CLASSIFICATION OF WEB PAGES IN YIOOP

WITH ACTIVE LEARNING

A Thesis

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Shawn C. Tice

June 2013

The Designated Thesis Committee Approves the Thesis Titled

CLASSIFICATION OF WEB PAGES IN YIOOP
WITH ACTIVE LEARNING

by

Shawn C. Tice

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

June 2013

Dr. Chris Pollett	Department of Computer Science
Dr. Mark Stamp	Department of Computer Science
Dr. Cay Horstmann	Department of Computer Science

ABSTRACT

CLASSIFICATION OF WEB PAGES IN YIOOP WITH ACTIVE LEARNING

by Shawn C. Tice

This thesis project augments the Yioop search engine with a general facility for automatically assigning “class” meta words (e.g. “class:advertising”) to web pages based on the output of a logistic regression text classifier. Users can create multiple classifers using Yioop’s web-based interface, each trained first on a small set of labeled documents drawn from previous crawls, then improved over repeated rounds of active learning.

Acknowledgements

Thanks to my wife Marisa, my advisor Dr. Chris Pollett, my committee members Dr. Mark Stamp and Dr. Cay Horstmann, my superbly supportive employers at iFixit, and to taxpayers everywhere, but especially in California.

Table of Contents

Chapter

Chapter 1

Introduction

The Internet is decentralized by design, and it is arguably this single fact that has shaped its growth into the massive web of servers and clients familiar to its current denizens. The Internet is not—as one might expect—like a library, each web site catalogued and cross-referenced by topic and author, but rather like an inconceivably vast information bazaar. In this bazaar the merchants are only vaguely distributed according to the wares that they have for sale, and the space is so enormous that it would be infeasible for an individual to visit them all. Thus, in the absence of a master index or even a table of contents, one's only options for finding information on the Internet are to type in arbitrary addresses and to follow links between pages in the hope that they lead somewhere useful. Search engines have emerged to provide the missing index, and they go about doing it by mimicking the behavior of a user following links between pages, keeping track of each page's content as they go.

Yioop is a search engine like the popular Google and Yahoo search engines, but designed to be operated on a smaller scale—by an individual or small organization. Like all search engines, Yioop performs three main tasks: *crawling* web pages (downloading new pages linked to from visited ones), *indexing* the downloaded pages (associating with each word a “posting list” of the documents and offsets where it occurs), and *querying* the resulting index (computing the set of documents present in all or some of the posting lists of the queried words). Where commercial search engines like Google can index tens of billions of pages, however, Yioop can only index hundreds of millions. Yioop’s advantage is that it can crawl, index, and query according to the unique requirements of the individual.

This trade-off makes Yioop particularly suited to indexing private intranets or relatively small hand-picked collections of public websites, and providing a customizable interface for querying the indexed documents. For example, a small e-commerce company might use Yioop to index their product pages and provide site visitors with a customized product search. Because the company controls when indexing happens and how it is carried out, they can keep the index up to date as they add and remove products.

When searching for something like a product, a query of a few words will usually suffice to retrieve the relevant documents, but often the sought-after information is not so easily expressed. Consider, for example, a search for documents of a particular media type, such as images or videos. The media type is a property of a document rather than something that would necessarily show up in the document text, and consequently a naive indexing strategy based only on a document’s words would often fail to retain such information, rendering it inaccessible to a query. Yioop holds on to this document-level information by adding *meta words* to the index and associating them with documents via the same mechanism used to index any other word. A meta word is created simply by affixing a document property (such as “image”) to a common prefix (such as “media”), using a colon to delimit the two. Examples of meta words are “media:image”, “filetype:pdf”, and “os:linux.” Without meta words it would be difficult—if not impossible—to write a query that reliably retrieves only those web pages that were served by a machine running the Linux operating system. And yet the server’s operating system is among the easier data to identify and retain when indexing web pages.

Imagine, instead, that you would like to restrict a query to web pages that contain partisan rhetoric, or to web pages that are free of advertising. These are rather complicated but useful document properties—difficult to specify with a query consisting only of words that might occur in relevant documents, but not readily available to the search engine during indexing in the way that properties like file type and operating system are. In fact, a human would probably have difficulty assigning such properties to some particularly ambiguous documents, and if asked to do so for millions of distinct documents, any human would certainly make some mistakes. Of course, the reality is that no human could endure even reading that many documents, so it would be useful for a search engine to annotate documents with such properties as best it could, and make those properties available as meta words to be used in queries.

The primary goal of this project is to augment the Yioop search engine with the capability to be taught to recognize complex document properties, and to appropriately assign those properties to crawled documents using the meta word mechanism. Learning to recognize document properties like “contains partisan rhetoric” is the domain of machine learning, and more specifically, of text classification. Text classification usually requires a large training set—that is, example documents that have already been determined to either have or *not* have a particular property—to achieve reasonable accuracy, and because example documents are often hard to come by, this project’s secondary goal is to leverage Yioop’s inverted index and web interface to simplify the process of finding and labeling them. Furthermore, since it is unlikely that any user would be willing (let alone eager) to search for more than a few hundred examples, this project’s final major goal is to specialize for the case of a small training set by trying to help the user find those documents that—once labeled—would most improve the classifier’s accuracy.

The remainder of this thesis will proceed as follows: Chapter 2 sets the scene, describing relevant prior work and explaining the terminology and ideas used throughout the remaining chapters. Chapter 3 elaborates on the introduction, laying out in precise terms the project requirements, and chapter 4 explains in detail how each requirement was satisfied—describing both the major design decisions and the details of their concrete implementations within the Yioop framework.

Chapter 5 presents the results of several experiments carried out to measure how well the design decisions and implementation worked in practice. Finally, chapter 6 closes with suggestions for future work and some general remarks on the project outcome and the role of machine learning in the field of information retrieval.

Chapter 2

Background

This chapter formally defines the classification task and explains how it relates to the stated goal of learning to recognize complex document properties. It presents several classification algorithms at a high level, with special attention paid to those that are well-suited to text classification in particular. Following this, feature selection is introduced as a method to ease the computational costs of text classification and improve the probability estimates of some classifiers. The discussion turns from there to obtaining labeled examples from a large pool of unlabeled documents when there is a user available to assign labels. Here, the application of active learning techniques provides an opportunity to reduce the work that the user must do to obtain an accurate classifier, thus making progress toward the second and third goals listed in the introduction. Finally, the chapter closes with an overview of the Yioop framework in order to familiarize the reader with its terminology and basic operation. All of this material will inform the development of project requirements in chapter 3, and provide the reader with the necessary background to follow the presentation of the design and implementation in chapter 4.

2.1 Text Classification

Within the field of machine learning, *classification* is the problem of training a learning algorithm on a set of labeled examples belonging to two or more classes so that, when given a new unlabeled instance, the algorithm may assign the correct label. This is referred to as a *supervised learning* problem because the training examples must be labeled—usually by a human. An *unsupervised learning* problem, by contrast, takes unlabeled instances and derives both a set of labels and a procedure for assigning one or more of those labels to each instance. The unsupervised learning equivalent of classification is called *categorization*.

Both classification and categorization have applications to information retrieval, where they can be used to weed out uninteresting documents (e.g., those containing advertising), to label documents so that searches may be restricted to a particular class (e.g., pages containing partisan rhetoric), and to group related documents together (e.g., news stories about the same event). The primary goal of this project is to extend Yioop with the first and second capabilities, both examples of the classification task. Web page classification is a special case of text classification, and most of what applies to the latter applies to the former as well. To simplify the discussion, then, the rest of this section focuses on text classification, with the understanding that the same ideas translate to web page classification with minor modifications.

In text classification the labeled examples are usually documents, where each document, $\mathbf{d}$, is represented as a vector of *terms* (referred to more generally as *features* or *variables*). Taking the offsets of terms within a document into account would result in far too many variables to consider, so most classifiers favor a *bag of words* approach, where only a term's presence in a document is considered—not where it occurs. Sometimes the input is simplified further still, discarding how often the term occurs, and retaining only whether it occurred at all.

Using this approach, a popular form for the input to the classifier to take is a binary matrix $\mathbf{X} \in \{0, 1\}^{m \times n}$, where the rows represent documents and the columns represent terms. A zero in cell $\mathbf{X}_{ij}$ indicates that term t^j does not occur in document $\mathbf{d}^i$, while a one indicates that the term does occur in the document. Note that n is equal to the *vocabulary* size—the number of distinct terms across all documents. Binary features are just one choice, though; term weights are another popular choice. The weight can be the number of times a term occurs in a document, or something more complicated such as a combination of the term frequency and inverse document frequency.

The main challenge that text classification presents is the number of features (distinct terms) that must be considered—anywhere from one to ten thousand for a normal problem. A large number of features gives rise to two practical difficulties. First, it is more likely that training will result in over-fitting to the examples, and second, it takes more work to both train a classifier and use it to classify new instances. The next section provides an explanation of the source of these difficulties by formulating the classification task as an attempt to find a boundary between points in high-dimensional space.

2.1.1 Classification as a Boundary Problem

Each row of the aforementioned input matrix **X** can be interpreted as a vector describing a point in high-dimensional space. If there were only two training points from which to build a classifier—one from each class—any boundary separating the two points would serve as a solution to the classification problem. Given a new point, one need only figure out which side of the boundary the point lies on, and assign it the same label as the training point which lies on that same side. If there were only one variable, then the points would lie along a line, and the boundary separating two classes of points would be another point on that line. If there were two variables, then the points could lie anywhere on the Cartesian plane, and the separating boundary would be limited to a curve (potentially straight) on that plane. Figure 2.1 depicts two datasets, one described by a single variable, and the other by two variables. The black and white points represent instances of two distinct classes, and their positions along the line and within the Cartesian plane encode their features.

In general, the more variables there are, the more likely it is that some hyperplane exists which completely separates the points belonging to two classes. If there are fewer training points than there are variables, then there is *always* a

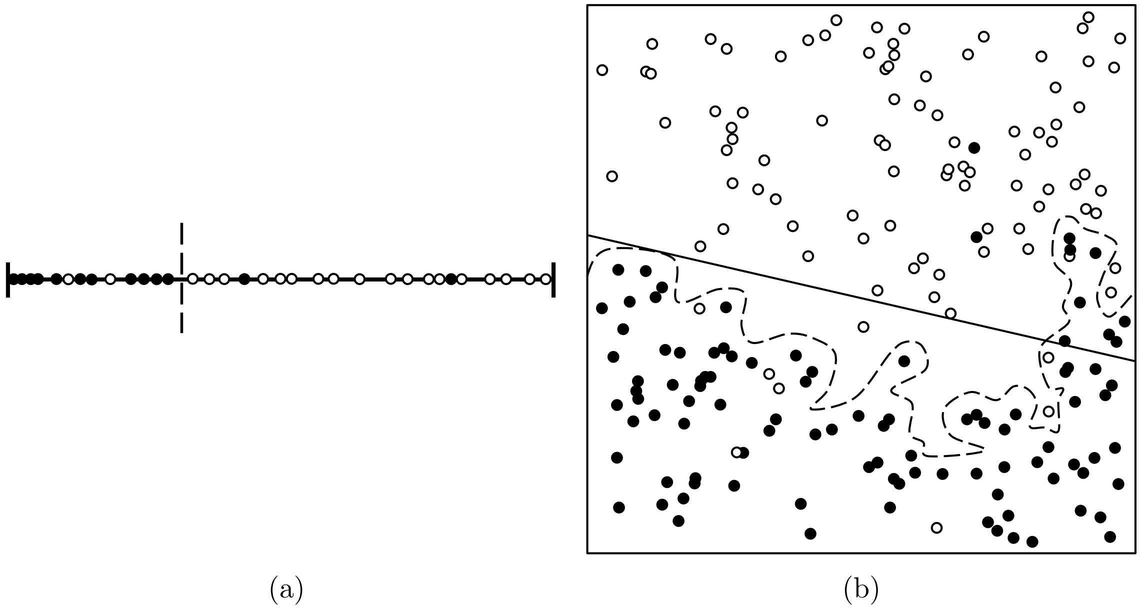

Figure 2.1: Two datasets depicted as sets of points. The color of each point indicates its class, and its position encodes one (a) or two (b) variables that describe it. Potential class boundaries are shown for each dataset; note that none of them completely separates the two classes, and that although the dashed line in (b) makes fewer mistakes than the solid line, it is probably a less meaningful division.

hyperplane that perfectly divides the points (assuming that the same point may not belong to two classes). While this may seem like a boon, it is usually problematic because the training points are not perfectly representative of all points belonging to the two classes. When the dividing boundary is perfectly tailored to the training points, it becomes more likely that a new point will fall on the wrong side because of some minor deviation. Thus to avoid over-fitting when there are more variables, one either needs more training points (making the training data more representative of all instances), or a way to artificially constrain the dividing hyperplane.

Gathering more training examples usually requires a constant amount of work per example (but for some problems there may be no more readily-available examples), while constraining the hyperplane requires special consideration when designing and implementing the classification algorithm.

Even with a sufficient amount of training data or a suitable classification algorithm, problems which contain more variables will require more work to arrive at a solution. Some algorithms scale better than others, but they all must somehow take into account each variable across all training points when searching for a solution, and thus the more variables (and the more training points) that there are, the more work the algorithm must perform. For more complex classification algorithms,[1] finding a near-optimal solution to problems with thousands of variables may be infeasible.

A text classification problem usually has as many variables as there are unique terms in all of the training documents, and for any kind of natural (i.e., human) language problem that number can easily reach into the low tens of thousands. The large number of variables and the statistically complex processes which give rise to natural language text make text classification a particularly challenging problem. Fortunately, because efficient and accurate text classification has so many practical applications it has been the focus of a lot of research, and a number of algorithms have now been either tailored specifically to text classification, or shown to be suitable for the task.

[1] For example, many classification algorithms perform some kind of optimization on the dividing boundary with the goal of minimizing the error of misclassified points in the training data.

The next two subsections discuss, respectively, two well-known text classification algorithms: Naive Bayes and logistic regression. The former is often used as a baseline for a new problem, and the latter has been shown to be effective for text classification [8]. These algorithms together form the core of Yioop's new classification system, where they are used for separate but complementary tasks. Section 2.1.4 gives an overview of several additional classification algorithms that were considered for this project, but ultimately deemed ill-suited for one reason or another.

2.1.2 Naive Bayes

The basic formulation of the Naive Bayes text classification task is to find $\mathrm{P}(c|\mathbf{d}) = \mathrm{P}(c|\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n)$ where c is a class label, and $\mathbf{d}_1 \ldots \mathbf{d}_n$ are the terms present in a particular document $\mathbf{d}$ (recall that transformed term features such as $\mathrm{TF} \cdot \mathrm{IDF}$ may be used as well). By Bayes' theorem, the probability that document $\mathbf{d}$ belongs to class c given the presence of a *single* term t is

$$\mathrm{P}(c|t) = \frac{\mathrm{P}(c)\mathrm{P}(t|c)}{\mathrm{P}(t)}.$$

For purposes of classification, the denominator may safely be ignored because it does not depend on c, leaving just $\mathrm{P}(c)$ and $\mathrm{P}(t|c)$, both of which can be computed from the training data. To find $\mathrm{P}(c|\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n)$, however, one must consider the conditional probabilities between terms since—intuitively—the presence of one term often affects the probability of seeing other related terms. For example, the presence of “Pythagorean” would likely increase the probability of seeing “theorem”.

Unfortunately, the number of possible relationships between terms grows exponentially with the number of terms, making the automatic computation of these conditional probabilities infeasible (at present). An alternative is to use human knowledge to inform the model—and Bayesian networks employ this strategy—but an even simpler and more common approach is to assume that terms are independent of one another. Although clearly false, this assumption suffices in most cases because the true conditional probabilities lack the impact to change the final outcome. The probability of seeing a set of independent terms together is given by the product of their individual probabilities, so that the final probability of document $\mathbf{d}$ belonging to class c is

$$\mathrm{P}(c|\mathbf{d}) = \mathrm{P}(c|\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n) \propto \mathrm{P}(c) \prod_{i=1}^{n} \mathrm{P}(\mathbf{d}_i|c).$$

This quantity is easily computed in a single pass through the training data, but it is actually possible to compute a column vector $\boldsymbol{\beta} \in \mathbb{R}^n$ from the training data such that $\boldsymbol{\beta} \cdot \mathbf{d}'$ gives the log-odds that the vector $\mathbf{d}'$ (representing a new document) belongs to class c [2, p. 347]. Thus the classification task is reduced to a single vector product. In order to derive a classification decision from the classifier, the log-odds are computed and converted to a probability p; if $p \geq .5$ then the document is assigned to class c, and to $\neg c$ otherwise.

Naive Bayes is often used as a baseline classifier because it is simple to implement, efficient to train, and produces a column vector that can be used to quickly classify new instances. Although it may not accurately estimate P($c|\mathbf{d}$), it often comes close enough to make a reasonable *hard classifier*, which determines only whether a document is in c or not in c. A *soft classifier*, by contrast, gives a fairly approximate estimate of how well a document fits into a particular class.

In general, more complex classification algorithms can obtain greater accuracy when trained on the same data as a Naive Bayes classifier, but the Naive Bayes classifier will take less time to train. This property of Naive Bayes classifiers suits them to situations where there is a large amount of data and limited time in which to train. The next section introduces logistic regression, which compliments the strengths of Naive Bayes: it can often achieve greater accuracy than Naive Bayes given the same training set, but requires more work (and thus time) to do so.

2.1.3 Logistic Regression

As with Naive Bayes, the goal of logistic regression is to estimate P($c|\mathbf{d}$), but the method for doing so is different. Logistic regression trains a classifier by explicitly maximizing the likelihood of $h(\boldsymbol{\beta} \cdot \mathbf{d})$ for all documents $\mathbf{d}$ in the training data, where h is called the *hypothesis function.* For logistic regression, h is chosen to be the logistic link function

$$h(z) = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}}.$$

This function has range $[-\infty, \infty]$ and a domain $[0, 1]$, making it a convenient approximation of a probability. In contrast to Naive Bayes, logistic regression training involves optimizing $\boldsymbol{\beta}$ to maximize the likelihood of the training data. Without going into detail about any particular optimization algorithm, the basic idea is to start with some reasonable value for $\boldsymbol{\beta}$ (all zeroes is a common choice), and to find a new value that results in a slightly better likelihood for the training data, then to iterate until $\boldsymbol{\beta}$ cannot be significantly improved. This final stage of the process is called convergence.

Convergence cannot be guaranteed, but this will not be a problem for most classification problems. The main issues that prevent convergence are a poor choice of parameters to a particular optimization routine, highly-correlated variables, or too little training data, all of which can usually be fixed. In the first case, the parameters must simply be tuned to the particular problem, and in the second case, highly-correlated variables can often be identified using standard statistical methods and collapsed into a single variable. The third problem is related to over-fitting, and often results in a perfectly-separable training set—one for which it is possible to perfectly classify every example.

To address this last problem, many practical implementations of logistic regression restrict the values of $\boldsymbol{\beta}$ to be close to zero. This restriction is called regularization, and it effectively combats over-fitting and the issue of too little training data relative to the number of variables, but adds to the model complexity.

In general, logistic regression with regularization is an excellent tool for text classification. It can obtain significantly better accuracy than Naive Bayes classification using the same training and test data, but can still be trained relatively efficiently (depending on the chosen optimization algorithm). Since the final product of training with a logistic regression classifier is the same as that for Naive Bayes—namely a parameter vector $\boldsymbol{\beta}$, the method and time required to classify a new document are essentially the same. Furthermore, the value $p = \boldsymbol{\beta} \cdot \mathbf{d}'$, where $\boldsymbol{\beta}$ is derived using logistic regression, is a much better approximation to the probability that $\mathbf{d}'$ belongs to a class than the same value calculated using a $\boldsymbol{\beta}$ derived using Naive Bayes.

The next section briefly introduces the specific logistic regression implementation used by this project: lasso logistic regression—which is just logistic regression with a particular regularization strategy.

Lasso Logistic Regression

A standard way to implement regularization is to impose a prior Gaussian distribution with mean zero and small variance on each parameter $\boldsymbol{\beta}_j$. Finding the maximum a posteriori (MAP) estimate of $\boldsymbol{\beta}$ is then equivalent to ridge regression [4] with the logistic link function. From an optimization perspective, this approach can be thought of as adding a penalty on the absolute magnitudes of the $\boldsymbol{\beta}$ terms. Making a particular $\boldsymbol{\beta}_j$ larger (or smaller) may increase the likelihood of the data, but the improvement must be balanced against the decreased likelihood of the parameter itself.

Genkin, Lewis, and Madigan propose a similar approach [3], but instead of a Gaussian prior, they suggest a Laplace prior with mean and mode both equal to zero. In contrast to the Gaussian prior, the Laplace prior produces more $\boldsymbol{\beta}_j$ parameters that are exactly equal to zero for the same amount of prior variance (a parameter), yielding stronger regularization and a sparser $\boldsymbol{\beta}$ vector, which combats overfitting when the training set is small, and can provide an alternative to feature selection (discussed in section 2.2). Genkin et al. call this approach lasso logistic regression, after the LASSO technique introduced by Tibshirani for linear regression, which used a least squares estimate subject to a constraint on the sum of the absolute values of the $\boldsymbol{\beta}$ parameters.

To find the MAP estimate of $\boldsymbol{\beta}$ with a Laplace prior Genkin et al. propose a modified form of the CLG convex optimization algorithm [11] developed by Zhang and Oles. CLG is a *cyclic coordinate descent* algorithm, which means that it optimizes the objective function one parameter at a time, holding all other parameters constant. Multiple iterations through all parameters are carried out, with a check for convergence at the end of each iteration. Because the optimal value for each parameter depends on the other parameters, and these are constantly changing, only a single update is carried out on each parameter per iteration. Upon convergence, the dot product of the $\boldsymbol{\beta}$ vector and a new feature vector $\mathbf{x}$ gives a point estimate of the log likelihood that $\mathbf{x}$ is a positive instance of the target class.

2.1.4 Other Classification Algorithms

There are many classification strategies and variations beyond the two presented in the previous sections. This section briefly describes a few alternatives that might have worked in place of Naive Bayes and logistic regression, but which were passed over because they were a bad fit for some aspect of the project.

The first alternative classification algorithm investigated was support vector machines (SVMs) [9, p 256], which share a great deal in common with logistic regression, but take a different theoretical approach to achieve what amounts to regularization. Support vector machines try to find a hyperplane that separates the two classes of training examples while maximizing the margins between the hyperplane and the examples closest to it from both classes. SVMs have been shown to be extremely effective text classifiers [5], but they offer only moderate improvements over good logistic regression implementations despite greatly increased complexity and training time. As discussed further in Chapter 3, training performance was a major concern for this project, and consequently the additional complexity to implement and time to train an SVM was deemed too great a cost for the expected marginal increase in accuracy.

The second alternative considered and discarded was a neural network with hidden nodes [9, p. 246]. Such a neural network usually has one input node for each variable, one or more layers of some number of hidden nodes, and one output node for each possible class (two in the case of text classification with two classes). Each node uses a function to map its inputs to a single output value (the logistic function is a popular choice to accomplish this), then feeds those outputs to each node in the next layer, applying a different weight for each destination node. Training modifies the weights used for each edge connecting two nodes to maximize the likelihood of the training data. Classification is accomplished by feeding the values of a new instance into the input nodes, propagating them through the network, and choosing the class represented by the output node with the largest final value.

Perhaps the most important benefit of neural networks is that they exhibit low bias, and can effectively learn any function given a suitably-large training set. They pay for this benefit, however, with a much more expensive training phase (again, relative to logistic regression), and a less theoretically-motivated parameterization. There are only general guidelines by which to set up the actual topology of a neural network, and it is unclear how many hidden nodes should be used to train a text classifier for an arbitrary problem. In practice, the topology to use for a particular problem is usually determined by experimentation, but in this case it would be unreasonable to expect a user of Yioop to try different network topologies in order to train a classifier for the class of interest.

The third and final approach considered was actually a class of algorithms called ensemble methods. These methods attempt to combine the results of several different classifiers on the same input in order to balance out the strengths and weaknesses of each. There is compelling evidence that ensemble methods generally produce results that are no worse than those of their best constituent classifier, and much better on some inputs [2, p. 376], but the extra accuracy comes at the cost of extra time spent training and running multiple classifiers for each class. Nonetheless, ensemble methods are a very promising avenue for improving classification accuracy; they were passed over only because there was insufficient time to implement and experiment with them.

2.2 Feature Selection

Feature selection is simply the process of choosing those features (i.e., terms in text classification) that best predict the class of an instance, and completely ignoring all other features. It has been shown that the text classification task rarely benefits from reducing the number of features used, but that good feature selection can result in only a minor loss in accuracy [10]. The primary reason to discard features is to drastically reduce both training and classification time by restricting an instance to only a very small subset of its original features. Additionally, using a limited feature set with the Naive Bayes algorithm can make it into a better soft classifier [2, p. 348].

There are a number of statistics that attempt to measure how informative a feature is [10]. This project uses the χ^2 measure, which computes the lack of independence between terms and the classes of documents that they appear in. If a term is highly dependent on some class, then it is considered to be informative, and ranked higher than a term which is not dependent on any class. The least informative term is one which occurs the same number of times in documents of all classes. The χ^2 measure of informativeness for a term, t, and class, c, is defined as

$$\chi^2(t,c) = \frac{N \times (AD - CB)^2}{(A+C) \times (B+D) \times (A+B) \times (C+D)},$$

where A is the number of times that t occurs in documents belonging to class c, B is the number of times t occurs in documents that *do not* belong to class c, C is the number of documents belonging to class c that do not contain t, D is the number of documents that do not contain t *and* do not belong to class c, and N is the total number of documents. The informativeness for a term irrespective of class is defined as the maximum over all classes (an alternative would be to use the average across classes).

Having computed this measure for each term, only sufficiently informative terms are selected for use in training and classification, where the cutoff may be specified either as a fixed number of terms ranked by decreasing informativeness, or as a threshold on the value of the χ^2 measure. Provided that the statistics that the measure relies on are maintained for the data set and do not have to be computed online, the complexity of the algorithm is the product of the number of terms and the number of categories.

2.3 Pool-Based Active Learning

The primary goal of this thesis is to extend Yioop to allow a user to create arbitrary classifiers and use them in a crawl. Because creating a useful classifier requires at least a moderately-sized set of labeled training examples, a necessary step toward achieving that goal is providing some means to acquire labeled examples. This is often one of the more challenging aspects of machine learning in practice, though an adequate set of labeled examples is an assumption in most machine learning research.

The simplest approach is to ask the user to provide all examples as input to the system, but this places the maximum burden upon the user—a reliably bad idea. Instead, the user could collaborate with the classification system to find training examples, an approach called *active learning.* Given a body of unlabeled documents to draw from, the classification system can choose documents that it would like labels for, and present these to the user for labeling. Newly-labeled examples are used to improve the classifier until the user either tires of labeling documents or is happy with the classifier's accuracy. As the former case is more likely, the document selection process should attempt to maximize the benefit derived from each requested label.

As a baseline, consider selecting documents at random. With this scheme, the document selected at step i depends on the document selected at step $i - 1$ only in that it cannot be the same, taking minimal advantage of previously-labeled documents. One way to take better advantage of the existing labeled training set is to select documents that a classifier trained on it has trouble classifying. At a minimum, basing the next selection off of a classifier trained on the current training set implies that each new labeled document affects following selection decisions. The effect is expected to be positive because knowing the true label for difficult-to-classify documents should have the best chance of improving the classifier. In contrast, asking for the true label of a document about which the classifier is already confident is unlikely to yield much new information about the class of interest. The challenge, then, becomes identifying those difficult-to-classify documents.

McCallum and Nigam propose *pool-based* sampling, which stands in contrast to *stream-based* sampling, where documents are considered in order, and once passed over will never be considered for labeling again. Pool-based sampling instead repeatedly scans the entire pool of unlabeled documents, looking for the single "best" one to request a label for. This approach places nearly all of the burden of finding documents to label on the classification system rather than on the user—usually a desirable arrangement.

Both the stream-based and the pool-based approach require a precise definition of how “difficult” a document is to classify in order to choose the “best” one. McCallum and Nigam suggest employing a *Query-by-Committee* (QBC) strategy, which works by classifying the same document with several slightly-different classifiers (the *committee*) and selecting those documents for which there is the most disagreement among committee members. Others have suggested QBC before, and it has been shown to successfully identify documents which, when labeled, improve classification accuracy beyond what would have been achieved with randomly-selected documents.

To measure disagreement between the k committee members $m_1, m_2, \ldots, m_k$, McCallum and Nigam propose *Kullback-Leibler divergence to the mean* [7]. Each committee member m first produces a distinct discrete class distribution $\mathrm{P}_m(C|\mathbf{d})$ for a document $\mathbf{d}$, where C is a random variable over the possible classes (in the simplest case c and $\neg c$). Then KL divergence to the mean measures the certainty of disagreement between the members by computing the mean KL divergence between each distribution and the mean of all of the distributions:

$$\frac{1}{k}\sum_{m=1}^{k} D(\mathrm{P}_m(C|\mathbf{d}) \parallel \mathrm{P}_{avg}(C|\mathbf{d})),$$

where $\mathrm{P}_{avg}(C|\mathbf{d})$ is the mean class distribution amongst all committee members:

$$\mathrm{P}_{avg}(C|\mathbf{d}) = \frac{1}{k}\sum_{m=1}^{k} \mathrm{P}_m(C|\mathbf{d}).$$

KL divergence, $D(\cdot \parallel \cdot)$, is an asymmetric measure of the difference between two distributions, usually interpreted as the extra bits of information that would be required to send messages sampled from the first distribution using a code constructed to be optimal for the second distribution. Given two discrete distributions P and Q, each with n components, the KL divergence between them is defined as

$$D(P \parallel Q) = \sum_{i=1}^{n} P(i) \ln \frac{P(i)}{Q(i)}.$$

Once the KL divergence to the mean has been computed for each document in the pool of unlabeled documents, the document with the largest divergence—indicating the greatest disagreement—is selected to be labeled next. McCallum and Nigam suggest, however, that this procedure tends to prefer documents which might be thought of as outliers from the other documents in the pool. They propose augmenting disagreement with density, which attempts to capture how similar a particular document is to other documents in the pool.

2.3.1 Density-Weighted Pool-Based Sampling

Under this scheme, the best candidate for labeling is the one with the maximum product of density and disagreement—the document which strikes a balance between being difficult to classify and being representative of other documents. The “overlap” between two documents is approximated by the exponentiated negative KL divergence between their respective word distributions:

$$Y(\mathbf{d}^i, \mathbf{d}^h) = e^{-\gamma D(\mathrm{P}(W|\mathbf{d}^h)\|\lambda \mathrm{P}(W|\mathbf{d}^i)+(1-\lambda)\mathrm{P}(W))},$$

where W is a random variable over words in the vocabulary, $\mathrm{P}(W|\mathbf{d})$ is the maximum likelihood estimate of words sampled from document $\mathbf{d}$, $\mathrm{P}(W)$ is the marginal distribution over words, λ is a parameter that specifies how much smoothing to apply to the encoding distribution (to handle the case that a word occurs in $\mathbf{d}^h$ but not in $\mathbf{d}^i$), and γ is a parameter that specifies how "sharp" the measure is. Note that the negative sign in front of the sharpness parameter, γ, converts a larger KL divergence into a smaller exponential, and thus a smaller overlap value.

The density, Z, of a document $\mathbf{d}^i$ is simply the geometric mean of the overlap between $\mathbf{d}^i$ and each other document $\mathbf{d}^h$ in the document pool $\mathcal{D}$:

$$Z(\mathbf{d}^i) = e^{\frac{1}{|\mathcal{D}|}\sum_{\mathbf{d}^h \in \mathcal{D}} \ln Y(\mathbf{d}^i, \mathbf{d}^h)}.$$

Calculating densities in this manner for each document in the pool is much more expensive than computing the per-document disagreement values because every document must be compared to every other document, resulting in a number of computations of the KL divergence between two documents that is quadratic in the pool size. The extra work has, however, been shown to result in an extra five to eight percentage points of accuracy over using just disagreement to select the same number of documents for labeling. The effect is especially dramatic when the number of labeled documents is small, suiting the density-weighted pool-based method to cases where the cost of labeling a document is especially high (e.g., when it is expected that the user will not want to label many documents). This is the expectation for users training novel web page classifiers, and so density-weighted pool-based sampling appears to be a good fit for this project.

2.4 The Yioop Framework

The final section of this chapter briefly describes the major components of the Yioop framework in preparation for the discussion of the project requirements, which were heavily influenced by Yioop's capabilities and limitations. The classification task—especially the collection of training examples—looks quite different when embedded within a web framework and made accessible to a user who is not expected to have any expertise in machine learning or information retrieval. Indeed, in combination with the goal of catering to a novice user, the restriction to a web application environment had the greatest impact on the overall design and implementation of the final classification system. As such, an awareness of Yioop's structure will be integral to understanding the decisions made throughout the rest of the thesis.

Yioop is a web application written in the popular PHP scripting language, with as few external dependencies as possible. Being a web application, it requires an operational web server, and users are expected to connect to this server via a web browser to carry out most administrative tasks. Yioop also has a large collection of libraries and a few executable scripts to perform less interactive tasks, such as crawling and indexing web pages. These computationally-intensive tasks are designed to be distributed across multiple computers, each equipped with a Yioop installation, and coordinated by a single master called the *name server*.

Like any web application, Yioop is best divided at a high level into server-side and client-side components. The following subsections consider each of these collections of components separately, starting with the client side.

2.4.1 Client Side

Yioop performs the vast majority of its work on the server, only using the client (a web browser) to provide a graphical interface for changing settings, managing web crawls, and so on. Yioop builds this graphical interface using standard HTML and CSS, so Yioop's client side consists of primarily-static HTML generated by PHP scripts and completely-static CSS and JavaScript files.

In most cases, the client connects to the server to request a page, and the server responds with HTML, which in turn specifies other resources such as CSS and JavaScript files that the client will need. Once the web browser has fetched these extra resources and rendered the page, the user interacts with it in the browser, filling in text fields, toggling radio buttons, and so on; the web browser does not communicate with the server at all during this time. When done with the task at hand, the user clicks a “submit” button, and the browser sends a form containing the user’s modifications (and perhaps some hidden values passed along in the original HTML) to the server.

The form values, and the URL to which the form is sent, are the primary means the client (and thus the user) has of effecting change on the server. Critically, the server keeps no state between connections from a particular client, so each time the client connects, the server must reestablish—either from scratch, or from some serialized form that had been previously saved to disk—any data structures it needs in order to carry out the task put to it by the client. This limitation of the underlying HTTP protocol has a large impact on the time that it takes to build a classifier, as well as on the overall design of the process.

2.4.2 Server Side

Yioop’s server-side components are considerably more complex than its client-side ones, owing to the rather complex nature of crawling, indexing, and querying web sites; they break down roughly into *controllers*, *models*, *views*, *libraries*, and *executables*. Controllers handle requests made to the server, and may be thought of as collections of related functions, where each function serves as an entry point to the application, similar to the *main* function of a standard C executable. These entry points are called activities, and each activity is responsible for processing and responding to a specific kind of request.

Activities carry out their duties primarily by manipulating models and libraries to change the application state, then passing the new state into a view, which is responsible for generating the HTML sent back to the client. Libraries serve the familiar purpose of grouping together related functions, and models are classes with the specific task of providing an interface to manipulate persistent objects, stored either in a database or in a normal file on disk. Finally, executables are scripts that run independently of the web server, and their primary purpose is to carry out web crawls.

Instances of two separate executables coordinate to perform a crawl: *queue servers* and *fetchers*. Fetchers are—naturally enough—responsible for fetching and processing web pages, while queue servers are responsible for maintaining the list of web pages to be crawled, coordinating the activities of the fetchers, and integrating the fetchers' work into a master index. Usually there will be one queue server per machine involved in a crawl, and several fetchers per queue server. All communication between machines is coordinated by the name server—the same machine that users connect to in order to administer Yioop and make queries. The name server is thus the hub of the entire system and a single point of failure, but because it is usually only serving web pages, responding to requests, and passing messages between queue servers and fetchers, in a multi-machine environment it is typically under relatively light load. Nearly all of the heavy work of connecting to remote servers, processing web page text, building indices, and processing queries is pushed off to the queue servers and fetchers on other machines.

Chapter 3

Requirements

Recall that the primary goal of this thesis is to augment Yioop with the capability to be taught to recognize complex document properties, and to appropriately assign those properties to crawled documents using the meta word mechanism. It should be more than simply *possible* to teach Yioop to recognize new document classes, though. It should be relatively easy for a user familiar with Yioop's administrative interface to do so. Toward that end, users should be able to leverage Yioop's existing indices and its web interface to find examples and use them to train a new classifier. After building a new classifier, it should also be relatively easy to use Yioop's web interface to begin a crawl that uses it.

In order to meet these goals, the classification system must satisfy each of the high-level criteria listed below (these are just general criteria—in each case details will be provided in the sections that follow):

- It must be *effective*. Provided a sufficiently-representative training set, the classifier should classify new documents with reasonable accuracy.
- It must be *efficient*. Classifying documents during a crawl should not significantly reduce the number of documents that may be fetched per hour.

- It must be *responsive*. When the user is building up a new classifier, the delay between any action the user performs and a useful response from the system should be small.

- It must be *usable*. Beyond being responsive, the system should be usable by a novice after reading some brief documentation. A new user familiar with Yioop's crawl process should be able to create and use a new classifier. Additionally, the system should permit recovery from mistakes when labeling documents.

The following sections discuss each of these criteria in detail. It is difficult to derive concrete targets for most of them, but at a minimum some notion of a range of reasonable behavior is provided for each, as well as suggestions for quantitatively measuring on a relative scale how well the system performs on a particular criterion. Thus, for example, while it might not be possible to say how responsive the system is on an absolute scale, it should at least be possible to say how a specific change to the system affects its responsiveness.

3.1 Effectiveness

Classification effectiveness (i.e., accuracy) *is* widely reported for specific data sets, and so it is often possible to place a classifier's effectiveness on an absolute scale. Furthermore, Yioop provides a facility for indexing records stored in custom formats in text files and databases, making it relatively easy to import existing data sets, classify them, and compare the results with those reported in the literature. Doing this with the final classification system should yield accuracy in the neighborhood of the best reported results.

It is unlikely that classification accuracy will climb quite as high as the best results because the system must be implemented in PHP, and due to this and other factors discussed in section 4, must sometimes sacrifice accuracy in exchange for responsiveness and efficiency. Most reported results, on the other hand, come from classifiers implemented in C or C++, where training time is not limited, and the primary goal is to maximize accuracy. Having extra time to train on a large data set often means better accuracy, and this is especially the case when employing optimization methods such as those used by logistic regression. Furthermore, previous research has shown that some machine learning approaches are simply better-suited to text classification, and it may be that the final classification system cannot use the *best* approach due to other constraints.

In any case, for most text classification problems involving a small number of classes it is common to achieve at least 80% accuracy, and so that should be a target for the final system. In general, accuracy should be within a range of five to ten percent of reported results for a particular problem.

3.2 Efficiency

Classification of web pages is a useful addition to Yioop that should not come at the cost of significantly degrading its performance on any of its core tasks. For example, it would be unacceptable for classifying web pages at crawl time to reduce the number of pages crawled per hour by fifty percent. As the cost of classification decreases, however, it becomes harder to judge whether or not it is acceptable. Would a ten percent reduction in throughput be too much? Five percent?

It is difficult to choose any particular number ahead of time as the dividing line between success and failure. The true test is whether an actual job that requires classification may be feasibly carried out using the final system. As a very rough guide, however, adding classification probably should not reduce crawling throughput by more than ten percent. This requirement is easily tested by measuring the time that it takes to index the same set of documents with and without classification. The same test will suffice to determine how changes to the classification system affect efficiency.

3.3 Responsiveness

Like efficiency, responsiveness is hard to quantify, but often "you know it when you see it." Training a classifier—especially on more than a few hundred examples—is an expensive operation, and so it is reasonable to expect a noticeable delay between the time that training starts and the time that it completes. The more interactive process of searching for and labeling training examples, on the other hand, should probably not involve large delays lest the user grow frustrated. Unfortunately, there is an inherent conflict between optimizing work such as picking the *best* document to label next, and responsiveness. A successful classification system must balance the two objectives, reducing the user's work without taking too long to do so.

At the very least, any necessary delay should be communicated to the user, for example by displaying a "loading" message that updates periodically. This strategy only carries one so far, though, and at some point the user will grow frustrated whether informed of a delay or not. Thus there are practical limits on how long any computation may take during the more interactive classifier construction phase, but it is hard to say exactly what those limits are. The standard thirty second execution time limit on PHP processes started by the web server provides a reasonable upper bound, but this is probably already more than most users' patience can bear.

Again, although it is difficult to concretely specify what constitutes "success," one can at least measure a proxy for responsiveness and seek always to improve it to the extent that it can be done without negatively impacting other criteria. One reasonable proxy is simply the time between initiating an action and seeing the desired consequence of that action (e.g., the time between labeling a document and being presented with the next document to label). This value can often be approximated by measuring the time between sending a request to the server and receiving a response.

3.4 Usability

If efficiency and responsiveness are difficult to set expectations for, then doing so for usability is perhaps too much to ask. The only test here is whether or not a new user can read some brief documentation (i.e., a few paragraphs) and then use the system to classify documents of interest to that user. Nonetheless, there are some universal usability guidelines that do not require quantitative validation, and the design of the classification system should take these into account.

First, as mentioned previously, the user should be able to recover from mistakes when labeling documents; that is, the user should be able to change the label assigned to a document, and to remove labeled documents from the training set. Second, when performing heavy computational work, the system should notify the user so that it is clear nothing has gone wrong. Finally, and perhaps most importantly, work that the user has performed (e.g., labeling documents) should be safe from loss under all but the most extreme circumstances.

These are only guidelines, and while it is possible to verify that the classification system follows them, there is no guarantee that doing so will make it usable. Regardless, one must plan for usability to the extent that one's foresight allows; these guidelines are an important part of the project requirements, and should be considered in the design. After all, an efficient, effective classification system has little practical utility if users cannot reliably employ it to classify documents.

Chapter 4

Design & Implementation

This chapter describes the design and implementation of the classification system that was motivated by the goals developed in the introduction, and the requirements outlined in chapter 3. First, the system design is sketched out at a high level, and then each component is described in greater detail. The description of a component is itself further divided into two topics: its design, and its implementation. These two topics were interleaved rather than separated into individual chapters in order to highlight the motivations behind each implementation decision without being repetitive or taxing the reader's memory.

4.1 System Overview

At a high level, the design of the classification system breaks down into four major components, each one largely independent of the others:

- managing the creation and deletion of classifiers,
- building a training set for a classifier,
- training a classifier,
- and using one or more trained classifiers in a crawl.

Before describing each of these components in detail, it will be helpful to first clarify what a classifier in this system actually looks like. Because building a new classifier requires a lot of work, a classifier's representation is independent of any particular crawl, and any other classifier. If a user has built an excellent classifier for identifying advertising material, he or she should be able to use that classifier over the course of several crawls. Furthermore, the user should be able to—at the same time—make use of another classifier that identifies pages containing partisan rhetoric. If, on the other hand, a particular crawl calls for just one classifier, then the user should be able to select it without discarding the work done to create the other. Thus classifiers are first-class entities with an interface for suggesting new documents to label, adding labels to documents, and classifying new documents. In concrete terms, a classifier is an instance of a class with associated persistent data stored on disk.

For simplicity (mostly in the document labeling interface, but the decision impacts the representation of a classifier in general), all classifiers are binary—each one may be trained to recognize only the presence or absence of a single class. Thus each classifier decides whether a given document is a *positive* instance of the class, or a *negative* instance. This is not a trivial limitation, for it provides no means to distinguish between dependent and independent classes. As an example, a user might want to identify different news categories such as entertainment, sports, and finance, where each category is mutually exclusive with the others. With only the capability to create multiple binary classifiers, it is possible that some documents might be classified as belonging to both the entertainment and the sports categories, whereas the user would prefer that those documents be classified only as the more likely of the two. Such a preference could be accommodated relatively easily by providing a way to group classifiers together, but that feature was left for future work.

Managing Classifiers

The presentation of classifiers just given—as independent entities that persist between crawls, and that have the well-defined job of deciding between the presence or absence of a class in a given document—suggests a fairly standard *create*, *read*, *update*, *delete* (CRUD) interface for managing them. When users select the "Classifiers" activity tab in the Yioop administrative interface they are presented with a listing of existing classifiers. Each classifier in the list may be selected for editing or deletion, and a text box is provided to enter the label of a new classifier. The "create" and "delete" operations proceed as one would expect for nearly any persistent entity, but the "edit" operation is much more involved.

Building a Training Set

The page dedicated to editing a classifier provides a mechanism to change the classifier's label and view some of its relevant details, but the bulk of the interface is geared toward helping users find and label example documents in order to improve classification accuracy. This is one of the more complex aspects of the classification system, and it will be discussed in detail further on, but from the user's perspective it amounts to selecting an existing index (from a previous crawl) to draw documents from, and marking each in a series of documents as either "in the class" or "not in the class." In addition to specifying an index, the user may also provide a query that documents must match in order to be considered for labeling.

Each time the user labels a document, the classifier is re-trained and the pool of unlabeled documents is re-evaluated to identify the next best candidate for labeling; this candidate is then presented to the user. Instead of choosing a label, the user may skip a document, in which case the classifier is not re-trained, and the next best candidate is selected. In either case, the record representing the old document is updated to reflect the decision, and shifted down to make room for a new record. The old record remains visible and active, however, so that the user may either change the label or retroactively skip the document, thus allowing recovery from mistakes, as discussed in section 3.4.

Training a Classifier

Each time a fixed number of new documents are labeled, the classifier checks its own accuracy by splitting its labeled examples into a larger training set and a smaller test set, training on the training set, then checking its accuracy on the test set; this is done several times—each time with a different split of the data—and the average accuracy obtained is presented to the user. When satisfied that the classifier is accurate enough, the user can *finalize* it, which results in one last round of training on the full set of labeled examples. This step may take a while, but once it has completed the user is free to use the trained classifier in a crawl.

Using Classifiers

Over time, a user might build a number of accurate classifiers, any subset of which are relevant to a particular crawl. Because classifiers do not interact with one another, including an extra classifier in a crawl does not impact the performance of any other classifier, but it does reduce crawl-time efficiency, unnecessarily working against the criterion established in section 3.2. To help prevent this situation, there is a simple mechanism for selecting the classifiers to be used for the next crawl: a list of check boxes, one per classifier, where a selected box indicates that the associated classifier should be used. Because this list is specific to a crawl, rather than to a classifier, it does not appear on any of the normal classifier pages. Instead, the list is available to the user on the “Page Options” page, which provides administrators with options for configuring how fetched pages will be processed during the next crawl.

Once a set of classifiers has been chosen, and the other crawl parameters set appropriately, a new crawl using the selected classifiers can begin. The current configuration for each active classifier is sent to the fetchers, where it is used to construct a classifier instance that will remain in memory for the duration of the crawl. After a fetcher downloads and processes a page, as a final step it gives the active classifiers a chance to label it. Each classifier computes an approximate probability that the page belongs to its class, and if this estimate exceeds a threshold then the classifier adds the appropriate meta word to the record for that page. These meta words are associated with the page in the index, and may be searched for later to limit query results to the pages associated with those meta words.

The following four sections dive into the details behind the design and implementation of each major component just outlined.

4.2 Managing Classifiers

The system for managing the creation, viewing, and deletion of classifiers is quite simple, and most of its design with regard to operation was covered in the overview. It does remain, however, to describe how classifiers are represented in memory and stored on disk.

4.2.1 Design

The entity that until this point has been referred to as a classifier is in fact a container class for all of the data required to carry out the training and application of a classification algorithm. In addition to the data, each classifier instance also maintains references to one or two instances of classes that implement the actual classification algorithms. This arrangement facilitates the use of two different classification algorithms for the different tasks to which they are suited without duplicating data or complicating the code that makes use of a classifier. These two different tasks are, roughly, searching for the next document to label when building an example set, and actually classifying new documents at crawl time. The former task requires a very efficient classification algorithm that might sacrifice some accuracy, while the latter task has no serious time constraint, and benefits from maximum accuracy.

Classifiers[1] are stored on disk as directories of files containing serialized and sometimes compressed data. Storing classifiers in a directory structure makes them easy to back up and copy between machines, while splitting them across several files allows for partial reconstitution, depending on the task they are required for.

For example, when classifying at crawl time, a classifier has no need of the huge matrix of example data that it uses during training to estimate the $\boldsymbol{\beta}$ vector discussed in section 2.1.2; it would thus be a tremendous waste—both of resources and of time—to load that matrix into memory during classification. A similar argument applies when listing all classifiers on the main classifiers page, where only the most basic summary information about each classifier is required. A simpler design would reconstitute each classifier in its entirety, including several megabytes of training data, the full vocabulary, the $\boldsymbol{\beta}$ vector, and so on.

[1] That is, the containers mentioned previously. The class that actually implements an algorithm such as logistic regression will be referred to generally as a “classification algorithm.”

4.2.2 Implementation

When a user clicks on the “Classifiers” activity tab in Yioop’s web-based administrative interface, a request is sent to the name server, and ultimately routed to the new `manageClassifiers`[2] activity in the `AdminController` class. This activity scans a directory on disk for any existing classifiers, unserializes the appropriate skeleton files describing each classifier, and passes the aggregated summary information off to a view, which generates HTML to display the list of classifiers and their basic information. This HTML is embedded in Yioop’s generic administrative template (which provides HTML to display the header, activity tabs, and so on), and the result is finally sent back to the client, where it is rendered into the page that the user sees. When the user submits the form to create a new classifier, or clicks the link to delete a classifier, this same basic execution path is followed, except that in the former case a new, empty classifier instance is constructed and saved to disk, and in the latter case the appropriate directory on disk is deleted.

In memory, each classifier is an instance of the `Classifier` class, which provides a simplified interface to manage the training set, to select a new document for labeling out of a pool of unlabeled documents, and to both train and use two different kinds of classification algorithms. The training set is maintained internally as a sparse matrix[3] where the rows represent documents and the columns represent

[2] A monospace font is used to indicate the name of a class, method, or other entity taken directly from the project source code. This convention is used throughout the rest of the report.

[3] That is, as a map from row indices to maps from column indices to values, rather than as, for

terms, plus a vector of labels for the documents. Within this matrix (and elsewhere), terms are mapped to numeric indices to make more efficient use of memory and to simplify checks for equality; each classifier has its own instance of a `Features` class responsible for keeping track of the mapping between the two sets of indices. The `Features` instance also keeps track of feature and label statistics, such as how often a given feature appears in documents with a given label, and how many documents overall have a particular label. These statistics are maintained as new documents are added to or removed from the training set so that they may be queried efficiently.

The pool of unlabeled documents is simply stored as an array, but some parallel structures are also maintained in order to facilitate more efficient calculation of document densities. Finally, each classification algorithm is represented as an instance of a subclass of the `ClassificationAlgorithm` class. It may seem strange that an algorithm should be instantiated, but this is done to keep the algorithm's free parameters and the $\boldsymbol{\beta}$ vector computed during training in a single place for serialization. All of these properties of a classifier are persistent—that is, they are stored on disk between requests.

The directories that represent a classifier on disk are named after the class labels of the classifiers they store. Thus the labels must be unique, and given a

example, a vector of vectors. Under the former scheme, zero entries are left out of the maps, so that an $m \times n$ matrix of zeroes would take up $O(1)$ memory, instead of $O(mn)$ memory, as would be the case under the latter scheme. A sparse vector is represented similarly, as a map from indices to values.

label, the associated classifier can be efficiently found on disk and reconstituted. Whenever classifiers are first created or loaded from disk they contain only summary information such as the class label, the total number of examples in their training sets, and their last estimated accuracy. The extra data stored on disk (such as the matrix representing the training set, the `Features` instance, etc.) must be explicitly loaded by calling the appropriate classifier method; there are several related methods for this purpose—each one tailored to prepare the classifier for a particular activity, such as training or classification. When the classifier is stored back to disk, only the files relevant to the current activity are written to.

At present there is no attempt to guarantee that operations on classifiers are atomic, and this could certainly cause issues if several users attempted to work with the same classifier on the same instance of Yioop, all at the same time. As an example, it is possible to lose information if two users submit requests to add a new document label sufficiently close together. Should one request arrive before the earlier request has written the modified classifier back to disk, the second request will read in stale information, nullifying the first request's effect when it eventually writes its own view of the classifier to disk. This and a number of similar conflicts that can occur when operations on classifiers are carried out concurrently limit the system, in practice, to a single user.

4.3 Building a Training Set

Building a representative set of example documents for a particular class is every bit as important as choosing a powerful learning algorithm, for even the best algorithm cannot learn how to recognize a class without being exposed to examples of the variation amongst documents that belong to it. Since it is a major goal of this project to—as much as possible—relieve users of the responsibility to find examples, this component of the classification system will play a major role in its overall success or failure. The following subsections describe the interface that the system exposes for finding and labeling documents, as well as the work that occurs on the server to actually identify candidate documents for labeling, choose the next document to be labeled, and keep track of the labels that a user assigns.

4.3.1 Client-Side Design

As mentioned briefly in the overview, the interface for building a training set can be found on the classifier "edit" page, and this is where users will spend most of their time when creating a new classifier. The top of the page provides some summary information and a place to change the class label, but the rest of the page is dedicated entirely to finding and labeling documents. This process begins with the selection of a Yioop crawl index to draw examples from, and an optional query that selected documents must match.

Using existing indices as a source for documents has several advantages. First and foremost, indices can be efficiently queried, which allows users to quickly and easily narrow down the pool of candidate documents to those matching one or more words. This capability can be very helpful early on, since presumably the user can make some informed guesses about which words are likely to be good indicators for a class. A user trying to find examples of documents containing advertising might, for instance, search for phrases such as “one weird trick” or “limited offer”. Another advantage of using indices is that they provide a convenient and uniform way to group, store, and access documents; furthermore, there is already a mechanism built into Yioop for importing and indexing records from a variety of formats (e.g., data base records, specialized archive formats, and plain text files).

Having selected an index and optionally specified a query, the next step is to decide between bulk and manual labeling. The former case is simpler, and is useful if one already has a training set for a particular problem. In this case, all candidate documents (i.e., documents in the index that optionally match a query) are given the same label—specified by the user as either positive or negative. To take advantage of this method, the user would import the positive examples from the training set into one index, and the negative examples into another,[4] then perform two bulk labeling operations, one for each index. Bulk labeling may take a while if there are a lot of candidate documents, but it is extremely efficient relative to manual labeling. Of course, this is only possible because the examples have already

[4] Or, alternatively, add a special meta word to each example specifying whether it is positive or negative. Then the positive examples can be selected by specifying the positive meta word as the query.

been pre-sorted into positive and negative groups; usually this will only be the case for data sets used as benchmarks for machine learning research.

Manual Labeling

The more common case will be that a user wants to build a classifier for some novel class, and has no preexisting set of examples to leverage. This is the scenario that active learning (introduced in section 2.3) was developed for, the goal being to reduce the number of documents that the user must label by always selecting for labeling the candidate document that the classifier can learn the most from. If active learning is working well, then the classifier should be able to gain more accuracy from each document added to the training set than if documents to be labeled were selected randomly. From the user's perspective, however, the interface is the same.

After selecting an index, specifying an optional query, and choosing manual labeling, the user is presented with a record summarizing the first document to be labeled. The record includes the document title, its URL,[5] a summarized description (similar to the brief description one sees under each search result on Google and other popular search engines), and whether the current classifier thinks this document is in the class of interest or not, and with what confidence. To the left of the record, the user is presented with three buttons: "in the class", "not in

[5] For simplicity, the term "URL" here is used somewhat loosely. Because Yioop supports crawling archived records stored in databases and other files, not every document has a URL such as "http://www.example.com". Instead, some documents may have a string of the form "record:*hash*", where the *hash* uniquely identifies the record.

the class", and "skip". Clicking one of these options sends the user's choice—paired with the document's unique identifier—to the server.

The server adds the document and its associated label to the training set, re-trains the classifier on the improved training set, selects the next document to be labeled, and sends it back to the client (i.e., the web browser), where the display is updated. The record for the just-labeled document is shaded according to the choice the user made, and pushed down to make room for the new record. Because the old record is still visible, the user may change its label or retroactively skip it at any time, causing a new request to be sent to the server to update the classifier accordingly.

JavaScript and Asynchronous Requests

Manually labeling documents is the most interactive aspect of building a classifier, and as such it is important that its interface is both responsive and easy to use. Normal HTTP requests are not suited to highly interactive communication between the client and server because they require the interface on the client to be rebuilt from scratch with each new request. Doing this for each document labeled would not only waste time on the client, it would create extra work for the server, which would have to repeatedly generate a lot of HTML that remains the same between labeling operations (such as the administrative frame, the form for changing the class label, and so on).

Fortunately, modern web browsers support JavaScript, Document Object Model (DOM) manipulation, and the `XmlHttpRequest` object, which enables JavaScript code to make asynchronous requests to the same server from which the JavaScript was served, and read the server response, all without reloading the page. These resources together make it possible to write web pages that operate much like desktop applications. The JavaScript code on these pages captures common events such as button clicks, suppresses the default action, and instead makes an asynchronous request to send some data (such as the label selected for a document) to the server; the code then parses the server's response to determine how to update the client display. The interface for finding and labeling documents makes heavy use of this pattern in order to improve responsiveness. In fact, the only operation on the classifier "edit" page that causes it to be reloaded is changing the class label.

4.3.2 Client-Side Implementation

The implementation of the client interface for building the training set is not particularly germane to this thesis, being more suited to a discussion of web design. A brief sketch of the organization will, however, benefit the reader's overall understanding of how the classifier system fits into the Yioop framework.

Yioop consists mostly of static pages containing forms that are filled out on the client and sent to the server, causing a new page to be loaded. Most of the classification system does not deviate from this behavior, but the interface for labeling documents certainly does. The JavaScript that implements this interface resides in a single static file that is linked to the classifier “edit” page as a resource. When the page is loaded, that file is requested, parsed, and executed by the web browser. This initiates a JavaScript thread of execution that continues to run until the page is navigated away from or reloaded. That thread takes care of sending requests for new documents to label, displaying the record associated with a document, shifting old records down, and sending new labels back to the server. These requests are sent to a special `ClassifierController` that does not output HTML, but instead responds to requests with data encoded in JavaScript Object Notation (JSON).[6] Thus the data sent back and forth between the web browser and the server is drastically reduced relative to what would be required if a new page were sent and rendered each time the user labeled a document.

The JavaScript that manages the interactive aspects of the client interface is complemented by a small collection of additions to the global style sheet that Yioop uses to style its bare HTML. These additions specify of a variety of small visual tweaks, such as changing font sizes and colors, setting the background color for document records that have been marked as positive or negative examples, and setting the spacing between elements.

[6] The JSON format has a similar purpose to XML in that it is designed to be easily machine-readable, but unlike XML, it can be parsed by evaluating it as JavaScript—a facility built into every modern web browser.

4.3.3 Server-Side Design

Each request made to the server to ask for the next document to be labeled, or to add a label to a document, results in a flurry of activity and computation that eventually settles down into a relatively compact response. Usually this response contains a concise summary of the next document selected for labeling, and updated classifier statistics such as the number of positive and negative examples, the estimated accuracy, and the number of documents remaining in the candidate pool.[7]

Generating such a response requires a series of computations that—unfortunately—can easily add up to an unacceptable delay on the client. As a representative example, using density-weighted pool-based document sampling (see section 2.3.1), the server must work its way through the following steps in order to select the first document to be labeled from the candidate pool generated by choosing a new index and query:

(1) Load the relevant classifier into memory from disk, including the matrix representing all training documents, the vector of training labels, the `Features` instance responsible for mapping between terms and feature indices, and an instance of the Naive Bayes classification algorithm that has been previously trained on the current training set.

[7] Recall that the candidate pool consists only of those documents in the currently-selected index that also optionally match a query. Thus the candidate pool may be quite large or quite small, depending on the size of the selected index and the specificity of the query, if any.

(2) Create a new iterator for traversing through the documents in the selected index that match the provided query, if any, and iterate through these documents, checking each one against the set of existing training documents (using a unique key), and removing any duplicates.

(3) Compute the density of each remaining candidate document, requiring approximately N^2 computations of the KL divergence between two documents, where N is the size of the candidate pool.

(4) Create a small committee of new classification algorithm instances, each one with a slightly different $\boldsymbol{\beta}$ column vector.

(5) For each candidate document, first classify it with each of the committee members, then compute the disagreement between them. Multiply this disagreement score by the document's density score, and keep track of the document with the largest product of the two.

(6) Use the current classification algorithm to classify the best candidate.

(7) Output the selected candidate and relevant statistics for the current classifier.

The user has nothing to do while waiting for this process to complete, so in order to maintain responsiveness, it is important that these operations be carried out as quickly as possible. Many of them, however, do not take a constant amount of time, and instead scale with either the number of candidate documents, or the size of the existing training set. For example, if the training set contains several thousand labeled documents, it can take quite a while to load all of those from disk and into memory. Similarly, if there are several thousand candidate documents, classifying each one a small number of times can add up to a significant delay, and computing densities takes even longer.

Early experiments made it clear that PHP is not up to the task of carrying out all of these operations on an arbitrary number of documents, and identified the document density calculations as the limiting factor. With an empty training set, loading fifty candidate documents and selecting the first document to be labeled takes just under two seconds, while loading two hundred candidates takes a little under twenty-two seconds. The latter delay is getting close to the default thirty second PHP execution time limit for web requests, and regardless, is too long to make a user wait for useful feedback. Furthermore, these delays represent the best case, since there is no large training set to load from disk, and no expensive training phase to be carried out, as when adding a new document label. It thus became clear early on that the pool of candidate documents would need to be constrained to a maximum size, certainly less than two hundred documents, and that speed would be of primary importance in the design of the server-side procedure for selecting documents for user labeling.

The remainder of this section breaks the description of the server-side design

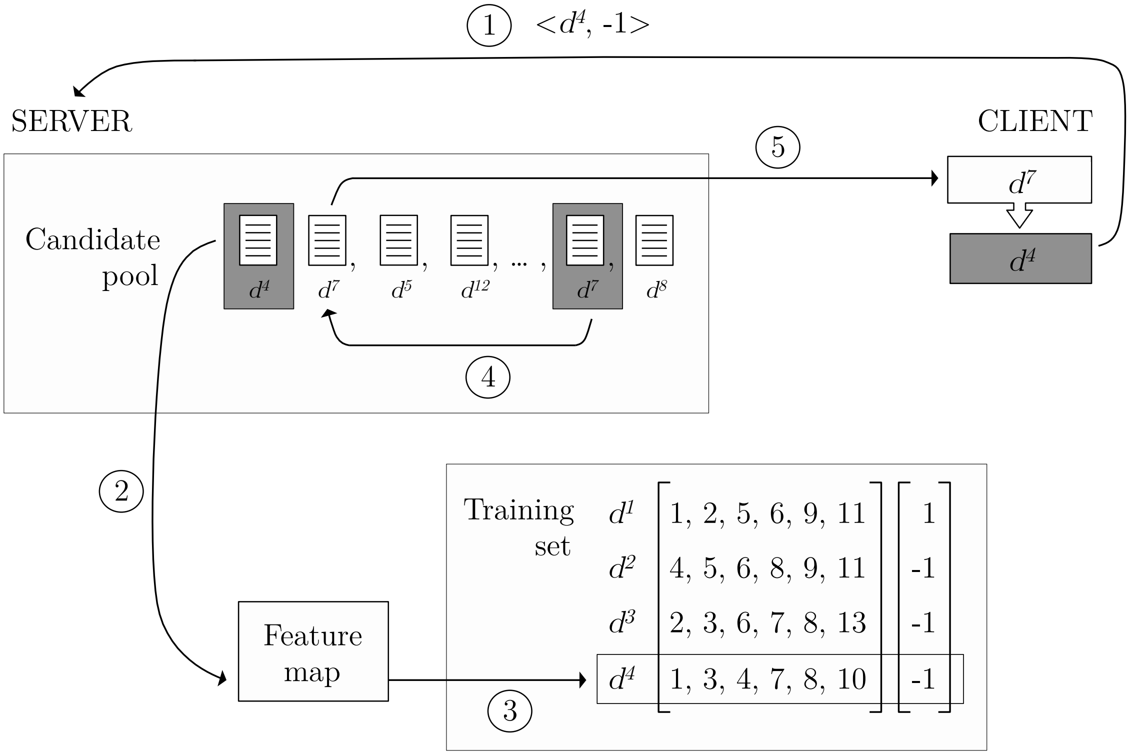

Figure 4.1: An overview of the labeling system. The darker gray boxes indicate state before a user adds a new label, and the lighter gray boxes indicate state at the end of the labeling operation. The circled numbers give the order of operations, starting after the user clicks a link to mark the document d^4 as *not* in the target class. On the server, d^7 is moved to the front of the candidate pool, replacing d^4, which is converted to a feature vector and placed in the training set. On the client, the record associated with d^7 takes the place of the (now-old) record for d^4, pushing it down the page.

down into four parts: first, selecting documents to label; second, keeping track of document labels; third, estimating classifier performance as the training set grows; and last, feature selection. Figure 4.1 provides a high-level overview of the labeling process, to be used as a reference throughout the discussion. It depicts the relevant structures on the server and the client, and how they change when a user labels a new document.

Selecting Documents to Label

Selecting documents for the user to label requires—at a minimum—a pool of candidate documents. The classification system fills this pool from an iterator over documents that come from the user-selected index, that optionally match a query, and that have not been labeled already. Yioop has a convenient mechanism for iterating over documents matching a query, so the classification system only needs to take care of filtering out documents that have already received a label. Such documents can occur in the iterator stream because two iterators created for different queries (or even twice for the same query) have no memory of documents that have been iterated over before.

Conveniently, a classifier has just such a memory in its collection of training documents. Each document is uniquely identified by its URL, so by storing documents in a hash table keyed by document URL, the classifier can efficiently look up whether a given document has already been labeled. Any document in the iterator stream that is also in the hash table is simply skipped. Note that this system does not keep track of documents the user has manually skipped, since these documents are never added to the training set; consequently it is quite possible for a user to skip a document, then enter a new query and see the same document again. This behavior may sometimes be annoying, but it is also useful because it allows users to skip a document without making a final decision.

Iterating through candidate documents in this manner leverages Yioop's existing indices and web interface to help users find examples, but it does not reduce the number of documents that the user must label in order to achieve a target accuracy. To make progress toward that goal, the classification system uses density-weighted pool-based document sampling, as discussed in section 2.3.1. A small committee of k Naive Bayes classifiers is sampled once, then used to measure the disagreement for each candidate document; this disagreement score is multiplied by the document density, and the document with the largest product of disagreement and density is selected for labeling.

The committee uses the Naive Bayes classification algorithm because it is efficient, and it is possible to sample its $\boldsymbol{\beta}$ vector from the Dirichlet distribution specified by the existing training data. This is accomplished by, for each term t^i, drawing weights from two Gamma distributions—Gamma$(1 + N(t^i, pos), 1)$ and Gamma$(1 + N(t^i, neg), 1)$—where $N(t^i, pos)$ is the number of times that term t^i occurs in positive examples, $N(t^i, neg)$ is the number of times it occurs in negative examples, and the counts are incremented by one to smooth features that do not occur at all in one class or the other. These drawn weights are treated as the new counts for each feature, and used in the usual way to build the $\boldsymbol{\beta}$ vector for the instance (see section 2.1.2).

As discussed previously, calculating candidate document densities takes time quadratic in the size of the candidate pool, so the pool must be limited to a maximum size, set by a constant. The larger this constant, the farther out into the full stream of candidate documents the search for the best candidate can go, but the longer it will take. Thus the maximum pool size captures a trade-off between helpfulness and responsiveness. Ideally, it should be set so as to minimize the number of documents that the user must label to obtain a desired accuracy, subject to the constraint that the time to select a document never exceeds the PHP execution time limit or the user's patience. If the number of candidate documents is larger than the pool size, then each time a document is labeled and removed, it is replaced by a new candidate and the document densities of *all* candidates are recalculated (because the removal of one document and addition of another will usually change the probability of seeing each term).

Managing Labeled Documents

The second half of building the training set is keeping track of the labels that the user assigns to documents, as well as the documents themselves. Each classifier maintains two separate sets of documents: the pool of candidates for labeling, and the training set, which contains all labeled documents. The candidate pool is frequently cleared out and replenished as the user labels documents and changes the index or query used to identify candidates. The training set, on the other hand, generally only grows as documents from the candidate pool are labeled and subsequently migrated over.

When a user first selects a label for a document in the web browser, a request containing the document's URL and the selected label is sent to the server. The server removes the associated document from the candidate pool (replenishing the pool if there are more candidates), adds the document's terms to the `Features` instance that keeps track of the vocabulary, and stores a transformed version of the document—along with its label—in a hash table keyed by the document's URL. The transformed document is a binary feature vector—represented sparsely—where the indices correspond to terms, and a value of one or zero indicates a term's presence or absence, respectively. These feature vectors are essentially the rows of the matrix $\mathbf{X}$ defined in section 2.1.

When a user changes a document's label, as opposed to adding a label for the first time, the training set is updated to reflect the new label, and the classifier's `Features` instance is updated as well, to maintain accurate counts for each of the terms that the document contains (since, for example, each term in the document may have changed from appearing in a positive example to appearing in a negative example). The process for retroactively skipping a document is similar, the main differences being that a row is removed from the training set altogether, and rather than just being shifted around, the counts maintained by the `Features` instance strictly decrease.

Gauging Classifier Performance

In addition to updating the candidate pool, updating the vocabulary, adding a new example document to the training set, and finding the next document to label, each time a new document receives a label the server also trains several instances of the Naive Bayes classification algorithm on the newly-augmented training set. The Naive Bayes instances thus trained are used to classify the next document selected for labeling (to give the user some insight into the classifier's current bias for a given document), and to provide an estimate of the accuracy that can be expected from a classifier trained on the current training set. Naive Bayes is used instead of logistic regression because there is already a lot going on, and the optimization step employed by logistic regression would simply take too long to justify the extra accuracy. This is not the classification algorithm that will ultimately be used to classify new web pages—just a tool to help the user gauge how useful the current training set is.

The current training set's quality is estimated by setting aside a randomly-selected fifth of its documents as a test set, then training a new Naive Bayes instance on the remaining documents. The trained instance is used to classify the documents in the test set, and the accuracy is recorded. This process is repeated four times, each time rotating the previous test set into the new training set, and an equal-sized section of the previous training set into the new test set. At the end, all documents in the full training set have been used for both training and testing, and the mean of the recorded accuracies serves as a proxy for the usefulness of the overall training set. This measure is very noisy when the training set is small, but becomes more reliable as the number of documents in the test sets grows.

Feature Selection

The preceding sections have often mentioned training a classification algorithm on the current training set, but this is in fact a simplification that glosses over a non-essential, but practical, extra step. As mentioned in the introduction to text classification (see section 2.1), a lot of problems have a vocabulary containing as many as ten thousand distinct terms, and this can be a problem both for training *and* for classification, depending on the algorithm used. A large number of features simply slows training down, and while this is not a serious problem for the Naive Bayes algorithm, it can severely limit the practicality of logistic regression, especially when time is at a premium.

Naive Bayes, on the other hand, is very sensitive to a large number of features when classifying a document. The independence assumption tends to overvalue the contributions of individual features, so that the consideration of a large number of features biases the probability estimate to one extreme or the other. This behavior often does not go so far as to make the classifier wrong, but it does make it overly-confident. That undue confidence can, in turn, harm the effectiveness of the disagreement measure used to select documents for labeling, since nearly every document garners maximum disagreement.

A simple approach to combat both of these negative effects resulting from a large number of features is to get rid of some of them, and this is exactly what the classification system does before actually running any classification algorithm over the training set. Specifically, the top N most informative positive and negative features according to the χ^2 feature selection algorithm are kept, and all other document features are ignored. The number N should be a relatively small integer (e.g., sixty) in order to both reduce training time for logistic regression and make Naive Bayes a reasonable soft classifier.

This reduced feature set is another `Features` instance, derived from—but independent of—the full feature set. The reduced set is used to sample the committee for document selection, to train both the Naive Bayes and the final logistic regression instances, and to classify new documents. In fact, the full feature set is only kept around to maintain the term statistics used to select successively better feature subsets as the training set grows. If, instead, only the most informative features were kept each time a new example was added to the training set, there would be no way for an initially-uninformative feature to gain ground over time and eventually become an informative feature.

4.3.4 Server-Side Implementation

The concrete implementation details of the server-side process for building the training set are, like the client-side details, relatively minor. The entry point for all requests made by the client is the `classify` activity of the `ClassifierController` class, referenced previously. This activity has two sub-activities reflected in the structure of the previous sections: `getdocs` and `addlabel`. Both sub-activities begin by loading the relevant classifier from disk, along with nearly all of its associated data, and end by sending a JSON-encoded response back to the client. The response always provides summary statistics for the classifier, whether they have had reason to change or not, and the client always updates its display with these potentially-new numbers.

Candidate documents are retrieved from an index using a *crawl mix*, which is essentially an iterator over search results whose progress can be saved to disk, making it possible to pick up iteration from where it left off on a previous request. This interface is exactly what is needed to fill the candidate pool initially, then replenish it as labeled documents are removed and placed in the training set. Crawl mixes are stored in the Yioop database, so each time a new index or query is requested by the client, the `classifier` activity deletes any previous crawl mix record for the current classifier, and inserts a new one. The record for the last index and query selected remains until the classifier is deleted, but it is never visible to the user.

The committee size, k, used for measuring classification disagreement of candidate documents is three; this number was adopted from McCallum and Nigam, who found that larger committee sizes provide little benefit. The smoothing parameter, λ, and the sharpness parameter, γ, used in the calculation of document density are 0.5 and 3.0, respectively; again, these values were adopted from McCallum and Nigam. The candidate pool is limited to fifty documents, based on experimentation with how long it takes to calculate densities for that many documents. For efficiency, the statistics—such as marginal word probabilities—used to compute document densities are calculated once when the candidate pool is initially filled, and updated incrementally as labeled documents are removed from the pool and replaced by new candidates.

Standard PHP associative arrays are used as hash tables to store the documents in the training set by key (recall that the key is the document URL), and to store their labels as well. At most 250 randomly-selected documents are used to estimate accuracy, resulting in two hundred documents being used for training, and fifty for testing. This number was chosen by experimenting with the time that it takes to run five training and testing rounds on that many documents. Finally, only the top thirty and the top two hundred most informative features for documents are used with the Naive Bayes and logistic regression algorithms, respectively.

4.4 Training a Classifier

Once a training set has been established for a class of interest, all that remains to obtain a functioning classifier is a final round of training using the more aggressive logistic regression algorithm. The classification algorithm's output is the column vector $\boldsymbol{\beta}$ such that, for a new document $\mathbf{d}'$, $\boldsymbol{\beta} \cdot \mathbf{d}'$ is a measure of the likelihood that $\mathbf{d}'$ is a positive instance of the class. This column vector is used during a crawl to efficiently assign class labels to documents.

4.4.1 Design

Unlike the Naive Bayes text classification algorithm, which simply makes one pass through the training data, and one more pass through each of the features to build the vector $\boldsymbol{\beta}$, logistic regression attempts to solve for the $\boldsymbol{\beta}$ that maximizes the likelihood of the training data (see section 2.1.3). In practice, a solution to this problem is always approximated by iteratively improving an initial guess for $\boldsymbol{\beta}$, where each successive improvement requires evaluating a function over the entire training set. Consequently, logistic regression can often take a relatively long time to converge on a solution. Nonetheless, the extra time is usually worth it, as logistic regression often produces a more accurate classifier than Naive Bayes, and running the logistic regression algorithm does not require any interaction with the user, so responsiveness and usability are not a concern.

The primary concern, then, with letting logistic regression run to convergence is hitting the execution time limit set by PHP—thirty seconds by default. In order to avoid this limit, logistic regression, though initiated by a request from the web client, must be carried out in a new *training process* that continues to run even once the controller that initially handled the request has output its response and exited. When this training process first begins it sets a flag on the classifier and saves it to disk so that new requests will see that the classifier is currently being trained. After logistic regression completes and the optimized $\boldsymbol{\beta}$ vector has been saved, the training process removes the flag, signalling to future requests that the classifier has been successfully finalized. The client takes advantage of this flag to periodically check in with the server and update its display accordingly.

The specific algorithm used to maximize the likelihood of the training data is lasso logistic regression using a Laplace prior, as presented in section 2.1.3. This algorithm was shown to be particularly effective for text classification problems, and has the benefit of being relatively easy to implement in PHP, since it requires no involved matrix operations. The implementation was modified slightly from that proposed by Genkin, et al. to use a "parameter sleeping" strategy that attempts to avoid repeatedly updating parameters—components of the $\boldsymbol{\beta}$ vector—that appear to have stabilized at some value. This is done because each parameter update step is relatively expensive, and many parameters (even after feature selection) make only a minor contribution to the overall log likelihood.

4.4.2 Implementation

Yioop already has a method for launching a new PHP executable from a web request so that fetchers and queue servers may be started from the web interface. The classification system uses this same mechanism to manage the training of a logistic regression instance by adding a new executable script. The script takes the name of a classifier as an argument, loads the appropriate classifier into memory, sets the flag indicating that the classifier is being finalized, and initiates training on the classifier's logistic regression instance. A subset of the features are used in order to limit training time to a few minutes. Once the logistic regression process converges, the executable toggles the "training" flag off, and saves the trained classifier back to disk. The value used to check for convergence and whether or not a component of the $\boldsymbol{\beta}$ vector is sufficiently close to 0, ϵ, is set to 0.001.

4.5 Using a Classifier

Any finalized classifier may be used in a crawl by selecting its checkbox on the "Page Options" page of Yioop's administrative web interface. This final component of the classification system resides on that page because it is specific to a single crawl, whereas the other pages dealing with classification are for building classifiers that might be used on many crawls. Once one or more classifiers have been selected, starting the crawl will result in each crawled page being classified. If the score computed by any classifier for a page exceeds a threshold, then a "class:*label*" meta word (where *label* is the classifier's name) is added to the page's collection of meta words.

4.5.1 Design

When the user selects one or more finalized classifiers and saves the "page options," a compressed, serialized representation of each selected classifier prepared solely for classification (i.e., stripped of any resources used only during training), is saved to the file of crawl options. When the crawl starts, these options are distributed to the fetchers, where the classifiers are decompressed and unserialized. Thus all fetchers have identical copies of each active classifier, and they only load each classifier into memory once—at the start of a crawl.

During the crawl, after a page has been fetched and processed, it is passed in sequence to each active classifier. If a classifier determines that the page belongs to the class that the classifier has been trained to recognize, then it adds the appropriate "class:*label*" meta word to it. Thus, as a single page is passed along the chain of active classifiers, it may be marked as belonging to several independent classes. Additionally, each classifier adds a sequence of related meta words that represent the pseudo-probability that the page belongs to the class.

An example will serve better here than an explanation of the scheme. If an "ad" classifier estimated that the probability of a particular page containing advertising was .74, and the hard classification threshold were set to .5, the classifier would add the meta words “class:ad”, “class:ad:50plus”, “class:ad:60plus”, “class:ad:70plus”, and “class:ad:70” to the page. Later, a user could use these extra meta words to search for pages that were classified as containing advertising with successively higher levels of confidence (or, more usefully, to search for pages that were *not* classified as containing advertising with high confidence). The final meta word is included to make it possible to search for specific intervals, in addition to anything over a particular threshold.

4.5.2 Implementation

The only fields of a classifier that are relevant to classification at crawl time are the trained logistic regression instance and the reduced `Features` instance. The latter is required to map the terms of a new page into a feature vector like those used to train the classifier, and the former contains the $\boldsymbol{\beta}$ vector whose dot product with the feature vector yields an estimated probability that the page belongs to a class. In order to be considered an instance of a class, a page must have an estimated probability of belonging to the class, $p \geq .5$.

Chapter 5

Experiments

This chapter presents several experiments that were carried out to assess how well the classification system meets three of the four main criteria developed in chapter 3. The three criteria tested were effectiveness, efficiency, and responsiveness, leaving out usability. As discussed previously, the system's usability is difficult to test quantitatively, and is not the primary focus of this project. Thus, instead of testing usability with an experiment, a set of guidelines were developed and followed in the design and implementation of the system. See section 3.4 for a discussion of these guidelines, and the previous chapter for how they were incorporated into the system's design and implementation.

The remaining criteria were tested on a corpus of Internet advertisements [6] built to investigate the possibility of automatically blocking certain advertisements placed on content providers' websites through advertising networks such as Google AdWords.[1] The advertisements placed on twelve different websites were collected and shown to their corresponding content providers, who rated each on a 1–5 scale of acceptability; any advertisement rated below three was labeled as unacceptable.

[1] Content providers (e.g., blog owners, newspapers, special interest websites, etc.) often monetize their efforts by signing up with advertising networks, which automatically place their customers' advertisements on signed-up providers' websites. The network collects money from a customer wishing to place an advertisement, takes a cut, and pays the rest to the content providers on whose websites the advertisement was placed.

Each advertisement is represented by text collected from the advertisement itself, and from its landing page. All words were stemmed and additionally annotated according to whether they were found in the advertisement text, or on the landing page, either within the title or the headings. This corpus was chosen because it is representative of the kind of data that Yioop can extract from websites during crawls, and because detecting advertisements—especially some advertisements and not others—is a motivating use cases for adding web page classification to Yioop. Furthermore, the corpus authors also employed active learning methods to train a classifier on the corpus, and their published results provide a useful point of comparison to Yioop's own active learning approach.

5.1 Experimental Setup

The following sections are divided according to the performance criteria that they investigate. Each experiment uses the Internet advertisements corpus described previously, which contains 4,143 instances of textual representations of advertisements. Mesterharm and Pazzani used 3,000 instances for training and 1,143 for testing, repeating each experiment fifty times on fifty different permutations of the data. Unfortunately, because Yioop's classification system cannot comfortably handle training on that many instances, the size of the training set in the experiments presented here is limited to 500 instances, though the number of test instances is kept the same. For similar reasons, each experiment is repeated ten times on ten different permutations of the training and test data, instead of fifty.

Although Yioop's classification system is implemented as a web interface, as a practical matter the experiments are mostly carried out using a command-line tool that automates the training and classification process. This setup will provide a lower bound on those experiments that measure time as the dependent variable, since directly manipulating the server state removes all of the overhead introduced by the web browser, and by its communication with the server. In these cases, some anecdotal evidence is given for the delay that the user can expect when using the web interface to carry out an equivalent operation.

5.2 Effectiveness

Classification effectiveness is measured by the classifier's error rate when classifying test instances. The first experiment measures error rate as a function of the number of training instances, where it is expected that the error rate will decrease as the number of training examples increases. Both a lasso logistic regression (with and without active learning) and a Naive Bayes classifier were run on successively larger training sets, up to a maximum size of 500, with a reduced feature set of the top two hundred most informative features selected according to the χ^2 algorithm. The Naive Bayes algorithm would not be used in practice to classify web pages, but is shown here for comparison with the lasso logistic regression implementation. With each addition of ten new training documents, each

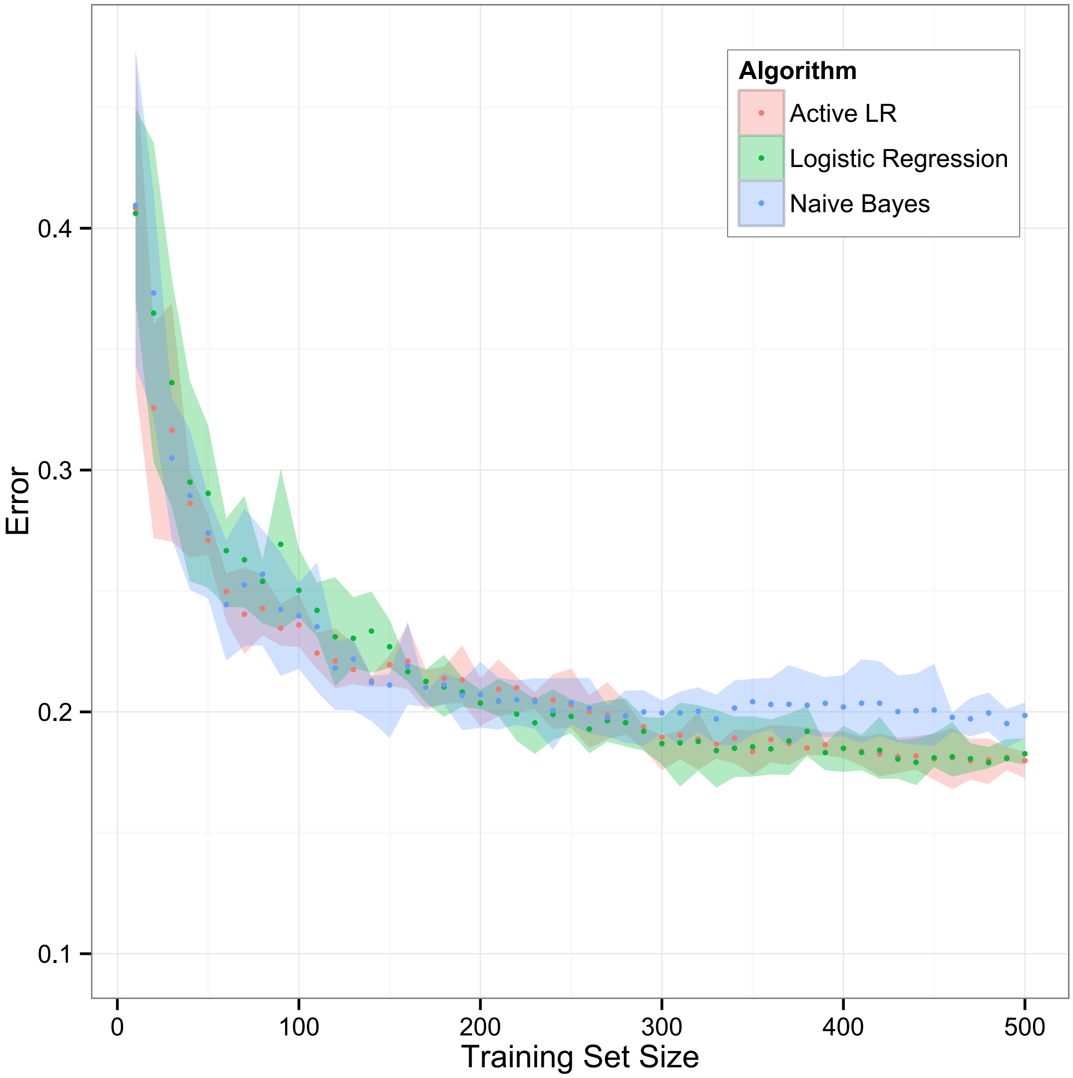

Figure 5.1: Classification error by training set size, using lasso logistic regression (with and without active learning) and Naive Bayes classifiers. The ribbons are bound by the lower and upper quartiles of the distribution of error observations at each test point, and the points are the means. Note that the y-axis reaches its minimum at .1, and not at zero.

classifier was used to classify all 1,143 test documents, and the error rate recorded.

The results of ten different runs with ten different permutations of the training and test data were averaged, with the results shown in figure 5.1.

Mesterharm and Pazzani report error rates starting at approximately .4 for as few as ten training instances, and dropping down to approximately .1 for 3,000 training instances. Because the lasso logistic regression implementation that the Yioop classification system employs cannot efficiently train on the full 3,000 instances, and even with fewer instances uses feature selection and other optimizations in order to cut down on training time, it is expected that the error rate achieved at each step will be somewhat worse. This expectation is borne out by the figure, which shows that logistic regression approaches an error rate of .18 as the number of training instances reaches five hundred. While certainly not ideal, this is an acceptable error rate considering the relatively small size of the training set, and good enough to be useful for classifying web pages. It is interesting to note that while the Naive Bayes algorithm ultimately falls behind logistic regression, it stays very close up to a little under two hundred training instances, and is consistently slightly better. This behavior is likely due to the logistic regression algorithm overfitting to the relatively small number of training examples despite regularization.

Logistic regression with active learning appears to—on average—provide a clear benefit over the other algorithms when the training set is very small (less than 100 documents), but rather quickly loses its edge, and appears to converge with logistic regression without active learning. This behavior is likely due to the relatively small candidate pool that Yioop's active learning algorithm presently uses—only fifty documents. With the current strategy and pool size, the active learning algorithm never has more than a 49-document lead over the other algorithms, and as the size of training set grows this advantage appears to become less valuable. Either increasing the pool size, or throwing away some portion of the pool on each iteration in order to draw in more documents (a hybridization of the online and pool-based approaches) would likely help the active learning algorithm maintain its lead.

5.2.1 Feature Selection

The classification system uses feature selection with the Naive Bayes algorithm in order to make it a better soft classifier, and with the logistic regression algorithm to speed up training time. Feature selection rarely—if ever—*reduces* the error rate, but it can often significantly reduce training and classification time without greatly *increasing* the error rate. A second experiment was carried out to see how expanding or reducing the feature set affects the error rate. The setup is essentially the same as the first experiment, but with a maximum of only 250 training documents, and the classification algorithm fixed to logistic regression. Five separate trials were conducted with classifiers limited to using the top 25, 50, 100, 200, and 400 most informative features; again, each trial was repeated ten times, and the results averaged to create figure 5.2.

There is no clear benefit to adding more features. All of the error rates are

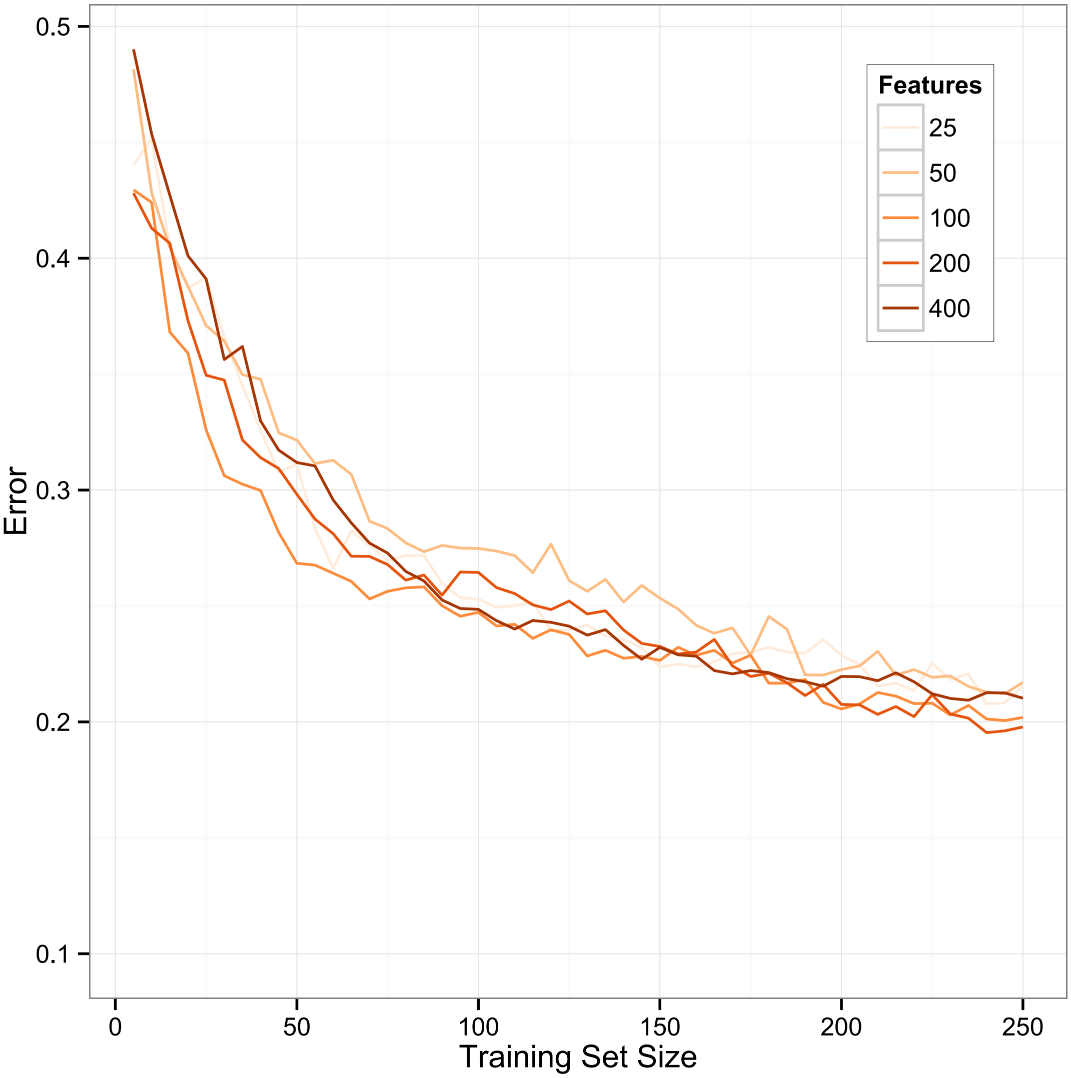

Figure 5.2: Classification error by training set size, for several logistic regression classifiers (trained without active learning) using successively more features. Note that the y-axis reaches its minimum at .1, and not at zero.

fairly close throughout, but with one hundred features showing a small advantage early on, and two hundred taking the lead when there are 250 documents in the training set. Using four hundred features appears to yield slightly worse performance than using either one or two hundred, but the gap narrows as more documents are added. Using less than one hundred features seems to be clearly worse throughout. These results suggest a policy of using as many features as possible, subject to the constraint that they do not make training and classification too costly. To maximize performance for training sets of all sizes, it may be beneficial to use more aggressive feature selection for a new classifier, and gradually increase the number of allowed features as examples are added.

5.3 Efficiency

Having determined that the lasso logistic regression classifier is effective, the next question is how much it slows down the crawl process. Classifying documents using either the Naive Bayes or the lasso logistic regression classifier should be efficient since both algorithms simply convert an incoming document to a feature vector and compute its dot product with the $\boldsymbol{\beta}$ column vector. Thus to the extent that crawling is slowed down, it should be a function of the number of features in the $\boldsymbol{\beta}$ vector, which is controlled by feature selection. The more features that are used to train the final classifier, the longer it should take to classify a page.

In order to explore the effect of classification on crawl times, a third experiment was carried out using three otherwise-identical lasso logistic regression classifiers trained without active learning, and with the maximum number of features set to 50, 200, and 400, respectively. These classifiers were used to classify documents during three separate crawls, all over the same set of documents, and the average time to process a fetched page recorded. As a baseline for comparison, a fourth crawl with no classification was performed. Because there is no way to automate the crawl process, this experiment was not repeated.

Table 5.1: The time to process a single fetched page without classification, and using three classifiers trained with 50, 200, and 400 features, respectively.

# Features	None	50	200	400
Time per page (s)	0.0030	0.0033	0.0034	0.0040

As table 5.1 makes clear, the extra overhead from classification at crawl time is insignificant. One could easily classify web pages using several classifiers, each trained with several thousand features, and the impact on the number of pages crawled per hour would be negligible. Thus, given a trained classifier, adding labels to documents via meta words at crawl time is efficient and perfectly viable.

5.4 Responsiveness

The final criterion to measure quantitatively is responsiveness in the interface for manually labeling documents. Recall that when the user submits a request for a new document to label, or to provide a label for a document, the server must do a lot of work that depends on the size of the candidate pool, the size of the training set, and the amount of feature selection. The user has nothing to do while waiting for the server to respond, so it is important that it respond relatively quickly.

The major contributors to this response time are loading a classifier from disk, refreshing the candidate pool, selecting a new candidate, training a Naive Bayes instance on the updated training set, and saving the classifier back to disk. As discussed in section 4.3, the most significant factor among these is refreshing the candidate pool, since doing so requires calculating new candidate document densities. This operation takes time quadratic in the number of candidate documents, which completely dominates all other operations. Figure 5.2 illustrates this point by measuring the time to load a new candidate and the time to label a document while varying the size of the candidate pool.

Table 5.2: The time to load a new candidate and label a document with document pool sizes of 50, 100, and 200.

Pool Size	Load Time (s)	Label Time (s)
50	1.59	1.60
100	5.61	5.39
200	21.52	20.61

Note that the time to label a document is approximately the same as the time to load in an initial document set. This is because both operations require refreshing the candidate pool and calculating document densities. The labeling operation is perhaps slightly faster because it only adds a single new document to the candidate pool before calculating densities, rather than having to fill the pool entirely. As more documents are added to the training set, the times listed in table 5.2 will increase due to the necessity of loading more data in from disk and training Naive Bayes on a larger training set, but they will continue to be dominated by the density calculations.

Chapter 6

Conclusion

The Yioop web page classification system exhibits classification performance close to published results for the same data set, and can be used without significantly slowing down the crawl process. Thus, at the very least, provided that the user can build a representative training set, the resulting classifier should be accurate enough to be useful. In order to help build such a training set, a web interface is provided to guide the user through identifying candidate documents and adding labels to them. The back end to this interface searches through a pool of candidate documents on the user's behalf, and attempts to identify the document for which having a label would most improve accuracy.

Yioop's facility for searching and iterating through arbitrary crawl indices provides an excellent foundation for labeling documents, and demonstrates how well-suited the search engine environment is to the task of building training sets. Not only does it provide a convenient means for the user to direct the search for example documents, it provides the classification system with a way to store and access those documents, which it would otherwise have to do itself. Moving beyond training, the search engine is also an excellent place to apply classification, as it essentially provides a steady stream of new documents to label, and a convenient mechanism—meta words—to store assigned labels for future use. Thus the search engine and classification system have a kind of symbiotic relationship, where the search engine provides a framework for managing documents, and the classification system (with the user's help) augments that framework with the capability to identify complex document properties.

6.1 Future Work

One of the benefits of Yioop's administrative web interface is that multiple users can access it at the same time in order to carry out different tasks. Presently however, as discussed in section 4.2, the way in which classifiers are stored to disk can result in unexpected, nondeterministic behavior if two users attempt to modify a classifier at the same time. One way to resolve this issue would be to introduce finer-grained control over what gets saved to disk, so that two users could coordinate the collection of training examples. Another approach would be to attach a notion of ownership to classifiers, so that only one user would ever have write access; a generalized facility for managing access to resources in the Yioop framework is presently being investigated by another graduate student.

With regard to classification itself, lasso logistic regression appears to achieve good accuracy, but is limited by its expensive training time as the size of the training set (either the number of examples or the number of features) grows. A promising alternative approach to training a logistic regression classifier on a large training set is stochastic gradient descent, which works by repeatedly updating the $\boldsymbol{\beta}$ vector in order to maximize the likelihood of individual, randomly-selected training examples [1]. This method requires far fewer passes over the entire training set, but after an appropriate number of iterations, still approximates an optimal solution to the logistic regression problem.

Finally, the present system is limited to making independent binary classification decisions on crawled documents, which makes it impossible for two classifiers to assign mutually exclusive labels. This decision simplifies the interface for building a single classifier, but restricts the overall space of classifiers that may be created. As mentioned in section 4.1, one way to maintain the simplicity of the current interface, yet enable the creation of mutually exclusive classifiers, would be to provide a mechanism to group classifiers together. This could be implemented, for example, by enabling a classifier to be specified as a *meta classifier* composed of other classifiers. Given a document to label, the meta classifier would poll its constituents and combine their decisions according to a chosen rule. This same mechanism could be used to implement more complex ensemble classifiers, as discussed in section 2.1.4.

Bibliography

[1] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[2] S. Büttcher, C. Clarke, and G. Cormack. *Information retrieval: Implementing and evaluating search engines.* The MIT Press, 2010.

[3] A. Genkin, D. Lewis, and D. Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007.

[4] A. E. Hoerl and R. W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[5] T. Joachims. Text categorization with support vector machines: Learning with many relevant features. *Machine learning: ECML-98*, pages 137–142, 1998.

[6] C. Mesterharm and M. J. Pazzani. Active learning using on-line algorithms. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 850–858. ACM, 2011.

[7] F. Pereira, N. Tishby, and L. Lee. Distributional clustering of english words. In *Proceedings of the 31st annual meeting on Association for Computational Linguistics*, pages 183–190. Association for Computational Linguistics, 1993.

[8] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[9] P. Tan et al. *Introduction to data mining.* Pearson Education India, 2007.

[10] Y. Yang and J. Pedersen. A comparative study on feature selection in text categorization. In *Machine Learning: International Workshop then Conference*, pages 412–420. Morgan Kaufmann Publishers, Inc., 1997.

[11] T. Zhang and F. J. Oles. Text categorization based on regularized linear classification methods. *Information retrieval*, 4(1):5–31, 2001.