Journal of Machine Learning Research 10 (2009) 2571-2613 Submitted 12/07; Revised 1/09; Published 11/09

Learning When Concepts Abound

Omid Madani MADANI@AI.SRI.COM
SRI International, AI Center
333 Ravenswood Ave
Menlo Park, CA 94025

Michael Connor CONNOR2@UIUC.EDU
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Wiley Greiner W.GREINER@LASOFT.COM
Los Angeles Software Inc
Santa Monica, CA 90405

Editor: Ralf Herbrich

Abstract

Many learning tasks, such as large-scale text categorization and word prediction, can benefit from efficient training and classification when the number of classes, in addition to instances and features, is large, that is, in the thousands and beyond. We investigate the learning of sparse class *indices* to address this challenge. An index is a mapping from features to classes. We compare the index-learning methods against other techniques, including one-versus-rest and top-down classification using perceptrons and support vector machines. We find that index learning is highly advantageous for space and time efficiency, at both training and classification times. Moreover, this approach yields similar and at times better accuracies. On problems with hundreds of thousands of instances and thousands of classes, the index is learned in minutes, while other methods can take hours or days. As we explain, the design of the learning update enables conveniently constraining each feature to connect to a small subset of the classes in the index. This constraint is crucial for scalability. Given an instance with l active (positive-valued) features, each feature on average connecting to d classes in the index (in the order of 10s in our experiments), update and classification take $O(dl\log(dl))$.

Keywords: index learning, many-class learning, multiclass learning, online learning, text categorization

1. Introduction

A fundamental activity of intelligence is to repeatedly and rapidly categorize. Categorization (classification or prediction) has a number of uses; in particular, categorization enables inferences and the taking of appropriate actions in different situations. Advanced intelligence, whether of animals or artificial systems, may require effectively working with myriad classes (concepts or categories). How can a system quickly classify when the number of classes is huge (Figure 1), that is, in the thousands and beyond? In nature, this problem of rapid classification in the presence of many classes may have been addressed (for evidence of fast classification in the visual domain, see Thorpe et al. 1996 and Grill-Spector and Kanwisher 2005). Furthermore, ideally, we seek systems that *efficiently*

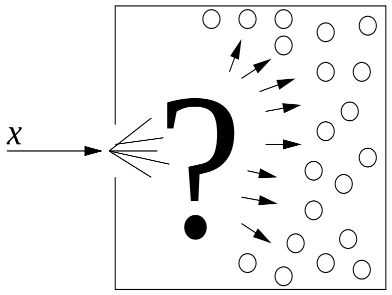

Figure 1: The problem of quick classification in the presence of myriad classes: How can a system quickly classify a given instance, specified by a feature vector $x \in R^n$, into a small subset of classes from among possibly millions of candidate classes (shown by small circles)? How can a system *efficiently learn* to quickly classify?

learn to efficiently classify in the presence of myriad classes. Many tasks can be viewed as instantiations of this *large-scale many-class* learning problem, including: (1) classifying text fragments (such as queries, advertisements, news articles, or web pages) into a large collection of categories, such as the ones in the Yahoo! topic hierarchy (`http://dir.yahoo.com`) or the Open Directory Project (`http://dmoz.org`) (e.g., Dumais and Chen, 2000; Liu et al., 2005; Madani et al., 2007; Xue et al., 2008), (2) statistical language modeling and similar prediction problems (e.g., Goodman, 2001; Even-Zohar and Roth, 2000; Madani et al., 2009), and (3) determining the visual categories for image tagging, object recognition, and multimedia retrieval (e.g., Wang et al., 2001; Forsyth and Ponce, 2003; Fidler and Leonardis, 2007; Chua et al., 2009; Aradhye et al., 2009). The following realization is important: in many prediction tasks, such as predicting words in text (statistical language modeling), training data is abundant because the class labels are not costly, that is, *the source of class feedback (the labels) need not be explicit assignment by humans* (see also Section 3.1).

To classify an instance, applying binary classifiers, one by one, to determine the correct class(es) is quickly rendered impractical with increasing number of classes. Moreover, learning binary classifiers can be too costly with large numbers of classes and instances (millions and beyond). Other techniques, such as nearest neighbors, can suffer similar drawbacks, such as prohibitive space requirements, possibly slow classification speeds, or poor generalization. Ideally, we desire scalable discriminative learning methods that learn compact classification systems that attain adequate accuracy.

One idea for achieving quick classification is to use the features of the given instance as cues to dramatically reduce the space of possibilities, that is, to build and update a mapping, or an *index*, from features to the classes. We explore this idea in this work. An index here is a weighted bipartite graph that connects each feature to zero or more classes. During classification, given an instance containing certain features, the index is used ("looked up") much like a typical inverted index for document retrieval would be. Here, classes are retrieved and ranked by the scores that they obtain during retrieval, as we describe. The ranking or the scores can then be used for class assignment. In this work, we explore the learning of such cues and connections, which we refer to as *index learning*. For this approach to be effective overall, roughly, two properties need to hold: To achieve adequate accuracy and efficiency and in many problems arising in practice, (1) each feature need only connect to a relatively small number of classes, and (2) these connections can be discovered efficiently. We provide empirical evidence for these conjectures by presenting efficient and competitive indexing algorithms.

We design our algorithms to efficiently learn sparse indices that yield accurate class rankings. As we explain, the computations may best be viewed as being carried out from the side of features. During learning, each feature determines to which relatively few classes it should lend its weights (votes) to, subject to (space) efficiency constraints. This parsimony in connections is achieved by a kind of *sparsity-preserving* updates. Given an instance with l active (i.e., positive-valued) features, each feature on average connecting to d classes in the index, update and classification take $O(dl\log(dl))$ operations. d is in the order of 10s in our experiments. The approach we develop uses ideas from online learning and multiclass learning, including mistake driven and margin-based updates, and expert aggregation (e.g., (e.g., Rosenblatt, 1958; Genest and Zidek, 1986; Littlestone, 1988; Crammer and Singer, 2003a), as well as the idea of the inverted index, a core data structure in information retrieval (e.g., Witten et al., 1994; Turtle and Flood, 1995; Baeza-Yates and Ribeiro-Neto, 1999).

We empirically compare our algorithms to one-versus-rest and top-down classifier based methods (e.g., Rifkin and Klautau, 2004; Liu et al., 2005; Dumais and Chen, 2000), and to the first proposal for index learning by Madani et al. (2007). We use linear classifiers—perceptrons and support vector machines—in the one-versus-rest and top-down methods. One-versus-rest is a simple strategy that has been shown to be quite competitive in accuracy in multiclass settings, when properly regularized binary classifiers are used (Rifkin and Klautau, 2004), and linear support vector machines achieve the state of the art in accuracy in many text classification problems (e.g., Sebastiani, 2002; Lewis et al., 2004). Hierarchical training and classification is a fairly scalable and conceptually simple method that has commonly been used for large-scale text categorization (e.g., Koller and Sahami, 1997; Dumais and Chen, 2000; Dekel et al., 2003; Liu et al., 2005).

In our experiments on six text categorization data sets and one word prediction problem, we find that the index is learned in seconds or minutes, while the other methods can take hours or days. The index learned is more efficient in its use of space than those of the other classification systems, and yields quicker classification time. Very importantly, we find that budgeting the connections of the features is a major factor in rendering the approach scalable. We explain how the design of the update makes this budget enforcement convenient. We have observed that the accuracies are as good as and at times better than the best of the other methods that we tested. As we explain, methods based on binary classifiers, such as one-versus-rest and top-down, are at a disadvantage in our many-class tasks, not just in terms of efficiency but also in accuracy. The indexing approach is simple: it requires neither taxonomies, nor extra feature reduction preprocessing. Thus, we believe that index learning offers a viable option for various many-class settings.

The contribution of this paper include:

- Raising the problem of large-scale many-class learning, with the goal of achieving both efficient classification and efficient training

- Proposing and exploring index learning, and developing a novel weight-update method in the process

- Empirically comparing index learning to several commonly used techniques, on a range of small and large problems and under several evaluation measures of accuracy and space and time efficiency, and providing evidence that very scalable systems are possible without sacrificing accuracy

This paper is organized as follows. In Section 2, we discuss related work. In Section 3, we describe and motivate the learning problem, independent of the solution strategy. We explain the index, and describe our implementation and measures of index quality, in terms of both accuracy and efficiency. We then report on the NP-hardness of a formalization of the index learning problem. In Section 4, we present our index learning approach. Throughout this section, we discuss and motivate the choices in the design of the algorithms. In particular, the consideration of what each feature should do in isolation turns out to be very useful. In Section 5, we briefly describe the other methods we compare against, including the one-versus-rest and top-down methods. In Section 6, we present a variety of experiments. We report on comparisons among the techniques and our observations on the effects of parameter choices and tradeoffs. In Section 7, we summarize and provide concluding thoughts. In the appendices, we present a proof of NP-hardness and additional experiments.

2. Related Work

Related work includes multiclass learning and online learning, expert methods, indexing, streaming algorithms, and concepts in cognitive psychology.

There exists much work on multiclass learning, including nearest neighbors approaches, naive Bayes, support vector machine variants, one-versus-rest and output-codes (see, for example, Hastie et al., 2001; Rennie et al., 2003; Dietterich and Bakiri, 1995); however, the focus has not been scalability to very large numbers of classes.

Multiclass online algorithms with the goal of obtaining good rankings include the multiclass and multilabel perceptron (MMP) algorithm (Crammer and Singer, 2003a) and subsequent work (e.g., Crammer and Singer, 2003b; Crammer et al., 2006). These algorithms are very flexible and include both additive and multiplicative variants, and may optimize an objective in each update; some variants can incorporate non-linear kernel techniques. We may refer to them as prototype methods because the operations (such as weight adjustments and imposing various constraints) can be viewed as being performed on the (prototype) weight vector for each class. In our indexing algorithms it is the features that update and normalize their connections to the classes. This difference is motivated by efficiency (for further details, see Sections 4.1 and 4.4, and the experiments). Similar to the perceptron algorithm (Rosenblatt, 1958), we use a variant of mistake driven updating. The variant is based on trying to achieve and preserve a margin during online updating. Learning to improve or optimize some measure of margin has been shown to improve generalization (Vapnik, 2000). On use of margin for online methods, see for instance Krauth and Mezard (1987), Anlauf and Biehl (1989), Freund and Schapire (1999), Gentile (2001), Li and Long (2002), Li et al. (2002), Crammer et al. (2006) and Carvalho and Cohen (2006). In our setting, a simple example shows that keeping a margin can be beneficial over pure mistake-driven updating even when considering a single feature in isolation (Section 4.3.1).

The indexing approach in its focus on features (predictors) has similarities with additive models and tree-induction algorithms (Hastie et al., 2001), and may be viewed as a variant of so-called expert (opinion or forecast) aggregation and weight learning (e.g., Mesterharm, 2000; Freund et al., 1997; Cesa-Bianchi et al., 1997; Vovk, 1990; Genest and Zidek, 1986). In the standard experts problems, all or most experts provide their output, and the output is usually binary or a probability (the outcome to predict is binary). In our setting, a relatively small set of features are active in each instance, and only those features are used for voting and ranking. In this respect, the problem is in the setting of the "sleeping" or "specialist" experts scenarios (Freund et al., 1997; Cohen and Singer, 1999). Differences or special properties of our setting include the fact that here each expert provides a partial class-ranking with its votes, the votes can change over time (not fixed), and the pattern of change is dependent on the algorithm used (the experts are not "autonomous"). In a multiclass calendar scheduling task (Blum, 1997), Blum investigates an algorithm in which each feature votes for (connects to) the majority class in the past 5 classes seen for that feature (the classes of the most recent 5 instances in which the feature was active). This design choice was due to the temporal (drifting) nature of the learning task. Feature weights for the goodness of the features are learned (in a multiplicative or Winnow style manner). Mesterharm refers to such features (or experts) as sub-experts (Mesterharm, 2000, 2001), as the performance can be significantly enhanced by learning a good weighting for mixing (aggregating) the experts' votes,[1] and it is shown how different linear threshold algorithms can be extended to the multiclass weight learning setting. The classifier is referred to as a *linear-max* classifier, since the maximum scoring class is assigned to the instance (as opposed to a linear-threshold classifier). Mesterharm's work includes the case where the experts may cast probabilities for each class, but the focus is not on how the features may compute such probabilities (it is assumed the experts are given). Learning different weights for the features can complement indexing techniques. Section 4.3.2 gives a limited form of differential expert weighting (see also Madani, 2007a).

The one-versus-rest technique (e.g., Rifkin and Klautau, 2004) and use of a class hierarchy (taxonomy) (e.g., Liu et al., 2005; Dumais and Chen, 2000; Koller and Sahami, 1997) for top-down training are simple intuitive techniques commonly used for text categorization. The use of the structure of a taxonomy for training and classification offers a number of efficiency and/or accuracy advantages (Koller and Sahami, 1997; Liu et al., 2005; Dumais and Chen, 2000; Dekel et al., 2003; Xue et al., 2008), but also can present several drawbacks. Issues such as multiple taxonomies, evolving taxonomies, unnecessary intermediate categories on the path from the root to deeper categories, or unavailability of a taxonomy are all difficulties for the tree-based approaches. In our experiments, we find that index learning offers both several efficiency advantages and ease of use (Section 6). No taxonomy or separate feature-reduction pre-processing is required. Indeed, our method can be viewed as a feature selection or reduction method. On the other hand, researchers have shown some accuracy advantages from the use of the taxonomy structure (e.g., top-down) compared to "flat" one-versus-rest training (in addition to efficiency) (e.g., Dumais and Chen, 2000; Liu et al., 2005; Dekel et al., 2003) (this depends somewhat on the particular method and the loss used). Our current indexing approach is flat (but see Huang et al. 2008, for a two-stage nonlinear method using fast index learning for the first stage). One advantage that classifier-based methods such as one-versus-rest and top-down may offer is that the training can be highly parallelized: learning of each binary classifier can be carried out independent of the others.

The inverted index, for instance from terms to documents, is a fundamental data structure in information retrieval (Witten et al., 1994; Baeza-Yates and Ribeiro-Neto, 1999). Akin to the TFIDF weight representation and variants, the index learned is also weighted. However, in our case, the classes (to be indexed), unlike the documents, are implicit, indirectly specified by the training instances (the instances are not the "documents" to be indexed), and the index construction becomes

1. Theoretical work often focuses the analysis on learning the best expert, and the use of term "subexpert" is introduced by Mesterharm to differentiate.

a learning problem. As one simple consequence, the presence of a feature in a training instance that belongs to class *c* does not imply that the feature will point to class *c* in the index learned. We give a baseline algorithm, similar to TFIDF index construction in its independent computation of weights, in Section 4.2. Indexing has also been used to speed up nearest neighbor methods, classification, and retrieval and matching schemes (e.g., Grobelnik and Mladenic, 1998; Bayardo et al., 2007; Fidler and Leonardis, 2007). Indexing could be used to index already trained (say linear) classifiers, but the issues of space and time efficient learning remain, and accuracy can suffer when using binary classifiers for class ranking (see Section 6.1). Learning of an unweighted index was introduced by Madani et al. (2007), in which the problem of efficient classification under myriad classes was motivated. This two-stage approach is explained in Section 6.4.1, and we see in Section 6.4.1 that learning a weighted index to improve ranking appears to be a better strategy than the original approach in terms of accuracy, as well as simplicity and efficiency. Subsequent work on indexing by Madani and Huang (2008) explores further variations and advances to feature updating (e.g., supporting nonstationarity and hinge-loss minimization), taking as a starting point the findings of this work on the benefits of efficient feature updating. It also includes comparisons with additional multiclass approaches. This paper is an extension of the work by Madani and Connor (2008).

The field of data-streaming algorithms studies methods for efficiently computing statistics of interest over data streams, for example, reporting the items with proportions exceeding a threshold, or the highest *k* proportion items (sometimes called "hot-list" or "iceberg" queries). This is to be achieved under certain efficiency constraints, for example, with at most two passes over the data and poly logarithmic space (e.g., see Fang et al., 1998; Gibbons and Matias, 1999). Note that in the case of a single feature, if we only value good rankings, computing weights may not be necessary, but in the general case of multiple features, the weights become the votes given to each class, and are essential in significantly improving the final rankings. An algorithm similar to our single-feature update for the Boolean case is used as a subroutine by Karp et al. (2003), for efficiently computing most frequent items. In some scenarios, drifts in proportions can exist, and then online and possibly competitive measures of performance may become important (Borodin and El Yaniv, 1998; Albers and Westbrook, 1998). In this ranking and drifting respect, the feature-update task has similarities with online list-serving and caching (Borodin and El Yaniv, 1998), although we may assume that the sequence is randomly ordered (at minimum, not ordered by an adversary). Some connections and differences between goals in machine learning research and space-efficient streaming and online computations are discussed by Guha and McGregor (2007).

Statistical language modeling and similar prediction tasks are often accomplished by n-gram (Markov) models (Goodman, 2001), but the supervised (or discriminative) approach may provide superior performance due to its potential for effectively aggregating richer feature sets (Even-Zohar and Roth, 2000; Madani et al., 2009). Prior work has focused on discriminating within a small (confusion) set of possibilities. In the related task of *prediction games* (Madani, 2007a,b), Madani proposes and explores an integrated learning activity in which a system builds its own classes to be predicted and to help predict. That approach involves large-scale long-term online learning, where the number of concepts grows over time, and can exceed millions.

Concepts and various phenomena associated with them have been studied extensively in cognitive psychology (e.g., Murphy, 2002). A general question that motivated our work, and that appears heretofore uninvestigated, is the question of computational processes required for a system to effectively deal with a huge number of concepts. Three prominent theories on the nature of the representation of concepts are the classical theory (logical representations), the exemplar theory (akin to nearest neighbors), and the prototype theory (akin to linear feature-based representations). Prototype theory is perhaps the most successful in explaining various observed phenomena regarding human concepts (Murphy, 2002). Interestingly, our work suggests a predictor-based representation for efficient recall/recognition purposes, that is, the representation of a concept, at a minimum for recall/retrieval purposes, is distributed among the features (predictors or cues). However, the predictor-based representation remains closest to the prototype theory.

3. Many-Class Learning and Indexing

In this section, we first present the learning setting and introduce some notation in the process. Next, we motivate many-class learning and the indexing approach. In Section 3.2, we define the index and how it is implemented and used in this work. We then present our accuracy and efficiency evaluation measures in Section 3.3. Before moving to index learning (Section 4), we analyze the computational complexity of a formulation of index learning in Section 3.4.

A learning problem consists of a collection S of instances, where S can denote a finite set, or, in the online setting, a sequence of instances. Each training instance is specified by a vector of feature values, $\mathbf{v_x}$, as well as a class (or assigned label) that the instance belongs to,[2] c_x. Thus each instance x is a pair $\langle \mathbf{v_x}, c_x \rangle$. $\mathcal{F}$ and $\mathcal{C}$ denote respectively the set of all features and classes. Our proposed algorithms ignore features with nonpositive value,[3] and in our experiments feature values range in $[0,1]$. $\mathbf{v_x}[f]$ denotes the value of feature f in the vector of features of instance x, where $\mathbf{v_x}[f] \geq 0$. If $\mathbf{v_x}[f] > 0$, we say feature f is *active* (in instance x), and denote this aspect by $f \in x$. Thus, an instance may be viewed as a set of active features, and the input problem may be seen as a tripartite graph (Figure 2). The number of active features is denoted by $|x|$. We also use the expression $x \in c$ to denote that instance x belongs to class c (c is a class of x).

As an example, in text categorization, a "document" (e.g., an email, an advertisement, a news article, etc.) is typically translated to a vector by a "bag of words" method as follows. Each term (e.g., "ball", "cat", "the", ”victory", ...) is assigned an exclusive unique integer id. The finite set of words (more generally phrases or ngrams), those appearing in at least one document in the data set, comprise the set of features $\mathcal{F}$. Thus the vector $\mathbf{v_x}$ corresponding to a document lives in an $|\mathcal{F}|$ dimensional space, where $\mathbf{v_x}[i] = k$ iff the word with id i (corresponding to dimension i) appears k times in the document, where $k \geq 0$ (other possibilities for feature values include Boolean and TFIDF weighting). Therefore, in typical text categorization tasks, the number of active features in an instance is the number of unique words that appear in the corresponding document. The documents in the training set are assigned zero or more true class ids as well. Section 6 describes further the feature representation and other aspects of our experimental data. For background on machine learning in particular when applied to text classification, please refer to Sebastiani (2002) or Lewis et al. (2004).

2. In this paper, to keep the treatment focused, and for simplicity of evaluation and algorithm description, we treat the multiclass but single class (label) per instance setting. However, two of our seven data sets include multilabel instances. Whenever necessary, we briefly note the changes needed, for example, to the algorithm, to handle multiple labels. However, the multilabel setting may require additional treatment for better accuracy.
3. A partial remedy is to replace each feature that can also have negative values by two features, one having value $\max(v,0)$, the other $\max(0,-v)$.

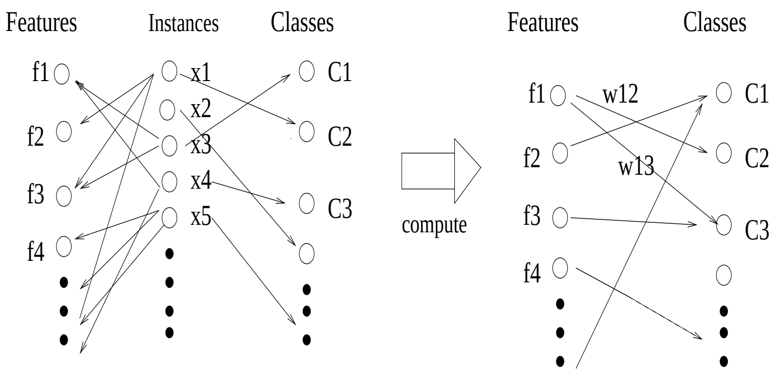

Figure 2: A depiction of the problem: the input can be viewed as a tripartite graph, possibly weighted, and perhaps only seen one instance at a time in an online manner. Our goal is to learn an accurate efficient index, that is, a sparse weighted bipartite graph that connects each feature to zero or more classes, such that an adequate level of accuracy is achieved when the index is used for classification. The instances are ephemeral: they serve only as intermediaries in affecting the connections from features to classes. The index to learn is also equivalent to a sparse weight matrix (in which the entries are nonnegative in our current work) (see Sections 3.2 and 3.2.1).

3.1 The Level of Human Involvement in Teaching and Many-Class Learning

Learning under myriad-classes is not confined to a few text-classification problems. There are a number of tasks that could be viewed as problems with many classes and, if effective many-class methods are developed, such an interpretation can be quite useful. In terms of the sources of the classes, we may roughly distinguish supervised learning problems along the following dimensions (the roles of the teacher):

1. The source that defines the classes of interest, that is, the space of the target classes to predict.
2. The source of supervisory feedback, that is, the source or the process that assigns to each instance one or more class labels, using the defined set of classes. This is necessary for the procurement of training data, for supervised learning.

In many familiar cases, the classes are both human-defined and human-assigned. These include typical text classification problems (e.g., see Lewis et al. 2004 and Open Directory Project or Yahoo! directories/topics). In many others, class assignment is achieved by some “natural” or indirect activity, that is, the “labeling” process is not as explicit or controlled. The labeling is a by-product of an activity carried out for other purposes. One example of this case is data sets obtained from news groups postings (e.g., Lang, 1995). In this case, users post or reply to messages, without necessarily verifying whether their message is topically relevant to the group. Another example problem is predicting words using the context that the word appears (the words are the classes). In these problems, the set of the classes of interest may be viewed as human-defined, but the labeling is implicit (collections of written or spoken texts in the word prediction task). The extreme case where both the set of classes and the labeling is achieved with little or no human involvement is also possible, and we believe very important. For instance, Madani (2007b,a) explores tasks in which it is (primarily) the machine that builds its own many concepts, through experience, and learns prediction connections among them. This is a kind of *autonomous* learning. As human involvement and control diminishes over the learning process, the amount of noise tends to increase. However, training data as well as the number of classes can increase significantly. We have used the term "many-class" (in contrast to multiclass) to emphasize this aspect of the large number of classes in these problems.

Thus, in large-scale many-class learning problems, all the three sets S, $\mathcal{C}$, and $\mathcal{F}$ can be huge. For instance, in experiments reported here, $\mathcal{C}$ and $\mathcal{F}$ can be in the tens of thousands, and S can be in the millions. S can be an infinite stream of instances and $\mathcal{C}$ and $\mathcal{F}$ can grow indefinitely in some tasks (e.g., Madani, 2007a). While $\mathcal{F}$ can be large (e.g., hundreds of thousands), in many applications such as text classification, instances tend to be relatively sparse: relatively a few of the features (tens or hundreds) are active in each instance.

The number of classes is so large that indexing them, not unlike the inverted index used for retrieval of documents and other object types, is a plausible approach. An important difference from traditional indexing is that classes, unlike documents, are implicit, specified only by the instances that belong to them. An index is a common technique for fast retrieval and classification, for instance to speed up nearest neighbor or nearest centroid computations (e.g., Grobelnik and Mladenic, 1998; Bayardo et al., 2007; Gabrilovich and Markovitch, 2007; Fidler and Leonardis, 2007). Also, for fast classification when there is a large number of classes, after one-versus-rest training of linear binary classifiers (see Section 5 on one-versus-rest training), a natural and perhaps necessary technique is to index the weights, that is, to build an index mapping each feature to those classifiers in which the feature has nonzero weight. This approach is indirect and does not adequately address efficient classification and space efficiency,[4] and the problem of slow training time for one-versus-rest training remains. Here, we propose to learn the index edges as well as their weights directly. For good classification performance as well as efficiency, we need to be very selective in the choice of the index entries, that is, which connections to create and with what weights. Figure 3 presents the basic cycle of categorization via index look up and learning via index updating (adjustments to connection weights). We have termed the system that is learned a *Recall System* (Madani et al., 2007): a system that, when presented with an instance, quickly "recalls" the appropriate classes from a potentially huge number of possibilities.

3.2 Index Definition, Implementation, and Use

The use of the index for retrieval, scoring, and ranking (classification) is similar to the use of inverted indexes for document retrieval. Here, features "index" classes instead of documents. In our implementation, for each feature there corresponds exactly one list that contains information about the feature's connections (similar to inverted or posting lists Witten et al. 1994 and Baeza-Yates and Ribeiro-Neto 1999). The list may be empty. Each entry in the list corresponds to a class that the feature is *connected* to. An entry in the list for feature f contains the id of a class c, as well as the *connection or edge weight* $w_{f,c}$, $w_{f,c} > 0$. Each class has at most one entry in a feature's list. If a class c doesn't have an entry in the list for feature f, then $w_{f,c}$ is taken to be 0. The connection

4. Our experiments show that if we do not drop some of the connections during learning, training and classification time and space consumption suffer significantly.

```
Basic Mode of Operation:
 Repeat
   1. Get next instance $x$
   2. Retrieve, score, and rank classes via
      active features of $x$
   3. If update condition is met:
      3.1 Update index.
   4. Zero (reset) the scores of the retrieved classes.
                    (a)
```

```
Algorithm RankedRetrieval($x$, $d_{max}$)
 /* initially, for each class $c$, its score $s_c$ is zero */
 1. For each active feature $f$ (i.e., $\mathbf{v_x}[f] > 0$):
      For the first $d_{max}$ classes with highest
      connection weight to $f$:
      1.1. $s_c \leftarrow s_c + (r_f \times w_{f,c} \times \mathbf{v_x}[f])$
 2. Return those classes with nonzero score,
    ranked by score.
                    (b)
```

Figure 3: (a) The cycle of classification and learning (updating). During pure classification (e.g., when testing), step 3 is skipped. See part (b) and Section 3.2 for how to use the index, and Section 4 for when and how to update the index. (b) The algorithm that uses a weighted index for retrieving and scoring classes. See Section 3.2.

weights are updated during learning. Our index learning algorithms keep the lists small for space and time efficiency (as we explain in Section 4.1). For ease of updating and efficiency, the lists are doubly linked circular dynamic lists in our implementation, and are kept sorted by weight.

Figure 3(b) shows how the index is used, via a procedure that we name RankedRetrieval. On presentation of an instance, the active features score the classes that they are connected to. The score that a class c receives, s_c, can be written as

$$s_c = \sum_{f \in x} r_f \times w_{f,c} \times \mathbf{v_x}[f], \tag{1}$$

where r_f is a measure of the predictiveness power or the *rating* of feature f, and we describe a method for computing it in Section 4.3.2. Currently, for simplicity, we may assume the rating is 1 for all features.[5] Note that the sum need only go over the entries in the list for each active feature (other weights are zero). We use a hash map to efficiently update the class scores during scoring. In a sense, each active feature casts votes for a subset of the classes, and those classes receive and tally their incoming votes (scores). In this work, the scores of the retrieved classes are positive. The positive scoring classes can then be ranked by their score, or, if it suffices, only the maximum scoring class can be kept track of and reported. Note that if negative scores (or edge weights) were allowed, then, when some true class obtains a negative score, the system would potentially have to process (i.e., retrieve or update) all zero scoring classes as well, hurting efficiency (this depends on how update and classification are defined). The scores of the retrieved classes are reset to 0 before the next call to RankedRetrieval.

On instance x, and with d connections per feature in the index, there can be at most $|\mathbf{v_x}|d$ unique classes scored. The average computation time of RankedRetrieval is thus $O(d|\mathbf{v_x}|\log(d|\mathbf{v_x}|))$, where d denotes the average number of connections of a randomly picked feature (from a randomly picked instance). In our implementation, for each active feature, only at most the d_{max} classes (25 in our experiments) with highest connection weights to the feature participate in scoring.

5. After index learning, r_f can be incorporated into the connection weights $w_{f,c}$.

3.2.1 GRAPH AND LINEAR-ALGEBRAIC VIEWS OF THE INDEX

A useful way of viewing the index is as a directed weighted bipartite graph (Figure 2): on one side there are features (one node per feature) and on the other side there are the classes. The index maps (connects) each feature to a subset of zero or more classes. An edge connecting feature f to class c has a positive weight denoted by $w_{f,c}$, or $w_{i,j}$ for feature i and class j, and corresponds to a list entry in the list for feature f. Absent edges have zero weight. The *outdegree* of a feature is the number of (outgoing) edges of the feature. Small feature outdegrees translates to efficiency in retrieval (and updating as we will see).

In addition to the graph-theoretic view, the index can also be seen as a sparse non-negative (weight) matrix $\mathbf{W}$. Let the rows correspond to the features and let the columns correspond to the classes. Retrieval or classification involves efficiently computing[6] the vector of class scores $\mathbf{v_x^T W}$, and post-processing the resulting (sparse) score vector (e.g., sorting the positive scoring classes). Efficiency constraints translate to limiting the number of nonzero entries in each row. In the indexing algorithms of this paper, the sum of the entries in each row does not exceed 1. Lemma 1 below states that this restriction does not lose power, among the set of nonnegative matrices, for achieving good rankings.

3.3 Evaluating the Index

We evaluate index learning based on efficiency as well as the quality of classification (accuracy). In large-scale learning, both memory and time efficiency are important, and both at training as well as classification times.[7] Our other goal is to maintain satisfactory accuracy. In our experiments in Section 6.2 (on finite samples), we report on three measures of efficiency: training time T_{tr}, the size of the index learned, denoted by $|\mathbf{W}|$, meaning the number of edges or nonzero weights in the index, and the average number of edges d touched (processed) per feature during classification (a measure of work/speed during classification time). We next describe our classification accuracy measures.

We use the standard accuracy (i.e., one minus zero-one error), here denoted R_1, as well as other measures of ranking quality. R_1 allows us to compare to other published results. A method for ranking classes, given an instance x, outputs a sorted list of zero or more classes. In addition to weighted indices, we describe other methods for ranking the classes in Section 5. An instance may belong to multiple classes in some tasks (two of our data sets in Figure 8). To simplify evaluation and presentation, in this paper we only consider the highest ranked true class. Let k_x be the rank of the highest ranked true class after presenting instance x to the system. Thus $k_x \in \{1,2,3,\cdots\}$. If the true class does not appear in the ranked list, then $k_x = \infty$. We use R_k to denote *recall at (rank) k*, which measures the proportion of (test) instances for which one of the true classes ended in the top k classes:

$$R_k = \text{ recall at } k = E_x[k_x \leq k],$$

where E_x denotes expectation over the instance distribution and $[k_x \leq k] = 1$ iff $k_x \leq k$, and 0 otherwise (Iverson bracket). So we get a reward of 1 if the true class is within top k for a given instance, 0 otherwise, and R_k is the expectation. In our experiments, we will report on (average) recall at rank 1, R_1, and recall at rank 5, R_5, on held-out sets. R_1 is simply the standard accuracy, that is,

6. Feature ratings, can be incorporated in a diagonal matrix $\mathbf{R}$, where $\mathbf{R}[\mathbf{i},\mathbf{i}] = \mathbf{r_i}$ (the rating of feature i) and $\mathbf{R}[\mathbf{i},\mathbf{j}] = \mathbf{0}$, when $i \neq j$. Obtaining the scores would then be $\mathbf{v_x^T R W}$.
7. Note that in online learning, there isn't a sharp separation between the training and testing phases.

the highest ranked class is assigned to the instance, and R_1 measures the proportions of instances to which the true class was assigned.

We also report on the harmonic (mean) rank (HR) (reciprocal of mean reciprocal rank or MRR), defined as:

$$MRR = E_x \frac{1}{k_x}, \text{ and } HR = MRR^{-1}.$$

MRR gives a reward of 1 if a correct class is ranked highest, the reward drops to 1/2 at rank 2, and slowly goes down the higher the k (the lower the rank). If the right class is not retrieved, the reward is 0. MRR is the expectation or the empirical average of such reward over (test) instances, and we simply invert it to get a measure of ranking performance, the harmonic rank HR. The lower the HR, the better, and it has a minimum of 1 (rank 1 is best). MRR is a commonly used measure in information retrieval, such as in question answering tasks (e.g., Radev et al., 2002). In our experiments, we report the HR values so that the reader can quickly get an impression of the average class-ranking performance of the various methods.

Both R_k and MRR are appropriate for settings in which we value better ranks *significantly* more than worse ranks. Thus, if an index is perfect half the time, that is, ranks the correct class of the given instance at top (rank 1) half the time, but fully fails the rest of the time, that is, does not retrieve the correct class at all, then its HR value is 2. However, for an index that always retrieves the correct class, but ranks it third, the HR value is worse, at 3. Note that one could raise the fraction $\frac{1}{k_x}$ to a different exponent (instead of 1) to shift the emphasis in one direction or another. R_k does not reflect the quality of ranking within top k, and it simply cuts the reward off if the right class is outside top k. HR is a smoother measure. Our evaluation measure are from the point of view of an instance to be classified. This is appropriate with large numbers of classes and in many applications, such as personalization or text prediction, in which a given instance (a query, a page, etc.) should be classified into one or a few classes that the system is confident about. In a number of information retrieval tasks such as question answering and document retrieval, the extra emphasis on higher ranks is well motivated. We expect that the situation would be similar for typical many-class problems, such as text categorization. The common precision and recall measures used in machine learning are often computed from the point of view of a class: for each class, the instances are ranked according to the classifier's scores for the class. This is especially appropriate when we are interested in performance on a single class at a time. For instance, when we seek to rank or filter instances based on their degree of membership in a given class of interest (e.g., a news topic). Our indexing techniques are more appropriate for the problem of obtaining good rankings per instance, similar to some other multiclass ranking algorithms (e.g., Crammer and Singer, 2003a). However, existing techniques for improving precision/recall for imbalanced classes may be applicable (e.g., Li et al., 2002). We conclude this section with a simplifying property of non-negative matrices, for the purposes of ranking.

Lemma 1 *Let* $\mathbf{W}$ *be the non-negative matrix corresponding to an index (features correspond to the rows and classes are the columns). The ranking that* $\mathbf{W}$ *produces on nonzero scoring classes is not changed under positive scaling, that is,* $\alpha\mathbf{W}$*, for* $\alpha > 0$*, produces the same ranking.*

Proof The score for each class is obtained in the vector $\mathbf{v_x^T W}$. Therefore, the ranking obtained from $\mathbf{v_x^T}\alpha\mathbf{W} = \alpha\mathbf{v_x^T W}$, is the same as the ranking in the vector $\mathbf{v_x^T W}$, when $\alpha > 0$ and all entries in $\mathbf{v_x^T W}$ are non-negative. ■

The lemma implies that optimal matrices, for the objective of say maximizing R_1 on the training set, among non-negative matrices in which the entries in each row sum to at most 1.0, exist. The indexing algorithms presented in this paper learn non-negative weight matrices.

3.4 Computational Complexity of Index Learning

Can we efficiently compute an index achieving maximum training accuracy given any finite set S of instances? If we constrain the outdegree of each feature to be below a given constant (motivated by space and time efficiency), then the corresponding decision problem is NP-hard under plausible objectives such as optimizing accuracy (R_1):

Theorem 2 *The index learning problem with the objective of either maximizing accuracy (R_1) or minimizing HR on a given set of instances, and with the constraint of a constant upper bound (e.g., 1) on the outdegree of each feature is NP-Hard.*

The proof is by a reduction from the minimum cover problem (see Appendix A). A problem involving only two classes is shown NP-hard. We do not know whether the indexing problem is approximable in polynomial time however, or whether removing the constraint on the outdegree alters the complexity. Linear programming formulations exist with continuous objectives and no explicit outdegree constraint (Madani and Connor, 2007; Madani and Huang, 2008).

The next section describes very efficient online algorithms that perform well in our experiments. We motivate our choices in the algorithm design, but leave theoretical guarantees to future work.

4. Feature Focus Algorithms for Index Learning

Figure 4 present our main index learning technique. After first giving a quick overview of the approach, we motivate the choices in the design of the algorithm in the rest of this section.

On a given instance, after the use of the index for scoring and ranking (an invocation of RankedRetrieval), if a measure of *margin* (to be described shortly) is not large enough, an update to the index is made. The margin is the score obtained by the true class, minus the highest scoring incorrect (negative) class (either of the two scores can be zero). Our index learning algorithms may be best described as performing their updates from the features' side or features' "point of view" (rather than the classes' side or class prototypes), and hence we name the whole family *feature-focus* algorithms. As we will explain, this design was motivated by considerations of efficiency (Sections 4.1 and 4.4). The basic question for each feature is to which subset of classes it should connect (possibly none), and with what weights. Figure 4(d) gives a generic feature updating scheme and Figure 4(c) gives the instantiation we use in our experiments. Initially, all weights are zero. Note that when a weight is zeroed, the connection is removed. This means that, in our index implementation, the list entry corresponding to the edge is removed from the list of the edges of the feature.

We next motivate the design choices in FF. The problem of what each feature in isolation should do during learning turns out to be helpful and we first explore and discuss this single feature case. We then present the IND(ependent) method, a baseline in which effectively on every instance every feature updates. We then motivate mistake-driven updating, and in particular the use of margin.[8]

8. All the examples given to illustrate various aspects make the assumption of Boolean feature values, but the feature-focus algorithm as presented works with the more general nonnegative values.

```
/* The FF Algorithm */
Algorithm FeatureFocus($x$, $w_{min}$, $d_{max}$, $\delta_m$)
  1. RankedRetrieval($x$, $d_{max}$). /* retrieve/score */
  2. Compute the margin $\delta$:
     $\delta = s_{c_x} - s'_x$, where $s'_x = \max_{c \neq c_x} s_c$.
  3. If $\delta > \delta_m$, return. /* update not necessary */
  4. Otherwise, for each active $f \in x$:
     /* update active features' connections */
      4.1  FSU($x$, $f$, $w_{min}$).
                          (a)
```

```
Algorithm RankedRetrieval($x$, $d_{max}$)
  /* initially, for each class $c$, its score $s_c$ is zero */
  1. For each active feature $f$ (i.e., $\mathbf{v_x}[f] > 0$):
     For the first $d_{max}$ classes with highest
      connection weight to $f$:
       1.1. $s_c \leftarrow s_c + (r_f \times w_{f,c} \times \mathbf{v_x}[f])$
  2. Return those classes with nonzero score,
      ranked by score.
                          (b)
```

```
/* Feature Streaming Update (allowing "leaks") */
Algorithm FSU($x$, $f$, $w_{min}$) /* Single feature updating */
  1. $w'_{f,c_x} \leftarrow w'_{f,c_x} + \mathbf{v_x}[f]$ /* increase weight to $c_x$. */
  2. $w'_f \leftarrow w'_f + \mathbf{v_x}[f]$ /* increase total out-weight */
  3. $\forall c, w_{f,c} \leftarrow \frac{w'_{f,c}}{w'_f}$ /* (re)compute proportions */
  4. If $w_{f,c} < w_{min}$, then /* drop tiny weights */
       $w_{f,c} \leftarrow 0, w'_{f,c} \leftarrow 0$
                          (c)
```

```
Algorithm GenericWeightUpdate
 Each active feature:
  1. Strengthens weight to true class
  2. Weakens other class connections
  3. Drops weak edges (tiny weights)
                          (d)
```

Figure 4: (a) Pseudo-code for the Feature-Focus (FF) learning algorithm. The FF algorithm is invoked on every training instance. This corresponds to steps 2 and 3 in Figure 3(a). (b) The RankedRetrieval procedure for scoring and ranking (copied from Figure 3(b)). (c) Feature streaming update, or FSU: The connection weight of f to the true class c_x is strengthened. Others connections are weakened due to the division. All the weights are zero at the beginning of index learning. (e) Generic weight updating: on each training instance, each active feature strengthens its weight to the true class, weakens its other connections, and drops those that are too weak.

We conclude with a comparison of FF to existing online algorithms, in particular the perceptron algorithm and Winnow. The reader may wish to skip some of these sections at this point and go to the experiments (Section 6) on a first reading.

4.1 Updating for a Single Feature

Assume (training) instances arrive in a streaming fashion (from some infinite source), and assume the single label (per instance) setting. Fix one feature and imagine the substream of instances that have that feature active. Let us consider Boolean feature values only ($\mathbf{v_x}[f] \in \{0,1\}$) here for simplicity. Thus, we basically obtain a stream of observed classes, $<c^{(1)}, c^{(2)}, c^{(3)}, \cdots>$, for the given feature. Ignoring other features for now, and considering efficiency constraints, to which classes should this feature connect to, and with what weights? We next argue that our objective of a good ranking, subject to efficiency, reduces to computing the proportion in the sequence for those classes (if any) that exceed a desired proportion threshold.

In this single feature case, classes are ranked by the weight assigned to them by the feature. The constraint (of space efficiency) is that the feature may connect to only a subset of all possible classes, say d_{max} at most. The question is how the feature should connect so that an objective such as R_k or HR (harmonic rank) is maximized. We will focus on the scenario where the stream of classes is generated by an iid drawing from a fixed probability distribution.

It is not hard to verify that the best classes are the d_{max} classes with the highest proportions in the stream, or the highest $P(c)$ if the distribution is fixed and known (more precisely, $P(c|f)$, but f is fixed here) and the ranking should also be by $P(c)$. For a finite sequence on which we are allowed to compute proportions before having to connect the feature, this can easily be established.

Lemma 3 *A finite sequence of classes is given (class observations). To maximize HR, when the feature can connect to at most k different classes, a k highest frequency set of classes should be picked, that is, choose S, such that $|S| = k$ and $S = \{c|n_c \geq n_{c'}, \forall c' \notin S\}$), where n_c denotes the number of times c occurs in the sequence. The classes in S should be ordered by their occurrence counts to maximize HR. The same set maximizes R_k.*

Proof This can be established by a simple “swapping” or “exchange” argument. We look at the sum of rewards over the sequence rather than averages, as the sequence length is fixed. Consider maximizing R_k first. Let n_c denote the number of times class c appears in the sequence. For any chosen set S of size k, a pair of classes (c, c') is out of order if $n_c < n_{c'}$, but $c \in S$, and $c' \notin S$. Then R_k for S is improved if c is replaced by c', the improvement is $n_{c'} - n_c$. Similarly HR is improved for an ordered set S if a pair like above exists (improvement of $(n_{c'} - n_c)\frac{1}{j}$, where j denotes the rank of c in S), or a pair within the chosen set is out of order (improvement of $(n_{c'} - n_c)(1/j - 1/j')$, where $j', j' > j$, denotes the old rank of c'.). ■

For unbounded streams generated by iid drawing of classes from a fixed distribution over a finite number of classes, the empirical proportions of classes, over the sequence seen so far (of length at least k), take the place of the counts, in order to maximize expected HR or expected R_k on the unseen portion of the sequence.

We will use FSU (*Feature Streaming Update*, Figure 4(c)) in our main feature-focus algorithm. An FSU update takes at most two list traversals (involving finding or inserting the connection). With d connections per feature, a full update on an instance takes $\tilde{O}(d|x|)$. Note that when features are Boolean, FSU simply computes edge weights that approximate the conditional probabilities $P(c|f)$ (the probability that instance $x \in c$ given that $f \in x$ and FSU is invoked). Since the weights are between 0 and 1 and approximate probabilities, it eases the decision of assessing importance of a connection: weights below w_{min} are dropped at the expense of some potential loss in accuracy. FSU keeps total counts (w'_f and w'_{f,c_x}, which we will describe and motivate later). Note that w_{min} effectively bounds the maximum outdegree during learning to be $\frac{1}{w_{min}}$. We note that this space efficiency of FSU is central to making feature-focus algorithms space and time efficient (see Section 6.3.1). Given that FSU zeros some weights during its computation, it is instructive to look at how well it does in approximating proportions for the (sub)stream of classes that it processes for a single feature. This gives us an idea of how to set the w_{min} parameter and what to expect. Appendix A presents synthetic experiments and a discussion. To summarize, when the true probability (weight) w of interest is several multiples of w_{min}, with sufficient sample size, the chance of dropping it is very low (the probability quickly goes down to 0 with increasing $\frac{w}{w_{min}}$), and moreover, the computed weight is also close to the true conditional. See Section 6.3.3 on the effect of choice of $w_{min} \in \{0.001, 0.01, 0.1\}$ on accuracy on several data sets.

4.1.1 UNINFORMATIVE FEATURES, ADAPTABILITY, AND DRIFTING ISSUES

In FSU, we keep and update two sets of weights, the edge weights $w_{f,c}$ (not greater than 1), $w'_{f,c}$, as well as total weight w'_f. In case of binary features ($\mathbf{v}_\mathbf{x}[f] = 1$), we can simply think of w'_f as total count of times FSU has been invoked for the feature, and $w'_{f,c}$ as an under-estimate of the co-occurrence count in that stream ($w'_{f,c}$ can be less than the co-occurrence count, as it is reset to 0 if the edge is dropped). Note that if c_x is not already connected (for example in the beginning), $w_{f,c}$ and $w'_{f,c}$ are 0. An important point is that total weight w'_f is never reduced. This is useful as a way of down-weighing uninformative features (such as "the"). Thus, due to edge dropping, we may have the sum of proportions remain less than 1, $\sum_c w_{f,c} < 1$, even when $w'_f > 0$. We have found this alternative slightly better in our experiments than the case in which $w'_f = \sum_c w'_{f,c}$ (i.e., when w'_f is kept as the exact sum of the weights). See Section 6.3.5.

In case of non-Boolean feature values, similar to perceptron and Winnow updates (Rosenblatt, 1958; Littlestone, 1988), the degree of activity of the feature, $\mathbf{v}_\mathbf{x}[f]$, affects how much the connection between the feature and the true class is strengthened. We could use a *learning rate*, a multiplier for $\mathbf{v}_\mathbf{x}[f]$, to further control the aggressiveness of the updates. We have not experimented with that option.

Note also that as w_f grows, the feature may become less adaptive, as a new class will have to occur more frequently to obtain a strong weight ratio with respect to w_f. In particular, after $w_f > \frac{1}{w_{min}}$, a new class will be immediately dropped.[9] For long-term online learning, where distributions can drift (nonstationarity), this can slow or stop adaptation, and updates that effectively keep a finite memory or history are more appropriate. Note also that, if the same training instances can be seen multiple times (e.g., in multiple passes on finite data sets), with w_f growing, the fitting capability of the algorithms is curbed. This may be desired as a means of overfitting prevention. Other indexing updates have been developed, offering various trade-offs (see our discussion in Section 4.4, and Madani and Huang 2008, and Madani et al. 2009, in particular for a simple update appropriate for nonstationarity).

Before describing the main feature-focus algorithm, we describe a baseline algorithm we refer to as IND(ependent). This algorithm can be implemented in an offline (batch) manner. It is based on computing the conditionals $P(c|f)$.

4.2 Always Updating (the IND Algorithm)

One method of index construction is to simply assign each edge the class conditional probabilities, $P(c|f)$ (the conditional probability that instance $x \in c$ given that $f \in x$). This can be computed for each feature independent of other features. We refer to this variant as the IND ("INDependent") algorithm (Figure 5). Features are treated as Boolean here ($\mathbf{v}_\mathbf{x}[f] \in \{0,1\}$). After processing the training set (computing counts and then conditional probabilities), only weights exceeding a threshold p_{ind} are kept. The use of p_{ind} not only leads to space savings, but also can improve accuracy significantly. The best threshold p_{ind} (for improving accuracy) is often significantly greater than 0 (see Section 6.3.7). In our experiments with IND, we choose the best threshold by observing performance on a random 20% subset of the training set. We thus implemented the IND algorithm

9. At this point, updates can only affect classes already connected, and updates may improve the accuracy of their assigned weights, though there is a small chance that even classes with significant weights may be eventually dropped (this has probability 1 over an infinite sequence!). In any case, at this point or soon after, it is possible to stop updating. In our experiments, with finite data and small number of passes over the data sets, this was not an issue.

```
Algorithm IND($S$, $p_{ind}$) /* IND algorithm */
  1. For each instance $x$ in training sample $S$:
    1.1 For each $f \in x$:  /* increment counts for f */
      1.1.1 $n_f \leftarrow n_f + 1$
      1.1.2 $n_{f,c_x} \leftarrow n_{f,c_x} + 1$
  2. Build the index: for each feature $f$ and class $c$:
      2.1 $w \leftarrow \frac{n_{f,c}}{n_f}$.
      2.2 If $w \geq p_{ind}$, $w_{f,c} \leftarrow w$. (otherwise $w_{f,c} \leftarrow 0$. )
```

Figure 5: Pseudo-code for the IND(ependent) algorithm, implemented for the case of Boolean features only. The choice of p_{ind} affects accuracy significantly, and is picked using a held out set (see Section 4.2).

as a batch algorithm, that is, we computed the weights $P(c|f)$ exactly, not in an online streaming manner described[10] for FSU. The exact computation can be done on the relatively smaller data sets. IND is in fact the fastest algorithm on the smaller data sets, since the count updates are simple and there is no call to index retrieval during training. This counting phase for index construction can also be distributed. On larger data sets, IND runs into memory problems and becomes very slow during training, due to many features keeping connections to too many classes.[11] This aspect points to the importance of space efficiency for large-scale learning.

The IND algorithm, in its independent computations of weights for each feature, has similarities with the multiclass Naive Bayes algorithm (e.g., Rennie et al. 2003). Major differences include the computation of $P(f|c)$ (the reverse) in plain multiclass Naive Bayes, and that for classification, we are summing the weights (instead of multiplying under the independence assumption), similar to some techniques for expert opinion aggregation (Genest and Zidek, 1986; Cesa-Bianchi et al., 1997). We have found that summing improves accuracy. See Madani and Connor (2007) for a more detailed comparison to multiclass Naive Bayes. In its independent computation of weights, IND is also similar to inverted index construction using, for instance, TFIDF.

IND offers a nice baseline, but we can potentially do significantly better than computing proportions for each feature independently. Often features are inter-dependent. For instance, features can be near duplicates or redundant. In particular, with increasing feature vector sizes, the accuracy of methods that in effect assume feature independence can degrade significantly.

4.3 Mistake-Driven Updating Using a Margin (the FF Algorithm)

FF adds and drops edges and modifies edge weights during learning by processing one instance at a time,[12] and by invoking a feature updating algorithm, such as FSU. Unlike IND, FF addresses feature dependencies by not updating the index on every training instance. Equivalently, a feature updates its connection on only a fraction of the training instances in which it is active. This is motivated and explained next.

10. In case the instance belongs to multiple classes, step 1.1.2 is executed for each true class.
11. However, note that the FSU algorithm can be instead employed here to keep memory consumption in check.
12. The feature and class sets can also grow incrementally.

4.3.1 WHEN TO UPDATE?

FSU should not be invoked on every training instance. In particular, *"lazy"* or mistake-driven updating (not updating all the time) can, to some extent, address issues with feature dependencies. It can, for example, avoid over counting the influence of features that are basically duplicates by learning relatively low connection weights for each such feature (similar to a rational for mistake driven updates in other learning algorithms such as the perceptron). We next give a simple scenario, *case 1*, to demonstrate accuracy improvements that can be obtained by lazy updating.

Case 1. Imagine the simple case of two classes, c_1 and c_2, and two Boolean features, f_1 and f_2. Assume f_1 is perfect for c_1, $P(c_1|f_1) = P(f_1|c_1) = 1$, but that f_2 appears in instances of both classes, and $P(f_2|c_1) = 1$ (i.e., f_2 appears in all instances of c_1), but also $P(f_2|c_2) = 1$. Then, given only f_2, that is, an instance $x = \{f_2\}$ (x contains f_2 only), we want to rank c_2 higher. Now, if say $P(c_1) > P(c_2)$ (c_1 is more frequent than c_2), and we always invoked FSU, f_2 would also give a higher weight to c_1, ranking c_1 higher than c_2 on $x \in c_2$. An optimal solution, for accuracy R_1 or for HR, has the property that f_2 has a higher connection weight to c_2 than to c_1 (with $w_{f_1,c_2} = 0$, an optimal solution satisfies: $w_{f_1,c_1} > w_{f_2,c_2} > w_{f_2,c_1}$.). Now, if FF invoked FSU only when the correct class was not ranked highest, the connection weights in this example would converge to an optimal configuration. To see this, note that as soon as $x \in c_1$ is seen f_1 obtains a weight of 1 to c_1. Next, only updates on $x \in c_2$ will be performed, since c_1 is ranked correctly due to f_1 having a weight of 1 and f_2 keeping some nonzero weight to it. f_2 makes a stronger connection to c_2 than c_1 after at most 2 FSU invocations. R_1 in the optimal case would be 1.0 here, while it can approach 0.5 if we always update. Note that as fewer updates in general mean fewer connections (sparser indices), we may also save in space in this lazy update regime (see Section 6).

On the other hand, if we don't update at all when the right class is at rank 1, we may also suffer from suboptimal performance. This happens even in the case of a single feature. Thus *"proactive"* updating is useful too. The next case elaborates.

Case 2. Consider the single feature case and three classes c_1, c_2, c_3, where $P(c_1) = 0.5$, while $P(c_2) = P(c_3) = 0.25$. Thus c_1 should be ranked highest, for say maximizing R_1. This yields optimal $R_1 = 0.5$, and if we always invoke FSU, this will be the case after a few updates (we will soon get $w_{1,1} \approx 0.5$, and $w_{1,2} \approx w_{1,3} \approx 0.25$). If we don't update when true class is at rank 1, c_2 or c_3 can easily take the place of c_1 when an instance $x \in c_2$ or $x \in c_3$ is presented, but we need to reverse the situation subsequently when $x' \in c_1$ is presented, and instances belonging to c_1 are more frequent. In general, the connection weights from the feature to c_1, c_2, and c_3 will be similar in the purely mistake-driven updating regime, and on sequences that look like the worst case alternating sequence: $c_1, c_2, c_1, c_3, c_1, c_2, \cdots$, the running value of R_1 can approach 0. While random sequences are not as bad, we should still expect significant inferior performance. On randomly generated sample of size 2000 according to above class distribution, averaging over 100 80-20 splits, always (proactive) updating gave an average R_1 performance of 0.479 ± 0.02 (standard deviation of 0.02), while the lazy update gave 0.428 ± 0.09.

Therefore, not updating when the rank of the right class is adequate may cause unnecessary instability in behavior and inferior performance as well. Of course, we desire an algorithm that can perform well in the single feature case. Continued updating even when the true class is ranked highest is akin to keeping a kind of extended memory (in the connection weights).

We strike a balance between the two desirables by using the notion of *margin*. The margin on the current instance is the score of the positive class minus the score of the highest scoring negative class:

$$\delta = s_{c_x} - s'_x, \text{ where } s_{c_x} \geq 0, s'_x \geq 0, s'_x = \max_{c \neq c_x} s_c.$$

If the margin δ does not exceed a desired margin threshold δ_m, we update[13] (invoke FSU). Note that both s_{c_x} and s'_x can be 0. If we set the margin threshold to 0, we may fit more instances in the training set, and handle situations like case 1, but underperform for case 2 situations. With a sufficiently high margin, updates are always made and case 2 is covered, but fitting power (case 1) can suffer. There is a tradeoff, and a good question is what the best choice of threshold may be? The best choice depends on the problem and the feature vector representation. Individual edge weights are in the $[0,1]$ range, and when the instances are l_2 normalized, we have observed that on average top classes obtain scores in the $[0,1]$ range as well, irrespective of data sets or choice of margin threshold (Madani and Connor, 2007).

Our use of margin is somewhat similar to the use of margin for online algorithms such as perceptron and Winnow (e.g., Carvalho and Cohen, 2006; Crammer and Singer, 2003a), although our particular motivation from considering case 2, "stability" or keeping some "extended memory" for each feature, appears to be different.

4.3.2 RATING THE FEATURES: DOWN-WEIGH INFREQUENT FEATURES

It may be a good idea to down-weigh or eliminate those features' votes that are only seen a few times during learning, as their proportion estimates (connection weights) can be inaccurate and in particular higher than what they should be. Consider the first time FSU is invoked on a feature. After that update, such a feature gives a weight of 1 (the maximum possible) to the class it gets connected to. This is undesired. Of course, how much to down-weigh can depend on the problem, and how feature values are distributed. In our experiments, during scoring of the class, we multiply a feature's score for class c, $w_{f,c}$, by a rating r_f (see Equation 1 in Section 3.2), $r_f = \min(1, \frac{\#_f}{10})$, where $\#_f \geq 1$ denotes the number of times feature f has been seen so far. $\#_f$ is computed only during the first pass over training data. We show that on some problems, this option improves accuracy.

4.4 Summary and Relations to other Methods

The FF algorithm aggregates the votes of each features for ranking and classification. During learning, FF may be viewed as directing a stream of classes to each feature, so that each feature can compute weights for a subset of the classes that it may connect to. The stream, for example with the use of margin, may be hard to characterize and may show drifts during learning: it may initially be those instances in which the feature is active, but later it may correspond to a subsequence of those instances which are somewhat hard to classify. Features may be space constrained: they need to be space efficient in the number of connections they make as well as in computing their connection weights. This efficiency aspect is especially important in large-scale many-class learning.

The FF algorithm has similarities with online algorithms such as Winnow (Littlestone, 1988), as it normalizes (in general weakens some of) the weights, and the perceptron algorithm (Rosenblatt, 1958), as for example the updates are in part additive (ignoring the normalization or weakening). The important difference that changes the nature of the algorithm is that changes to weights are

13. For instances with multiple true classes, the margin is computed for each positive true class. Every active feature is updated for each true class for which its margin is below the margin threshold.

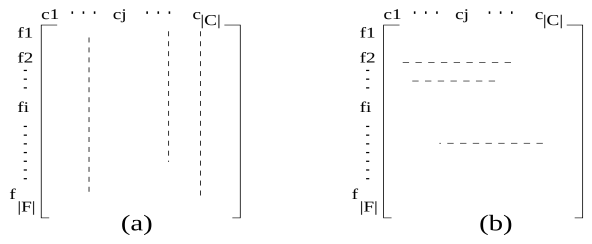

Figure 6: In learning a weight matrix for multiclass learning (here the features corresponding to the rows), prototype methods operate on the columns (part a), for instance in normalizing the (column) weight vectors, while feature-focus methods operate on the rows (part b), for example in ensuring that the number of nonzeros in each row remain within a budget.

done from the side of features, unlike Winnow or perceptron. The Winnow algorithm does the normalization from the side of the classes: each class is represented by a classifier (a class prototype), and each classifier has its feature weights normalized after each update. If normalization is done for all features, many features, whether or not active, get weakened. In a sense, the classifier ranks the features in order of importance for its own concept. A number of learning algorithms in the family of linear classifier learning algorithms, focus on the class side, for example, learning a prototype classifier for each class (e.g., Crammer and Singer, 2003a) (see Figure 6). This is a natural approach for binary classification. In our case, it is the classes whose connections to a feature may be weakened due to one or more classes being strengthened. In the FSU update given in this paper, this weakening happens irrespective of whether a class was ranked high (this aspect is similar to Winnow, but again, for class weights instead of feature weights). Alternative feature-focus updates are possible (e.g., Madani and Huang, 2008). It is best to view each feature as a voter or "expert", and the goal is to obtain good class rankings for each instance by adjusting the votes. A prototype for a class is more appropriate for ranking instances for that class.

To keep memory consumption in check, it seems most direct to constrain features not to connect to more than say 10s of classes, rather than somehow constraining the classes (class prototypes). It appeared harder to us to bound the number of features a class needs, and different classes may require widely varying number of features for adequate performance. See Raghavan et al. (2007) for an exploration of the number of useful features that different (binary) learning problems need for achieving (nearly) maximum accuracy. We also note that in many problems of interest, the number of classes, while large, is significantly smaller than the number of features. In many domains, as the number of classes grows, the number of features tends to grow too (possibly in a proportional manner). In the best of worlds, each feature could be predictive for at most one class. While reality is far from this idealized picture, and we anticipate many interactions, expecting that features may not require high outdegrees for good accuracy, can be a good first assumption. A fruitful future avenue may be exploring this assumption via modeling and developing theoretical arguments.

A second related reason is that constraining the feature outdegrees to remain relatively small (e.g., 10s) appears easier to implement and more time efficient in an online processing regime. Again, a class may require 100s of features and beyond for good performance. Therefore processing the needed classes, to examine importance of features, can take more time, per instance. Finally, we seek rapid categorization per instance, and constraining indegree of classes may not guarantee that the outdegree of commonly occurring features would be small. Constraining the degrees of classes is not directly related to the average time required for processing an instance. Given that an index is required for efficient classification, that is, efficient access from features to the relevant classes, one would need additional data structures (additional memory) for efficient prototype processing.

For the perceptron update, continued updating can increase weight magnitudes with no bound. This makes designing an effective weight management criterion difficult. False positive classes may obtain negative connections to features they weren't connected to (when ranked higher than the true positive). These extra connections hurt sparsity. Moreover negative connections may not be as useful in the task of ranking multiple classes, to the extent that they are useful in the binary-class case and when learning a single prototype, for ranking instances: in the many-class case, the true class could simply have higher positive weights to the appropriate features. Of course, our discussion does not preclude efficient algorithms that, nevertheless, perform their operations from the class side.

5. Techniques Based on Binary Classifier Training

We compare against hierarchical or top-down training and classification, a commonly used method when a taxonomy of classes, a tree from general classes at the top to specific classes, is available (Dumais and Chen, 2000; Liu et al., 2005). The hierarchical method reduces to one-versus-rest classification when the classes are flat (when there is just one level), which is another common method for multiclass classification (e.g., Rifkin and Klautau, 2004). We compare against one-versus-rest on relatively small sets, to see how indexing performs in more traditional classification settings. Note that the FF algorithm, while motivated by many-class learning, is a linear method applicable to few classes and in particular binary classification as well.

The one-versus-rest method simply trains a binary classifier for each class using all the data. During classification, all the classifiers are applied and their scores rather than classification outcomes are used for ranking.[14] We observed no advantage in obtaining probabilities here compared to using raw scores. The one-versus-rest method becomes quickly inefficient, at both training and classification times, as the number of classes increases (as all the classifiers need to be applied to a given instance).

Linear classifiers such as support vector machines (SVMs) often perform the best in very high dimensional problems such as text classification (Lewis et al., 2004; Sebastiani, 2002). We tested perceptrons and SVMs in one-versus-rest and top-down methods. We use single pass and multiple pass perceptrons (Rosenblatt, 1958) as well as committees of them. Here, each perceptron in the committee is represented as a sparse vector and random weight initialization, in $[-1,1]$, is used when a new feature is added to the prototype. Unless specified, we run the perceptron learning algorithm until the 0/1 error on training is not improved (computed at the end of each pass), for 5 consecutive passes. Perceptron committees often obtain performance close to SVMs (e.g., Carvalho and Cohen, 2006), although their training time can be less.

14. Note that the use of index for classification is one-versus-rest (or "flat" classification), but the index was not obtained by training binary classifiers.

```
Algorithm TopDownProbabilities($x$, $c$, $p$, $\tilde{C}_x$)
  1. For each class $c_i$ that is a child of $c$:
    1.1 $p_{c_i} \leftarrow p \times P_{c_i}(x)$. /* obtain probability */
    1.2 If $p_{c_i} \geq p_{min}$
      1.2.1 $\tilde{C}_x \leftarrow \tilde{C}_x \cup \{(c_i, p_{c_i})\}$
      1.2.2 TopDownProbabilities($x$, $c_i$, $p_{c_i}$, $\tilde{C}_x$)
```

Figure 7: Pseudo-code for top-down classification. $P_{c_i}(x)$ denotes the probability assigned to x by the classifier trained for c_i in the tree. For each instance x, the first call is TopDownProbabilities(x, root, 1.0, $\{\}$).

5.1 Hierarchical Training and Classification

Briefly, hierarchical training works by first training classifiers for the first level classes in a one-vs-rest manner (e.g., Dumais and Chen, 2000). Then the same procedure can be repeated for the children of each class residing in the 2nd level (in general, the level below), training each classifier only on the instances that belong to one of the siblings. Only the classifiers for the top level classes will be trained on all the instances. For ranking and categorization using the hierarchical approach, we use classifier probabilities. We obtain probabilities from classifier scores by the method of sigmoid fitting (Platt, 1999). This may require additional training time for improved accuracy. In the experiments, we report on the effect of increasing the number of sigmoid-fitting trials on one of the data sets (Reuters RCV1).

During classification, whenever a classifier is applied, we use the probability it assigns. The probabilities are multiplied along a path top-down (Figure 7). A path of candidate classes is terminated if the probability falls under some threshold p_{min}. All the classifier at the first level (corresponding to the classes at the top level) are applied to a given instance. During test time, we tried several thresholds: $p_{min} = 0.05 + 0.05k, k = 1, 2, \cdots$, and report results on the threshold giving highest accuracy R_1. All our ranking methods are evaluated on the deepest classes an instance is assigned to. For the evaluation of the top-down method, from the list of candidates obtained for a given test instance, any class whose child is also in the list is removed, and the remaining classes are sorted by their assigned probabilities. For the list of the true classes of the test instance, again only those true classes with no child in the list are kept. Then R_1, R_5 and HR are computed (for the highest ranked true positive class).

We note that if we don't use the probabilities and ranking, that is, use class assignments to follow a path, the classification performance greatly suffers. This is since classifiers (when having to assign a class) in the higher levels can make "premature" false positive and false negative mistakes (and false negative mistakes are very costly). This inferior performance has been noted before too (e.g., Dekel et al., 2003).

6. Experiments

In this section, after describing the data sets we use and the experimental set up, we report on comparisons with other approaches. We then report on several ablation experiments as well as comparisons to the simpler IND algorithm and a previous (unweighted) indexing method. We conclude the section with experiments on some properties of our index learning method and the indices

Data Sets	$\lvert S\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{C}\rvert$	$E_x\lvert\mathbf{v_x}\rvert$	$E_x\lvert C_x\rvert$
Reuters-21578	$9.4k$	$33k$	10	80.9	1
20 Newsgroups	$20k$	$60k$	20	80	1
Industry	$9.6k$	$69k$	104	120	1
Reuters RCV1	$23k$	$47k$	414	76	2.08
Ads	369k	301k	12.6k	27.2	1.4
Web	70k	685k	14k	210	1
Jane Austen	749k	299k	17.4k	15.1	1

Figure 8: Data sets: $|S|$ is number of instances, $|\mathcal{F}|$ is the number of features, $|\mathcal{C}|$ is the total number of classes, $E_x|\mathbf{v_x}|$ is the average (expected) number of unique active features per instance (avg. vector size), and $E_x|C_x|$ is the average number of class labels per instance.

learned (average class indegrees, performance on the training data, ...), and we give a few examples of the learned connections.

Figure 8 presents the data sets that we use, shown in order of class size $|\mathcal{C}|$. The first 6 are text categorization, and the last is a word prediction task. Ads refers to a text classification problem provided by Yahoo! Web refers to a web page classification into Yahoo! directory of topics. Jane Austen is 6 online novels of Jane Austen, concatenated (obtained from project Gutenberg (`http://www.gutenberg.org/`). The others are commonly used text categorization data sets (Lang, 1995; Rose et al., 2002.; Lewis et al., 2004).

On the first three small sets, we compare against one-versus-rest, and our purpose is mainly to compare accuracy. On the next 3, Reuters RCV1, Ads, and Web, we compare against the top-down method. In both the one-versus-rest and top-down methods, we deploy either single perceptron training, committee of 10 perceptrons, or a fast algorithm for learning linear SVMs (Keerthi and DeCoste, 2005). We could not run the SVM on the Web data as it took longer than a day, and had to limit our SVM experiments on Ads.[15] For the final word prediction data, the task is to predict each word given features derived from the surrounding words in the sentence. For this problem, since the classes (the words) do not form a hierarchy and one-versus-rest is too inefficient, we only show performance of the indexing method.

All instance vectors are l_2 (cosine) normalized. For text categorization data, the features are standard unigram or bigram words and phrases. The feature vectors were obtained from publicly available sources in the cases of Reuters RCV1 (Lewis et al., 2004), and the newsgroups (from Rennie[16]). For RCV1, we used the training split only (23k documents) to be able to experiment with the slower algorithms. We obtained the Ads and Web data sets from Yahoo! For the web data set, to obtain a sufficient number of instances per class, we cut the taxonomy at depth 4, that is, we only considered the true classes up to depth 4. To simplify evaluation, we used the lowest true class(es) in the hierarchy the instance was classified under at test time. Thus an instance in Reuters RCV1 corpus on average is assigned two true classes. We note that in many practical text

15. There has been further advances on speeding up linear SVM training algorithms since the submission of this paper (e.g., Shalev-Schwartz et al., 2007; Hsieh et al., 2008). The perceptron training timings in our tables may be a better indicator of the training times for the more recent algorithms.
16. Obtained from people.csail.mit.edu/jrennie/20Newsgroups.

categorization applications such as personalization, classes at the top level are too generic to be informative/useful. For top-down training, we trained the classifier on the internal classes as well. Web and Ads had just over 20 classes in the first level (after root), while Reuters RCV1 has two (we used both the Industry and Topic trees). The Jane Austen (word prediction) data set was obtained by processing Jane Austen's six online novels: the surrounding neighborhood of each word, 3 words on one side, 3 on the other, and their conjunctions, provided the features (about 15 many).

We report on the average performance over ten trials. In each trial a random 10% of the data is held out. The exception is the newsgroup data set where we use the 10 train-test splits by Rennie et al. (2003), each 80-20, and we used their vector representations, to be able to compare directly with their results. We used a 2.4GHz AMD Opteron processor with 64 GB of RAM, with light load.

Figures 9 and 10 present the algorithms' performance under both accuracy and efficiency criteria. As a simple baseline, we report the performance of FrequencyBaseline (FB) as well, which ranks classes simply based on the frequency of the classes in the training data set.

For the FF algorithm, we used w_{min} =0.01 for the minimum weight threshold during learning, and d_{max} =25 (max-outdegree during look up). Note that d_{max} of 25 means a class is retrieved as along as it is within the first 25 highest weight connections of some active feature, even if its weight is not much higher than w_{min}. During training, the FF algorithm looks for a true (positive) class within the first 50 top scoring classes. If it is not found, the score of the positive class is assumed 0. We report on performance after pass 1 with 0 margin threshold ($\delta_m = 0$), as well as best performance in R_1 within the first 10 passes, with $\delta_m \in \{0, 0.1, 0.5\}$. We did not optimize on the choice of these parameters, for example, we may do better for lower d_{max} values (see Section 6.3.2). Note that a δ_m value above 0.5 basically means to update on most instances as index edge weights are less than 1, and thus class score differences tend not to be much higher than 1.0, when instances are l_2 normalized. For the SVM, we report the best performance in R_1 over the regularization parameter $C \in \{1, 5, 10, 100\}$ for the first three small data sets and $C \in \{1, 10\}$ for the large ones. Often $C = 1$ and 10 suffices (accuracies are close). There are 10 perceptrons in the committee (often, 5 to 20 suffices for attaining much of the accuracy).

6.1 Accuracy Comparisons

We first observe that the FF algorithm is competitive with the best of others. In particular it achieves the highest R_5 in 5 of the 6 categorization domains, and the highest average R_1 in 4 of 6 cases. We have observed that comparison based on the HR results often yields similar rankings of the algorithms tested as does R_1. We limit the discussions to R_1 and R_5. In particular R_1 (plain accuracy or one minus zero-one error) is a simple commonly used performance measure. For the classifier based methods, observe that there is a good separation from perceptron to SVMs, suggesting that the classification tasks are challenging. The performance of FF on newsgroup ties the best performance achieved by Rennie et al. (2003), and they used special feature vector representations, for the linear SVM as well as their methods, to reach that performance.[17] In the case of RCVI, for top-down training, we experimented with using a fixed sigmoid, that is, no sigmoid fitting, as well as sigmoid fitting using one or more trials of obtaining scores (score-class pairs). When not fitting, we used fixed values of 0 bias and -2 slope: $\frac{1}{1+e^{-2s}}$, where s denotes the score of classifier on the instance.

17. On the industry data set, we found that the classes have similar proportions, close to 0.01. As we keep only 10% for test, and there are only just under 10k instances in the whole set, we see that the performance of Frequency Baseline is very low. The classes with the highest proportion in training are not the classes with the highest proportion on the test set.

	Rank (HR)	R_1	R_5	T_{tr}	d	$\|\mathbf{W}\|$
Small Reuters, 10 classes						
δ_m=0, p=1	1.082	0.860	0.998	0s	4.9	55k
δ_m=0.5, p=1	1.066	0.884 $\pm$0.009	0.997	0s	5	73k
perceptron	1.08	0.871	0.995	4s	10	74k
committee	1.06	0.891	0.999	40s	10	74+
SVM C=1	1.052	**0.906** $\pm$0.009	0.998	11s	10	74+
FreqBaseline	1.6	0.42	0.86	-	-	-
News Groups, 20 classes						
δ_m=0, p=1	1.137	0.798	0.978	2s	10	113k
δ_m=0.5, p=1	1.085	**0.865** $\pm$0.005	**0.987**	2s	10	171k
perceptron	1.229	0.728	0.928	20s	20	189k
committee	1.122	0.830	0.970	220s	20	189+
SVM C=1	1.1020	0.852 $\pm$0.005	0.975	92s	20	189+
FreqBaseline	3.33	0.05	0.25	-	-	-
Industry, 104 classes						
δ_m=0, p=1	1.114	0.861	0.942	4s	16.7	124k
δ_m=0.5, p=3	1.094	**0.886**$\pm$0.008	**0.949**	16s	15.8	196k
perceptron	1.488	0.595	0.773	55s	104	330k
committee	1.17	0.816	0.904	610s	104	330+
SVM C=10	1.112	0.872 $\pm$0.009	0.933	235s	104	330+
FreqBaseline	31	0.005	0.03	-	-	-

Figure 9: Comparisons on the smaller data sets. T_{tr} is training time (s=seconds, m=minutes, h=hours), d is the number of "connections" touched on average per feature of a test instance, and $|\mathbf{W}|$ denotes the number of (nonzero) weights in the system (see Section 6.2). The first two rows for each set report on FF, the first row being FF with 0 margin threshold, after one pass (p=4 means trained for four passes). Some example standard deviations for R_1 are also shown.

Thus, at score of 0, the (probability) value obtained is 0.5, and score of 1, the probability is $\frac{1}{1+e^{-1}} \approx 0.73$. When sigmoid fitting, we used one or more 80-20 splits of the training data, trained on 80, obtained the scores on the remaining 20, pooled the scores from different trials and fitted a sigmoid on the points. We then trained the classifier on the whole set. With more trials, we got better results on RCV1, but with diminishing returns, and this takes more time. For Ads, we could run the SVM with no fitting. Committee and perceptron used 3 trials. We performed the binomial sign test to compare the performance of FF (the second row for each data set) against the best of others (this is the SVM result, when available) as follows. We paired the R_k values on the same splits of data, 10 many for each data set, and counted the number of wins and losses of FF. The bold-faced R_1 and R_5 values indicate significance with confidence level $p \leq 0.05$ (i.e., either 9 or 10 wins). We observe that FF is superior with statistical significance in many cases, and only in one case, R_1 on the smallest data set, does it have statistically significant inferior performance.

The competitive and even superior accuracy of the FF algorithm provides evidence that improving class ranking on each instance, in the context of other classes that may be relevant (other retrieved classes), and at the same time, keeping the index sparse, is a good strategy or learning bias for our high performance categorization task. Methods based on binary classifiers can be at

	Rank (HR)	R_1	R_5	T_{tr}	d	$\|\mathbf{W}\|$
Reuters RCV1, 414 classes						
$\delta_m = 0$, p=1	1.181	0.763	0.955	6s	15.1	181k
$\delta_m = 0.1$, p=4	1.164	0.787±0.008	**0.952**	24s	12.9	223k
perceptron	1.418	0.621	0.815	70s	38	760k
committee	1.197	0.769	0.918	750s	36	760k+
C=1,0fit	1.26	0.72	0.89	94s	36	4meg
C=1,1 trial	1.18	0.779	0.936	200s	36	4meg
C=1,3 trials	1.17	0.782	0.937	400s	36	4meg
C=1,4 trials	1.17	0.783 ±0.01	0.939	520s	36	4meg
FreqBaseline	4.58	0.082	0.348	-	-	-
Ads, 12.6k classes						
$\delta_m = 0$, p=1	1.269	0.706	0.892	27s	7.8	814k
$\delta_m = 0.1$, p=4	1.254	**0.725**±0.003	**0.890**	92s	6.7	1meg
perceptron	1.738	0.517	0.642	0.5h+	80	5meg
committee	1.424	0.652	0.758	5h+	80	5meg+
SVM C=10, 0 fit	1.424	0.665 ±0.003	0.774	12h+	80	5meg+
FreqBaseline	35.56	0.012	0.033	-	-	-
Web, 14k classes						
$\delta_m = 0$, p=1	2.22	0.346	0.575	64s	8	1.6meg
$\delta_m = 0$, p=2	2.21	**0.352**±0.007	**0.576**	128s	8	1.5meg
perceptron	6.69	0.098	0.224	1h+	250	14meg
committee	3.78	0.207	0.335	12h+	190	14meg+
FreqBaseline	10.4	0.053	0.126	-	-	-
Jane Austen, 17.4k classes						
$\delta_m = 0$, p=1	2.71	0.272 ±0.002	0.480	40s	8.7	1.5meg
$\delta_m = 0.1$, p=4	2.73	0.279 ±0.002	0.462	160s	9.1	1.6meg
$\delta_m = 0.5$, p=4	3.01	0.243 ±0.002	0.425	160s	9.1	1.6meg
FreqBaseline	10.3	0.037	0.15	-	-	-

Figure 10: Comparisons on the larger data sets. T_{tr} is training time (s=seconds, m=minutes, h=hours), d is the number of “connections” touched on average per feature of a test instance, and $|\mathbf{W}|$ denotes the number of (nonzero) weights in the system (see Section 6.2). The first two rows for each set report on FF, the first row being FF with 0 margin threshold, after one pass (p=4 means trained for four passes). Some example standard deviations for R_1 are also shown.

a disadvantage because the task of choosing whether a single class should be assigned or not, in isolation, can be error-prone, especially with large numbers of classes. Classifier scores can be used for ranking classes, but the classifiers were not obtained with the objective of a good ranking of the classes for each instance (and their scores may not be calibrated):[18] a typical binary classifier such as an SVM is trained to yield a separation among instances (a good class prototype). The scores of such a classifier are more suitable for ranking the instances for the corresponding class than ranking classes for each instance. Top-down classification can help accuracy in that the top classifiers may effectively discover features useful for making general distinctions, and lower-level classifiers can similarly use features for making fine distinctions among a smaller set of sibling classes. On the

18. However, for the one-versus-rest experiments, we also evaluated rankings using the probabilities obtained via sigmoid fitting, instead of using the raw classifier scores, but saw no change or inferior accuracies (not shown).

other hand, top-down classification inherits the problems of one-versus-rest (for the children of every parent node, the problem is akin to one-versus-rest). Furthermore, the errors of the intermediate classifiers along a classification path can add up. Indexing achieves a kind of direct "flat" classification (akin to one-versus-rest, but not via binary classifiers). Features that are directly predictive of classes can be discovered, skipping error-prone intermediary classifiers. On the other hand, distinguishing thousands of classes via a single linear classifier (the index) can be error-prone.[19] It is ultimately an empirical question which of these fairly different learning techniques may outperform others.

6.2 Efficiency and Ease of Use

We see that the training times of the FF algorithm is dramatically lower than others, and the ratio grows with data set size, reaching or exceeding two orders of magnitude.

Our measure of work, d, is the expected number of "connections" touched per feature of a randomly drawn instance during categorization. For example, for the ads data set, on average just under 8 connections (classes) are touched during index look up per feature, or 8×27 total are touched per instance (the average number of features of a vector is 27, see Figure 8), while for top-down ranking, 80 classifiers are applied on average (over 22 at the top level) during the course of top-down ranking/classification. We are assuming the classifiers have a memory-time efficient hashed representation. Again, we see that the indexing approach can have a significant advantage here.

In the case of the index, the space consumption $|\mathbf{W}|$ is simply the number of edges (positive weights) in the bipartite graph. In the case of classifiers, we assumed a sparse representation (only nonzero weights are explicitly represented), and in most cases used a perceptron classifier, trained in a mistake driven fashion as a lower estimate for other classifiers.[20] On the smaller data sets, the difference is not important. However, we see that on the large categorization data sets the classifier based methods can consume significantly more space. We also note that for the FF algorithm, with higher δ_m, the index size increases. This is caused by more updates being performed with higher δ_m, and more updates tends to increase edge additions. This does not appear to increase the work d though.

The FF algorithm is very flexible. We could run it on our workstations for all the data sets (with 2 to 4 GB RAM), each pass taking no more than a few minutes at most. This was not possible for the classifier based methods on the large data sets (inadequate memory). In general, the top-down method required significant engineering effort (encoding the taxonomy structure, writing the classifiers to file for largest data sets, etc). Liu et al. (2005) also report on the considerable engineering effort required and the need for distributing the computation.

6.3 Effects of Various Options and Parameters

In this section, we investigate the effects of various parameters and options on accuracy and efficiency. For each option, we show performance on a subset of data sets to show the difference that using that option can make. In each case, unless otherwise specified, the remaining parameters (such as choice of margin) are as in Figures 9 and 10 for best performance, and as before we report

19. Huang et al. (2008) explore a multi-stage data-driven classification approach, using fast indexing for the first stage.
20. We have observed that the committee of perceptrons can be converted into a single linear classifier by weight averaging after training without degrading accuracy.

	No Constraints	Default Constraints	T_{tr} (single pass)
Small Reuters	0.884 $\pm$0.008	0.884 $\pm$0.008	0s vs 0s
News Group	0.866 $\pm$0.006	0.865 $\pm$0.005	3s vs 2s
Industry	0.889 $\pm$0.009	0.886 $\pm$0.008	9s vs 4s
RCV1	0.787 $\pm$0.007	0.787 $\pm$0.008	$[40s - 50s]$ vs 6s
Ads	0.716	0.711	45m vs 27s
Web	0.327	0.347	2h vs 64s
Jane Austen	0.276	0.274	1h vs 41s

Figure 11: No constraints on d_{max} (maximum outdegree) nor w_{min} (w_{min} set to 0), compared to the default settings. Accuracies (R_1 values) are not affected much, but efficiency suffers greatly. The rough training times for a single pass are compared.

averages and standard deviations for 10 random trials of 90-10 splits (except for news groups, for which the 80-20 split is given).

6.3.1 REMOVING EFFICIENCY CONSTRAINTS

We designed the FF algorithm with efficiency in mind. It is instructive to see how the algorithm performs when we remove the efficiency constraints (w_{min} and d_{max}). Note however that such constraints may actually help accuracy somewhat by removing unreliable weights and preventing overfitting.

In these experiments, we set w_{min} to 0 and d_{max} to a large number (1000). We show the best R_1 result for choice of margin threshold $\delta_m \in \{0, 0.1, 0.5\}$, over the first 5 passes, and compare to default values for the efficiency constraints. We observe that the accuracies are not affected. However FF now takes much space and time to learn, and classification time is hurt too. On the web data, for instance, the number of edges in the index grows to 6.5meg after first pass (it was about 1.5meg before). The average number of edges touched per feature grows to 1633, versus 8 for the default, thus 200 times larger, which explains the slow-down in training time.

For the ads, web, and Jane Austen, due to the very long running times, we ran FF for only a few trials, sufficient to convince ourselves that the accuracy does not change (see also next section). We report the result (with or without constraints) from the first pass of a single trial, on the same split of data.

6.3.2 OUTDEGREE CONSTRAINT

Figure 12 shows accuracy against the degree constraint, d_{max}, on the 3 large categorization data sets. We see that accuracy may in fact improve with lower degrees (RCV1 and Web). At outdegree constraint of 3 for RCV1, the number of edges in the learned index is around 80k instead of 180k (for the default $d_{max} = 25$), and the number of classes (connections) touched per feature is under 3, instead of 15 (Figure 10).

In general, it may be a better policy to use a weight threshold, greater than w_{min}, instead of a max outdegree constraint, for more efficient retrieval, as well as more reduction in index size, without loss in ranking accuracy.

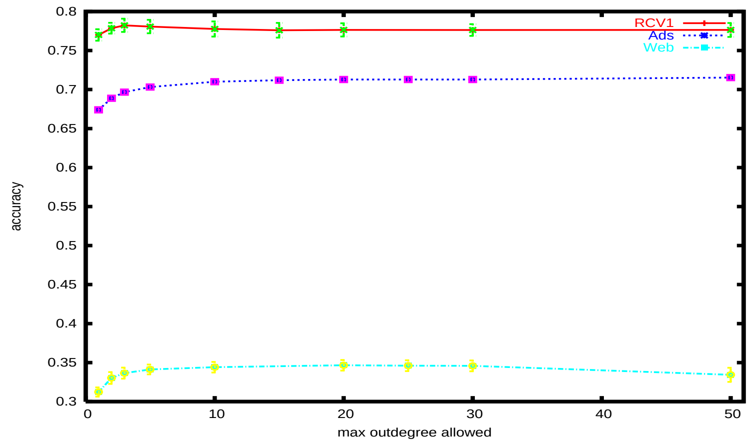

Figure 12: Accuracy (R_1) after one pass against the outdegree constraint.

	0.001	0.01	0.1
RCV1	0.786 ±0.009	0.787 ±0.008	0.761 ±0.008
Ads	0.728 ±0.002	0.725 ±0.003	0.701 ±0.003
Web	0.332 ±0.005	0.352 ±0.003	0.30 ±0.006

Figure 13: The effect of w_{min} on accuracy. We took the best R_1 within the first 5 passes. The standard deviations are also shown. The value $w_{min} = 0.1$ is significantly inferior, while setting w_{min} to 0.001 does not lead to significant improvements.

6.3.3 THE MINIMUM WEIGHT CONSTRAINT

We noted in Section 4.1 that a w_{min} value of 0.01 can be adequate if we expect most useful edge weights to be in say $[0.05, 1]$ range, while a w_{min} value of 0.1 is probably inadequate for best performance. Figure 13 shows the R_1 values for $w_{min} \in \{0.001, 0.01, 0.1\}$ on the three bigger text categorization data sets. Other options were set as in Figure 10, and the best R_1 value within first 5 passes is reported.

Note that while w_{min} =0.1 is inferior, the bulk of accuracy is achieved by weights above 0.1, and $w_{min} \leq 0.01$ does not make a difference on these data sets.

6.3.4 MULTIPLE PASSES AND CHOICE OF MARGIN

Figure 14 shows accuracy (with standard deviations over 10 runs for two plots) as a function of the number of passes and different margin values, in the case of Reuters RCV1. As can be seen, different margin values can result in different accuracies. In some data sets, accuracy degrades somewhat right after pass 1, exhibiting possible overfitting as training performance increases.

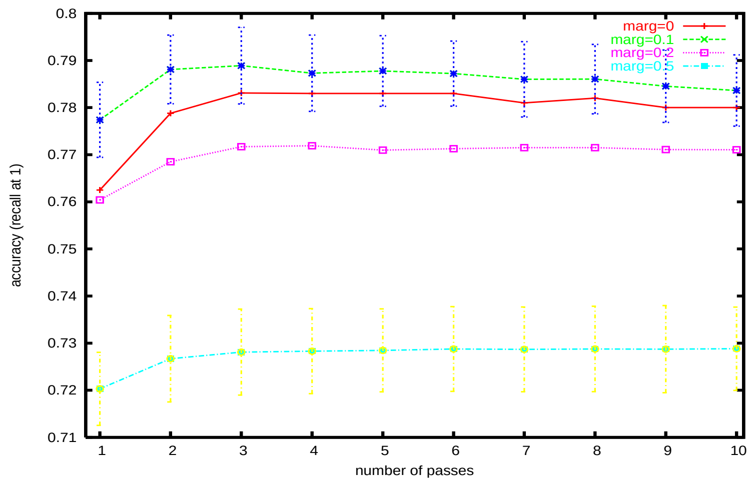

Figure 14: Reuters RCV1: Accuracy (R_1) for margin threshold $\delta_m \in \{0, 0.1, 0.2, 0.5\}$ against the number of passes.

	No Leakage	Allow Leakage
News Group	0.866 $\pm$0.005	0.865 $\pm$0.005
RCV1	0.780 $\pm$0.008	0.787 $\pm$0.008
Ads	0.696 $\pm$ 0.002	0.725 $\pm$ 0.003

Figure 15: On some data sets, allowing weight leakage when dropping edges can significantly improve accuracy.

6.3.5 Disallowing Weight "Leaks"

An uninformative feature such as "the" should give low votes to all classes. However, since the outdegree is constrained for memory reasons, if we imposed a constraint that the connection weights of a feature should sum to 1, then "the" may give significant but inaccurate weights to the classes that it happens to get connected with. Allowing for weight leaks is one way of addressing this issue. Figure 15 compares results. For the NO case in the figure (not allowing), whenever an edge from f to c is dropped, its weight, $w'_{f,c}$, is subtracted from w'_f. Thus $w'_f = \sum w'_{f,c}$ when we don't allow leaks, and $w'_f \geq \sum w'_{f,c}$ when we allow them.

6.3.6 Down Weighing Little Seen Features

Figure 16 shows the effect of down weighing infrequent features (the default option, see Section 4.3.2), against treating all features as equal (not using the option). Down-weighing infrequent features can significantly help.

	No Down-Weigh	With Down-Weigh
Newsgroup	0.860 $\pm$0.005	0.865 $\pm$0.005
RCV1	0.758 $\pm$0.007	0.787 $\pm$0.008
Web	0.327 $\pm$0.006	0.352 $\pm$ 0.007

Figure 16: Down-weighing infrequent features can significantly improve accuracy.

	IND	Bool FF p=1	best Bool FF	best FF
News Group	0.846 $\pm$0.006	0.860 $\pm$0.006	0.860 $\pm$0.005	0.865 $\pm$0.005
Industry	0.799 $\pm$0.01	0.839 $\pm$0.011	0.867 $\pm$0.008	0.886 $\pm$0.009
RCV1	0.686 $\pm$0.009	0.76 $\pm$0.01	0.780 $\pm$0.008	0.789 $\pm$0.008

Figure 17: Comparisons with IND and the effect of using feature values or treating them as Boolean and no l_2 normalization. The last column (best FF) contains results when default FF (with the use of feature values, l_2 normalization) is used (from Figure 9). Boolean FF with the right margin can significantly beat IND in accuracy, and use of feature values in FF appears to help over Boolean representation.

6.3.7 IND AND BOOLEAN FEATURES

IND treats features independently and as Boolean, but computes the conditionals exactly. Thus IND is similar to Boolean FF with high margin and $w_{min} = 0$, but IND also has a post-improvement step of adjusting p_{ind} (using the training set), which we have observed can improve the test accuracy of IND significantly (in addition to reducing index size). In these experiments p_{ind} was chosen from,

$$\{0.01, 0.02, \cdots, 0.09, 0.1, 0.15, 0.2, 0.25, \cdots, 0.6\}.$$

Here we compare IND against FF with Boolean values (and feature vectors are not l_2 normalized). This allows us to see how much using features values helps, as well as a comparison to a simpler heuristic of computing the conditional probabilities exactly and dropping the small values afterward. Figure 17 shows the results. For Newsgroup, Industry and RCV1, the best value of p_{ind} was respectively $0.01, 0.1$, and 0.3. To see the effect of edge removal on the accuracy of IND, if we chose $p_{ind} = 0$ (did no edge removal), we would get R_1 averaging below 0.58 on RCV1 (instead of current 0.69).

To achieve the best performance with Boolean features for FF on newsgroup, we had to raise the margin threshold to 7.0. Margin threshold of 1 or below gave significantly inferior results of 0.82 or below. Note that the scores that the classes receive during retrieval can increase significantly with Boolean features (compared to using the feature values in l_2 normalized vector representation).

We conclude that IND can be significantly outperformed by FF with an appropriate margin.

6.4 Other Experiments

Here, we first compare to an older indexing method (Madani et al., 2007) and then report and discuss some properties of the FF learning algorithm, such as the training performance, average scores of the top class, and a few example connections learned.

	No Classifiers	With Classifiers	FF
News Group	0.681 $\pm$0.007	0.768 $\pm$0.006	0.86
Industry	0.658 $\pm$0.009	0.795 $\pm$ 0.01	0.88$\pm$ 0.008

Figure 18: The performance of the non-ranking indexer algorithm, learning an unweighted index, with and without classifiers (first two columns) (Madani et al., 2007). The goal of improving class rankings, via learning a weighted index, simplifies indexing and improves classification accuracy.

6.4.1 COMPARISON TO OLDER INDEX LEARNING

The first idea for use of an index was to drastically lower the number of candidate classes to a manageable set when classifying a given instance, say 10s, and then use classifiers, possibly trained using the index as well (for efficient training), to precisely categorize the instances (Madani et al., 2007). Here, we briefly compare using that method, which we will refer to as *unweighted* indexing, against our current FF method. We have already noted that (binary) classifiers appear inferior for class ranking, especially as we increase the number of classes, in our comparisons in one-versus-rest experiments. Here, we present results showing that adding an intermediate index trained as described by Madani et al. (2007) does not improve accuracy. Furthermore, FF is significantly faster and easier to use.

The unweighted indexing algorithm of Madani et al. (2007) uses a threshold t_{tol} during training and updates the index only when more than t_{tol} many false positive classes are retrieved on a training instance or when a false negative occurs. In that work, class-feature weights are computed only to decide whether a connection or an edge should go into the index. We report accuracy under two regimes when we test unweighted indexing: (1) as a baseline, when only using the class-feature weights (without training classifiers), (2) when classifiers are also trained, here committee of perceptrons, trained in an online manner in tandem with the learning of the index, and the classes are ranked using the scores of the retrieved classifiers on the instance. For further details on that algorithm, please refer to Madani et al. (2007). Note that if we use the classifiers for direct classification (and not ranking) we obtain significantly inferior accuracy.

Figure 18 shows the results on the newsgroup and Industry data sets. When using no classifiers, we obtained the best R_1 performance with t_{tol} = 5 (out of $t_{tol} \in \{2,5,20\}$) on the newsgroup and Industry data sets. The accuracy improves with more passes, but reaches a ceiling in under 20 passes, and we have reported the best performance over the passes. With the addition of classifiers, the best R_1 is obtained when we don't use the indexer (see Figure 9), but the results from using the indexer can be close as the number of classes grows and with tolerance set in 10s. We have shown the result for t_{tol} =5 for newsgroup, and t_{tol} =20 for Industry. We observe that we require classifiers for the unweighted index learning method, to significantly improve accuracy, and the combination still lags behind FF in accuracy.

We note that while unweighted indexing without classifier training is fast, classifier training adds significant space and time overhead. Training was an order of magnitude slower than FF on the two data sets we reported on, and the classifiers also require 10 or more training passes to reach best performance.

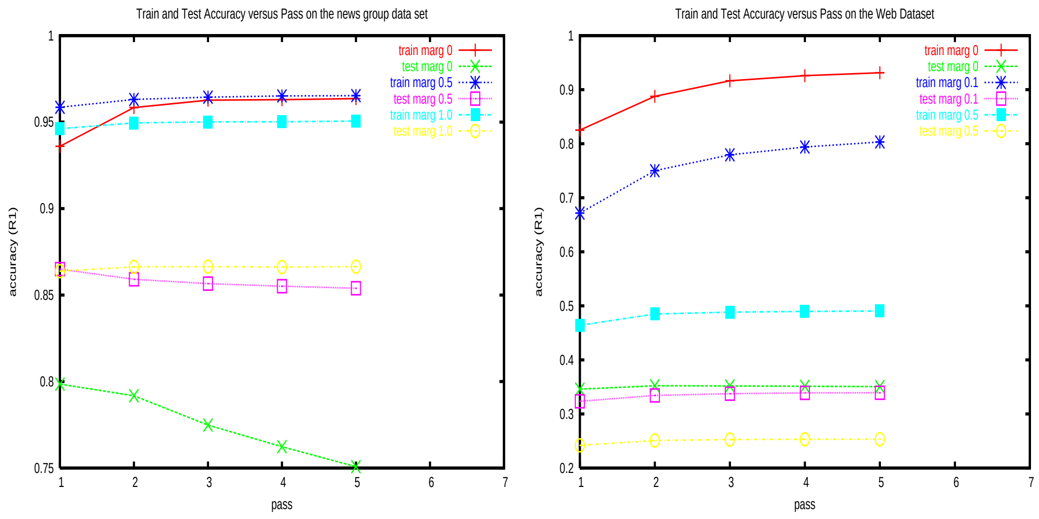

Figure 19: Train and test accuracy versus the number of passes, on the newsgroup (left) and web (right) data sets. Increasing the margin threshold can help control overfitting, but may not result in best test performance.

6.4.2 TRAINING VERSUS TEST PERFORMANCE OF FF

Figure 19 shows the train and test R_1 values as a function of pass. For the training performance, at end of each pass, the R_1 performance is computed on the same training instances rather than on the held-put sets. The higher the margin threshold, the less the capacity for fitting and therefor the less the possibility of overfitting. In the case of the newsgroup, we see that we reach the best performance with a relatively high margin threshold of $\delta_m \approx 1$, and the test and training performance remain roughly steady with more passes, unlike the case for δ_m =0. For the web data set, we see that the difference between train and test performance also decreases as we increase the margin threshold, but the best test performance is obtained with margin threshold of 0.

6.4.3 LEARNING CLASS PROTOTYPES

FF does not necessarily learn good (binary) classifiers or class prototypes, that is, the incoming weights into a class c_i (the vector $(w_{1,i}, \cdots, w_{f_{|F|},i})$), may make a poor class prototype vector. For example, we used such “prototypes” for ranking instances for each class in Reuters-21578 and newsgroup. The ranking quality (max F1) was significantly lower than that obtained from a single perceptron or a linear SVM trained for the class (5 to 10% reduction in absolute value of Max-F1 compared to perceptron on Reuters-21578 classes). On the newsgroup data set, the Max-F1 performances were comparable to single perceptrons but lagged the performance of SVMs.

6.4.4 CLASS INDEGREES

In Section 4.4 it was mentioned that prototypes may require more (nonzero) weights and processing time than features, and thus feature-based methods could have an efficiency advantage over prototype-based methods (even when adjusted for the average vector length times average feature outdegree). Of course, this all depends on the details of what processing needs to be performed for a given algorithm and what the average numbers come out to. It may be useful to look at the average indegree of a class during the FF algorithm on our data sets.

Let the indegree of a class, that is, the length of the prototype vector, be the number of features that have a significant edge to the class (within the highest d_{max} edges for each feature). After one pass of training, the indegree for the top ranking class (averaged over test instances), for the Newsgroup, Industry, RCV1, Ads, and Web was respectively: 6k, 2k, 4k, 530, and 14k. The true class had a lower but somewhat similar average indegree, except for the web, where the true class had an average indegree of 6700. Furthermore, in general, the average indegree of classes at given rank goes down with increasing rank. This is plausible: concepts with relatively higher indegree (i.e., more connections) tend to beat others in the score received: they tend to be ranked closer to the top.

Observe that the uniform averages (indegree of a class picked uniformly at random) is significantly lower for the big data sets, due to the skew in class frequencies. The uniform averages can be computed from Figures 9 and 10, for example, for the Web data it is: $\frac{1.5meg}{14k} \approx 100$.

6.4.5 EXAMPLE FEATURES AND CONNECTIONS

On RCV1, there were about 300 feature-class connections with weight greater than 0.9 (strong connections). Examples included: “figurehead” to the class “EQUITY MARKETS”, “gunfir” to the class “WAR, CIVIL WAR“, and ”manuf” (manufacturing) to “LEADING INDICATORS”. Examples of features with relatively large “leaks”, that is, with $w_f = \sum_c w_{f,c} < 0.25$, and thus likely uninformative, included “ago”, “base”, “year”, and “intern”.[21]

7. Conclusions

We raised the challenge of large-scale many-class learning and explored the approach of index learning. In this index-learning context, we began with the informal conjecture that (1) each feature need only connect to a relatively small number of classes, and (2) these connections can be discovered efficiently. We provided evidence that there exist very efficient online learning algorithms that nevertheless enjoy competitive and at times better accuracy performance than other commonly used methods. The algorithms may best be viewed as performing the computations from the side of features (the predictors) rather than the classes (the predicted). Each feature computes that choice of classes it may connect to and the connection weights. In particular, for very large-scale problems, each feature is space constrained in performing its computations and in the number of classes to which it can connect.

Much work remains in terms of advancing the algorithms and developing an understanding of their successes and limitations, including developing insights into the possible regularities in naturally occurring data that could explain the observed successes. We intend to further investigate index-learning algorithms, including different update methods and objectives, to develop theoretical properties, and to explore applications to various domains.

21. The feature “the” was probably dropped (a “stop” word) during the tokenization of this data set (Lewis et al., 2004).

Acknowledgments

Thanks to the anonymous reviewers of the paper for their valuable feedback and suggestions, which improved the presentation. Thanks to Scott Corlett, Dennis DeCoste, Scott Gaffney, Jian Huang, Sathiya Keerthy, David Kempe, Ravi Kumar, John Langford, Chih-Jen Lin, Kishore Papineni, Hema Raghavan, Lance Riedel, Mohammad Salavatipour, and the machine learning group at Yahoo! Research for suggestions, pointers, and discussions, and Pradhuman Jahala, Xiaofei He, and Cliff Brunk for providing us the web data.

Appendix A. NP-Hardness

For the purpose of establishing hardness, the problem is specified by a finite set of instances, wherein each instance is assigned a class and specified by the set of its active features. The features need only be Boolean. Of course, more general problems are at least as hard. We show NP-hardness when a fixed upper constraint is imposed on the outdegree on each feature in the index. The problem is NP-hard under either objective of maximizing accuracy or maximizing the MRR reward on the given set. For MRR, for each instance, the reward is the reciprocal rank $\frac{1}{k_x}$, that is, the rank of the correct class in the ranking returned by the index. On a single instance, the reward could be 0, if the class is not retrieved, and maxes at 1, if the correct class has rank 1. Note that MRR in Section 3.3 is simply the average reward per instance. For accuracy (R_1), the reward is either 1, if the correct class is ranked highest, or otherwise 0. The decision problem is then to determine whether a weighted index (a weighted bipartite graph) satisfying the outdegree constraint exists that yields a total reward, $\sum_{x \in X} r(c_x)$, exceeding a desired threshold.

Theorem 4 *The index learning problem with the objective of either maximizing accuracy (R_1) or minimizing HR on a given set of instances, with the constraint of a constant upper bound, such as 1, on the outdegree of each feature is NP-Hard.*

Proof The reduction is from the SET COVER problem (Garey and Johnson., 1979). We reduce the SET COVER problem to problem of computing an index wherein each feature can connect to at most 1 class.

An instance I of SET COVER consist of a set $U = \{e_1, \ldots, e_n\}$ of elements and a set $\mathcal{S} = \{S_1, \ldots, S_m\}$ of subsets of U. The goal is to find a smallest subset $\mathcal{S}' \subseteq S$ such that $\bigcup_{S_i \in \mathcal{S}'} = U$. Given a SET COVER instance I, we construct an instance of the indexing problem with only two classes c_1 and c_2 such that there is a SET COVER solution of size C for I iff there is an index (with the maximum outdegree of 1 constraint), such that the maximum total reward, the number of instances for which the right class is ranked highest, is $|U| + |\mathcal{S}| - C$.

In the constructed indexing problem, there is one feature f_i corresponding to each set $S_i \in \mathcal{S}$, for a total of m features. There is also one instance x_j for each element $e_j \in U$ ($1 \leq j \leq n$), and x_j contains feature f_i (x_j is connected to f_i) iff the element e_j belongs to the set S_i. These instances, called the "original instances", belong to class c_1. In addition, there are m "extra" instances, one for each set (or each feature). Each of these extra instances contains only the feature it corresponds to, and belongs only to class c_2 (see Figure 20).

Here, in constructing an index, we need to decide for each feature whether to connect the feature to c_1 or to c_2 (we can only connect to one of the two), and with what weights. Now, if a cover of size C exists, then we can easily obtain an index yielding reward of $|U| + |\mathcal{S}|$ - C: we connect the

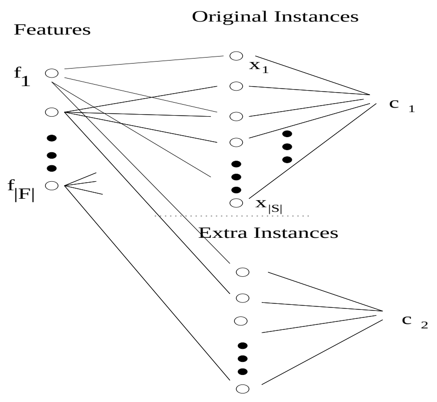

Figure 20: Reduction of the minimum set cover problem to index learning.

features in the cover (i.e., those features whose corresponding sets are in the cover) to c_1, each with weight of $|\mathcal{S}|$, and we connect all the other features to c_2 with a relatively small weight of say 1. In this way, for any original instance ($|U|$ many), c_1 is ranked highest, as at least one of its feature (the one(s) in the cover) connects to c_1 with high weight. For only $|\mathcal{S}| - C$ many of the extra instances, the correct class is missed, thus the total reward is $|U| + |\mathcal{S}| - C$.

For the reverse direction, we want to show that if an index with reward R exists, then there is a cover size $C \leq |U| + |\mathcal{S}| - R$. Assume an index is given with reward R. Note that lowering the connection weights to c_2 does not degrade the reward. So assume all such weights are at fixed minimum value v_{min}. Next, we note that any index can be converted to one in which all the original instances are “covered”, that is, the index ranks the right class highest: take any original instance for which this is not the case, and take one of its features that is connected to c_2 (there must be at least one), drop that edge, and connect it to c_1 with high enough weight so that c_1 is ranked highest. The weight can simply be $v_{min}|\mathcal{S}|$. This operation does not degrade total reward as we lose on exactly one extra instance, but gain on at least one original instance. We may repeat this operation until all original instances are covered, and the reward is now $R' \geq R$. Now, we see that $R' = |U| + |\mathcal{S}| - n$, where n is the number of those extra instances for which c_2 is not retrieved, equal to the number of features covering the original instances (connecting to c_1), or the cover size in the original problem is $C = n = |U| + |\mathcal{S}| - R' \leq |U| + |\mathcal{S}| - R$. ■

Observe that the NP-hardness remains and is easier to show if we use the maximum incoming score rule for class retrieval (each class gets the maximum of its incoming edge weights) instead of the sum. This reduction does not establish NP-hardness of constant-ratio approximability of class ranking (due to the subtraction), which remains an interesting open problem. For instance, either a constant-ratio approximation to loss (for problems with high accuracy) or accuracy (for problems with high loss) would be interesting. A similar reduction for the problem of computing an *unweighted* index shows that problem is NP-hard even to approximate (Madani et al., 2007).

Appendix B. Approximation Consequences of Edge Dropping

Consider the setting of Section 4.1 wherein a feature wants to compute the proportions of the (sufficiently frequent) classes in the stream it observes. There are two causes for inaccuracies in computing proportions:

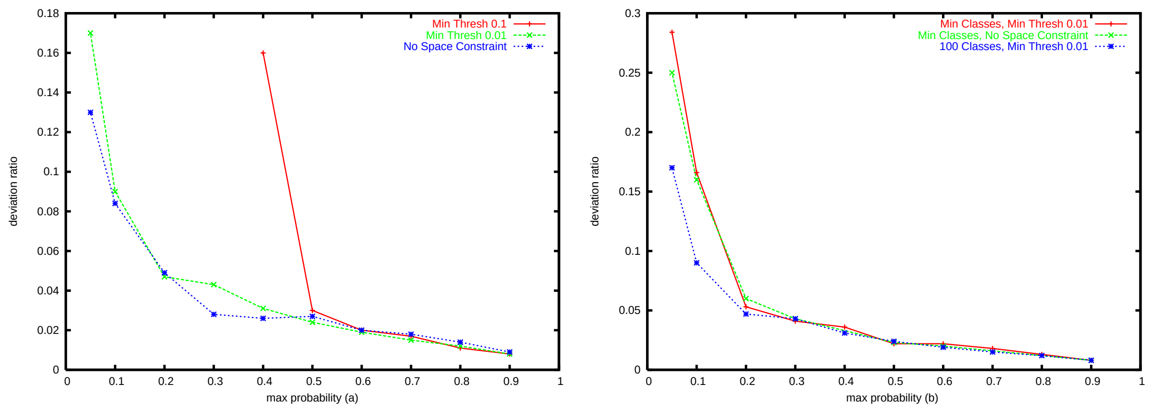

Figure 21: The performance of FSU under different allowances w_{min}. FSU computes the class proportions from processing a stream of 1000 class observations. For a choice of highest true probability p^*, the remaining probability mass $(1-p^*)$ is spread evenly over remaining classes. This is done under two regimes of generating classes. In (a) the number of unique classes is fixed at $|\mathcal{C}| = 100$, and in (b) it is $|\mathcal{C}| = \lceil \frac{1}{p^*} \rceil$ (i.e., modeling the situation in which other classes tie or have close probabilities to the maximum). The experiment, consisting of picking a class distribution and generating a 1000 draws, is repeated for 200 trials. In each trial, the deviation ratio of the highest proportion value computed by FSU, $\tilde{p}_{c_1}$, from the true maximum probability, p^*, is computed. This deviation (ratio) is $\frac{|\tilde{p}_{c_1} - p^*|}{p^*}$. The average deviation over the 200 trials is plotted against p^*. In the plot of part (b), the deviation is also compared to the case of 100 classes and $w_{min} = 0.01$. We note that $w_{min} \approx 0.01$ appears satisfactory for $p^* \geq 0.05$, while $w_{min} \approx 0.1$ performs well for a much smaller range.

- Finite samples (at any given time only a finite sample has been observed).
- Setting small weights (below w_{min}) to 0 (dropping edges) to save memory.

As FSU may drop and reinsert edges repeatedly, its approximation of actual proportions suffers from more than the issue of finite samples alone. We want to get an idea of this extra loss that we incur compared to the case when memory is not an issue (when no edges are dropped). Intuitively, FSU should work well as long as the proportions we are interested in sufficiently exceed the w_{min} threshold. The probability that a class with say probability p is not seen in some $\frac{1}{w_{min}}$ trials is $(1-p)^{1/w_{min}}$, and as long the ratio $\frac{p}{w_{min}}$ is high (several multiples), for example, $p > 4w_{min}$, this probability is relatively small. For example, for $w_{min} = 0.01$, and $p = 0.05$, the probability of not seeing such a class for a stretch of 100 consecutive trials is 0.006. More generally, the chance of being set to 0 (dropped) for a class with occurrence probability p quickly diminishes as we increase the ratio $\frac{p}{w_{min}}$, and therefore the cause of inaccuracies due to finite memory (the outdegree constraint on features) is mitigated.

We conducted experiments to see how much the proportion estimation by FSU deviates from true proportions and in particular compared that deviation to the deviations when FSU is not memory constrained (when w_{min} is set to 0). Figures 21 and 22 show the results. The experiments differ on how we generated the classes and computed the deviations. In the first of these experiments, to generate the true-class distribution, for some fixed number of classes $|\mathcal{C}|$, one class is given a

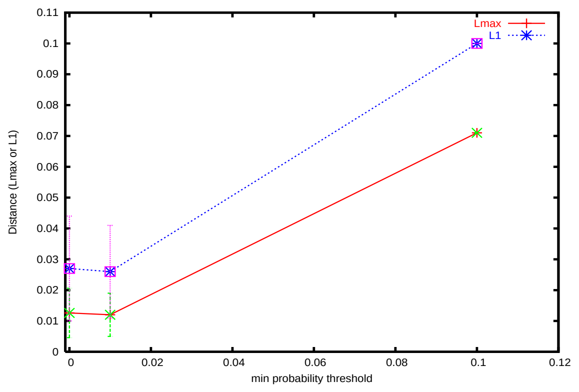

Figure 22: The performance of FSU, under different allowances w_{min}. The vector of true class probabilities is generated by uniformly and sequentially picking from $[0,1]$, and keeping track of total mass p (which should not exceed 1). If for latest generated class the probability drawn is greater than remaining mass $1-p$, the remaining mass is assigned instead, and class generation is stopped. FSU is evaluated after seeing a stream of 1000 classes iid drawn from such a source. The l_1 and l_∞ (or l_{max}) distances between the vector of empirical proportions that FSU computes and the true probabilities vector, averaged over 200 trials, is reported. FSU with w_{min} =0.01 yields distances comparable to FSU with $w_{min} = 0$, but w_{min} =0.1 yields significantly inferior estimates.

highest probability p^*, and the remaining classes obtain the remaining probability mass divided uniformly: $\frac{1-p^*}{|\mathcal{C}|-1}$. We then generated a stream of 1000 class observations (1000 iid draws) from such a distribution, and gave it to FSU with different values of w_{min}. We computed the deviation ratio: $\frac{|\tilde{p}_{c_1}-p^*|}{p^*}$, where c_1 denotes the class ranked highest by FSU, and $\tilde{p}_{c_1}$ is its assigned probability (highest computed probability). We averaged this deviation (ratio) over 200 trials of repeating the experiments. Figure 21(a) shows the averages when $|\mathcal{C}| = 100$ (so all classes except for one, obtain $\frac{1-p^*}{99}$). Figure 21(b) shows the results for $|\mathcal{C}| = \lceil \frac{1}{p^*} \rceil$ (e.g., for when $p^* \geq 0.5$, $|\mathcal{C}| = 2$, and when $p^* = 0.05, |\mathcal{C}| = 20$). Thus Figure 21(b) shows how FSU with limited w_{min} compares when the classes have similar proportions.

In the second set of experiments, we generated the probability for each class uniformly from the $[0,1]$ interval, keeping track of the total probability p used up during the course of generation. If the newest class gets a probability greater than $1-p$, $1-p$ is assigned to it and class generation, for selecting a distribution, is stopped. We then sampled iid to get a sequence of 1000 class observations. We compared the vector of class proportions that FSU computed using l_1 or l_∞ distance against the vector of true probabilities. We averaged the distances over 200 trials. We plot the results for FSU under different w_{min} constraints. We see that a threshold of $w_{min} \geq 0.1$ is not appropriate if the proportions we are interested in may be below 0.5, but a threshold of $w_{min} \approx 0.01$ does well, if we are interested in true proportions that are greater than 0.05 say. We compared a number of other statistics, such as the maximum deviation from true probability, and the probability that the deviation is larger than a threshold, and FSU with $w_{min} = 0.01$ performed similarly to $w_{min} = 0$ on the distributions tested. The reason as alluded to earlier is that those classes with proportions significantly greater than w_{min} have a high chance of being seen early and frequently enough in the stream and not being dropped.

Thus, as long as we expect that the useful proportions are a few multiples away from the w_{min} we choose, FSU is expected to compute proportions that are close to ones computed by the FSU with w_{min} set to 0 (no space constraints). Further, we expected that most often the important feature connection weights that determine the true classes during ranking have fairly high weight. Note also that the constraint of finite samples also points to the limited utility of trying to keep track of relatively low proportions: for most useful features, we may see them below say a 1000 times (in common data sets), and commonly occurring features tend not to be discriminative. Finally, vector length is a factor: if there tend to exist strong features-class connections, the influence of the weaker connections on changing the ranking will be limited, in particular when the number of active features is adequately small. Thus, in many practical learning problems, expecting that most useful proportions (weights) are in a relatively small interval, say $[0.05, 1]$ (or that the features do not require high outdegree) may be reasonable (see Section 6.3.3). In general however, some experimentation may be required to set the w_{min} parameter.

References

S. Albers and J. Westbrook. Self-organizing data structures. In A. Fiat and G. Woeginger, editors, *Online Algorithms: The State of the Art*, pages 31–51. Springer LNCS 1442, 1998.

J. K. Anlauf and M. Biehl. The adatron: an adaptive perceptron algorithm. *Europhysics Letters*, 1989.

H. Aradhye, G. Toderici, and J. Yagnik. Video2text: Learning to annotate video content. In *IEEE Int. Conf. on Data Mining (ICDM) Workshop on Internet Multimedia Mining*, 2009.

R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

R. Bayardo, Y. Ma, and R. Srikant. Scaling up all-pairs similarity search. In *Proc. Int. World Wide Web Conference (WWW)*, 2007.

A. Blum. Empirical support for winnow and wighted majority algorithms: Results on a calendar scheduling domain. *Machine Learning*, 26:5–23, 1997.

A. Borodin and R. El Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

V. R. Carvalho and W. Cohen. Single pass online learning. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2006.

N. Cesa-Bianchi, Y. Freund, D. P. Helmbold, D. Haussler, R. Schapire, and M. Warmuth. How to use expert advice. *Journal of the ACM*, 44(3):427–485, 1997.

T. Chua, J. Tang, R. Hong, H. Li, Z. Luo, and Y. Zheng. Nus-wide: A real-world web image database from national university of singapore. In *Proc. of ACM Conf. on Image and Video Retrieval (CIVR'09)*, 2009.

W. W. Cohen and Y. Singer. Context-senstive learning methods for text categorization. *ACM Trans. on Information Systems (TOIS)*, 17:141–173, 1999.

K. Crammer and Y. Singer. A new family of online algorithms for category ranking. *Journal of Machine Learning Research (JMLR)*, 3:1025–1058, 2003a.

K. Crammer and Y. Singer. Ultraconservative online algorithms for multiclass problems. *Journal of Machine Learning Research*, 3:951–991, 2003b.

K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7:551–585, 2006.

O. Dekel, J. Keshet, and Y. Singer. Large margin hierarchical classification. In *Proc. Int. Conf. on Machine Learning (ICML)*, 2003.

T. G. Dietterich and G. Bakiri. Solving multiclass learning problems via error-correcting codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

S. Dumais and H. Chen. Hierarchical classification of web content. In *Proc. Int. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR)*, 2000.

Y. Even-Zohar and D. Roth. A classification approach to word prediction. In *Proc. of the 1st North Amercian Association of Computational Linguistics (NAACL)*, 2000.

M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. In *Proc. 24th Int. Conf. Very Large Scale Data Bases (VLDB)*, 1998.

S. Fidler and A. Leonardis. Towards scalable representations of object categories: Learning a hierarchy of parts. In *Proc. of IEEE Int. Conf. on Vision and Pattern Recognition (CVPR)*, 2007.

D. A. Forsyth and J. Ponce. *Computer Vision*. Prentice Hall, 2003.

Y. Freund and R. E. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296, 1999.

Y. Freund, R. Schapire, Y. Singer, and M. Warmuth. Using and combining predictors that specialize. In *Proc. ACM Symposum on Theory of Computing (STOC)*, 1997.

E. Gabrilovich and S. Markovitch. Computing semantic relatedness using wikipida-baed explicit semantic analysis. In *Proc. Int. Joint Conf. on AI (IJCAI)*, 2007.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

C. Genest and J. V. Zidek. Combining probability distributions: A critique and an annotated bibliography. *Statistical Science*, 1(1):114–148, 1986.

C. Gentile. A new approximate maximal margin classification algorithm. *Journal of Machine Learning Research*, 2:213–242, 2001.

P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science: Special Issue on Eternal Memory Algorithms and Visualization*, 1999.

J. T. Goodman. A bit of progress in language modeling. *Computer Speech and Language*, 15(4): 403–434, October 2001.

K. Grill-Spector and N. Kanwisher. Visual recognition, as soon as you know it is there, you know what it is. *Pscychological Science*, 16(2):152–160, 2005.

M. Grobelnik and D. Mladenic. Efficient text categorization. In *Text Mining Workshop at European Conf. on Machine Learning (ECML)*. 1998.

S. Guha and A. McGregor. Space-efficient sampling. In *Proc. Int. Conf. on Artificial Intelligence and Statistics (AISTATS)*, 2007.

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer-Verlag, 2001.

C. J. Hsieh, K. J. Chang, C. J. Lin, and S. Sathiya Keerthi. A dual coordinate descent method for large-scale linear SVM. In *Proc. Int. Conf. on Machine Learning (ICML)*, 2008.

J. Huang, O. Madani, and C. Lee Giles. Error-driven generalist+experts (EDGE): A multi-stage ensemble framework for text categorization. In *Proc. ACM Conf. on Information and Knowledge Management (CIKM)*, 2008.

R. M. Karp, C. H. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Systems (TODS)*, 28:51–55, 2003.

S. Keerthi and D. DeCoste. A modified finite newton method for fast solution of large scale linear svms. *Journal of Machine Learning Research (JMLR)*, 6:341–361, 2005.

D. Koller and M. Sahami. Hierarchically classifying documents using very few words. In *Proc. Int. Conf. on Machine Learning (ICML)*, 1997.

W. Krauth and M. Mezard. Learning algorithms with optimal stability in neural networks. *J. of Physics A*, 20, 1987.

K. Lang. Newsweeder: Learning to filter netnews. In *Proc. Int. Conf. on Machine Learning*, 1995.

D. D. Lewis, Y. Yang, T. G. Rose, and F. Li. RCV1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research (JMLR)*, 5:361–397, 2004.

Y. Li and P. M. Long. The relaxed online maximum margin algorithm. *Machine Learning*, 46(1-3), 2002.

Y. Li, H. Zaragoza, R. Herbrich, J. Shawe-Taylor, and J. Kandola. The perceptron algorithm with uneven margins. In *Proc. Int. Conf. on Machine Learning (ICML)*, 2002.

N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1988.

T. Liu, Y. Yang, H. Wan, H. Zeng, Z. Chen, and W. Ma. Support vector machines classification with very large scale taxonomy. *SIGKDD Explorations*, 7, 2005.

O. Madani. Exploring massive learning via a prediction system. In *AAAI Fall Symposium Series: Computational Approaches to Representation Change During Learning and Development*, 2007a.

O. Madani. Prediction games in infinitely rich worlds. Technical Report 2, Yahoo! Research (and workshop on Utility Based Data Mining (UBDM)'06), June 2007b.

O. Madani and M. Connor. Ranked Recall: Efficient classification by efficient learning of indices that rank. Technical Report 3, Yahoo! Research, 2007.

O. Madani and M. Connor. Large-scale many-class learning. In *SIAM Conf. on Data Mining (SDM)*, 2008.

O. Madani and J. Huang. On updates that constrain the features' connections during learning. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2008.

O. Madani, W. Greiner, D. Kempe, and M. Salavatipour. Recall systems: Efficient learning and use of category indices. In *Proc. Int. Conf. on Artificial Intelligence and Statistics (AISTATS)*, 2007.

O. Madani, H. Bui, and E. Yeh. Efficient online learning and prediction of users' desktop actions. In *Proc. Int. Joint Conf. on AI (IJCAI)*, 2009.

C. Mesterharm. A multiclass linear learning algorithm related to Winnow. In *Proc. Neural Information Processing Systems (NIPS)*, 2000.

C. Mesterharm. Transforming linear-threshold learning algorithms into multiclass linear learning algorithms. Technical Report dcs-tr-460, Rutgers, 2001.

G. L. Murphy. *The Big Book of Concepts*. MIT Press, 2002.

J. Platt. Probabilities for support vector machines and comparisons to regularized likelihood methods. In A. Smola, P. Bartlett, B. Schlkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers*, pages 61–74. MIT Press, 1999.

D. R. Radev, H. Qi, H. Wu, and W. Fan. Evaluating web-based question answering systems. In *Proc. Int. Conf. on Language Resources and Evaluation (LREC)*, 2002.

H. Raghavan, O. Madani, and R. Jones. When will a human in the loop accelerate learning? quantifying the complexity of classification problems. In *Int. Workshop on AI for Human Computing, at IJCAI*, 2007.

J. Rennie, L. Shih, J. Teevan, and D. Karger. Tackling the poor assumption of Naive Bayes text classifiers. In *Proc. Int. Conf. on Machine Learning (ICML)*, 2003.

R. Rifkin and A. Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research (JMLR)*, 5, 2004.

T. G. Rose, M. Stevenson, and Miles Whitehead. The reuters corpus vol. 1 - from yesterday's news to tomorrow's language resources. In *Proc. Int. Conf. on Lang. Resources and Evaluation (LREC)*, 2002.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review,* 65(6):386–408, 1958.

F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34: 1–47, 2002.

S. Shalev-Schwartz, Y. Singer, and N. Srebro. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. In *Proc. Int. Conf. on Machine Learning (ICML)*, 2007.

S. Thorpe, D. Fize, and C. Marlot. Speed of processing in the human visual system. *Nature*, 381: 520–522, 1996.

T. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing & Management*, 31(6), 1995.

V. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, 2000.

V. G. Vovk. Aggregating strategies. In *Annual Workshop on Computational Learning Theory*, 1990.

J. Z. Wang, J. Li, and G. Wiederhold. SIMPLIcity: Semantics-sensitive integrated matching for picture libraries. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(9):947–963, 2001.

I. H. Witten, T. C. Bell, and A. Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*. John Wiley & Sons, 1994.

GR. Xue, D. Xing, Q. Yang, and Y. Yu. Deep classification in large-scale text hierarchies. In *Proc. Int. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR)*, 2008.