Large-Scale Many-Class Learning

Omid Madani * Michael Connor †

Abstract

A number of tasks, such as large-scale text categorization and word prediction, can benefit from efficient learning and classification when the number of classes (categories), in addition to instances and features, is large, that is, in the thousands and beyond. We investigate learning of sparse category indices to address this challenge. An index is a weighted bipartite graph mapping features to categories. On presentation of an instance, the index retrieves and scores a small set of candidate categories. The candidates can then be ranked and the ranking or the scores can be used for category assignment. We present novel online index learning algorithms. When compared to other approaches, including one-versus-rest and top-down learning and classification using support vector machines, we find that indexing is highly advantageous in terms of space and time efficiency, at both training and classification times, while yielding similar and often better accuracies. On problems with hundreds of thousands of instances and thousands of categories, the index is learned in minutes, while other methods can take orders of magnitude longer. As we explain, the design of the algorithm makes it convenient to maintain a constraint on the number of prediction connections a feature is allowed to make. This constraint is crucial in yielding efficient learning and classification.

1 Introduction

A fundamental activity of intelligence is to repeatedly and rapidly categorize. This task is especially challenging when the number of categories (classes) is very large, *i.e.*, in the tens of thousands and beyond. The approach of applying binary classifiers, one by one, to determine the correct classes is quickly rendered impractical with increasing number of classes. On the other hand, it appears that this problem of rapid classification in the presence of many classes might have been solved in nature (*e.g.*, [29, 13]). Furthermore, we seek systems that *efficiently learn to efficiently classify* in the presence of myriad classes. Again, the approach of learning binary classifiers is prohibitive with large numbers of classes and instances (millions and beyond). Other techniques, such as nearest neighbors, can suffer from a mix of drawbacks, including prohibitive space requirements, slow classification speeds, or poor generalization. Ideally, we seek highly scalable supervised (discriminative) learning methods that learn compact predictive models and achieve sufficient accuracy. A variety of tasks can be viewed as instances of *(large-scale) many-class* learning, including: (1) classifying text, such as queries, advertisements, news articles, or web pages into a large collection of categories, such as the Yahoo! topic hierarchy (http://dir.yahoo.com) [5, 19, 23], (2) determining the visual categories for image tagging and object recognition [30, 8], and (3) language modeling and similar prediction problems [12, 6, 20]. The following observations are important. In many tasks, such as in text prediction, training data is abundant, as the class labels are not costly. In general, neither the source of class feedback (the labels) nor what constitutes as a useful class to predict need be humans. In particular, the machine can build its own numerous concepts [21, 20].

In this work, we explore an approach based on learning an efficient index into the categories. An index here is a weighted bipartite graph that connects each feature to zero or more categories. 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, categories are retrieved and scored. We design our algorithms to efficiently learn space-efficient indices that yield accurate rankings. In particular, as we explain, the computations are carried out from the side of features. During learning, each feature determines to which relatively few categories it should lend its weights (votes) to, subject to efficiency constraints. At classification time, the features' votes are aggregated to score a relatively small set of candidate categories. The scores can then be used for ranking and/or class assignment.

We compare our algorithms empirically against one-versus-rest and top-down (hierarchical) classifier based methods, and an earlier proposed (unweighted) index learning method of [23]. We use linear classifiers, perceptrons as well as support vector machines, in the one-versus-rest and top-down methods. One-versus-rest is a simple method that has been shown to be quite competitive in accuracy in multi-class settings (with say 10s of classes), when properly regularized binary classifiers are used [27], and linear support vector machines achieve the state of the art in accuracy in many text classification problems (*e.g.*, [28, 18]). The top-down approach requires a taxonomy of classes. It is a conceptually simple and fairly scalable method that has com-

*SRI International, AI Center (madani@ai.sri.com). Research conducted while the author was at Yahoo! Research.

†UIUC, Computer Science (connor2@uiuc.edu). A major portion of the research conducted at Yahoo! Research.

monly been used for text categorization (*e.g.*, [17, 19, 5, 4]).

Our experiments provide evidence that there exist highly efficient and yet accurate algorithms, via index learning, for many-class problems. In our experiments on 6 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 and days. The index learned is more efficient in its use of space than the other classification systems, and results in quicker classification time. Very importantly, we find that budgeting the features' connections is a major property that renders the approach highly scalable. We explain how the algorithm's design makes this budget enforcement convenient. We have observed that the accuracies are as good and often better than the best of others. As we explain, one-versus-rest learning of binary classifiers is at a disadvantage in our many-class tasks, not just in terms of efficiency but also in accuracy. The indexing approach is simple to use: it requires neither taxonomies, nor extra feature reduction preprocessing. Thus, we believe that index learning offers a viable alternative to other methods, in particular for large-scale many-class problems.

Paper Organization. Sec. 2 presents the problem setting, including the semantics of index use as well as performance criteria, and reports on NP-hardness of a formalization. Sec. 3 presents the index learning algorithms and provides arguments that motivate the design. Sec. 4 presents our comparisons, algorithm behavior, and the effects of various parameters. Sec. 5 discusses related work, including similarities and differences to existing online methods, and Sec. 6 concludes. Some material, such as proofs, are omitted due to space limitations, and can be found in the technical report [22].

2 Index Learning

A learning problem consists of a set S of instances, each training instance specified by a vector of feature weights, F_x, as well as a category (class) that the instance belongs to[1], c_x. Thus each instance x is a pair $\langle F_x, c_x \rangle$. F and Y denote respectively the set of all features and categories. Our current indexing algorithms ignore features with nonpositive value, and in our data sets, features do not have negative value. $F_x[f]$ denotes the weight of feature f in the vector of features of instance x, where $F_x[f] \geq 0$. If $F_x[f] > 0$, we say feature f is *active* (in instance x), and denote this aspect by $f \in x$. The number of active features or vector length is denoted by $|F_x|$. Thus, an instance may be viewed as a set of active features. We also use the expression $x \in c$ to denote that instance x belongs to category c (c is a category of x). References on machine learning for text classification include Sebastiani [28] and Lewis *et. al.* [18].

Basic Mode of Operation:
Repeat
1. **Get next instance** x
2. **Retrieve, score, and rank categories via active features of** x
3. **If update is required, update the index.**

Figure 1: The cycle of categorization and learning (updating). During pure categorization (*e.g.*, in our tests), step 3 is skipped. See Figure 2 and Sec. 2 for how the index is used for scoring/classification, and Sec. 3 for when and how to update the index.

2.1 Index Definition and Use.

An index can be viewed as a directed weighted bipartite graph (a matrix): on one side, features (one node per feature) and on the other, categories. The index maps (connects) features to a subset of zero or more categories. An edge connecting feature f to category c has a positive weight denoted by $w_{f,c}$, or $w_{i,j}$ for feature i and category j. The *outdegree* of a feature is simply the number of (outgoing) edges of the feature. Index use here is similar to its use for inverted indices for document retrieval. It is also a way of performing efficient dot products. On presentation of an instance, the (nonnegative) score that a category obtains is determined by the active features: $s_c = \sum_{f \in x} w_{f,c} \times rating(f) \times F_x[f]$, where we explain the feature rating in Sec. 3.6. The categories may then be ranked by score. The top scoring category can then be assigned to the instance.[2] When features do not have large outdegrees, the ranked retrieval operation (scoring + ranking) can be implemented efficiently (Figure 2). For each active feature, only at most the d_{max} (maximum outdegree) categories with highest connection weights to the features participate in scoring. As there can be at most $|F_x|d_{max}$ classes scored, ranked retrieval takes $O((|F_x|d_{max}) \log(|F_x|d_{max}))$. The extra $\log(|F_x|d_{max})$ factor is due to the sorting cost and may not be needed (*e.g.*, when we are only interested in the score of the true category and the highest scoring category).

2.2 Performance Desiderata.

In large-scale many-class problems, all the sets S, Y, and F can be very large. In experiments reported here, Y and F can be in the tens of thousands, and S can be in the millions. In text and other prediction problems (*e.g.*, language modeling [12, 6]), the sizes exceed millions. The challenge is designing memory and time efficient learning and classification algorithms that do not sacrifice accuracy. We report on three measures of efficiency: training time T_{tr}, the size of the index learned

[1] Some problems have multiple true categories per instance (2 of our 7 data sets). In this paper, to simplify discussions, we focus on the single class (label) per instance setting. Whenever necessary, we briefly note the changes needed, *e.g.*, to the algorithm, to handle multiple labels. However, the multilabel setting may require additional treatment for improved performance under different accuracy criteria.

[2] Or information such as rank and the scores can be used to obtain probabilities for confidence assignment.

Algorithm RankedRetrieval(x, d_{max})
1. $\forall c, s_c \leftarrow 0$**, /* implicitly initialize scores */**
2. For each active feature f **(*i.e.,*** $F_x[f] > 0$**):**
For the first d_{max} **categories with highest connection weight to** f**:**
2.1. $s_c \leftarrow s_c + (w_{f,c} \times rating(f) \times F_x[f])$
3. Return those categories with nonzero score, ranked by score.

Figure 2: Using a weighted index for retrieving, scoring, and ranking categories.

$|\mathcal{I}|$ (total number of nonzero weights), and average number of edges touched per feature during classification $\bar{e}$, where $\bar{e} \leq d_{max}$. Both classification and update times are functions of $\bar{e}$. We next describe quality of categorization.

A method for ranking categories (indexing or otherwise), given an instance x, outputs a sorted list of 0 or more categories. We only consider the highest ranked true category (in case of multiple true categories). Let k_x be the rank of the highest ranked true category for instance x in a given ranking. Thus $k_x \in \{1, 2, 3, \cdots\}$. If the true category does not appear in the ranked list, then $k_x = \infty$. We use R_k to denote recall at (rank) k, $R_k = E_x[k_x \leq k]$, 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). We report on empirical R_1 and R_5, on held-out sets. R_1 is simply accuracy and allows us to compare against published results. We also tracked harmonic rank HR, the reciprocal of MRR, $MRR = E_x \frac{1}{k_x}$, and $HR = MRR^{-1}$, commonly used in IR literature. We obtained similar results in terms of comparisons whether we used R_1 or HR.

2.3 Computational Complexity. Computing an optimal index is NP-hard (worst-case), under a constant upper bound on feature outdegree (*e.g.*, 1), and whether optimizing R_1 or HR on a given training set. The proof, given in [22], is via reduction from the Set Cover problem [10]. In particular, a problem involving only two categories is shown NP-hard, by basically showing an equivalence to a feature selection problem. Approximability remains open, and the result does not preclude existence of algorithms that *converge* to good indices. The problem of checking whether a separator matrix exists (an index with 0 training error), when there is no outdegree constraint on the features, can be formulated as a linear program, and thus can be solved in polynomial time [22]. It is not clear how to effectively move the desired constraints on every feature's connection weights into a single objective (akin to regularization in SVM formulation, wherein a norm constraint on the weight vector to be learned is included). The next section describes an efficient algorithm in which each active feature performs a computation that improves the score of the true class on a given instance, while abiding by the constraints.

3 The Feature-Focus Algorithm

Our index learning algorithm may be best described as performing its operations from the features' side (rather than the classes' side), hence we name it *feature-focus* or FF. This design was motivated by considerations of efficiency in classification as well as updates, as we explain below. FF repeatedly inputs an instance, calls RankedRetrieval to get the ranked list of candidate categories with their scores, and then, if necessary, updates the index, *i.e.*, calls each active feature to update its connections. We first describe the computation (the update) that a single feature performs and then explain when the update is invoked.

3.1 The Single Feature Case. Imagine instances arrive in a streaming manner and assume $|F| = 1$, and Boolean ($F_x[f] \in \{0, 1\}$) for simplicity,[3] *i.e.*, we simply obtain a stream of categories. We will focus on the scenario where categories are generated by an iid drawing from a fixed probability distribution. Later we touch briefly on possibility of drifting in the category proportions. The question is to which categories the feature should connect, and with what weights, so that an objective such as R_k or HR is optimized. We seek simple space and time efficient algorithms, as the algorithm can be executed by millions of features. The feature may connect to no more than say d_{max} categories.

LEMMA 3.1. *A finite sequence of categories is given. To optimize HR or R_k on the sequence, when the feature can connect to at most k categories, a k highest frequency set of categories should be picked, i.e., choose S, $|S| = k$, such that $S = \{c | n_c \geq n_{c'}, \forall c' \notin S\}$), where n_c denotes the number of times c occurs in the sequence. The categories in S should be ordered by their occurrence counts to maximize HR.*

The proof is by an "exchange" argument: For any ordered set, HR (or R_k) is improved by exchanging a lower proportion category in the set, with a higher proportion category outside the set or improving the ordering within a set (if such candidate pairs exist). To maximize expectations on unseen portion of the stream, the highest proportion categories seen so far should be chosen. Thus a single feature should try to compute the highest proportion categories in its stream.

3.2 Proportion Approximation The weight update algorithm, called *Feature Streaming Update* (FSU), is given in Figure 3. With Boolean features, FSU simply computes

[3] Alternatively, consider the substream of instances that have the same feature active.

edge weights that approximate the conditional probabilities $P(c|f)$ (the probability that instance $x \in c$ given that $f \in x$). More generally, $\forall f, \sum_c w_{f,c} \leq 1$. This eases the decision of assessing importance of a connection: weights below w_{min} are dropped at the expense of potential loss in accuracy, and w_{min} bounds the maximum outdegree to $\frac{1}{w_{min}}$. This space efficiency of FSU is crucial in making FF space and time efficient (Sec. 4.5). There are two sources of inaccuracies in computing proportions: (1) finite samples (not algorithmically controllable), and (2) edge dropping (zeroing small weights). FSU should work well as long as a useful proportion w sufficiently exceeds the w_{min} threshold, as the likelihood of dropping diminishes with increasing $\frac{w}{w_{min}}$. In synthetic experiments, under different regimes for generating categories and distance measures, we have observed little difference in quality of computed proportions between FSU with $w_{min} = 0$ (no memory constraint), and $w_{min} = 0.01$ (the default in our experiments of Sec. 4), if the proportions we are interested in exceed 0.05 (significantly greater than 0.01). However $w_{min} \geq 0.1$ is not appropriate. Figure 4 shows one such experiment, an average of 200 trials (for experiments using other generation schemes, see [22]). Here, in each trial, we first generated a category distribution vector "uniformly", *i.e.,* each new category's probability is uniformly picked from $[0, 1]$ (Beta(1,1)), and we keep track of the total probability p used up during the distribution generation, and if the newest category gets a probability greater than $1 - p$, $1 - p$ is assigned to it and distribution generation is stopped. We then sampled iid from the generated distribution to get a sequence of 1000 categories, and ran FSU on it. We computed the l_1 and l_∞ distance between the FSU's category proportions vector and the true probabilities.

The constraint of finite training sample 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 1000 times (in common data sets). Furthermore, FSU is not necessarily invoked every time a feature is active, as we describe below. Finally, if there tend to exist strong feature-category connections, the weaker connections' influence on changing the ranking will be limited (this also depends on vector length). In practice, expecting that most useful proportions (weights) to be in say $[0.05, 1]$ may often suffice (see Sec. 4.3).

3.3 Uninformative Features, Adaptability, and Drifting. FSU keeps edge weight $w_{f,c}$ (not greater than 1), and "counts" $w'_{f,c}$ and w'_f (think Boolean features), all initially 0. Total weight w'_f is never reduced, thus an uninformative feature (such as "the") may develop low connections with total weight significantly less than 1 (weight "leaking"). As w'_f grows, the feature becomes less adaptive: when $w'_f > \frac{1}{w_{min}}$, a new category will be immediately dropped

/* Feature Streaming Update (allowing "leaks") */
Algorithm FSU(x, f, w_{min})
1. $w'_{f,c_x} \leftarrow w'_{f,c_x} + F_x[f]$ **/* increase weight to c_x.*/**
2. $w'_f \leftarrow w'_f + F_x[f]$ **/* 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 small weights */**
   $w_{f,c} \leftarrow 0, w'_{f,c} \leftarrow 0$

Figure 3: Feature Streaming Update (FSU): The connection weight of f to the true category c_x, w_{f,c_x}, is strengthened. Other connections are weakened due to the division. All the weights are zero at the beginning of index learning.

[4]. For long-term online learning, where distributions can drift, this can thwart learning, and updates that effectively keep a finite memory are more appropriate. On the other hand, this aspect has a regularizing effect that can prevent overfitting, for example in case of multiple passes on the same training data. Figure 5 shows a simple variant of FSU using finite memory, wherein each feature computes the exponential moving average of category proportions in its invocation stream [7]. For this version too, one can verify that the weights remain in [0,1], summing to no more than 1 for each feature, and the weight of the true concept is always increased with an update, unless already at 1.0. We leave systematic comparison of different FSU methods to future work, and use the FSU of Figure 3 here.

3.4 When to Update? The FF algorithm addresses some aspects of feature dependencies by not updating the index (invoking FSU) on every training instance. Lets consider two scenarios to motivate our choice of when to update. **Scenario 1.** Imagine only two categories, c_1 and c_2, and two Boolean features, f_1 and f_2, with $P(c_1|f_1) = P(f_1|c_1) = 1$ (f_1 is perfect for c_1), $P(f_2|c_1) = 1$ and also $P(f_2|c_2) = 1$. Thus, an instance x either has f_2 only ($x \in c_2$), or both f_1 and f_2 ($x \in c_1$). Assume $P(c_1) > P(c_2)$, thus $P(c_1|f_2) > P(c_2|f_2)$. If we always update, c_1 is always ranked higher, but an optimal index ranks c_2 higher when only f_2 is present. If FF invoked FSU only when the correct category was not ranked highest, *mistake-driven updating*, an optimal index is obtained: A first update on $x \in c_1$ allows for $w_{1,1} = 1$, and at most two updates on $x \in c_2$ allows for $w_{2,2} > w_{2,1}$. **Scenario 2.** Mistake driven updating can also lead to suboptimal performance. Consider the single feature case and three categories c_1, c_2, c_3, where $P(c_1) = 0.5$,

[4] At this point, updates can only affect categories already connected, and updates may improve the accuracy of their assigned weights, though there is a small chance that even categories with significant weights may be eventually dropped (this has probability 1 over an infinite sequence!). At this point or soon after, we could stop updating. With our finite data and small number of passes, this was not an issue.

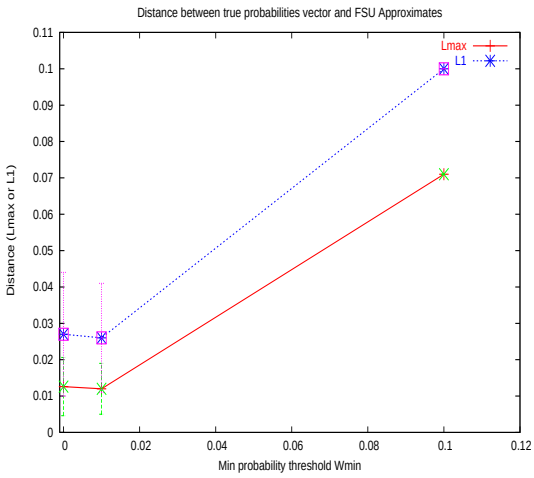

Figure 4: The performance of FSU, under different allowances w_{min}. FSU is evaluated after processing a stream of 1000 categories iid drawn from a fixed probability distribution (generated uniformly). The l_1 and l_∞ (or l_{max}) distances between the vector of empirical proportions that FSU computes 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.

/* A Finite Memory Version of Feature Streaming Update */
Algorithm FSU2(x**,** f**,** β**,** w_{min}**)**
1. $\forall c$**,** $w_{f,c} \leftarrow (1-\beta)w_{f,c} + [c = c_x]\beta$ **/* weaken and boost */**
2. If $w_{f,c} < w_{min}$**, then /* drop small weights */**
$w_{f,c} \leftarrow 0$

Figure 5: A simple finite memory verion of Feature Streaming Update (FSU), for the Boolean case, where β is a learning rate, $0 \leq w_{min} < \beta \leq 1$, and $[c = c_x]$ denotes the Iverson bracket (Kronecker delta, value of 1 if $c = c_x$, 0 otherwise). Each feature computes the exponential moving average of the category proportions in its invocation stream.

while $P(c_2) = P(c_3) = 0.25$ (thus, maximum $R_1 = 0.5$). If we don't update when the true category is at rank 1, on an alternating sequence: $c_1, c_2, c_1, c_3, c_1, c_2, \cdots$, the running value of R_1 can approach 0 (as connection weights will be similar). Random sequences are not as bad, but in synthetic experiments, in 200 trials, in each trial generating a 2000 sequence according to scenario 2 and 80-20 train-test splits, always invoking FSU gave average $R_1 = 0.479 \pm 0.02$, while mistake-driven gave 0.428 ± 0.09. Therefore, pure mistake-driven updates may cause unnecessary instability in weights and inferior performance.

3.5 Use Margin. We strike a balance between the two extremes by using a *margin*, defined on the current instance as the score of the positive category minus the score of the highest scoring negative category: $\delta = s_x - s'_x$, where $s_x \geq 0, s'_x \geq 0, s'_x = \max_{c \neq c_x} s_c$. If $\delta \leq \delta_m$, we update.[5] Setting $\delta_m = 0$, yields mistake driven updating (fitting capacity increases), and a high value corresponds to always updating. A good choice of δ_m is problem dependent. As individual edge weights are in $[0, 1]$ range, with l_2-normalized vectors, choice of $\delta_m \in [0, 1]$, and in particular $\delta_m \in \{0, 0.1, 0.5\}$, sufficed in our experiments for finding a good value: 0 corresponds to pure mistake-driven, and 0.5 to almost always updating.

3.6 Downweigh Infrequent Features (Feature Rating Adjustment). Down-weighing those features' votes that have only been seen a few times during learning can improve accuracy, as their proportions are likely somewhat inaccurate. As it stands, FSU of Figure 3 normalizes the weights, so a feature seen only once gives a vote of 1. Here, during category retrieval, we simply multiply a feature's vote, $w_{f,c}$, by $rating(f) = \min(1, \frac{n_f}{10})$, where $n_f \geq 1$ denotes the number of times feature f has been seen so far (n_f is updated only during the first training pass, in case of multiple passes).

We show in our experiments that the different aspects of FF, choosing a proper δ_m, and downweighing features, can make a significant difference.

3.7 Run-Time Complexity and Implementation. The edges for each feature are stored in a sorted doubly-linked list, and reordering, after an edge weight is boosted, amounts to a possible deletion and then an (re)insertion into the list. The cost of FSU is therefore $O(d_{max})$ for each invocation. Edges that are dropped are garbage collected. Thus, an update takes $O((|F_x|d_{max})$ per instance, and FF enjoys the same overall complexity per instance as its retrieval part: $O((|F_x|d_{max}) \log(|F_x|d_{max}))$.

3.8 Convergence. Convergence of the FF algorithm, given existence of a good index, is open. However, if the categories are perfect disjunctions or more generally noisy-OR (basically assuming independence) with $P(c|f)$ probabilities exceeding w_{min}, FSU computes the appropriate proportions with high likelihood.

3.9 A Baseline Algorithm: IND. As one baseline, we report on experiments with a simple heuristic, IND (for INDependent), in which the conditionals $P(c|f)$ are computed exactly. After processing all the training data, IND drops weights below a threshold p_{ind}. We have observed this reduction improves IND's accuracy (in addition to obvious space efficiency). Thus, IND is the version of FF that al-

[5] For instances with multiple true categories, the margin is computed for each. Every active feature is updated for each true category for which its margin is below the margin threshold.

Data Sets	$\|S\|$	$\|F\|$	$\|Y\|$	$E_x\|F_x\|$	$E_x\|Y_x\|$
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
Austen	749k	299k	17.4k	15.1	1

Figure 6: Data sets: $|S|$ is number of instances, $|F|$ is the number of features, $|Y|$ is the total number of categories, $E_x|F_x|$ is the average (expected) number of unique active features per instance (avg. vector size), and $E_x|Y_x|$ is the average number of true categories per instance.

Algorithm TopDownProbabilities(x**,** c**,** p**,** $\tilde{Y}_x$**)**
1. For each category 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 $\tilde{Y}_x \leftarrow \tilde{Y}_x \cup \{(c_i, p_{c_i}\})$
1.2.2 TopDownProbabilities(x**,** c_i**,** p_{c_i}**,** $\tilde{Y}_x$**).**

Figure 7: Using top-down classification to obtain a ranked list of candidate classes with assigned probabilities, when a taxonomy is available. For each instance x, the first call is TopDownProbabilities(x, root, 1.0, {}). $P_{c_i}(x)$ denotes the probability assigned by classifier for c_i to instance x.

ways invokes FSU with w_{min} =0. IND shares similarities with Naive Bayes and other algorithms that employ the technique of connecting features to categories based on measures such as information gain or tfidf: each feature's connections are computed independent of other features. The similarities and differences are further discussed in [22]. IND does not require the retrieval step during learning: it always updates all features, and need only increment counts, thus it takes $O(|F_x|)$ per update (or per instance). Thus, IND has faster training time on smaller datasets. However, IND gets into memory difficulties and consequently significantly slows down, on our bigger data sets (on typical machines with 4 or 8 GIGs of memory), as it does not drop edges (weights) until after all training instances are processed. We will observe that IND or always updating can lead to significant loss in accuracy (Sec. 4.4 and 4.6).

4 Experiments

In this section, we first compare with commonly used approaches, then present experiments on effects of various parameters, on comparisons to variants of index learning, and on some properties of the algorithm and the index learned. Figure 6 displays our data sets. The first 6 sets are text categorization. Ads refers to advertisement classification, provided by Yahoo! Web refers to web page classification into Yahoo! directory of topics. The others are benchmark categorization [18, 26]. All instances are l_2 (cosine) normalized, using standard features (unigrams, bigrams,..), obtained from publicly available sources (*e.g.*, [18, 26]). On the first three small sets, we compare against one-versus-rest learning (*e.g.*, [27]). Here efficiency is not a significant issue, and we primarily want to see how FF compares on accuracy. On the next 3, (Reuters) RCV1, Ads and Web, one-versus-rest becomes prohibitive, and we use top-down (hierarchical) learning and classification using the available taxonomy (*e.g.*, [5, 19]). Both approaches are based on training binary classifiers, and both are very commonly used techniques. To simplify evaluation for the case of the taxonomy, we used the lowest true category(ies) in the hierarchy the instance was categorized under. In many practical applications such as personalization, topics at the top level are too generic to be useful. For top-down training, we had to train a classifier for each internal category as well, however, and applied sigmoid fitting to obtain probabilities [25]. When classifying an an instance (Figure 7), top-down yields a ranked list. We remove any category that is a parent of another in the list. Top-down requires a threshold on probability, p_{min}, for deciding when to stop following a path in the taxonomy. We picked the probability threshold from $\{0.01, 0.02, \cdots, 0.1, 0.2, ..\}$, that gave the best accuracy R_1. For further details on these algorithms, please see our technical report [22].

The last data set, Austen, is a word prediction task [12, 6], the concatenation of 6 online novels of Jane Austen (obtained from project Gutenberg (http://www.gutenberg.org/). Here, each word is its own class, and there is no class hierarchy. Each word's surrounding neighborhood of words, 3 on one side, 3 on the other, and their conjunctions constituted the features (about 15 many). We include this prediction task to further underscore the potential of scalable learning.

Figure 8 presents the algorithms' performance under both accuracy and efficiency criteria. The default parameters for FF: w_{min} =0.01, d_{max} =25, and we report the best performance within first 10 passes for the choice of $\delta_m \in \{0, 0.1, 0.5\}$, in addition to possibly other δ_m choices for comparison. For classifier-based methods, we used state of the art linear SVMs (a fast variant [16]), and report the best R_1 over a few values for the regularization parameter C, $C \in \{1, 5, 10, 100\}$ for first three small data sets and $C \in \{1, 10\}$ for the large ones. Often $C = 1$ and 10 suffices (accuracies are close). 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. We also used faster but less accurate single perceptron and committee of 10 perceptrons (each perceptron takes about 10 to 20 passes on the training data for best performance). 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 newgroup data set where we use

the ten 80-20 train-test splits provided by Rennie *et. al.* [26]. We used a 2.4GHz AMD Opteron processor with 64 GB of RAM, with light load in all experiments.

4.1 Accuracy Comparisons. We first observe that the FF algorithm is competitive with the best of others. We note that the accuracy of FF (R_1) on newsgroup ties the best performance in [26] (with the same vectors and train-test splits), who used an enhanced feature representation. We performed the binomial sign test, pairing FF versus SVM (or committee if unavailable) on the 10 splits, and the winners at 95% confidence (meaning at least 9 wins out of 10) are boldfaced[6] The competitive and even superior ranking and ultimately classification performance of the FF algorithm provides evidence that the goal of learning to improve category ranking is a good classification strategy in multiclass learning, especially in the presence of many classes. Methods based on binary classifier training can be at a disadvantage as a binary classifier are trained to discriminate instances for a given class, and are thus more suited for ranking instances for their class.

4.2 Efficiency and Ease of Use. FF is significantly faster, and the advantage grows with data size, exceedingly two orders of magnitude in our experiments. Note that for the one-versus-rest method, we simply need to train $|Y|$ binary classifiers for $|Y|$ classes. The simplest perceptron algorithm that we tried required on average 10 to 20 passes to obtain its highest accuracy on its training data. Thus, one-versus-rest becomes prohibitive for large $|Y|$. For the top-down (hierarchical) method, each binary classifier need only train on the instances belonging to sibling classes. Thus, binary classifiers are trained on all the instances only for the classes in the first level of taxonomy (*e.g.,* in the order of 10s of classes for Web and Ads). Still, 1000s of classifiers should be trained, and, to achieve descent accuracy, the required training work can exceed 100s of passes (10s of passes needed for each class or classifier). Also, we note that the bigger data sets (*e.g.,* Web) are actually subsets of the available data. We are using these subset to be able to experiment with the slower algorithms and to compare accuracies.

Our measure of work, $\bar{e}$, is the expected number of edges touched per feature of a randomly drawn instance during classification. For instance, for the Ads set, on average just under 8 connections (categories) are touched during index use per feature (avg 8×27 per vector), while for top-down, 80 classifiers are applied on average (over 22 at the top level) during the course of ranking. We are assuming the classifiers have an efficient hashed representation. We see

	R_1	R_5	T_{tr}	$\bar{e}$	$\lvert\mathcal{I}\rvert$
Reuters-21578 (10 classes)					
δ_m=0, p=1	0.860	0.998	0s	4.9	55k
δ_m=0.5, p=1	0.884	0.997	0s	5	73k
perceptron	0.871	0.995	4s	10	74k
committee	0.891	0.999	40s	10	74k+
SVM C=1	**0.906**	0.998	11s	10	74k+
News Groups (20 classes)					
δ_m=0.5, p=1	**0.865**	**0.987**	2s	10	171k
δ_m=0, p=1	0.798	0.978	2s	10	113k
perceptron	0.728	0.928	20s	20	189k
committee	0.830	0.970	220s	20	189k+
SVM C=1	0.852	0.975	92s	20	189k+
Industry (104 classes)					
δ_m=0.5, p=3	**0.886**	**0.949**	16s	15.8	196k
perceptron	0.595	0.773	55s	104	330k
committee	0.816	0.904	610s	104	330k+
SVM C=10	0.872	0.933	235s	104	330k+
Reuters RCV1 (414 classes)					
δ_m=0.1, p=4	0.787	**0.952**	24s	12.9	220k
δ_m=0.5, p=4	0.728	0.922	24s	12.9	250k
perceptron	0.621	0.815	70s	38	760k
committee	0.769	0.918	750s	36	760+
SVM C=1	0.783	0.939	520s	36	4meg
Ads (12k classes)					
δ_m=0.1, p=4	**0.725**	**0.890**	92s	6.7	1meg
perceptron	0.517	0.642	0.5h+	80	5meg
committee	0.652	0.758	5h+	80	5meg+
SVM C=10	0.665	0.774	12h+	80	5meg+
Web (14k classes)					
δ_m=0, p=2	**0.352**	**0.576**	128s	8	1.5meg
perceptron	0.098	0.224	1h+	250	14meg
committee	0.207	0.335	12h+	190	14meg+
Austen (17k classes)					
δ_m=0, p=1	0.272	0.480	40s	8.7	1.5meg
δ_m=0.5, p=4	0.243	0.425	160s	9.1	1.6meg

Figure 8: Comparisons. T_{tr} is training time (s=seconds, m=minutes, h=hours), $\bar{e}$ is the number of edges touched on average per feature of a test instance, and $|\mathcal{I}|$ denotes the number of (nonzero) weights in the learned system. The first one or two rows on each set report on FF, the first being the best choice, and the second another δ_m to compare, where $p = i$ means results after pass i on training. On the top three (smaller) data sets, comparison is to one-versus-rest, on the bottom three, the comparison is with top-down (hierarchical) classification.

[6] Standard deviations are not shown due to space limits, but see the tables in subsequent sections.

$w_{min} \rightarrow$	0.001	0.01	0.1
RCV1	0.786 $\pm$0.009	0.787 $\pm$0.008	0.761 $\pm$0.008
Ads	0.728 $\pm$0.002	0.725 $\pm$0.003	0.701 $\pm$0.003
Web	0.332 $\pm$0.005	0.352 $\pm$0.003	0.30 $\pm$0.006

Figure 9: 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.

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

Figure 10: 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 is utilized (using feature values, l_2 normalization, Figure 8). 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.

FF has a significant advantage here. $\bar{e}$ affects both update and classification times. The space consumption $|\mathcal{I}|$ is simply the number of edges (positive weights) in the bipartite graph, for index learning. 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. On the larger data sets the classifier based methods can consume significantly more space.

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 much engineering (*e.g.*, encoding the taxonomy structure for training/classification). The work of Liu *et. al.* [19] also reflects the considerable engineering effort required and the need for distributing the computation.

4.3 Minimum Weight Constraint. We noted in Sec. 3.1 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 9 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 8, and the best R_1 value within first 5 passes is reported.

Note that while w_{min} =0.1 is significantly 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.

4.4 IND and Boolean Features. Here we compare IND against FF with Boolean values, and raw feature vectors, *i.e.*, not l_2 normalized. Thus, this compares FF, which handles feature dependencies via the idea of margin and mistake-driven updating, with a simpler "optimized" heuristic of computing the conditional probabilities exactly, but independently, and afterwards dropping the small weight values, below a parameter p_{ind}, to improve on accuracy in addition to space savings. Here, we also get an idea of the extent that using feature values helps over treating them as Boolean. In these experiments p_{ind} was the best value (yielding highest accuracy on a held-out set from training) from the set,

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

IND is never better than FF. Figure 10 shows the results for a few data sets. For Newsgroup, Industry and RCV1, the best value of p_{ind} was respectively $0.01, 0.1$, and 0.3. To get an idea of the positive effect of edge removal for IND's accuracy performance, on RCV1, when we chose $p_{ind} = 0$ (did no edge removal), IND yields R_1 values averaging below 0.58 (instead of 0.69 with $p_{ind} = 0.3$).

To achieve the best performance with raw Boolean features for FF on the newsgroup data, we had to experiment with several threshold values, values around 7.0 giving best results. Margin threshold of 1 or less gave significantly inferior results of 0.82 or less. Note that the scores that the categories receive during retrieval can increase significantly with raw feature values, compared to using the feature values in l_2 normalized representation. We conclude that IND can be significantly outperformed by FF with an appropriate margin, and l_2 normalization has advantages over raw representation.

4.5 Lifting the Efficiency Constraints. We designed the FF algorithm with efficiency in mind. It is very useful to see how the algorithm performs when we lift 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 the first pass (it was about 1.5meg with default constraints). 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

	No vs. Default Constraints	T_{tr} (single pass)
Small Reuters	0.884 vs. 0.884	0s vs. 0s
News Group	0.866 vs. 0.865	3s vs. 2s
Industry	0.889 vs. 0.886	9s vs. 4s
RCV1	0.787 vs. 0.787	45s vs. 6s
Ads	0.716 vs. 0.711	45m vs. 27s
Web	0.327 vs. 0.347	2h vs. 64s
Jane Austen	0.276 vs. 0.274	1h vs. 41s

Figure 11: No constraints on d_{max} (maximum outdegree) or 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.

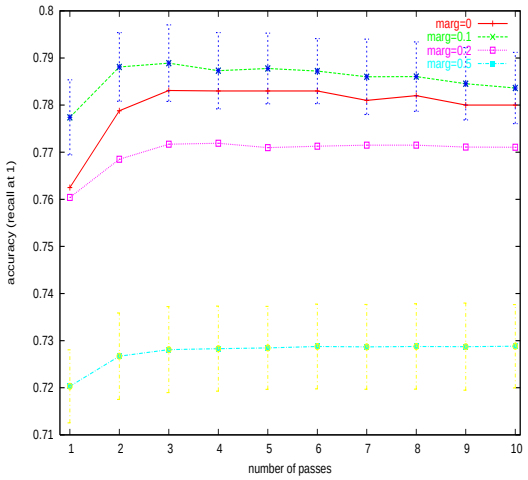

Figure 12: Reuters RCV1: Accuracy (R_1) for margin threshold $\delta_m \in \{0, 0.1, 0.2, 0.5\}$ against the number of passes.

running times, we ran FF for only a few trials, sufficient to convince ourselves that the accuracy does not change. We report the result (with or without constraints) from the first pass of a single trial, on the same split of data.

4.6 Multiple Passes and the Choice of Margin. Figure 12 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 (see also Sec. 4.11).

4.7 Outdegree Constraint During training, each feature keep several edges (possibly more than what is needed) so that the important concepts are given sufficient chance to develop their connection to the feature. But at any

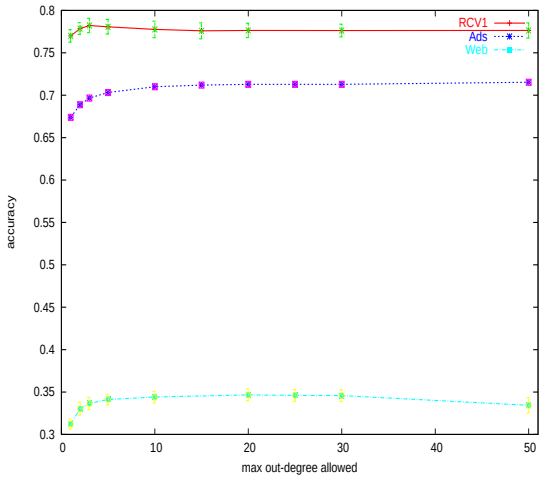

Figure 13: Accuracy (R_1) after one pass against the outdegree constraint.

given time, only a subset of these edges, d_{max} many (the d_{max} highest weight ones), may take part in scoring during classification. By lowering d_{max}, classification becomes faster, and the effective index (say after training is done) takes less space, but accuracy can be hurt. For each d_{max} value, a different index may be learned (more appropriate for that value), since the update patterns of FF can change. The capability to change and tune d_{max} as appropriate is useful for scenarios when the target machine that will run the index (once learned) has limited memory, or when classification speed is important. Figure 13 shows that accuracy changes smoothly with changes in the degree constraint d_{max}. We observe that accuracy may actually improve somewhat 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 categories (connections) touched per feature is under 3, instead of 15 (compared to Figure 8).

4.8 Disallowing Weight "Leaks" An uninformative feature such as "the" should give low votes to all categories. 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 categories that it happens to get connected with. Allowing for weight leaks is one way of addressing this issue. Figure 14 compares results. For the NO case in the figure (not allowing leaks), 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.

	No	Yes
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 14: Allowing weight "leakage" (Yes) when dropping edges can significantly help at times over disallowing it (NO).

	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

Figure 15: Comparison to a previous non-ranking unweighted index learning algorithm [23]. The objective of ranking via weighted index learning improves ranking (and ultimately classification) accuracy.

4.9 Down-weighing Infrequent Features. This option (see Sec. 3.6) can help significantly, for example R_1 on RCV1 goes from 0.787 ± 0.008 down to 0.758 ± 0.007 (other parameters fixed), if we don't down-weigh. On the Web, R_1 goes from 0.352 ± 0.006 down to 0.327 ± 0.007.

4.10 Comparison to Unweighted Index Learning. An effective index first serves to drastically lower the number of candidate categories. The first idea for learning a good index was to then use binary classifiers, possibly trained using the index as well (for efficient training), to precisely categorize the instances [23]. Here, we briefly compare FF with that method, call it *unweighted* index learning. We already noted that (binary) classifiers appear inferior for ranking and ultimately classification (Sec. 4.1), especially as we increase the number of classes. Here, we show that adding an intermediate index trained as in [23] does not improve accuracy, and furthermore FF is significantly faster overall, in addition to being simpler.

The indexer algorithm of Madani *et. al.* uses a threshold t_{tol} during training and updates the index only when either more than t_{tol} many false positive categories are retrieved on a training instance or the true class wasn't retrieved. We report accuracy under two regimes: (1) when only using the category-feature weights (without training classifiers), (2) when training classifiers, here committee of 10 perceptrons, in an online manner as the index is learned, and ranking categories using the scores of the retrieved classifiers on the instance. We note that the category-feature weights in that work were not learned for a ranking objective: they were only used for determining which unweighted edges should go into the index. We report performance when using such weights just as a baseline, as learning such an index without the classifier is very quick. We also compare when classifiers are used for ranking. Note that if we use the classifiers directly for class-assignment, instead of ranking, accuracy significantly degrades (false-positive and false-negative errors increase significantly). For further details on that algorithm, please refer to [23].

Figure 15 shows the accuracies 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 index (see Figure 8), but the results from using the index 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 note that while the unweighted index learning is fast here, the accuracies are low. Adding the classifier learning significantly slows down the training compared to using FF (an order of magnitude on the data we are reporting on), and the classifiers also often require 10 or more passes to reach best performance, and yet the accuracy is still not as good as FF.

4.11 Training versus Test Performance of FF. Figure 16 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 margin threshold of $\delta_m \approx 1$ (or higher), 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 the 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.

4.12 Class Prototypes. FF does not learn (binary) classifiers or class prototypes, *i.e.*, the incoming weights into a category c_i (the vector $(w_{1,i}, \cdots, w_{f_{|F|},i})$), make a poor category prototype vector. For example, we used such "prototypes" for ranking instances for each category in Reuters-21578 and newsgroup. The ranking quality (max F1,..) was significantly lower than that obtained from a single perceptron trained for the class (5 to 10% reduction in Max F1 on Reuters-21578 categories).

A category's indegree (the length of the prototype vector) is defined as the number of features that have a significant edge to the concept (within the first d_{max} edges for each feature). After one pass of training, the indegree for the top ranking category (averaged per test instance), for the Newsgroup, Industry, RCV1, Ads, and Web is respectively:

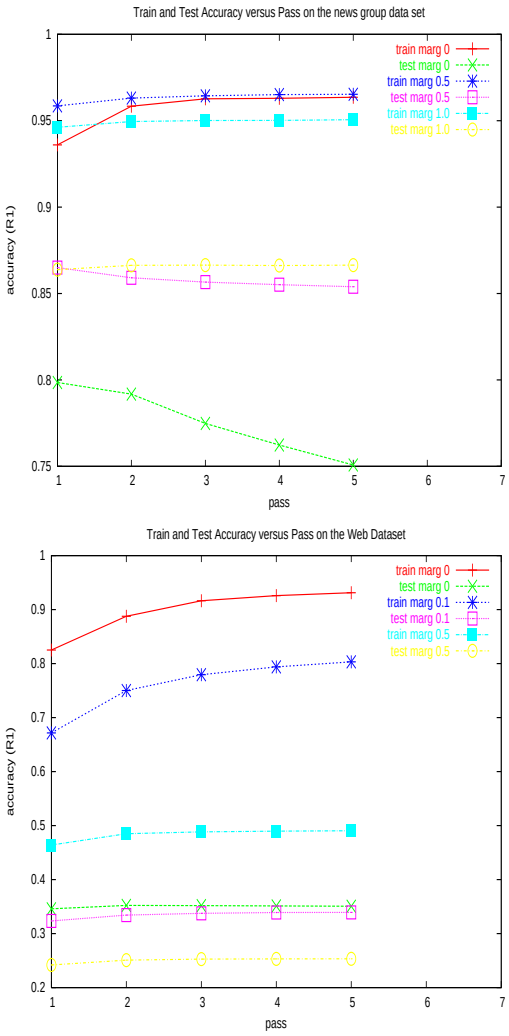

Figure 16: Train and Test Accuracy versus the number of passes, on the newsgroup (top) and web (bottom) data sets. Increasing the margin threshold can help control overfitting, but may not result in best test performance.

6k, 2k, 4k, 530, and 14k. The true category has a lower but somewhat similar indegree, except for the web, where the true category has an average indegree of 6700. The uniform averages (indegree of a randomly picked category) is significantly lower for the big data sets, due to the skew in category frequency. The uniform averages can be computed from Figure 8, for example, for Web it is: $\frac{1.5meg}{14k} \approx 100$.

4.13 Example Features and Connections. On RCV1, there were about 300 feature-category connections with weight greater than 0.9 (strong connections). Examples included: "figurehead" to the category "EQUITY MARKETS", "gunfir" to the category "WAR, CIVIL WAR", and "manuf" (manufacturing) to "LEADING INDICATORS". Examples of features with relatively large "leaks", *i.e.*, with $w_f = \sum_c w_{f,c} < 0.25$, and thus likely uninformative, included "ago", "base", "year", and "intern".[7]

5 Related Work and Discussion

Related work includes multiclass learning, indexing, feature selection, expert aggregation, and online and streaming algorithms, from which we can only discuss a few. Learning of an index was introduced in [23], but, as explained in Sec. 4.10, the index learned wasn't weighted and the objective was not direct classification via use of the index. Our comparisons show that our current approach is simpler as well as more efficient and accurate.

There exist a variety of multi-class approaches, but these approaches have not focused on large-scale many-class learning, and in particular the problem of efficient classification. The multiclass perceptron and related algorithms (*e.g.*, [3, 2]) share the ranking goal. These methods are best viewed as learning "prototypes" (weight-vectors), one for each class, and on an update certain prototypes' weights for the active features get promoted or demoted/weakened. We will refer to them as *prototype methods*, in contrast to our *predictor-based* methods. Our first message in this paper is evidence for the existence of very scalable algorithms for many-class problems, and it is possible that scalable discriminative methods other than predictor-based ones exist as well. It is conceivable that some kind of prototype regularization may render prototype methods efficient for large-scale many-class learning, but here are some difficulties that we see: Computing prototypes while simultaneously preserving efficiency may be more challenging as constraining the indegree of classes may not guarantee small feature-outdegrees (important for efficient classification and updates) and different classes can require widely different and large indegrees for good accuracy, as some classes are more typical or generic than others (Sec. 4.12). Furthermore, updates that involve prototype weight normalization or sparsity control require extra data structures (*e.g.*, pointers) for efficient prototype processing, in addition to the space required for the index (the feature to category mapping), complicating the approach.

Feature-focus, as a predictor-based method, has similarities with additive models and tree-induction [14], and is in the family of so-called expert (opinion or forecast) aggregation and learning algorithms (*e.g.*, [24, 9, 11])). Here, the experts (features) vote for several outcomes, are not always active (aka "sleeping" experts or "specialists", *e.g.*, [1, 9]), change their votes during learning, and can be space budgeted. Previous work has focused on learning mixing weights for the experts, while we have focused on how the

[7] The feature "the" is likely to have been dropped (a "stop" word) during tokenization of this data set [18].

experts may efficiently update their votes. Learning and using different weights for the experts (features) is a technique that we leave for future work, but down-weighing infrequent features (Sec. 3.5) is a form of that. An algorithm similar to FSU is also utilized for finding frequent items in a finite stream [15].

Taxonomies offer a number of potential efficiency and accuracy advantages [19, 5], but also present several drawbacks when used as a means for training and classification, including: they are unavailable for many tasks, there can be multiple ones and the structure may not be a tree, the implementation is somewhat complex (the structure has to be programmed in), and intermediate higher level categories in the tree may not be necessary or could hurt accuracy at the lower level, but more useful (informative), categories.

6 Conclusions

We raised the challenge of large-scale many-class learning, and provided evidence that there exist very efficient online supervised learning algorithms that nevertheless enjoy competitive or better accuracy performance compared to several other commonly used methods. There is much to do in extending the algorithms as well as developing an understanding of their success and limitations. For instance, it would be useful to allow feature's out-degree constraints to change dynamically on a per feature basis, possibly as a function of the predictiveness (utility) of the feature. We plan to investigate variations of index learning algorithms, to strengthen theoretical guarantees, and to explore applications to other domains.

Acknowledgements

Many thanks to Dennis DeCoste, Scott Gaffney, Sathiya Keerthy, John Langford, Chih-Jen Lin, Kishore Papineni, Lance Riedel, and the machine learning group at Yahoo! Research for valuable discussions, and Pradhuman Jahala, Xiaofei He, and Cliff Brunk for providing us the web data. Thanks to anonymous reviewer for feedback on improving the presentation.

References

[1] W. W. Cohen and Y. Singer. Context-senstive learning methods for text categorization. In *SIGIR*, 1996.

[2] K. Crammer, O Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *JMLR*, 7, 2006.

[3] K. Crammer and Y. Singer. A family of additive online algorithms for category ranking. *JMLR*, 3, 2003.

[4] O. Dekel, J. Keshet, and Y. Singer. Large margin hierarchical classification. In *ICML*, 2003.

[5] S. Dumais and H. Chen. Hierarchical classification of web content. In *SIGIR*, 2000.

[6] Y. Even-Zohar and D. Roth. A classification approach to word prediction. In *NAACL*, 2000.

[7] B. S. Everitt. *Cambridge Dictionary of Statistics*. Cambridge University Press, 2nd edition edition, 2003.

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

[9] Y. Freund, R. Schapire, Y. Singer, and M. Warmuth. Using and combining predictors that specialize. In *STOC*, 1997.

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

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

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

[13] 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.

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

[15] R. M. Karp, C. H. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Systems*, 2003.

[16] S. Keerthi and D. DeCoste. A modified finite Newton method for fast solution of large scale linear SVMs. *JMLR*, 2006.

[17] D. Koller and M. Sahami. Hierarchically classifying documents using very few words. In *ICML*, 1997.

[18] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li. RCV1: A new benchmark collection for text categorization research. *JMLR*, 5:361–397, 2004.

[19] 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.

[20] O. Madani. Exploring massive learning via a prediction system. In *AAAI Fall Symposium Series: Computational Approaches to Representation Change*, 2007.

[21] O. Madani. Prediction games in infinitely rich worlds. In *AAAI Fall Symposium Series*, 2007.

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

[23] O. Madani, W. Greiner, D. Kempe, and M. Salavatipour. Recall systems: Efficient learning and use of category indices. In *AISTATS*, 2007.

[24] C. Mesterharm. A multi-class linear learning algorithm related to Winnow. In *NIPS*, 2000.

[25] J. Platt. Probabilities for SVMs and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*. MIT Press, 1999.

[26] J. Rennie, L. Shih, J. Teevan, and D. Karger. Tackling the poor assumption of Naive Bayes text classifiers. In *ICML*, 2003.

[27] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *JMLR*, 5, 2004.

[28] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 2002.

[29] S. Thorpe, D. Fize, and C. Marlot. Speed of processing in the human visual system. *Nature*, 381, 520-522.

[30] J. Z. Wang, J. Li, and G. Wiederhold. SIMPLIcity: Semantics-sensitive integrated matching for picture libraries. *IEEE PAMI*, 23(9):947–963, 2001.