TECHNICAL REPORT
YR-2007-3

RANKED RECALL: EFFICIENT CLASSIFICATION BY EFFICIENT LEARNING OF INDICES THAT RANK

Omid Madani
Yahoo! Research
3333 Empire Ave
Burbank, CA 91504
{madani@yahoo-inc.com}

Michael Connor
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{connor2@uiuc.edu}

June 14, 2007

Yahoo! Research Report No. YR-2007-3

Yahoo! Research Report No. YR-2007-3

RANKED RECALL: EFFICIENT CLASSIFICATION BY EFFICIENT LEARNING OF INDICES THAT RANK

Omid Madani
Yahoo! Research
3333 Empire Ave
Burbank, CA 91504
{madani@yahoo-inc.com}

Michael Connor[*]
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{connor2@uiuc.edu}

June 14, 2007

ABSTRACT: Efficient learning and categorization in the face of myriad categories and instances is an important challenge. We investigate algorithms that efficiently learn sparse but accurate *category indices*. An index is a weighted bipartite graph mapping features to categories. Given an instance, the index retrieves, scores, and ranks a set of candidate categories. The ranking or the scores can then be used for category assignment. We compare index learning against other classification approaches, including one-versus-rest and top-down classification using support vector machines. We find that the indexing approach is highly advantageous in terms of space and time efficiency, at both training and classification times, while retaining competitive accuracy. 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.

[*] Part of this research was performed while the author was at Yahoo! Research.

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 problem occurs in a number of incarnations and applications, including: (1) classifying text, such as queries, documents, or web pages into a large collection of categories, such as the Yahoo! topic hierarchy (http://dir.yahoo.com) [MGKS07, LYW$^+$05, DC00], (2) language modeling and related prediction problems [Mad07, Goo01, EZR00], and (3) determining the visual categories for image tagging and object recognition [WLW01, FP03]. Furthermore, ideally we desire systems that *efficiently learn* to efficiently classify. This leads to challenges in ensuring that both learning of categories and categorization of items be efficient in their usage of time and space, in addition to being sufficiently accurate. Our setting is supervised learning: the categories are given and training data is available.

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 ranked by the scores that they obtain during retrieval, as we describe. The ranking or the scores can then be used for category assignment. We design our algorithms to efficiently learn space-efficient indices that yield accurate rankings. In particular, 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 categories it should lend its weights (votes) to, subject to efficiency (space) constraints. At classification time, the features' votes are aggregated to compute the final ranking of the categories.

We empirically compare our algorithms against one-versus-rest and top-down classifier based methods. We use linear classifiers, perceptrons as well as support vector machines, in the one-versus-rest and top-down methods. 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 other methods can take hours and days. The index learned is more efficient in its use of space than the other methods tested, and results in quicker classification time. The approach is simple to use: it does not require taxonomies or extra feature reduction preprocessing. Finally, we have observed that the accuracies are competitive and at times better than the best of others. Thus, we provide evidence that index learning offers a number of significant efficiency and simplicity advantages, without sacrificing accuracy.

This paper is organized as follows. We define the learning problem in Section 2. This includes the semantics of index use as well as performance criteria, in terms of both accuracy and efficiency. We then establish NP-hardness of a formalization of the problem. We present our index learning algorithms in Section 3. We discuss and motivate our choices in the design of the algorithms.

In Section 4 we briefly describe the other ranking methods we compare against, including one-versus-rest and top-down learning and classification methods. Section 5 presents our experiments. We report on comparisons among methods, and experiments on observations on effects of choice of parameters, and various tradeoffs. Section 6 discusses related work, and Section 7 concludes. Appendices contain proof details and further explorations.

2. The Index Learning Problem

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 proposed algorithms ignore features with nonpositive weight, and in the data sets we consider, features do not have negative weight. $F_x[f]$ 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 $f \in x$. 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). For background on machine learning in particular when applied to text classification, please refer to Sebastiani [Seb02] or Lewis *et. al.* [LYRL04].

The input problem may be viewed as a tripartite graph (Figure 1). In our large scale learning problems, all the three sets S, Y, and F can be very large. For instance, in experiments reported here, Y and F can be in the tens of thousands, and S can be in the millions. Prediction problems, such as predicting words and n-grams in text, can involve a practically unbounded stream of instances and growing sets of categories and features reaching millions and beyond [Mad07]. While F is large, each instance is sparse: relatively few of the features (often tens or hundreds) are active in each instance. The challenge is designing memory and time efficient learning and classification algorithms that do not sacrifice accuracy.

The number of categories is so large that indexing them, not unlike the inverted index used for retrieval of documents and other objects, is a plausible idea. An important difference from traditional indexing is that categories, unlike documents, are implicit, specified only by the (training) instances that belong to them. Another difference is that for good classification performance as well as efficiency, we need to be very selective in the choice of the index entries, *i.e.*, which connections (edges) to make and with what weights. Figure 2 presents the basic cycle of categorization via index look up and learning via index updating (adjustments to connection weights).

We have termed the learned outcome a *Recall System* [MGKS07, MG06]: a system that, when presented with an instance, quickly "recalls" the appropriate categories. In particular, we refer to

[1]In this paper for simplicity of evaluation and algorithm description, we treat multiclass but single class (label) per instance setting. Most of our data sets are single class per instance. Whenever necessary, we briefly note the changes needed to the algorithms to handle multiple labels.

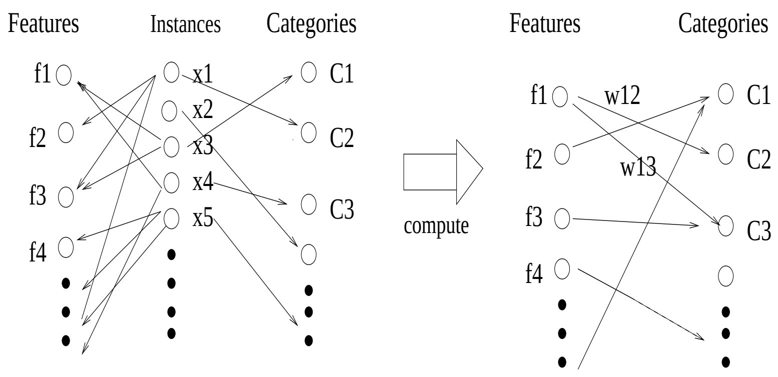

Figure 1: A depiction of the problem: the input can be viewed as a tripartite graph, possibly weighted, and perhaps only seen one instance at a time in an online manner. The goal is to learn an accurate efficient index, *i.e.*, a sparse weighted bipartite graph that connects (maps) each features to zero or more categories, 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 categories.

our present use of index for retrieving and ranking the categories, to be described next, as *ranked recall*.

2.1. Index Definition and Use

An index can be viewed as a directed weighted bipartite graph (or a matrix): on one side there are features (one node per feature) and on the other side there are 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 *out-degree* of a feature is the number of (outgoing) edges of the feature.

The use of the index for retrieval is similar to the use of inverted indexes for document retrieval. It is also a way of performing efficient dot products, and then sorting the scores, here positive. On presentation of an instance, the (nonnegative) score that a category obtains is determined by the active features:

$$s_c = \sum_{f \in x} r_f \times w_{f,c} \times F_x[f],$$

where r_f is a measure of the predictive goodness of feature f, and describe a method for computing it in Section 3.3.2 (currently assume it is 1 for all features[2]).

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

```
Basic Mode of Operation:
 Repeat
   1. Get next instance $x$
   2. Retrieve and rank categories via
   active features of $x$
   3. If update condition is met:
     3.1 Update index.
```

(a)

```
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. $s_c \leftarrow s_c + (r_f \times w_{f,c} \times F_x[f])$
  3. Return those categories with nonzero score,
       ranked by score.
```

(b)

Figure 2: (a) The cycle of categorization and learning (updating). During pure categorization (*e.g.*, in our tests), step 3 is skipped. See Section 2.1 for how to use the index (and part (b)), and Section 3 for when and how to update the index. (b) The algorithm that uses a weighted index for retrieving and ranking categories. See Section 2.1.

The positive scoring categories are then sorted by their score, or a priority queue can be used during retrieval . In a sense, each active feature casts votes for a subset of the categories, and categories receive and tally their incoming votes (scores). When features do not have large out-degrees in the index, the ranked retrieval operation (scoring + ranking) can be implemented efficiently (Figure 2(b)). Here, each feature (record) points to the list of categories it connects to along with the associated weights. For each active feature, only at most the d_{max} (maximum out-degree) categories with highest connection weights to the features participate in scoring. Note that if negative scores (edge weights) were allowed, then when some category obtains a negative score, the system would potentially have to process (*i.e.*, retrieve or update) all zero scoring categories as well, hurting efficiency.

2.2. Performance Criteria

In large scale learning, both memory and time efficiency are important, and both at training as well as categorization times.[3] We seek the following informal desiderata from index learning:

1. The index learned should be adequately sparse, *i.e.*, should not contain too many edges (space efficiency).
2. Efficient classification time: the out-degree of features should be small enough so that index use (look up) is fast on average.
3. We seek accurate indices.

[3]Note that in online learning, there isn't a sharp separation between the training and testing phases.

4. We seek space and time efficient algorithms for learning good indices. In particular, during online learning, when the index is continually used and updated, we want the efficiency criteria 1 and 2 to hold.

It may seem difficult to satisfy multiple criteria of space and time efficiency both during learning and classification (system use) times, but we note that in online learning, learning and classification times coincide, thus efficient training phase is equivalent to efficient classification phase. This reduces the number of constraints and can make the design task easier. In our experiments in Section 5.2 we report on three measures of efficiency: training time T_{tr}, the size of the index learned $|\mathcal{I}|$ (number of edges in it), and number of edges $\bar{e}$ touched during classification (a measure of work/speed during classification time). We next describe quality of categorization.

A method for ranking categories, given an instance x, outputs a sorted list of 0 or more categories. In addition to weighted indices, we describe other methods for category ranking in Section 3. An instance may belong to multiple classes in some domains (Table 9). To simplify evaluation and presentation, in this paper we only consider the highest ranked true category. 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, which measures the average proportion of (test) instances for which one of the true categories ended in the top k categories:

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

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 category 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. Note that recall at rank 1 is basically the standard accuracy measure when we are in the single label setting. That is, the highest ranked category is assigned to be the category of the instance, and R_1 measures the proportions of instances for which we obtained the right category. R_1 is a simple standard measure that allows to compare to published results.

We also use mean reciprocal rank (MRR) and the harmonic (mean) rank (HR) (reciprocal of MRR), defined as follows:

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

MRR gives a reward of 1 if a correct category 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 category 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.

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, *i.e.*, ranks the correct category of the given instance at top (rank 1) half the time, but fully fails the rest of the time, *i.e.*, does not retrieve the correct category at all, then its HR value is 2. However, for an index that always retrieves the correct category, 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 category is outside top k, while HR is a smoother measure.

2.2.1. Discussion of other Accuracy Criteria Our evaluation measure are from the point of view of an instance to be categorized (*i.e.*, category ranking), geared toward evaluating a categorization (ranking) system. The question is, how well does the system categorize a given instance, on average. Since the number of categories is very large and we are using an index, our evaluation measures are the same as the ones typically used in information retrieval. This is appropriate with large numbers of categories and in many applications, such as personalization (*e.g.*, [WM06, MD05]), or text prediction, in which a given instance (a query, a page, etc.) should be categorized into one or a few categories that the system is confident about. A number of similar evaluation measures (category ranking), but more appropriate for the multilabel setting, are used in [CS03b].

The common precision and recall measures used in text classification and IR are also useful here. They are often computed from the "point of view of a category": for each category, the instances are ranked according to classifier's scores for the category. This is especially appropriate when we are interested in categories one at a time. For instance, when we seek to rank or filter instances based on their degree of membership in a given category of interest. To use the index for ranking of instances with respect to a category, we need the scores to be consistent across instances for the same class, and the current index scores may not be the best choice. One possibility is to map scores (using extra information such as rank) to probabilities. Learning the mapping can be achieved in an online fashion. Confidence (probability) values can then be used for ranking instances. We leave exploring this and other evaluation criteria and computing probabilities to future work.

2.3. Computational Complexity

Can we efficiently compute an index with maximum accuracy given any finite set S of instances? If we constrain the out-degree 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:

Theorem 2.1. *The index learning problem with the objective of either maximizing accuracy (R_1) or minimizing HR, and with the constraint of a constant upper bound (e.g., 1) on each feature's out-degree is NP-Hard.*

The proof is by reduction from the minimum cover problem (see Appendix A). It works much like the proof given in [MGKS07]. In particular, a problem involving only two categories is shown NP-hard. We do not know whether the ranking problem is efficiently approximable however, or whether removing the constraint on the out-degree alters the complexity. Linear programming formulations exist with continuous objectives and no out-degree constraint. Appendices A and B discuss further.

The next section describes a very efficient online algorithms that perform well in our experiments, and we motivate our choices in its design and discuss its convergence in special cases (Section 3.3.4).

3. Feature Focus Algorithms

Our index learning algorithms may be best described as performing their operations from the features' side or features' "point of view" (rather than the classes' side), and hence we name the whole family *feature-focus* algorithms. As we will explain, this design was motivated by considerations of efficiency (Section 3.3.3). The basic question for each feature is to which subset of categories it should connect to (possibly none), and with what weights. We will first describe the computation that a single feature performs. Then we will describe two algorithms, IND and FF, based on this computation.

3.1. The Case of a Single Feature

Assume instances arrive in a streaming manner (from some infinite source), and each instance is described by a single category and the vector of features active in it. In this section, assume $|F| = 1$, *i.e.*, every instance has one and the same feature[4] and we assume Boolean ($F_x[f] \in \{0, 1\}$). Thus, we basically obtain a stream of categories. We next argue that our objective of a good ranking, subject to efficiency, reduces to computing the proportion in the sequence for those categories (if any) that exceed a desired proportion threshold.

In this single feature case, categories 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 categories, say d_{max} at most. The question is to which categories the feature should connect to, and with what weights, so that an objective such as R_k or HR (harmonic rank) is maximized. We will focus on the scenario where the stream of categories is generated by an iid drawing from a fixed probability

[4]Alternatively, imagine the substream of instances that have the same feature active.

distribution. Later we touch briefly on considerations of drifting in the category proportions in the stream.

It is not hard to verify that the best categories are the d_{max} categories 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 finite sequences, this can easily established.

Lemma 3.1. *A finite sequence of categories is given. To maximize HR, when the feature can connect to at most k categories, a k highest frequency set of categories should be picked, i.e., 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 categories 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 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 category c appears in the sequence. For any chosen set S of size k, a pair of categories (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 categories from a fixed distribution over a finite number of categories, the empirical proportions of categories, 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.

3.1.1. Finite Memory Constraints

We seek space-efficient and fairly simple online algorithms, as the stream of categories can be long or unbounded, and we need to do such computations for each of millions of features. Our generic weight update algorithm and an explicit instantiation, call it *Feature Streaming Update* (FSU) are given in Figure 3. We will use FSU in our main feature-focus algorithm. 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 $f \in x$). 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 out-degree 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 5.3.1).

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

(a)

```
/* 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 */
        $w_{f,c} \leftarrow 0, w'_{f,c} \leftarrow 0$
```

(b)

Figure 3: (a) Generic weight updating: when an update is required, each active feature strengthens its weight to the true category, weakens its other connections, and drops those that are too weak. An effect of this is that each feature attempts to improve the ranking and scores it assigns to the categories it is connected to, subject to space constraints. (b) An instantiation of weight updating: FSU. The connection weight of f to the true category c_x is strengthened. Others connections are weakened due to the division. All the weights are zero at the beginning of index learning.

It is instructive to look at how well FSU does in approximating proportions for the (sub)stream of categories that it processes for a single feature. This gives us an idea of how to set the w_{min} parameter and what to expect. There are two causes of inaccuracies to the true proportions:

- Finite samples.
- Setting small weights 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 category 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), *e.g.*, $p > 4w_{min}$, this probability is relatively small. For example, for $w_{min} = 0.01$, and $p = 0.05$, the probability of not seeing such a category for a stretch of of 100 trials is 0.006. More generally, the chance of being set to 0 (dropped) for a category with occurrence probability p diminishes with increasing the ratio $\frac{p}{w_{min}}$ to be several multiples, and therefore the cause of inaccuracies due to finite memory 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 similar deviations computed when FSU is not memory constrained (w_{min} is set to 0). Figures 4 and 5 show the results. The experiments differ on how we generated the categories and computed the deviations.

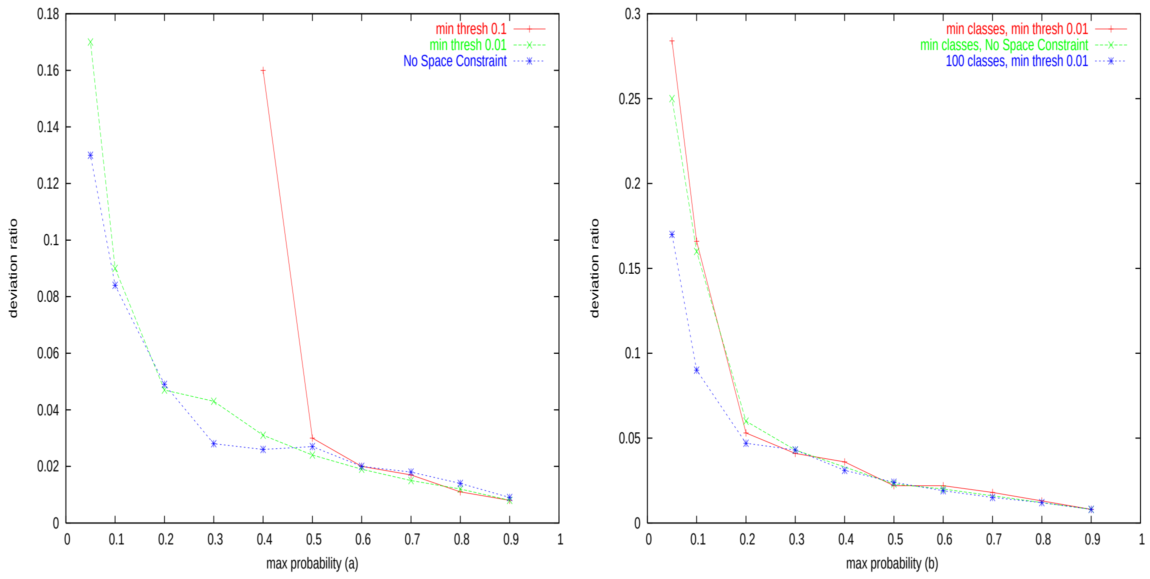

Figure 4: The performance of FSU under different allowances w_{min}. FSU computes the category proportions when seeing a stream of 1000 categories, under two regimes of generating categories. For a choice of highest true probability p^*, the remaining probability mass $(1 - p^*)$ is spread evenly over remaining categories. In (a) the number of categories is $|Y| = 100$, and in (b) it is $|Y| = \lceil \frac{1}{p^*} \rceil$ (*i.e.*, when other categories tie or have close probabilities to the maximum). The experiment is repeated for 200 trials and the average deviation, $\frac{|\tilde{p}_{c_1} - p^*|}{p^*}$, from the true maximum probability, p^*, is plotted against the maximum probability p^*. The number of trials is 100. $\tilde{p}_{c_1}$ is the highest proportion computed by FSU. In (b), the deviation is also compared to the case of 100 categories 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.

In first of these experiments, for some number of categories $|Y|$, we gave one category a highest probability p^*, and the remaining categories obtain the remaining probability mass divided uniformly: $\frac{1-p^*}{|Y|-1}$. We generated a random stream of 1000 categories 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 category ranked highest by FSU, and $\tilde{p}_{c_1}$ is its assigned probability (highest computed probability). We averaged this deviation over 200 trials of repeating the experiments. Figure 4(a) shows the averages when $|Y| = 100$ (so all categories except for 1, obtain $\frac{1-p^*}{99}$). Figure 4(b) shows the results for $|Y| = \lceil \frac{1}{p^*} \rceil$ (*e.g.*, for when $p^* \geq 0.5$, $|Y| = 2$, and when $p^* = 0.05$, $|Y| = 20$). Thus Figure 4(b) shows how FSU with limited w_{min} compares when the classes have similar proportions.

In the second set of experiments, we generated each category's probability uniformly from the $[0, 1]$ interval, keeping track of the total probability generated p during the course of generation. If

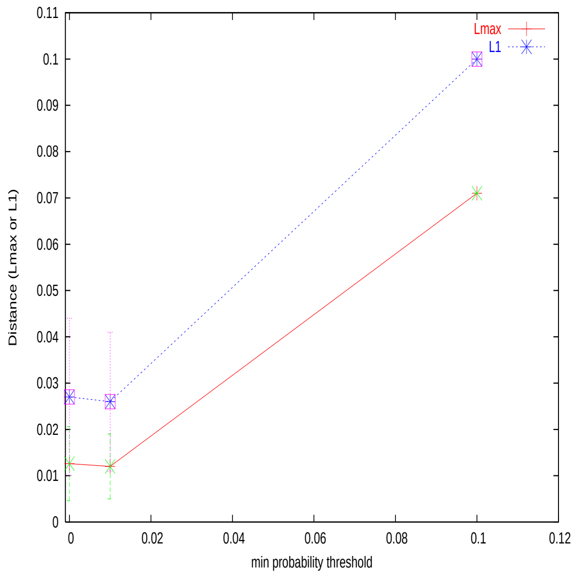

Figure 5: The performance of FSU, under different allowances w_{min}. The vector of true category probabilities is generated by uniformly and sequentially picking from $[0, 1]$, and keeping track of total mass p (which should not exceed 1). If for latest generated category the probability drawn is greater than remaining mass $1 - p$, the remaining mass is assigned, and category generation is stopped. FSU is evaluated after seeing a stream of 1000 categories 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.

the newest category gets a probability greater than $1-p$, 1-p is assigned to it and category generation is stopped. We then sampled iid to get a sequence of 1000 categories. We compared the category proportions that FSU computed using l_1 or l_∞ distance against the true proportions. 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 proportions 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 $w_{min} = 0.01$ performed similarly to $w_{min} = 0$ on the distributions tested. The reason as alluded to earlier is that those categories with proportions significantly greater than w_{min} have a high chance of being seen early 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). Furthermore, we expected that most often the important feature connection weights that determine the true categories during ranking have fairly high weight, say great then 0.05. 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). Also, FSU may not be invoked every time a feature is activated, as we describe below. Finally, vector length is a factor, if there tend to exist strong features-category connections, the weaker connections' influence on changing the ranking will be limited, in the number of active features is adequately small. Thus, in many practical learning problems (certainly the ones we tested), expecting the most useful proportions (weights) to be in say $[0.05, 1]$ may be reasonable (see Section 5.3.3).

3.1.2. 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 ($F_x[f] = 1$), 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 5.3.5. In case of non-Boolean feature values, similar to perceptron and Winnow updates [Ros58, Lit88], the degree of activity of the feature, $F_x[f]$, affects how much the connection between the feature and the true category is strengthened. We could use a *learning rate*, a multiplier for $F_x[f]$, 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 category 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 category will be immediately dropped [5]. For long-term online learning, where distributions can drift, this can slow or stop adaptation, and updates that effectively keep a finite memory or history are more appropriate [6]. In our current experiments, with finite data sets

[5]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!). In any case, at this point or soon after, it's possible to stop updating. With our finite data and small number of passes, this was not an issue.

[6]We could set an upper bound on w_f, *i.e.*, not increase it beyond say 1000, and make other appropriate modifications

```
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 category $c$:
     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 6: Pseudo-code for the IND(ependent) algorithm.

randomly permuted before training, this appears not to be an issue. 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. In general, there are other weight update algorithms with their trade-offs (see our discussion in Section 3.3.3), and we leave further exploration on different tasks for the future.

Before describing our main feature-focus algorithm, we describe the IND ("INDependent") algorithm, which is an offline (batch) algorithm based on computing the conditionals $P(c|f)$. IND is one of the baselines we use to compare performance.

3.2. The IND Algorithm

One method of index construction is to simply assign each edge the category 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. Features are treated as Boolean here ($F_x[f] \in \{0, 1\}$). After processing the training set (computing counts and then conditional probabilities), only weights exceeding a threshold p_{ind} are kept. This not only leads to space savings, but improves accuracy. The best threshold p_{ind} (for improving accuracy) is often significantly greater than 0. 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 as a batch algorithm (see Figure 6), *i.e.*, we computed the weights $P(c|f)$ exactly, not in an online streaming manner described[7] above. 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. On larger data sets, it runs into memory problems and becomes very slow during training.

to the updates.

[7]In case the instance belongs to multiple categories, step 1.1.2 is executed for each true category.

```
Basic Mode of Operation:
 Repeat
  1. Get next instance $x$
  2. Retrieve and rank categories via
  active features of $x$
  3. If update condition is met:
    3.1 Update index.
                 (a)
```

```
Algorithm FeatureFocus($x$, $w_{min}$, $d_{max}$, $\delta_m$) /* FF */
  1. RankedRetrieval($x$, $d_{max}$).
  2. Compute the margin $\delta$.
  3. If $\delta > \delta_m$, return.
  4. Otherwise, for each active $f \in x$:
    4.1 FSU($x$, $f$, $w_{min}$). /* update active features' connections */
                 (b)
```

Figure 7: (a) The cycle of categorization and learning (update) (repeated from Figure 2). (b) Pseudo-code for the feature-focus (FF) learning algorithm, which corresponds to steps 2 and 3 in (a).

This aspect points to the importance of space efficiency for large scale learning[8].

The IND algorithm has similarities with the multiclass Naive Bayes algorithm (*e.g.*, [Ren01]). Major differences include the computation of $P(f|c)$ (the reverse) in plain multiclass Naive Bayes, and that for retrieval, we are summing the weights (instead of properly multiplying), similar to some techniques for expert opinion aggregation [GZ86]. We have found that summing improves accuracy. Appendix C elaborates on the differences. IND can also be viewed as a feature selection algorithm, akin to using a heuristic such as information gain.

We can do significantly better than computing proportions for each feature independently. Often features are inter-dependent (such as various forms of redundancy), and especially with increasing feature vector sizes, performance of methods that assume independence can degrade significantly. We next describe the FF algorithm, which addresses some of the dependencies by not invoking FSU every time.

3.3. The Feature-Focus (FF) algorithm

Our main feature-focus (FF) algorithm, shown in Figure 7(b), begins with an empty index, *i.e.*, no edges or connections. This means that all edge weights are implicitly 0 initially. It adds and drops edges and modifies edge weights during learning by processing one instance at a time[9], and by invoking a feature updating algorithm, such as FSU. Unlike IND, FF addresses feature dependencies by not updating the index (invoking FSU) on every training instance. Equivalently, a feature updates its connection on only a fraction of the training instances in which it's active. This is explained next.

3.3.1. When to Update? It is not best to invoke FSU on every training instance. In particular, *"lazy"* or mistake-driven updating (not updating all the time), can to some extent address feature

[8]But note that the FSU algorithm can be instead employed here to keep memory consumption in check.

[9]The feature and category sets can also grow incrementally.

dependencies. It can, for example, avoid over counting the influence of features that are basically duplicates (similar to one rational for mistake driven updates of 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 a simple case of two categories, 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 both categories, 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, *i.e.*, a instance $x = \{f_2\}$ (has f_2 only), we would like 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 (increased weights the same way), 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 category 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 make 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 5).

On the other hand, if we don't update at all when the right category 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 discussion elaborates.

Case 2. Consider the single feature case and three categories 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 category 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 this 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, over 100 80-20 splits, always (proactive) updating FSU 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 category 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.

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

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

If the margin δ does not exceed a desired margin threshold δ_m, we update[10] (invoke FSU). Note that both s_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 case 1, but underperform for case 2. With a sufficiently high margin, updates are always made and case 2 is covered. 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 we the vectors are l_2 normalized, we have observed that on average a top category obtains scores in the $[0, 1]$ range as well, irrespective of data set or choice of margin threshold. An alternative worth exploring is to set the margin threshold dynamically by keeping statistics on the scores of the top retrieved categories and their differences.

Our use of margin is somewhat similar to the use of margin for online algorithms such as perceptron and Winnow (*e.g.*, [CC06, CS03b]), although our particular motivation from considering case 2, "stability" or keeping some "extended memory" for each feature, appears to be different.

3.3.2. Rating Features: Downweigh Infrequent Features It may be a good idea to down-weigh or eliminate those features' votes that are only seen a few times, as their proportion estimates 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 to the category it gets connected to. This is undesired. Of course, how much to down-weigh can depend on the problem, and how features are generated. In our experiments, during category retrieval, we multiply a feature's vote, $w_{f,c}$, by $\min(1, \frac{\#_f}{10})$, 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 extension improves accuracy.

3.3.3. Summary and Discussion The FF algorithm aggregates the votes of each features for ranking and classification. During learning, FF may be viewed as directing a stream of categories to each feature, so that each feature can compute weights for a subset of the categories 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 be become some hard to categorize subsequence. Each feature computes the proportions in the stream it gets. Features may be space constrained: they need to be space efficient in the number

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

of connections they make as well as in computing their connection weights. This efficiency aspect is especially important in large scale learning.

The FF algorithm has similarities with online algorithms such as Winnow [Lit88], as it normalizes (in general weakens some of) the weights, and the perceptron algorithm [Ros58], as for example the updates are additive. The important difference that changes the nature of the algorithm is that changes to weights are done from the feature's side, unlike Winnow or perceptron. The Winnow algorithm does the normalization from the class's side: 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. 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 here, this weakening happens irrespective of whether a class was ranked high (this aspect is similar to Winnow, but again, for classes instead of features).

A number of learning algorithms in the family of linear classifier learning algorithms, focus on the class side, *e.g.*, learning a prototype classifier for each class. This is natural for binary classification problems. In online updating, the classifier needs to decide which features are important for it, via promotions (*e.g.*, additive or multiplicative strengthening) and demotions (*e.g.*, subtraction or normalization). For instance, the perceptron update is of the form: $\alpha x \pm \mathbf{p}$, where $\mathbf{p}$ here denotes a class prototype to which the vector of active features x is added, or subtracted, and α incorporates a learning rate. In our case, it is each feature that assesses the relevance of categories to it. It is best to view each feature as a ranking module or "expert", so that we obtain a good ranking of the categories, rather than each class having its own prototype (binary classifier). A prototype for a category is most appropriate for ranking instances for that category.

We focused on computations from features' point of view, as compared to the category point of view, due to several efficiency considerations. It is best to think in terms of online streaming processing. To keep memory consumption in check, it seems most direct to constrain features not to connect to say more than 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. A second related reason is constraining the feature out-degrees to remain reasonable 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, and thus processing the class, to examine importance of features, can take more time. Finally, we seek rapid categorization per instance, and constraining out-degree of categories may not guarantee that the out-degree of commonly occurring features would be small. Thus, constraining the degrees of categories would only have an indirect effect on the average time required for processing an instance. Section 3.1 also explains the benefits that the design of FSU provides for efficiency.

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 (*e.g.*, when ranked higher than the true positive), may obtain negative connections to features they weren't connected to. This hurts sparsity. Moreover negative connections may not be as useful in a many-class ranking task as they are useful in a binary prototype scenario: in the many class case, the true category could have higher weights to the appropriate features. Of course, our discussion does not preclude efficient algorithms that, nevertheless, perform their operations from the class side. We have also noted that there are a number of alternatives to the basic FSU and FF algorithms. For example, the edge updates, strengthening of the weights, can be multiplicative or take another form[11]. We mentioned the possibility for having updates that effectively keep finite memory in the counts and the weights, which appears more appropriate for keeping adaptability in very long term learning with potentially significant drift [Mad07]. We leave exploring alternatives to future work.

3.3.4. Convergence Assume the Boolean case. It is not hard to show that the FF algorithm converges in simplified cases, for example when every category is a perfect (noise-free) disjunction. Note that in this scenario, and in the single-class per instance setting, each feature is either irrelevant or relevant to at most one class. We know then that the true feature-category connections are never weakened. More generally, FF and IND should succeed on noisy-OR type of categories, as long as the level of noise is relatively small. A noisy-OR (or noisy-disjunction) category is true with some probability if one of the features in its disjunction is true, or otherwise (if none of the features are true) with some very small probability. For instance, assume category c is the noisy-OR of only two features, f_1 and f_2. Let $P_{c,min} = \min(P(c|f_1), P(c|f_2))$. We can extend the definition of $P_{c,min}$ to more than two features. As long as $P_{c,min}$ is relatively high (for example, $P_{c,min} \geq 0.1$), then we can conclude that with high probability, such a category is learned by FF using FSU with appropriate w_{min}, or by IND.

We basically assumed (almost) independence of features in the previous argument. It would be useful to prove convergence of FF type algorithms in more relaxed settings. Convergence and mistake bounds have been established for multiclass perceptron and related algorithms [CS03b, CDK$^+$06]. Another basic and related question is why common ("natural") data sets allow for simple effective learning algorithms. We leave these problems to future work.

4. Classifier Based Algorithms

We compare against hierarchical or top-down training and classification, a commonly used method when a taxonomy of categories, a tree from general categories at the top to specific categories, is available [DC00, LYW$^+$05]. The hierarchical method reduces to one-versus-rest classification when the categories are flat (there is one level), which is another common method for

[11]We have some experience with multiplicative updates. Some issues such as how to initialize weights are not completely settled, and our multiplicative version in general is outperformed by the additive.

multiclass classification [RK04]. 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 for large scale 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 category using all the data. During classification, all the classifiers are applied and their scores rather than classification outcomes are used for ranking[12]. 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.

Linear classifiers such as support vector machines (SVMs) often perform the best in very high dimensional problems such as text classification [LYRL04, Seb02]. We use perceptrons and SVMs in one-versus-rest and top-down methods. We use single pass and multiple pass perceptrons as well as committees of them. Here, each perceptron is represented as a sparse vector and random weight initialization, in $[-1, 1]$, is used when a new feature is added to the prototype [Ros58]. Unless specified, we run the perceptron learning algorithm until the 0/1 error on training is not improved (computed at end of each pass), for 5 consecutive passes. Perceptron committees often obtain performance close to SVMs (*e.g.*, [CC06]), although their runtime can be less.

4.1. Hierarchical Training and Classification

Briefly, hierarchical training works by first training classifiers for the first level categories in a one-vs-rest manner (*e.g.*, [DC00]). Then the same procedure can be repeated for the children of each category 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 categories will be trained on all the instances. For ranking and categorization using hierarchical categorization, we use classifier probabilities. We obtain probabilities from classifier scores by the method of sigmoid fitting [Pla99]. 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 8). A path of candidate categories is terminated if the probability falls under some threshold p_{min}. All the classifier at the first level (corresponding to the categories 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 the threshold giving highest accuracy R_1. All our ranking methods are evaluated on the deepest categories an instance is assigned too. For the evaluation of the top-down method, from the list of candidates obtained

[12]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($x$, $c$, $p$, $\tilde{Y}_x$)
  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 8: 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, $\{\}$).

for a given test instance, any category whose child is also in the list is removed, and the remaining categories are sorted by their assigned probabilities. For the list of the true categories of the test instance, again only those true categories with no child in the list are kept. Then R_1, R_5 and HR are computed (for the highest ranked true positive category).

We note that if we don't use the probabilities and ranking, *i.e.*, 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 (false negative mistakes are very costly). This inferior performance has been noted before too (see for instance [DKS03]).

5. Experiments

Figure 9 displays several dimensions of the data sets that we use, shown in order of class size. The first 6 sets are text categorization data sets, 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 other sets are standard text (categorization) data [RSW02, LYRL04, RSTK03].

On the first three small sets, we compare against one-versus-rest, and our focus is to compare accuracy. On the next 3, Reuters RCV1, Ads and Web, we use the top-down method. Both one-versus-rest and top-down methods use either single perceptron training, committee of 10 perceptrons, or a fast algorithm for learning linear SVMs [KD06]. 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. For the final task, word prediction, as the categories (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

Data Sets	$\lvert S\rvert$	$\lvert F\rvert$	$\lvert Y\rvert$	$E_x\lvert F_x\rvert$	$E_x\lvert Y_x\rvert$
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 9: 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 categories per instance.

available sources in the cases of Reuters RCV1 [LYRL04], and newsgroups from Rennie [RSTK03]. 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 datasets from Yahoo! For the web data set, to obtain a sufficient number of instances per category, we cut the taxonomy at depth 4, *i.e.*, we only considered the true categories up to depth 4. To simplify evaluation, we used the lowest true category(ies) in the hierarchy the instance was categorized under at test time. Thus an instance in Reuters RCV1 corpus on average is assigned two true categories. We note that in many practical text categorization applications such as personalization, categories at the top level are too generic to be informative/useful. For top-down training, we trained the classifier on the internal categories as well. Web and Ads had just over 20 categories 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: each word's surrounding neighborhood of words, 3 on one side, 3 on the other, and their conjunctions provided the features (about 15 many).

Figures 10 and 11 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 categories simply based on the frequency of the categories 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-out-degree during look up). Note that d_{max} of 25 means a category is retrieved as along as it's within the first 25 highest weight connections, even if its weight is not much higher than w_{min}. During training, we look for a positive category within the first 50 top scoring categories. If it's not found, the score of the positive category 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, *e.g.*, we may do better for lower d_{max} values (see Section 5.3.2). 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 category score differences tend not to be much higher than 1.0, when instances are l_2 normalized. The average score of top scoring category was below 1.0 on all data sets.

For the SVM, we report the best performance in R_1 over the regularization parameter set to $C = 1$ or $C = 10$. There are 10 perceptrons in the committee (often, 5 to 20 suffices for much of the 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 newsgroup data set where we use the 10 train-test splits of Rennie *et. al.* [RSTK03], each 80-20, to be able to compare to their results. We used a 2.4GHz AMD Opteron processor with 64 GB of RAM, with light load.

5.1. 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 times. 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 once case, R_1 on the smallest data set, does it have statistically significant inferior performance.

For the classifier based methods, we see there is a good separation from perceptron to SVMs, suggesting that the classification tasks are challenging.

The performance of FF on the newsgroup ties the best performance achieved by Rennie *et. al.* [RSTK03], and they used special feature vector representations, for the linear SVM as well as their methods, to reach that performance.

We observe that comparison based on the HR results often yields similar rankings of the algorithms tested as does R_1. We focus on R_1 (accuracy) as it is a more commonly used measure.

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

In the case of RCVI, for top-down training, we experimented with using a fixed sigmoid (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. For fitting, we used one or more 80-20 splits of training data,

	Rank (HR)	R_1	R_5	T_{tr}	$\bar{e}$	$\lvert\mathcal{I}\rvert$
Small Reuters						
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 ± 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** ± 0.009	0.998	11s	10	74+
FreqBaseline	1.6	0.42	0.86	-	-	-
News Groups						
δ_m=0, p=1	1.137	0.798	0.978	2s	10	113k
δ_m=0.5, p=1	1.085	**0.865** ± 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 ± 0.005	0.975	92s	20	189+
FreqBaseline	3.33	0.05	0.25	-	-	-
Industry						
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**± 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 ± 0.009	0.933	235s	104	330+
FreqBaseline	31	0.005	0.03	-	-	-

Figure 10: Comparisons. T_{tr} is training time (s=seconds, m=minutes, h=hours), w is the number of "connections" touched on average per feature of a test instance, and $|\mathcal{I}|$ denotes the number of (nonzero) weights in the system (see Section 5.2). 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.

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 this takes more time. For Ads, we could run the SVM with no fitting. Committee and perceptron used 3 trials.

The competitive and even superior ranking performance of the FF algorithm provides evidence that improving category ranking on each instance, in the context of other categories that may be relevant (other retrieved categories), while keeping the index sparse, is a good strategy or learning bias for our high performance categorization task. Binary classifier based methods may be at a disadvantage since the task of choosing whether a single category should be assigned or not, in isolation, appears hard and error-prone, with large numbers of categories. Classifier scores can be

	Rank (HR)	R_1	R_5	T_{tr}	$\bar{e}$	$\vert\mathcal{I}\vert$
			Reuters RCV1			
$\delta_m = 0$, 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	760+
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			
$\delta_m = 0$, 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	5+
SVM C=10, 0 fit	1.424	0.665 ± 0.003	0.774	12h+	80	5+
FB	35.56	0.012	0.033	-	-	-
			Web			
$\delta_m = 0$, 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	14+
FreqBaseline	10.4	0.053	0.126	-	-	-
			Jane Austen			
$\delta_m = 0$, 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 11: Comparisons. T_{tr} is training time (s=seconds, m=minutes, h=hours), $\bar{e}$ is the number of “connections” touched on average per feature of a test instance, and $|\mathcal{I}|$ denotes the number of (nonzero) weights in the system (see Section 5.2). 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.

used for ranking categories, but the classifiers were not obtained with the objective of a good ranking of the categories for each instance: in fact, a typical binary classifier such as an SVM is trained to yield a separation among instances (a good class prototype), which should be more suitable for ranking the instances for the corresponding class than ranking categories for each instance.

5.2. 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, to exceed two orders of magnitude.

Our measure of work, $\bar{e}$, 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 (categories) 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 9), 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 $|\mathcal{I}|$ 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[13]. On the smaller data sets, the difference is not significant. 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 $\bar{e}$ 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). The work of Liu *et. al.* also reflects the considerable engineering effort required and the need for distributing the computation [LYW$^+$05].

5.3. 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

[13]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 ± 0.008	0.884 ± 0.008	0s vs 0s
News Group	0.866 ± 0.006	0.865 ± 0.005	3s vs 2s
Industry	0.889 ± 0.009	0.886 ± 0.008	9s vs 4s
RCV1	0.787 ± 0.007	0.787 ± 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 12: No constraints on d_{max} (maximum out-degree) 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.

that using that option can make. In each case, unless otherwise specified, the remaining parameters (such as choice of margin) are as in Figures 10 and 11 for best performance, and as before we report averages and standard deviations for 10 random trials of 90-10 splits (except for news groups, for which the 80-20 split is given).

5.3.1. 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 performance 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.

5.3.2. Out-Degree Constraint Figure 13 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 out-degree constraint of 3 for RCV1, the number of edges in the learned

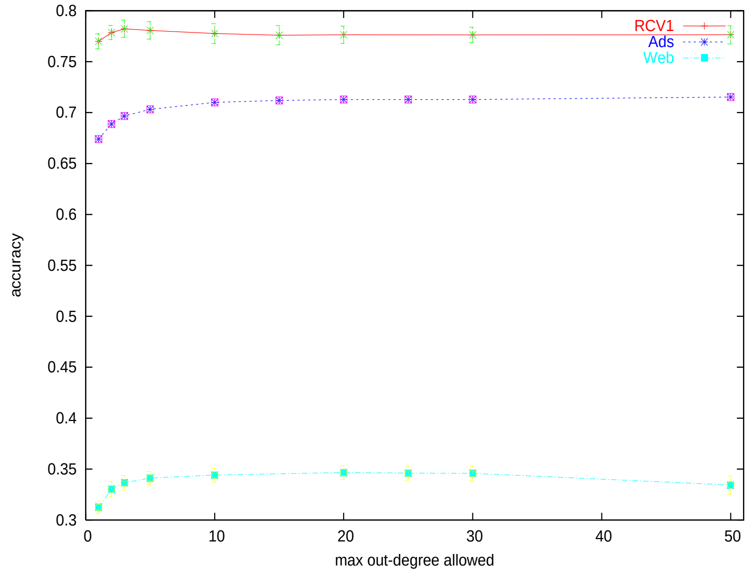

Figure 13: Accuracy (R_1) after one pass against the out-degree constraint.

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 (Figure 11).

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

5.3.3. Minimum Weight Constraint We noted in Section 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 14 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 11, 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.

5.3.4. Multiple Passes and Choice of Margin Figure 15 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.

	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 14: 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.

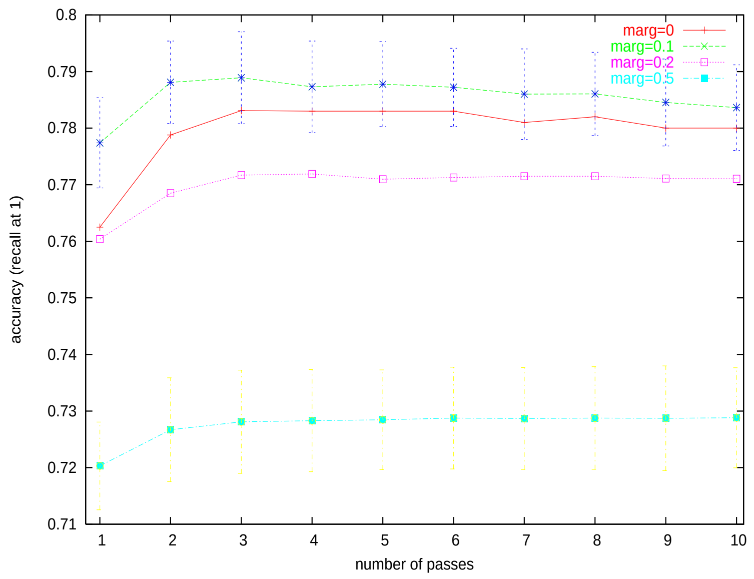

Figure 15: Reuters RCV1: Accuracy (R_1) for margin threshold $\delta_m \in \{0, 0.1, 0.2, 0.5\}$ against the number of passes.

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

Figure 16: Allowing leakage (Yes) when dropping edges can significantly help at times over disallowing it (NO).

	No	Yes
Newsgroup	0.860 ± 0.005	0.865 ± 0.005
RCV1	0.758 ± 0.007	0.787 ± 0.008
Web	0.327 ± 0.006	0.352 $\pm$ 0.007

Figure 17: Down-weighing infrequent features can significantly help.

5.3.5. Disallowing Weight "Leaks" An uninformative feature such as "the" should give low votes to all categories. However, since the out-degree 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 16 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.

5.3.6. Down Weighing Little Seen Features Figure 17 shows the effect of down weighing infrequent features (the default option, see Section 3.3.2), against treating all features as equal (not using the option). Down-weighing infrequent features can significantly help.

5.3.7. 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 IND's test accuracy performance 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

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

Figure 18: 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 use of values, l_2 normalization) is utilized (from Figure 10). 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.

afterward. Figure 18 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 for IND's accuracy performance, on RCV1, if we chose $p_{ind} = 0$ (did no edge removal), we would get R_1 averaging below 0.58 (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 categories receive during retrieval can increase significantly with Boolean features (compared to using the feature values in l_2 normalized vector format).

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

5.4. Other Experiments

Here, we first compare to an older indexing method [MGKS07, MG06] and then report and discuss some properties of the FF learning algorithm, such as the training performance, average scores of the top category, and the type of connections learned.

5.4.1. Comparison to Older Index Learning The first idea for use of an index was to drastically lower the number of candidate categories 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 [MGKS07]. Here, we briefly compare using that method for ranking categories against our current method. We have already noted that (binary) classifiers appear inferior for category ranking, especially as we increase the number of classes, in our comparisons in one-versus-rest experiments. Here, we give results showing that adding an intermediate index trained as in [MGKS07] does not improve accuracy, and furthermore FF is significantly faster.

The indexer algorithm of Madani *et. al.* uses a threshold t_{tol} during training and updates the index only when more than t_{tol} many false positive categories are retrieved on a training instance.

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

Figure 19: The performance of the non-ranking indexer algorithm [MG06]. The objective of ranking improves ranking (and ultimately classification) here.

We report accuracy under two regimes: (1) when only using the category-feature weights (without training classifiers), (2) when training classifiers, here committee of 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 the 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 reference. We also compare when classifiers are used for ranking. For further details on that algorithm, please refer to [MGKS07, MG06].

Figure 19 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 10), 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 note that while the index learning is fast here, the classifier learning still significantly slows down the experiments compared to using FF (an order of magnitude on the data we are reporting on), and the classifiers also require 10 or more passes to reach best performance.

5.4.2. Training versus Test Performance of FF Figure 20 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.

5.4.3. Poor 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

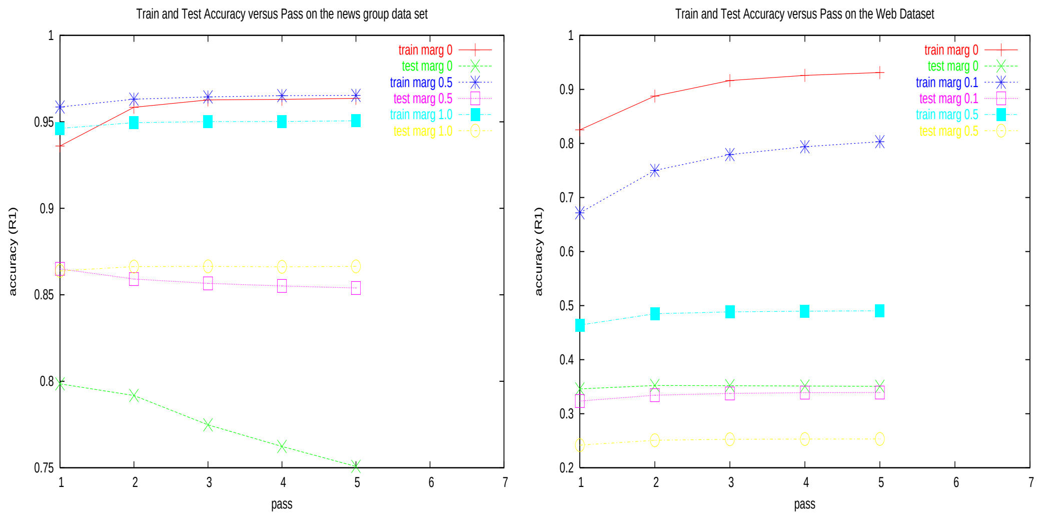

Figure 20: Train and Test Accuracy versus the number of passes, on the newsgroup (left) and web (right) data sets. The increasing the margin threshold can help control overfitting, but may not result in best test performance.

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 or a linear SVM trained for the class (5 to 10% reduction in absolute value of Max F1 compared to perceptron on Reuters-21578 categories).

5.4.4. Scores In Figure 21 we show average of top scoring category ranges on the test set, as well as the difference of top score from the category ranked second, and the (top) margin (highest scoring true positive score minus highest scoring negative score), after one pass, for several of the data sets, and for the choice of several margin thresholds 9during training). As the number of categories increases, the range of the top score as well as the difference and margins all decrease. This trend also correlates roughly with the accuracy performance (R_1).

5.4.5. Feature Occurrence Counts Figure 22 shows, on the larger data sets, the number of features seen at least once (active in some training instance) during the first training pass, and the proportion of that number that is seen below a certain threshold (100, 200, and 500), *i.e.*, the number of training instances in which the feature was active. We see that close to or more than 95% of features are seen below 100 times, and almost all (≈0.99) are seen below 500. Since the number of samples on almost all features is below several hundred, computing accurate category

	Avg Top Score	Difference From Next	Top Marg
Reuters	2	0.7-1.0	1.03
News Group	1.2-1.4	0.4-0.8	0.47
Industry	0.46-0.5	0.28-0.35	0.33
RCV1	0.4-0.6	0.14-0.27	0.14-0.15
Ads	0.23-0.35	0.09-0.18	0.10
Web	0.2-0.3	0.13-0.15	-0.02
Jane Austen	0.1-0.2	0.02-0.08	-0.04

Figure 21: Range of average scores of tops ranking category, range of the difference in score from next highest category, and average margin score of top positive category minus highest negative, on test instances. The ranges are reported for a few different margin thresholds during training (0, 0.1, 0.5, 1.0). As the number of classes increases, the average top score, the difference, and the top-margin all tend to decrease.

	count	100	200	500
RCV1	45k	0.948	0.968	0.984
Ads	290k	0.971	0.984	0.993
Web	6.4 mil	0.979	0.987	0.994
Jane Austen	3.0mil	0.971	0.985	0.994

Figure 22: Counts and proportions of features seen at least once during training, but below a given count threshold (100, 200, and 500).

proportions below say 0.05 in such streams can be of limited utility. Of course, the frequent features can have more of an impact on accuracy, but note that the commonly occurring features are often the uninformative ones.

5.4.6. Examples On RCV1, there were about 300 feature-category connections with weight greater than 0.9 (strong connections). Examples are: “figurehead” to the category “EQUITY MARKETS”, “gunfir” to the category “WAR, CIVIL WAR“, and ”manuf” (manufacturing) to“LEADING INDICATORS”. Examples of features with large “leaks” (with $w_f = \sum_c w_f < 0.25$) are “ago”, “base”, “year”, and “intern”.[14]

[14] The feature “the” is likely to have been dropped (a “stop” word) during tokenization of this data set [LYRL04].

6. Related Work and Discussion

Related work includes indexing, multiclass learning, feature selection, expert aggregation, online and streaming algorithms, and concepts in cognitive science.

Indexing has been used to speed up nearest neighbor methods as well as classification schemes (*e.g.*, [GM98, RBS07, GM07a]). Indexing can certainly be used to index already trained (linear) classifiers, but the issues of space and time efficient learning and classification remain. Learning of an index was introduced in Madani *et. al.* [MGKS07, MG06], where the problem of efficient classification and learning under myriad classes is motivated. As we saw, our experiments indicate that learning to rank appears to be better a strategy than classifier training, and we do not incur the overhead of classifier training. However, for some problems, some type of staged classification may be useful for improved accuracy.

One-versus-rest [RK04] or use of a class hierarchy (taxonomy) (*e.g.*, [LYW$^+$05, DC00, KS97]) for the top-down method are simple intuitive techniques commonly used for text categorization. We saw that index learning offers a number of efficiency advantages, as well as flexibility. No taxonomy or separate feature reduction pre-processing is required. Indeed, our method can be viewed as a feature selection or reduction method. We are not aware of other methods that perform efficient and effective feature selection in our setting of many classes and millions of instances.

Taxonomies offer a number of efficiency and/or accuracy advantages [KS97, LYW$^+$05, DC00, DKS03], but also present several drawbacks when used as an aid for training and classification. Issues such as multiple taxonomies, evolving taxonomies, unnecessary intermediate categories, or not having a taxonomy are all difficulties for the tree-based approaches. On the other hand, researchers have shown some accuracy advantages of top-down compared to flat one-versus-rest training (in addition to efficiency) [DC00, LYW$^+$05, DKS03] (this depends on the method and the loss used). Our current indexing approach is flat. Extensions may improve accuracy. An 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 is independent of others. Our method appears to be inherently sequential.

The multiclass (multilabel) perceptron (MMP) algorithm of Crammer *et. al.* [CS03b] shares the ranking and efficient online training, in particular when implemented as an index. However, MMP, and to the best of our knowledge, other existing algorithms (*e.g.*, [CS03b, CS03a, CDK$^+$06]), lack the weight reduction aspects, as problems with myriad classes had not been the focus in previous work. For example, in the MMP algorithm the features' connections can increase, in terms of the magnitude of weights as well as the number of nonzero weights, without bound. For large scale and long term learning [Mad07], this is a crucial issue. We saw in Section 5.3.1 that if we remove the constraints w_{min} and d_{max}, space and time efficiency greatly suffers. A main difference in design is that previous online algorithms are best seen as performing the computations with respect to each class (the prototypes), rather than the predictors (the features), and we explained the efficiency considerations that motivated our different design (see Section 3.3.3 and Section 3.1). Section 3.3.3 further compares and contrasts FF with existing online updates. We plan to conduct experiments to compare performance and further explore the algorithms in future work. Pros and Cons of efficient index learning and use versus other multiclass methods such as nearest neighbors are discussed in [MGKS07, MG06].

FF in its focus on features has similarities with additive models and tree-induction algorithms [HTF01], and is in the family of so-called expert (opinion or forecast) aggregation and learning algorithms (*e.g.*, [DMP$^+$06, CBFD$^+$97, GZ86]). In typical experts problems, all or most experts provide their output, and the output is usually binary or a probability (the outcome to predict is binary). Here only a small set of features are active in each instance, and only those features are used for ranking. In this respect, the problem is similar to the setting of the sleeping experts algorithm (*e.g.*, [CS96]). Differences or special properties of our setting include the fact that here each expert provides a partial 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"). Learning different weights for the experts (features) can be useful, in addition to our current scheme where each feature adjusts its votes subject to space budgets, and our preliminary experiments show promise for the approach of learning expert weights. We leave exploring this possibility for future work, but note that the technique of down-weighing infrequent features (see Section 3.3.2) is one form of differential expert weighting.

The field of data streaming algorithms studies methods for efficiently computing statistics of interest over data streams, for instance reporting the items with proportions exceeding a threshold, or the highest k proportion items (sometimes called "hot-list" or "iceberg" queries), under certain efficiency constraints, for instance at most two passes and poly logarithmic space (see for example [FSGM$^+$98, GM99]). 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 category, and are essential in significantly improving the final rankings. An algorithm similar to FSU for the Boolean case is used as a subroutine by Karp *et. al.* [KPS03], for efficiently computing most frequent items. In some scenarios, there can be drifts in proportions, and then online and possibly competitive measures of performance may become important [CA98, AW98]. In this ranking and drifting respect, the FSU task has similarities with online list-serving and caching [CA98], although we may assume that the sequence is random (*i.e.*, not ordered by an adversary). Some connections and differences between machine learning and space efficient streaming and online computations are discussed in [GM07b].

Statistical language modeling is accomplished mostly by n-gram (markov) models [Goo01], but the supervised (or discriminative) approach may provide superior performance due to its flexibility for incorporating expressive features (*e.g.*, [EZR00]). Prior work focused on discriminating within a small (confusion) set of possibilities [EZR00]. In prediction games [Mad07], learning tasks based on improving predictions and involving very large scale and long-term online learning are explored.

The common precision and recall measures are computed from the point of view of a category, to evaluate ranking of instances for the category [Seb02]. This is appropriate for tasks such as news or email filtering. Our task is the reverse: given an instance, quickly rank a few categories from the many thousands possible, appropriate for large scale high performance categorization. Section 2.2.1 further discusses accuracy criteria.

7. Conclusions

Learning problems with large numbers of categories in addition to instances and features present several challenges in striking a good balance between accuracy and efficiency. Our work provides evidence that there exist very efficient online supervised learning algorithms that nevertheless enjoy competitive or better accuracy performance with other commonly used methods. We have formulated the task of learning and classification under many classes and instances as one of efficient learning of a sparse feature-category index. The algorithms presented are primarily online, and they may best be viewed as performing the computations from the side of features (the predictors) rather than the classes (the predicted). Each feature performs computations to determine to which categories it may connect to and with what weights. In particular, in large scale learning, each feature is space constrained in performing its computations and in the number of categories it may connect to.

There is much to be done in terms of advancing the algorithms as well as developing an understanding of their success and limitations. We would like to investigate different feature update methods, index learning algorithms, and objectives, and to develop theoretical guarantees. We also intend to explore applications to different domains.

Acknowledgements

Many thanks to Dennis DeCoste, Scott Gaffney, Jian Huang, Sathiya Keerthy, Ravi Kumar, 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.

A. Computational Complexity

We show here that a basic formulation of the weighted index learning problem is NP-hard. For the purpose of establishing hardness, the problem is specified by a finite set of instances, each instance specified by exactly one category that it belongs to as well as 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. For the latter, for each instance, the reward is the reciprocal rank $\frac{1}{k_x}$, *i.e.*, the rank of the correct category in the ranking returned by the index. On a single instance, the reward could be 0, if the category is not retrieved, and maxes at 1, if the correct category has rank 1. Note that MRR in Section 2.2 is simply the average reward per instance. For accuracy (R_1), the reward is either 1, if the correct category is ranked highest, or otherwise 0. The decision problem is then to determine whether a weighted index (a weighted bipartite graph) satisfying the out-degree constraint exists that yields a total reward, $\sum_{x \in X} r(c_x)$, exceeding a desired threshold.

Theorem A.1. *The ranked recall problem with either objective of maximizing accuracy or minimizing HR, and with the constraint of a constant upper bound (such as 1) on each feature's out-degree is NP-Hard.*

Proof. The reduction is from the SET COVER problem [GJ79]. We reduce the SET COVER problem to problem of computing an index wherein each feature can connect to at most 1 category.

An instance $\mathcal{I}$ 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 $\mathcal{I}$, we construct an instance of the indexing problem with only two categories c_1 and c_2 such that there is a SET COVER solution of size C for $\mathcal{I}$ iff there is an index (with the maximum out-degree of 1 constraint), such that the maximum total reward, the number of instances for which the right category is ranked highest, is $|U| + |\mathcal{S}| - C$.

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 $e_j \in U$ ($1 \le j \le n$), 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 category 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 category c_2 (see Figure 23).

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 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 category 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 $C \le |U| + |\mathcal{S}| - R$. 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

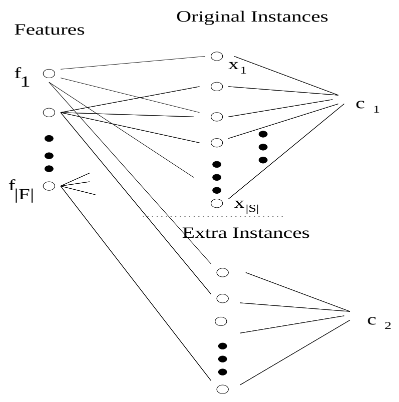

Figure 23: Reduction of minimum set cover problem to the ranked-recall problem.

minimum value v_{min}. Next, we note that any index can be converted to one where for all the original instances are "covered", *i.e.*, the index ranks the right category 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 loose 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 $C = n = |U| + |\mathcal{S}| - R' \leq |U| + |\mathcal{S}| - R$. □

Note that the NP-hardness remains and is easier to show if we use the maximum incoming score rule for category retrieval (each category gets the maximum of its incoming edge weights) instead of the sum. This reduction does not establish NP-hardness of constant-ratio approximability of category 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 to approximate [MGKS07]. With continuous objectives and constraints, there are polynomial time algorithms, as we briefly explore in the next section.

B. Linear Programming (Optimization) Formulations

When there is no limit on the out-degree, one can use a linear program to verify whether there is a "perfect" index, that is one that leads to the maximum reward possible (every instance gets its category ranked highest). However, the linear program is rather big, although special cases may be applicable (see below). Here, we assume features are Boolean and each instance belongs to 1 class for simplicity. For every instance one obtains $|C| - 1$ inequalities, each involving $O(|x|)$ variables: $\forall_{c_j \neq c_x} \sum_{f_i \in x} w_{ij} < \sum_{f_i \in x} w_{ic_x}$ (we could have an extra inequality that $\sum_{f_i \in x} w_{ic_x} > 0$). The question is whether the linear program is feasible. This can be answered in polynomial time.

One can also maximize the sum of the differences in scores, each instance contributing one difference (or margin). For each instance, the difference may be defined as the minimum of the differences between the score of the true category for that instance, and any other category. This is expressible as an LP, as follows:

$$\text{Objective: } Maximize \sum_{x \in S} \delta_x$$

$$\begin{aligned}
\forall x \forall_{c \neq c_x}, \delta_{x,c} &= \sum_{f \in x} w_{f,c_x} - \sum_{f \in x} w_{f,c} \\
\forall x, \forall_{c \neq c_x}, \delta_x &\leq \delta_{xc} \\
\forall f, \forall_c, w_{f,c} \geq 0 & \\
\forall f, \sum_c w_{f,c} \leq 1 &
\end{aligned}$$

One drawback is that there is no constraint on the out-degree, so we get $O(|C||F|)$ many inequalities, which can lead to memory and time ineffciencies. However, the last two constraints are useful for enforcing sparsity in the index learned. Another potential drawback is that maximizing this kind of margin may not be the best for maximizing accuracy: on a few instances, the margin can be large and positive, but for many the margins can be negative. Still, the general approach is worth further exploration and there can be remedies and reformulations, such as SVM soft-margin type formulations, for the potential accuracy (generalization) problems. In particular, a simplified problem that mitigates the efficiency issue is as follows: we may assume feature-category connections that should have nonzero weight are given to us, *e.g.*, by running IND or FF on the data, and we need only compute the weights for those connection to optimize an appropriate objective, such as maximizing accuracy or a choice of margin such as above. In practice, it may be that we have a good idea of what categories to connect a feature to, but optimizing the weights of those connections can lead to significant gains over simpler algorithms such as IND and even FF.

C. Comparison of IND to multiclass Naive Bayes

Here, we show the similarities and differences between our indexing algorithm that treats features independently, IND, and a version of multi-class naive Bayes, somewhat similar to improvements made in Rennie et al [RSTK03]. Assume, we sorted categories by $P(c|x)$, or equivalently, $1 - P(\bar{c}|x)$. We have:

$$P(\bar{c}|x) = \frac{P(x|\bar{c})P(\bar{c})}{P(x)}.$$

For sorting purposes, we may ignore $P(x)$ as it's the same for all categories. Assume features are Boolean, and let x_i denote the value of feature i for instance x (either 0 or 1). Now, making the independence assumption:

$$\begin{aligned} P(x|\bar{c})P(\bar{c}) &= \prod_i P(x_i|\bar{c})P(\bar{c}) && \text{(C.1)} \\ &= \prod P(x_i \cap \bar{c}) && \text{(C.2)} \\ &= \prod P(\bar{c}|x_i)P(x_i) && \text{(C.3)} \end{aligned}$$

Therefore,

$$\operatorname*{argmax}_c P(c|x) \equiv \operatorname*{argmin}_{\bar{c}} P(\bar{c}|x) \equiv \operatorname*{argmin}_{\bar{c}} \prod_i P(\bar{c}|x_i)P(x_i) \equiv \operatorname*{argmin}_{\bar{c}} \prod_i P(\bar{c}|x_i)$$

Finally, if we ignore the cases in which $x_i = 0$ (when the feature is not active) (here, we assume no category will have a high probability if a feature is 0, so the factors $P(c_j|x_i = 0)$ can be ignored), we get a multiplicative version of the indexer IND algorithm (focusing only on the active features), *i.e.*, properly multiply, instead of summing the conditional weights that FSU computes:

$$\operatorname*{argmax}_c P(c|x) \approx \operatorname*{argmin}_{\bar{c}} \prod_{x_i=1} P(\bar{c}|x_i) = \operatorname*{argmax}_c [1 - \prod_{x_i=1} (1 - P(c|x_i))]$$

However, the independence assumption can often hurt accuracy performance when feature vectors are relatively long, say 10s or 100s, and there are redundancies or when very predictive features do not occur in instances sufficiently often. When very predictive features for each category exist (on average, or for the common categories), then IND may perform adequately.

Bibliography

[AW98] 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.

[CA98] Online Computation and Competitive Analysis. *A. Borodin and R. El-Yaniv*. Cambridge University Press, 1998.

[CBFD+97] N. Cesa-Bianchi, Y. Freund, D.Helbold, D. Haussler, R. Schapire, and M. Warmuth. How to use expert advice. *JACM*, 44(3):427–485, 1997.

[CC06] V. R. Carvalho and W. Cohen. Single pass online learning. In *KDD*, 2006.

[CDK+06] K. Crammer, O Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *Jouornal of Machine Learning Research*, 7, 2006.

[CS96] W. W. Cohen and Y. Singer. Context-senstive learning methods for text categorization. In *SIGIR*, 1996.

[CS03a] C. K. Crammer and Y. Singer. Ultraconservative online algorithms for multiclass problems. *Jouornal of Machine Learning Research*, 3:951–991, 2003.

[CS03b] K. Crammer and Y. Singer. A family of additive online algorithms for category ranking. *Jouornal of Machine Learning Research*, 3, 2003.

[DC00] S. Dumais and H. Chen. Hierarchical classification of web content. In *SIGIR*, 2000.

[DKS03] O. Dekel, J. Keshet, and Y. Singer. Large margin hierarchical classification. In *ICML*, 2003.

[DMP+06] V. Dani, O. Madani, D. Pennock, S. Sanghai, and B.Galebach. An empirical comparison of expert aggregation techniques. In *UAI*, 2006.

[EZR00] Y. Even-Zohar and D. Roth. A classification approach to word prediction. In *NAACL*, 2000.

[FP03] D. A. Forsyth and J. Ponce. *Computer Vision*. Prentice Hall, 2003.

[FSGM+98] 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.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GM98] M. Grobelnik and D. Mladenic. Efficient text categorization. In *Text Mining Workshop at ECML*. 1998.

[GM99] 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.

[GM07a] E. Gabrilovich and S. Markovitch. Computing semantic relatedness using wikipida-baed explicit semantic analysis. In *IJCAI*, 2007.

[GM07b] S. Guha and A. McGregor. Space-efficient sampling. In *AISTATS*, 2007.

[Goo01] J. T. Goodman. A bit of progress in language modeling. *Computer Speech and Language*, 15(4):403–434, October 2001.

[GZ86] Christian Genest and James V. Zidek. Combining probability distributions: A critique and an annotated bibliography. *Statistical Science*, 1(1):114–148, 1986.

[HTF01] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer-Verlag, 2001.

[KD06] S. Keerthi and D. DeCoste. A modified finite newton method for fast solution of large scale linear svms. *JMLR*, 2006.

[KPS03] 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.

[KS97] D. Koller and M. Sahami. Hierarchically classifying documents using very few words. In *ICML*, 1997.

[Lit88] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1988.

[LYRL04] 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.

[LYW$^+$05] 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.

[Mad07] O. Madani. Prediction games in infinitely rich worlds. Technical Report 3, Yahoo! Research (and UBDM'06), June 2007.

[MD05] O. Madani and D. DeCoste. Contextual recommender problems. In *Utility Based Data Mining (UBDM) Workshop at KDD*, 2005.

[MG06] O. Madani and W. Greiner. Learning when concepts abound. Technical report, Yahoo! Research, 2006.

[MGKS07] O. Madani, W. Greiner, D. Kempe, and M. Salavatipour. Recall systems: Efficient learning and use of category indices. In *AISTATS*, 2007.

[Pla99] 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.

[RBS07] Y. Ma R. Bayardo and R. Srikant. Scaling up all-pairs similarity search. In *WWW*, 2007.

[Ren01] J. Rennie. Improving multi-class text classification with naive bayes. Master's thesis, MIT, 2001.

[RK04] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *JMLR*, 5, 2004.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[RSTK03] J. Rennie, L. Shih, J. Teevan, and D. Karger. Tackling the poor assumption of Naive Bayes text classifiers. In *ICML*, 2003.

[RSW02] T. G. Rose, M. Stevenson, and Miles Whitehead. The reuters corpus vol. 1 - from yesterday's news to tomorrow's language resources. In *Inter. Conf. on Lang. Resources and Evaluation*, 2002.

[Seb02] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 2002.

[WLW01] 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.

[WM06] S. Wedig and O. Madani. A large scale analysis of query logs for assessing personalization opportunities. In *KDD*, 2006.