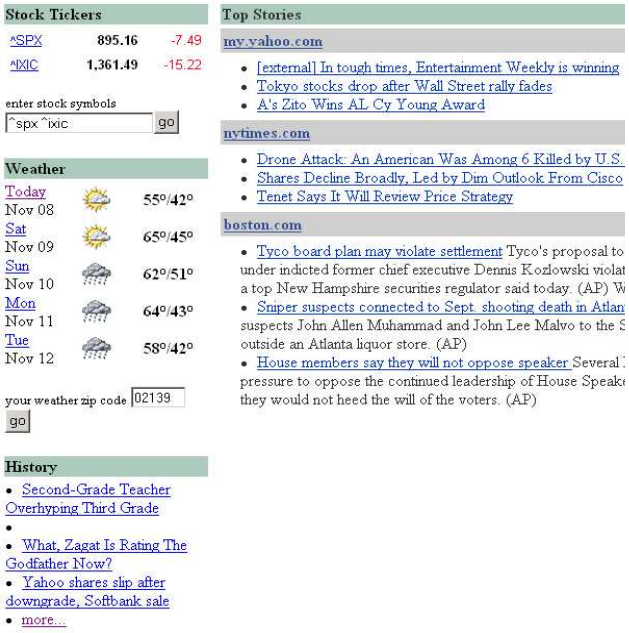


Figure 7-5: After inputting some favorite pages, the components of The Daily You interact to provide this summary page of highly recommended stories (right) and stocks, weather and a searchable history of viewed links (left).

Figure 7-6: The Daily You can also stream a news ticker of recommended news stories, for a low footprint summary of news relevant to you.

Its layout is similar to the layout in My Yahoo[8], though its functionality is somewhat greater. The left hand side of the page includes standard portal functionality, like customizable stock quotes and weather. The right hand side, like My Yahoo, contains a list of top stories from user-selected sites. However, My Yahoo has no learning—the top headlines from a given site are the same for all its millions of users. The Daily You's headlines are selected with the recommendation system. More interesting is that any site can be selected as a source for recommendations. One could add the Google news summary service, or Amazon.com, or even another Internet portal.

Another output choice of The Daily You is a news ticker (Figure 7-6). The news ticker simply takes your recommendations and scrolls them horizontally. Clicking on an item pops up the story in your browser. The news ticker is useful because it takes a small footprint at the top of the screen, but over time displays the same amount of information as the other views. The news ticker was built on Java code written by Gokhan Dagli [9].

[8] `http://my.yahoo.com`

[9] `http://javaboutique.internet.com/HMenus2`

Chapter 8

Conclusions

In this thesis, we presented several machine learning techniques which were adapted or created for use with Web applications. Our work might be summarized as "a little domain knowledge goes a long way." We used domain knowledge in various ways to speed up one algorithm, increase the accuracy of another algorithm, and to create new algorithms. All of these efforts yielded scalable algorithms that also had high accuracy and were practical for Web use.

We first used domain knowledge to improve the "punching bag of classifier's" accuracy. By carefully analyzing some of the systematic problems naive Bayes has when dealing with real textual data, we were able to produce a fast, easy-to-implement classifier with accuracy similar to the slower, but highly accurate SVM.

The SVM has the opposite problem from naive Bayes; it is accurate but does not scale well for even moderate-sized Web problems. For the SVM, we found that we could hybridize the SVM with the fast Rocchio algorithm. This produced a range of classifiers that ranged from fast (the "Rocchio" end) to accurate (the "SVM" end). Empirically, we found there were good intermediate stages that shared the speed of Rocchio while approximating the accuracy of the SVM.

In our examples of naive Bayes and the SVM, we were focusing on adjusting existing (text) algorithms for use with Web applications. However, as we observed in the Chinese newspaper example, humans can intuit a variety of information simply by looking at the visual organization of a page. This motivated a set of features, like the layout of a page, that we call tree-structured features.

These tree-structured features motivated a wide variety of classification and algorithmic challenges. We constructed a probabilistic model of trees that formalized our intuition that similar items are often near one another in a tree. In keeping with our goal of building algorithms that worked well with Web applications, we built algorithms that worked with this model that were scalable and dealt with precision problems that often occur in probabilistic models.

When we applied our algorithms to the real-world problems of ad-blocking and news recommendations, we found that they worked better and faster than standard algorithms (naive Bayes and the SVM) and standard features (anchor text and target page text). We then used these algorithms to build our news recommendation system, The Daily You.

Looking forward, we would like to further explore making practical, accurate algorithms for use with th Web. For example, we are interested in combining the generality of text algorithms with the speed and specificity of the tree algorithms. Interleaving those two algorithms could potentially lead to an algorithm that has the advantages of both approaches. This would lead to a better recommendation system, and a better application. Another interest is in recursive bundling, or the idea that larger bundles might guide the SVM towards the correct solution for smaller bundles.

Appendix A

In Chapters 4 and 3 we list several text experiments that we performed. This section describes the pre-processing and experimental setup that is common to both of those chapters.

A.1 Text Data Sets

We discuss four common text data sets that were used, as well as the pre-processing steps that converted those text data sets into the vector representation that is commonly used with the support vector machine.

The 20 Newsgroups data set is a collection of Usenet posts, organized by newsgroup category, which was first collected as a text corpus by Lang [34][1]. It contains 19,997 documents evenly distributed across 20 classes. We remove all headers and UU-encoded blocks, and skip stop-list words and words that occur only once[2]. Documents that are empty after pre-processing are removed.

In our experiments, we selected 600 documents randomly from each class to serve as training examples (times twenty classes yields 12,000 training documents). We repeated this random test/train split 10 times per result shown. This methodology is consistent with the pre-processing and testing procedures found in Rennie and Rifkin [44].

The Industry Sector data is a collection of corporate web pages organized into categories based on what a company produces or does[3]. There are 9649 documents and 105 categories. The largest category has 102 documents, the smallest has 27. We remove headers, and skip stop-list words and words that occur only once[4]. The vocabulary size is 55,197. Documents that are empty after pre-processing are removed.

In our experiments, we took 50% of the examples from each class as training examples, and the remaining 50% for test examples (for classes with an odd number of examples, the extra one went to the test set). This method preserves the ratio of documents within each class. Like our processing on the 20 Newsgroups data, we

[1]The 20 Newsgroups data set can be obtained from http://www.ai.mit.edu/~jrennie/20Newsgroups/.

[2]Our 20 Newsgroups pre-processing corresponds to rainbow options "–istext-avoid-uuencode –skip-header -O 2."

[3]We obtained the Industry Sector data set from http://www-2.cs.cmu.edu/~TextLearning/datasets.html.

[4]Our Industry Sector pre-processing corresponds to rainbow options "–skip-header -O 2."

also created 10 test/train splits. This method of pre-processing is consistent with the steps taken in Rennie and Rifkin [44].

Since 20 Newsgroups and Industry Sector are multi-class, single-label problems, we constructed a one-vs-all classifier for each category. This means 20 Newsgroups had 20 classifiers per test/train split and Industry Sector had 105 per test/train split. To assign a label to a document, we selected the most confident classifier.

The Reuters-21578 is a collection of Reuters newswire stories that is commonly used in text classification experiments[5]. We use the Mod-Apte split, which splits the data chronologically so that all the training documents were written before any of the testing documents. There are 90 categories with at least one document in both the training and the test set. After eliminating documents not labeled with at least one of these categories, we are left with 7770 training documents and 3019 test documents. After eliminating words from a standard stop list and words [6] that only appear once, we have a vocabulary size of 18,624. Reuters poses a multi-label problem. We construct a one-vs-all classifier for each category. A document is assigned a label for each classifier that produces a positive value.

Having a clear test-train split, like with Mod-Apte, has advantages in replicability. However, having exactly one way to run the experiment also means that we do not report statistical significance results, since standard deviation does not make sense with only one point.

Ohsumed is the largest data set that we used, containing approximately 230,000 documents and almost 267,000 features [22]. Ohsumed consists of MEDLINE documents from 1987 through 1991 which are published and assigned an identifier by humans. The data sets are available online[7]. The documents were downloaded, unzipped, but stopwords were not removed. The Ohsumed data was split by date; the first 179,215 documents were used for training and the last 49,145 were used for testing. Like the Reuters collection, each document may belong to multiple categories. For our experiments, we used the ten largest categories, which are "Human", "Male", "Female", "Adult", "Middle Age", "Support, Non-U.S. Goverment", "Animal", "Aged", "Case Report", and "Comparative Study." The experiments consisted of 10 separate binary experiments corresponding to each of those categories. For each of those experiments, the training data was labeled according to whether it was or was not in a category (i.e. was or was not labeled as "Human").

For 20 Newsgroups and Industry Sector, we use multi-class classification accuracy to compare different algorithms. This is simply the number of test documents in the correct class divided by the total number of test documents. For Reuters, we use precision-recall breakeven. To compute breakeven, for each class, we trade-off between precision and recall until we find their scores to be equal. For macro breakeven, we average these scores. For micro breakeven, we perform a weighted average, where the weight for a class is the number of testing examples in that class. For Ohsumed, the reported score was the average accuracy over the ten binary experiments performed.

[5] `http://daviddlewis.com/resources/testcollections/reuters21578`

[6] This corresponds to pre-processing with Rainbow using its standard stop list

[7] `ftp.ics.uci.edu/pub/machine-learning-databases/ohsumed`

A.2 SVM parameters

For our SVM-based experiments, we used SvmFu, which is known as a fast implementation of the SVM [46]. We used 3000 cache rows ("-c 3000") which has an effect on speed, but not accuracy. We set the kernel and input datatypes to floats ("-K float -D float"). We use the linear kernel for the SVM since the linear kernel empirically performs as well as non-linear kernels in text classification [59].

A.3 Statistical Significance

Having 10 test/train splits for each of the Industry Sector and 20 Newsgroups data sets lets us report statistical significance for each reported result. Each split yields a different accuracy result, and we report the means of those results along with the standard deviation. The mean (μ) is calculated as follows where a_i is the accuracy from test run i of n test runs ($n = 10$ in our 20 Newsgroups and Industry Sector results):

$$\mu = \frac{\sum_{i=1}^{n} a_i}{n}$$

The standard deviation (σ) is calculated as follows, where a_i is the accuracy from test run i of 10 test runs:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (a_i - \mu)^2}{n}}$$

We show statistical significance results for Ohsumed and the recommendation data sets in a different way. In Ohsumed and the recommendation data sets, each "run" is drawn from a different distribution than the Industry Sector and 20 Newsgroups sets.

By analogy, if one were comparing two cars on ten different race tracks of varying lengths, the means for each race would look very different, and the standard deviations would be large. Instead, you might notice whether one car consistently finished faster than the other, and built a statistical significance test based on comparing which car was faster in each race.

Statistics has a standard method for such comparisons [23]. For each test, one looks at the difference in accuracy between the two classifiers, then orders the differences in score by rank. Suppose we are trying to show that classifier 1 is better than classifier 2 over n distinct tests. Let $|Z_i|$ be the absolute value of the difference in accuracy of classifier 1 and classifier 2; so a high $|Z_i|$ means there is a large difference between the two classifiers. Order the $|Z_i|$ such that the smallest differences come first and the largest differences come last. Then $|R_i|$ runs from $1 \ldots n$ and represents the ordering of the differences between the two classifiers. If $|R_2 = 1|$ that means the

absolute differences between the two classifiers on trial 2 was the least of any of the trials.

Define a function ϕ_i that indicates whether or not Z_i is positive (i.e. confirms the hypothesis that classifier 1 is better than classifier 2):

$$\phi_i = \begin{cases} 1 & \text{if } Z_i > 0; \\ 0 & \text{if } Z_i < 0 \end{cases}$$

Then sum the product of R_i and ϕ_i over all the test examples:

$$T^+ = \sum_{i=1}^{n} R_i \phi_i.$$

To check for statistical significance, we compare T^+ with numbers in a table A.4 in Hollander and Wolfe [23]. When $n = 10$ (Ohsumed experiments), statistical significance above the 95% mark is reached when $T^+ > 44$.

For large n, as in our recommendation experiments (which had 176 samples), there were no table entries. Thus we adopted the methodology of Hollander and Wolfe [23] for larger n values.

First, define T^*:

$$T^* = \frac{T^+ - n(n-1)/4}{(n(n+1)(2n+1)/24)^{1/2}}$$

Then, from Hollander and Wolfe table A.1 [23], $T^* > .165$ implies statistical significance above the 95% mark.

Bibliography

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

[2] Regina Barzilay, Noemie Elhadad, and Kathleen R. McKeown. Inferring strategies for sentence ordering in multidocument news summarization. *Journal of Artificial Intelligence Research*, 17:35–55, 2002.

[3] Adam Berger. Error-correcting output coding for text classification. In *Proceedings of IJCAI-99 Workshop on Machine Learning for Information Filtering*, Stockholm, Sweeden, 1999.

[4] Daniel Billsus and Michael J. Pazzani. A hybrid user model for news story classification. In *Proceedings of the Seventh International Conference on User Modeling*, pages 99–108. Springer-Verlag New York, Inc., 1999.

[5] Christopher. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[6] Leo Breiman. Bias, variance, and arcing classifiers. Technical Report 460, Statistics Department, University of California, April 1996.

[7] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[8] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[9] Kenneth W. Church and William A. Gale. Poisson mixtures. *Natural Language Engineering*, 1(2):163–190, 1995.

[10] Michael Collins and Nigel Duffy. Convolution kernels for natural language. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Proceedings of Neural Information Processing Systems 14*, Cambridge, MA, 2002. MIT Press.

[11] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines*. Cambridge University Press, 2000.

[12] Pedro Domingos. When and how to subsample: Report on the kdd-2001 panel. *Knowledge Discovery and Data Mining Explorations*, 3(2), 2002.

[13] Pedro Domingos and Michael Pazzani. Beyond independence: conditions for the optimality of the simple Bayesian classifier. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML)*, 1996.

[14] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. Wiley and Sons, Inc., 1973.

[15] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley-Interscience, 2000.

[16] Susan Dumais, John Platt, David Heckerman, and Mehran Sahami. Inductive learning algorithms and representations for text classification. In *Seventh International Conference on Information and Knowledge Management*, 1998.

[17] William DuMouchel, Chris Volinsky, Theodore Johnson, Corinna Cortes, and Daryl Pregibon. Squashing flat files flatter. In *Knowledge Discovery and Data Mining*, pages 6–15, 1999.

[18] Rayid Ghani. Using error-correcting codes for text classification. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

[19] Shantanu Godbole, Sunita Sarawagi, and Soumen Chakrabarti. Scaling multi-class support vector machines using inter-class confusion. In *Proceedings of the Eighth International Conference on Knowledge Discovery and Data Mining*, 2002.

[20] D. Haussler. Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36(2):177–221, September 1988.

[21] David Heckerman. A tutorial on learning with Bayesian networks. Technical Report MSR-TR-95-06, Microsoft Research, March 1995.

[22] W. Hersh, C. Buckley, T. Leone, and D. Hickam. Ohsumed: An interactive retrieval evaluation and new large test collection for research. In *Proceedings of the 17th Annual International ACM Conference on Research and Development in Information Retrieval*, pages 192–201, 1994.

[23] Myles Hollander and Douglas A. Wolfe. *Nonparametric Statistical Methods*. John Wiley and Sons, 1973.

[24] G. Jeh and J. Widom. Scaling personalized web search, 2002.

[25] Thorsten Joachims. A probabilistic analysis of the rocchio algorithm with tfidf for text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning*, 1997.

[26] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *Proceedings of the Tenth European Conference on Machine Learning*, 1998.

[27] Thorsten Joachims. Making large-scale SVM learning practical. In B. Schlkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1999.

[28] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[29] Slava Katz. Distribution of content words and phrases in text and language modelling. *Natural Language Engineering*, 2(1):15–60, 1996.

[30] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[31] Daphne Koller and Mehran Sahami. Hierarchically classifying documents using very few words. In *Proceedings of the 14th International Conference on Machine Learning (ICML-97)*, pages 170–178, 1997.

[32] Nicholas Kushmerick. Learning to remove internet advertisements. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 175–181, Seattle, WA, USA, 1999. ACM Press.

[33] Nickolas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[34] Ken Lang. Newsweeder: Learning to filter netnews. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 331–339, 1995.

[35] Y. LeCun, L. Jackel, L. Bottou, A. Brunot, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, and V. Vapnik. Comparison of learning algorithms for handwritten digit recognition. In *International Conference on Artificial Neural Networks*, 1995.

[36] David D. Lewis. Naive (Bayes) at forty: the independence assumption in information retrieval. In *Proceedings of the Tenth European Conference on Machine Learning*, 1998.

[37] Andrew McCallum and Kamal Nigam. A comparison of event models for naive Bayes text classification. In *Proceedings of the AAAI-98 workshop on Learning for Text Categorization*, 1998.

[38] Andrew K. McCallum, Ronald Rosenfeld, Tom M. Mitchell, and Andrew Y. Ng. Improving text classification by shrinkage in a hierarchy of classes. In Jude W. Shavlik, editor, *Proceedings of ICML-98, 15th International Conference*

on Machine Learning, pages 359–367, Madison, US, 1998. Morgan Kaufmann Publishers, San Francisco, US.

[39] Dunja Mladenic. Feature subset selection in text-learning. In *10th European Conference on Machine Learning (ECML98)*, 1998.

[40] Andrew Y. Ng and Michael I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes. In *Advances in Neural Information Processing Systems 14*, 2002.

[41] Dmitry Pavlov, Darya Chudova, and Padhraic Smyth. Towards scalable support vector machines using squashing. In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining*, pages 295–299. ACM Press, 2000.

[42] Michael J. Pazzani and Daniel Billsus. Learning and revising user profiles: The identification of interesting web sites. *Machine Learning*, 27(3):313–331, 1997.

[43] Jason D. M. Rennie. Creating a web page recommendation system for haystack. Master's thesis, Massachusetts Institute of Technology, 2001.

[44] Jason D. M. Rennie and Ryan Rifkin. Improving multiclass text classification with the Support Vector Machine. Technical Report AIM-2001-026, Massachusetts Insititute of Technology, Artificial Intelligence Laboratory, 2001.

[45] Jason D. M. Rennie, Lawrence Shih, Jaime Teevan, and David R. Karger. Tackling the poor assumptions of the naive Bayes text classifier. In *Proceedings of the Twentieth International Conference on Machine Learning*, 2003.

[46] Ryan Rifkin. Svmfu. http://five-percent-nation.mit.edu/SvmFu/, 2000.

[47] J. Rocchio. Relevance feedback in information retrieval. In G. Salton, editor, *The SMART Retrieval System: Experiments in Automatic Document Processing*, pages 313–323. Prentice-Hall, 1971.

[48] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1994.

[49] Gerald Salton and Chris Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988.

[50] Robert E. Schapire and Yoram Singer. Boostexter: A boosting-based system for text categorization. *Machine Learning*, pages 135–168, 2000.

[51] Upendra Shardanand and Patti Maes. Social information filtering: Algorithms for automating "word of mouth". In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 1, pages 210–217, 1995.

[52] Lawrence Shih, Yu-Han Chang, Jason Rennie, and David Karger. Not too hot, not too cold: The bundled-svm is just right! In *Proceedings of the International Conference on Machine Learning Workshop on Text Learning*, 2002.

[53] Lawrence Shih and David Karger. Learning classes correlated to a hierarchy. Technical Report 2001-013, MIT AI Lab, May 2003.

[54] Lawrence Shih, Jason D. M. Rennie, Yu-Han Chang, and David R. Karger. Text bundling: Statistics-based data reduction. In *Proceedings of the Twentieth International Conference on Machine Learning*, 2003.

[55] Noam Slonim and Naftali Tishby. Agglomerative information bottleneck. In *Neural Information Processing Systems 12*, 1999.

[56] Vladimir Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, 1995.

[57] Vladimir N. Vapnik. *Statistical Learning Theory*. John Wiley & Sons, 1998.

[58] Geoffrey I. Webb and Michael J. Pazzani. Adjusted probability naive Bayesian induction. In *Australian Joint Conference on Artificial Intelligence*, pages 285–295, 1998.

[59] Yiming Yang and Xin Liu. A re-examination of text categorization methods. In *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval*, 1999.

[60] Yiming Yang and J. O. Pedersen. A comparitive study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning*, 1997.

[61] Tong Zhang and Frank J. Oles. Text categorization based on regularized linear classification methods. *Information Retrieval*, 4:5–31, 2001.