Web Page Classification in Yioop!

CS297 Progress Report

Shawn Tice

11 December 2012

1 Introduction

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 examples and derives both a set of labels and a procedure for assigning one or more of those labels to new instances. The unsupervised learning equivalent of classification is *categorization*.

Both classification and categorization have applications to information retrieval, where they can be used to weed out uninteresting documents (e.g. spam), to label documents so that searches may be restricted to a particular class (e.g. pages about a location), and to group related documents together (e.g. news stories about the same event). The Yioop search engine does not presently perform any classification or categorization, and this project aims to change that by introducing automatic classification of web pages at crawl time. The goal is to provide administrators with an interface to train classifiers on a collection of labeled examples (i.e. web pages) belonging to two or more classes, and to enable Yioop to use the trained classifiers to assign labels to crawled web pages using the mechanism of meta keywords. Because classification requires labeled training examples to learn from, and collecting such examples can often be the most difficult and tedious aspect of the task, the system should also help the administrator to efficiently find and label example documents, either on the open web or in an index from a previous crawl.

This classification system breaks down into a few major components: a classifier implementation, a web-based tool to interactively train classifier instances, and the application of one or more trained classifiers at crawl time to label new web pages. The first component can be implemented relatively independently of Yioop, while the latter two components require changes both to Yioop's administrator interface and its behavior at crawl time.

Over the course of this semester (Fall 2012) I've developed a plan for how web page classification will work in Yioop, explored the existing literature on text classification, and begun developing and experimenting with stand-alone implementations of Naive Bayes and logistic regression classifiers in PHP, as well as a method to make training a classifier more efficient, called feature selection. Next semester I will finalize these implementations and integrate them with a new Yioop configuration interface for training classifiers and selecting classifiers to be used for a new crawl. In the following sections I'll discuss each of the components of the proposed system in more detail. For each component I'll present my plans and report relevant findings from the literature I've reviewed, as well as anything I've discovered through my own experimentation. Finally, I'll briefly list the work that remains to be done next semester.

2 Classifier Implementation

Our goal is to allow Yioop administrators to train a collection of classifiers to assign a class label to new, unlabeled web pages at crawl time. Web page classification is a special case of text classification, and most of what applies to the latter applies to the former as well. My work so far has focused on text classification, so the rest of this section will be about that, with the understanding that the same ideas translate to web page classification with minor modifications.

In text classification the labeled examples are usually documents, and the variables are the terms that appear in a document. 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. One popular form for the input to the classifier to take is a binary matrix where the rows represent documents and the columns represent terms; a zero in cell A_{ij} indicates that term j does not occur in document i, and a one that it does. An alternative is to use some sort of term weight instead of a binary value. The weight might 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 large number of variables that must be considered—anywhere from one to ten thousand for a normal problem. A large number of variables gives rise to two practical difficulties: it's harder to avoid over-fitting to the training data, and it takes more work to compute an answer. To see why this is so, we'll consider the classification task from the perspective of finding boundaries between points in high-dimensional space, concentrating for now on deciding between only two classes.

2.1 Classification as a Boundary Problem

Each row of the input matrix presented previously can be seen 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 must 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 side. If there were only one variable, then the points would be distributed along a line, and the boundary separating two classes of points would be limited to one 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.

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 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's usually problematic because the training points aren't completely 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 in the solution, one either needs more training points, or a way to artificially constrain the solution. It's obviously more work to gather more training examples (for some problems this may even be impossible), and constraining the solution 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 find a solution. Some algorithms scale better than others, but they all must somehow take into account each variable across all training points when building 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 very well 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 there are now a number of algorithms which have either been tailored specifically to text classification, or have been shown to be suitable for the task. Next we'll discuss two well-known text classification algorithms which I spent this semester implementing and experimenting with: 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 well-suited to text classification [5].

[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.

2.2 Naive Bayes

The basic formulation of the Naive Bayes text classification task is to find $p(c|d) = p(c|t_{d,1}, t_{d,2}, \ldots, t_{d,n})$ where c is a class label, and $t_{d,1} \ldots t_{d,n}$ are the terms present in a particular document d (recall that transformed term features such as TF $\cdot$ IDF may be used as well). By Bayes' theorem, the probability that d belongs to c given the presence of a *single* term t_i is

$$p(c|t_i) = \frac{p(c)p(t_i|c)}{p(t_i)}$$

We can ignore the denominator because it doesn't depend on c, leaving just $p(c)$ and $p(t_i|c)$, both of which can be computed from the training data. To find $p(c|t_{d,1}, t_{d,2}, \ldots, t_{d,n})$, however, we 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 this is the domain of Bayesian networks, but an even simpler and more common approach is to make the assumption that terms are independent of one another. This is called the Bayesian independence assumption, and although it is clearly false, in most cases the true conditional probabilities don't have enough of an impact on the final probability to significantly change the answer from that obtained by applying the assumption. With terms treated independently, we can combine their probabilities with a product, giving the formula

$$p(c|t_{d,1}, t_{d,2}, \ldots, t_{d,n}) \propto p(c) \prod_{i=1}^{n} p(t_{d,i}|c)$$

This quantity is easily computed in a single pass through the training data, but it's actually possible to compute a column vector $\beta \in \mathbb{R}^n$ from the training data such that $\beta \cdot \vec{x}$ gives the log-odds that the vector $\vec{x}$ (representing a new document) belongs to class c [1, p. 347]. Thus the classification task is reduced to a single vector product. In order to derive a hard classification decision from the classifier, the log-odds are computed and converted to a probability p; if $p > 0.5$ then the document is assigned to class c, and to $\neg c$ otherwise.

Naive Bayes is often used as a baseline classifier because it's simple to implement, efficient to train, and efficient to use to classify new data. Although it may not accurately estimate $p(c|d)$, it often comes close enough to make a reasonable hard classification decision (i.e. the document is in c or not in c). 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 complete training. This enables a Naive Bayes classifier to work on larger training sets. It's usually the case, however, that there's a limited amount of data to train on, and *not* a limited amount of time in which to train. This is certainly the case for our particular problem, where we may begin with only a few positive examples of web pages belonging to a particular class. Next we'll consider logistic regression, which is still relatively efficient to train and apply to new instances, but more complicated to implement than Naive Bayes.

2.3 Logistic Regression

With logistic regression we are still trying to estimate $p(c|\vec{x})$, but the method for doing so is different. Logistic regression trains a classifier by explicitly maximizing the likelihood of $h(\beta \cdot \vec{x})$ for all $\vec{x}$ in the training data, where h is called the *hypothesis function*. Usually 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 a range from $-\infty$ to ∞ and a domain from 0 to 1, making it a convenient approximation of a probability. In contrast to Naive Bayes, logistic regression training involves iteratively improving β using some form of optimization. Without going into detail about any particular optimization algorithm, the basic idea is to start with some reasonable value for β, and to find a new value that results in a slightly better likelihood for the training data, then to iterate until β cannot be significantly improved. This final stage of the process is called convergence.

Convergence cannot be guaranteed, but this won't be a problem for most categorization 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 with a reasonable amount of effort. 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 tools and collapsed into a single variable. The third problem is related to over-fitting, and often leads to a perfectly-separable training set—one for which it's possible to perfectly classify every example.

In order to address this last problem, many practical implementations of logistic regression restrict the values of β to be close to zero. This restriction is called normalization, 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 normalization 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 β, the method and time required to classify a new document are essentially the same. A point in favor of logistic regression, however, is that its probability estimates are more accurate, making it a better *soft classifier*—one which can accurately predict how much more probable one class is than another.

2.4 Other Classifiers

I also researched several other classification algorithms, which I'll mention only briefly here before going on to discuss my implementations of and experiments with Naive Bayes and logistic regression text classification. The first alternative classification algorithm I investigated was support vector machines (SVMs) [6, p. 256], which share a great deal in common with logistic regression, but take a different theoretical approach to achieve what amounts to normalization. Support vector machines try to find a hyperplane which 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 [3], but they offer only moderate improvements over good logistic regression implementations despite greatly increased complexity and training time.

The second classification algorithm I considered and discarded was a neural network with hidden nodes [6, 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 (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's unclear how many hidden nodes should be used to train a text classifier that will work on different training sets. In addition to these drawbacks, I didn't encounter any evidence that neural networks are particularly suited to text classification, and so I decided not to pursue them any further.

Finally, I investigated ensemble methods, which attempt to combine the results of several different classifiers on the same input in order to balance out the strengths and weaknesses of each. I found evidence that ensemble methods generally produce results that are no worse than those of their best constituent classifier, and much better on some inputs [1, p. 376]. The trade-off, of course, is that these methods require training and running multiple classifiers for each possible class. I think these are a very promising avenue for improving classification accuracy, but I didn't have time this semester to implement them and perform my own experiments. If time permits, I'd like to revisit ensemble methods when I incorporate classification into Yioop.

2.5 Implementations and Experiments

The Yioop search engine is written entirely in PHP, and it's a project goal to minimize dependencies—especially those which require compilation. Consequently, the text classifier implementation should be written in PHP, which by default lacks an optimized linear algebra library. Even a simple and efficient text classification algorithm requires a significant amount of floating point calculation on moderately large training sets, so performance is an important issue. Furthermore, since one of our goals is to assist the user in finding web pages to be used in the training corpus, the classifier will have to run repeatedly on increasingly larger training sets, making the time to complete training another important issue.

This semester I implemented both a Naive Bayes text classifier and a logistic regression classifier, and carried out experiments to gauge their performance in terms of classification accuracy, training efficiency, and classification efficiency. I'll continue to work on these implementations, and ultimately they will be incorporated into Yioop, where they will be used to classify web pages at crawl time. All of my experiments thus far, however, have been on a corpus of spam and "ham" (not spam) email messages curated by the team that works on the open source SpamAssassin email spam filter. I used this corpus because I already had an Octave implementation of logistic regression built to work with it, whose performance I knew to be on par with the current state of the art. Thus the Octave implementation's performance could serve as a baseline by which to verify the PHP implementations.

Figure 1 gives the results of running my PHP implementations of Naive Bayes and lasso logistic regression on the spam corpus, as well as the results of running the Octave implementation of ridge logistic regression on the same corpus. I tested each PHP classifier on two different-sized training sets, with the smaller training set a random subset of the larger one. I also tested each of the PHP implementations with only the top 20% and top 6% of the features, selected according to the χ^2 feature selection algorithm. The PHP implementation of logistic regression was too slow to test on the full training set using either 100% or 20% of the features, so the only result I have is for 6%.

I found Naive Bayes to be as easy to implement as the literature makes it out to be, and the performance was exactly in line with what others have reported. On the small training set of 100 messages it achieved just less than 80% accuracy, and the accuracy goes up to approximately 95% on the larger training set of 4,000 messages. The fact that this simple algorithm gets up to 95% accuracy on the full training set suggests that the spam corpus might present a relatively "easy" text classification problem. Note that Naive Bayes seems to suffer more from aggressive feature selection than logistic regression

Classifier	Set	Feature Subset	Train	Classify	Accuracy
Naive Bayes	small	1.00	0.52s	0.19s	0.792
	small	0.20	0.25s	0.08s	0.762
	full	1.00	19.82s	0.21s	0.953
	full	0.20	7.26s	0.08s	0.900
	full	0.06	3.71s	0.05s	0.765
Log. Reg.	small	1.00	70.57s	0.20s	0.914
	small	0.20	1.76s	0.08s	0.924
	full	0.06	352.95s	0.05s	0.960
LR (Octave)	full	1.00	31.29s	0.00s	0.992

Figure 1: Results of running the PHP implementations of Naive Bayes and lasso logistic regression and the Octave version of ridge logistic regression on the spam dataset with various choices for the number of features and training examples to use. The full training set has 4,000 examples, 60% spam and 40% ham, and 1,895 features (distinct terms). The small training set is just a subset of the full one, with 100 examples and 1,649 features. The test set contains 1,000 instances, 692 of which are spam. The classification times are the time to classify *all* test instances, and all times do not include the time to tokenize the input.

does, indicating that it's dependent on small contributions from many terms to make the correct hard classification decision.

In contrast to Naive Bayes, my best implementation of logistic regression obtained approximately 91% accuracy on the small training set, and 96% accuracy on the full training set (with aggressive feature selection). In contrast to Naive Bayes, the logistic regression classifier's accuracy seems to be unaffected by feature selection, and actually benefits from it on the small training set, when using only 20% of the features. The Octave implementation of logistic regression—which uses a more sophisticated optimization routine—achieved approximately 99% accuracy on the full training set with no feature selection.

I had a much harder time implementing logistic regression classification in PHP than I did implementing Naive Bayes. This should be expected from the relative complexities of the two algorithms, but the optimization routine gave me a particularly hard time. The optimization algorithm used to iteratively improve β turns out to be the heart of a successful logistic regression classifier, and I underestimated the work that it would take to select, comprehend, and implement one. In the end I tried two separate approaches—one classifier based on simple gradient descent, and another based on lasso logistic regression using the cyclic coordinate descent optimization algorithm CLG [2].

The basic gradient descent approach ended up being far too fragile. It would often fail to converge unless the parameters were tuned precisely for a particular training set, and even when it could be made to converge, it would usually take a long time. These limitations led me to give up on this approach. The lasso logistic regression implementation, on the other hand, turned out to be much more robust, but far slower on similarly-sized training sets. It was prohibitively slow on the full-sized spam training set, and so my only results for logistic regression there had to come from a drastically-reduced set of features. Even using just 6% of the features, logistic regression took nearly six minutes to train on the full set, versus 20 seconds for the Naive Bayes classifier to train with *all* of the features, or just four seconds to train with 6% of them. Note however, that the time to classify is approximately the same in all cases, since it involves a simple dot product between β and the new instance.

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 [7]. Because PHP is often an order of magnitude slower than comparable compiled languages such as C++ for CPU-bound applications, and because in the course of building a training corpus we can expect the same classifier to be trained repeatedly on small but increasingly-larger training sets, I think that it will be important to trade off some accuracy for faster training times, and that feature selection is a promising route for accomplishing that. Just using Naive Bayes (which trains very quickly) would be an effective alternative, but the extra accuracy gained from using logistic regression seems worth pursuing. Furthermore, the fact that the Octave implementation managed to achieve 99% accuracy and took just ten more seconds to train than the Naive Bayes classifier gives me hope that logistic regression can be implemented more efficiently in PHP.

3 Training and Using Classifiers

This section outlines how the classifier will fit into the larger framework, and how exactly the system will assist the administrator to construct a training set. We consider the user interface changes that will need to be made to Yioop in order to allow administrators to train and select classifiers, as well as the (relatively simple) changes that will need to be made to classify pages at crawl time. Because I haven't actually implemented the interface yet, the plans outlined in this discussion are only tentative, and lack detail in some places.

3.1 Training

The first steps in training a classifier are to select the classes which should be identified, and then to build a training set. For reasons which I'll go into below, building a training set may be an involved process, so it will make sense to work on one class at a time. Furthermore, if the target classes and their corresponding training sets are kept independent of one another then it will be possible to easily create mixes of various classes, reusing the work done for each individual class several times. The high-level user interface components will support these actions:

- Create a classifier for a new class

- Delete a classifier
- Train a classifier
- Select a combination of classes to be used for a new crawl

In order to accurately classify new web pages the classifier will require a sufficiently large training set providing both positive and negative examples of the desired class. In some cases a training set may already exist, or may be easy to build because class examples are in abundant supply, but in other cases positive examples may be relatively rare and hard to find. Thus a major component of the user interface for training a classifier will be a mechanism to choose any of the following methods for specifying positive and negative training examples:

- A list of URLs which all belong to the same class
- A previous crawl of pages which all belong to the same class
- A query which retrieves at least one positive example of the target class, but which may return negative examples as well

The specified examples will be unified into a seed training set on which an initial classifier can be trained. It may be the case that the Yioop administrator was able to supply enough positive and negative training examples to build an accurate classifier and test it, in which case there's no need to search out more examples—the classifier can simply be used in the next crawl as-is. If the administrator is trying to identify a novel class, however, he or she will likely need some help searching for additional examples of that class.

The seed classifier can be used to automatically classify more pages drawn from a selected crawl index, with the goal of adding more positive and negative examples to the training set. Once a sufficient number of new examples have been added to the training set the classifier can be re-trained with the new data, hopefully yielding a more accurate classifier. This process can be repeated until either no more positive examples are found or the classifier has reached a sufficient level of accuracy.

A key step of the proposed procedure is identifying new positive and negative examples. Early classifiers which haven't been trained on very many examples will likely be wrong fairly often. Only a human can identify and correct misclassified pages, so the training interface will need to support human supervision of classification decisions. I propose that as new pages are found and classified they are presented to the administrator in a priority queue where pages which have been assigned with greater confidence to a class are displayed first. The interface might look a lot like a list of search results, but with the addition of three buttons which enable the administrator to provide feedback for each page:

- One button to indicate that the page is a positive example
- A second button to indicate that the page is a negative example

- And a third button to withhold judgement and remove the page from the list, in the case that the administrator can't decide whether it belongs to the class or not

This scheme allows the administrator to provide feedback on the classification of a page with a single click and—unless the administrator withholds judgement—provides the classifier with one new training example for each decision. I expect the main difficulty with this approach will be finding new positive examples in the selected index which have not already been classified. This will require both going further and further out into the results for a query and generating new relevant queries. The former task depends on the efficiency of Yioop's retrieval engine, and the latter on the administrator or on a procedure for generating new queries. I spent some time this semester investigating approaches for modifying queries to return more relevant documents [4], but I haven't yet carried out any experiments. I plan to begin by placing the burden on the administrator, then trying to remove that responsibility if time permits.

In any case, once a classifier for each positive class[2] has been trained, the last major task for the administrator is to specify which classifiers to use in the next crawl. All that's needed is a list of available classifiers, and the ability to add them to—and remove them from—a list of *active* classifiers.

3.2 Classifying Pages

We've been assuming that we can recognize two or more positive classes by training multiple single-class classifiers independently, but we haven't yet said how we'll combine these classifiers. The solution is a simple algorithm commonly called one-vs-rest [6, p. 306]. The basic idea is to, given a new instance to classify, try all of the independent classifiers separately and choose the class that's assigned with the most confidence. This requires running as many classifiers as there are potential classes, but all of the classifiers we'll use can assign a class to a new instance very efficiently, so this won't be a problem in practice.

A real difficulty, however, is dealing with new web pages which don't properly belong to *any* class of interest. In this case one of the trained classifiers will have greater confidence than the others, but we don't actually want to assign the corresponding class. One solution is to require that the maximum confidence be above a threshold before a class can be assigned, which is analogous to the normal procedure when dealing with a single class. The threshold is usually set to 0.5.

Once a positive class label has been selected, the corresponding meta tag must be added to the page. The tag will have the format "meta:class:*label*", where *label* is the label for the assigned class, and will enable users to restrict

[2]By a *positive class* I mean one which is not simply the negation of some other class. For example, if the domain is restricted to email messages, it would suffice to train one classifier to recognize spam, then label everything that isn't classified as spam as ham. In this case there would be only one positive class—*spam*. If, on the other hand, we were trying to identify two particular types of web pages, and there were some pages which belonged to *neither* class, then we would need two separate classifiers to recognize the two positive classes.

their queries to pages which have been determined to belong to a particular class. For example, a user might search for the syllabus for a class on information retrieval with the query "information retrieval meta:class:syllabus".

Another potentially useful meta tag would provide the confidence with which a class has been assigned (e.g. "meta:class-confidence:73"). This information could be used to eliminate all but the most likely class members, but its practical utility would depend on the soft classification accuracy of the classifier that assigned the class. Naive Bayes classifiers, for instance, make particularly bad soft classifiers [1, p. 348] because they aggregate many small factors which tend to polarize the final confidence score. Even with good soft classification accuracy, Yioop would require changes to take advantage of this information, as it's presently impossible to search for meta tags by any other relation than equality. Consequently, restricting a search to pages classified with confidence greater than 0.80 would require specifying the disjunction of "meta:class-confidence:$\{80, 81, \ldots, 100\}$.

4 Remaining Work

This semester I focused on experimenting with classifiers and—toward the end of the semester—feature selection, which can drastically reduce the number of variables a classifier must consider. I created a small framework for testing classifier and feature selection implementations, and implemented a Naive Bayes classifier, a lasso logistic regression classifier, and feature selection using the χ^2 statistic. I experimented with these implementations on a corpus of spam and ham email messages, and found that their performance was in line with what others have reported.

In the next and last semester I will integrate these classifier and feature selection implementations with Yioop, build the user interface discussed in the previous section, and modify Yioop to use a collection of classifiers to classify web pages at crawl time. If time permits, I'd like to continue to experiment with alternative optimization algorithms for logistic regression and term weight statistics for feature selection.

References

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

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

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

[4] I. Ruthven and M. Lalmas. A survey on the use of relevance feedback for information access systems. *The Knowledge Engineering Review*, 18(02):95–145, 2003.

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

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

[7] Y. Yang and J.O. 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.