

# Hierarchical Categorization for MALACH

J. Scott Olsson [olsson@math.umd.edu](mailto:olsson@math.umd.edu)

*Final report*

Supervisor: Doug Oard, [oard@umd.edu](mailto:oard@umd.edu)

May 18, 2004

# Contents

<b>1</b>	<b>Motivation</b>	<b>3</b>
<b>2</b>	<b>The <math>k</math>NN method</b>	<b>4</b>
2.1	The segment model . . . . .	4
2.1.1	Document mangling . . . . .	4
2.2	Term weighting . . . . .	5
2.3	How $k$ NN works . . . . .	6
2.4	Similarity Measure . . . . .	6
2.5	Class labeling . . . . .	6
2.5.1	Training data acquisition and storage . . . . .	7
2.6	Data structures . . . . .	7
2.6.1	AVL-Trees . . . . .	7
2.7	Complexity Analysis . . . . .	7
<b>3</b>	<b>The HPLC Method</b>	<b>8</b>
3.1	The segment model . . . . .	8
3.2	Training . . . . .	8
3.2.1	Maximum-Likelihood Parameter Estimation . . . . .	8
3.2.2	The Expectation-Maximization Algorithm . . . . .	9
3.3	Categorization . . . . .	11
<b>4</b>	<b>hiercat: an HPLC Implementation</b>	<b>11</b>
4.1	Data preparation and preprocessing . . . . .	11
4.1.1	Specifying the (Poly)Hierarchy . . . . .	11
4.1.2	Preprocessing Training Data . . . . .	12
4.1.3	Preprocessing Testing Data . . . . .	12
4.1.4	Specifying Training Labels . . . . .	12
4.2	Running hiercat . . . . .	12
4.3	Implementation Considerations . . . . .	13
4.3.1	Languages . . . . .	13
4.3.2	Parallelization . . . . .	13
4.3.3	Data Structures . . . . .	16
4.3.4	Complexity Analysis . . . . .	17
<b>5</b>	<b>Validation/Evaluation</b>	<b>20</b>
5.1	Definitions . . . . .	20
5.2	20 Newsgroups . . . . .	23
<b>6</b>	<b>Conclusions</b>	<b>27</b>
<b>7</b>	<b>Future work</b>	<b>27</b>
<b>8</b>	<b>Acknowledgements</b>	<b>28</b>

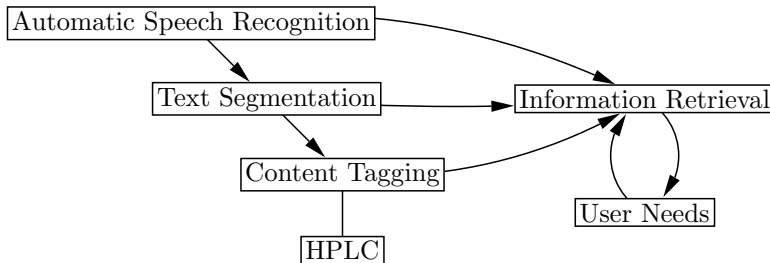


Figure 1: Several large subproblems of the MALACH project, and their relations.

### Abstract

Two automatic text classifiers (a non-hierarchical k-nearest neighbor and hierarchical probabilistic latent categorizer, of Gaussier, et. al [4]) are motivated and explained. Each categorizer has been implemented, and the implementation details are briefly outlined. The categorizers are validated on the popular 20 Newsgroups data set, and it is demonstrated that the hierarchical model improves classification accuracy.

## 1 Motivation

The MALACH Project (Multilingual Access to Large spoken ArCHives) is a joint collaboration between researchers at UMD, JHU, IBM, and the Survivors of the Shoah Visual History Foundation (VHF), whose purpose is to “dramatically improve access to large multilingual collections of recorded speech in oral history archives” [8].

Shortly after the release of *Schindler’s List* in 1994, the VHF began collecting and archiving video testimonies from Holocaust survivors and witnesses. To date, more than 116,000 hours of this film (including some 52,000 persons) has been collected and archived in over 180 terabytes of MPEG-1 video. This data set, the “world’s largest coherent archive of videotaped oral histories” [2], represents fertile ground for research in automatic speech recognition, machine translation, natural language processing and information retrieval.

In order to provide useful access to large spoken archives, many challenging subproblems must be addressed. Figure 1 contains a graph of some of the larger component problems and their relation. If we are to provide useful methods for information retrieval, we must first classify the segmented data in the archive, which requires us to have segmented the textual data, which of course requires us to have transcribed the original video data to text. That is to say, a video testimony must first be transcribed to textual data, broken into small text segments, and categorized, before robust information retrieval can occur.

Categorization may be conceived as a supervised learning problem, wherein a collection of vectors (representing a bag<sup>1</sup> of words in documents) termed the *training set* has for each vector an associated set of labels and the goal is to determine the labeling for other, unlabeled document vectors. These “unlabeled” vectors are termed the *test set*.<sup>2</sup>

In many document collections, documents may be naturally organized hierarchically (using, for example, is-a or part-whole dependency). While most categorizers make no use of this hierarchical structure, we expect that it might be leveraged to improve categorization performance—that is, to improve the discriminatory ability of the categorizer amongst labels.<sup>3</sup> This is especially true in a document collection with a large topic hierarchy and very many classes—properties exhibited strongly by the MALACH dataset.

To leverage our knowledge of the hierarchy, we must build a hierarchical text classifier. The model chosen here is the Hierarchical Probabilistic Latent Categorization (HPLC) model of Gaussier, et. al. [4] As

<sup>1</sup>ie, A multiset (a finite unordered set which respects multiplicity)

<sup>2</sup>In fact, the test set is also labeled, although the labels are reserved for evaluation purposes.

<sup>3</sup>From the hierarchical perspective, we may view categorization as walking from the root of the hierarchy down to some class (a leaf), where every node represents a new decision to be made.

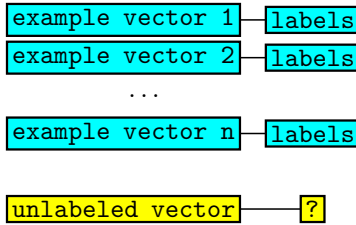


Figure 2: The vector space statement of the categorization problem, which naturally presents itself for non-hierarchical (flat) categorization problems. This is the view taken by  $k$ NN, while other models may state the problem differently (eg, documents may be viewed as mixtures of word probability distributions). This latter view is that taken by the present hierarchical model.

a baseline for comparison with flat (non-hierarchical) models, we first describe and implement a  $k$ -nearest neighbors ( $k$ NN) text classifier.

## 2 The $k$ NN method

### 2.1 The segment model

The  $k$ -nearest neighbors model views documents as vectors, where the  $i^{th}$  vector element represents the salience of term  $i$  in the associated document. Of course, documents come as documents, not as vectors, and so steps must be taken to produce the vectors from the original document data. This process can be roughly broken into two phases: document mangling (which includes document acquisition and formatting as well as dimensionality reduction steps such as stoplisting and stemming) and term weighting. An identical procedure is followed for training and test document mangling, while similarly motivated but slightly different (in form) term weightings are chosen for each.

#### 2.1.1 Document mangling

The MALACH scratchpad<sup>4</sup> data is stored in a flat file of tab delimited entrees, one field of which contains the scratchpad entry (also present are the document’s unique segment and interview identification numbers—used internally to, for example, associate category ID’s with segments in memory). Punctuation is stripped and capitalization is removed. The bulk of this pre-processing is currently being done in Perl (as data formats often change and because, computationally, mangling represents a very small fraction of the categorization cost—all of which may be performed once before several independent categorization runs are made, having perhaps different runtime parameters).

**Stoplisting** Stoplisting is the process of removing words which are expected to not contribute in any meaningful way to the semantic of a document (the canonical example being the word “the”). Stoplists (the list of words to be ignored) are most often (and here) produced by a statistical analysis of representative documents in the problem domain; in our case (as is commonly done) we simply stop on words which occur above some threshold frequency across the training collection. Some example stopwords used are: *the, about, he, she, and, from,* and *of*. These words are not expected to aid in categorization, and their removal can significantly reduce the size of the problem (both in terms of storage and computational cost).

<sup>4</sup>During the interviewing process, many MALACH interviewers took short notes (often summaries) for their own use. This metadata, referred to as the *scratchpad data*, is content rich and (it remains to be seen) will likely aid significantly in the categorization effort for the automatic speech recognition (ASR) data (of which we do not yet have sufficient data for testing).

**Stemming** Stemming also helps reduce the dimensionality of the feature space considered. In stemming, portions of words (namely suffixes) which are not expected to contribute significantly to their meaning are stripped off (on the assumption that words like *happy* and *happiness* contribute in a similar way to a documents semantic). These are typically heuristic algorithms hand tweaked for a particular language (as morphological rules in English will differ significantly from those in Latin or Greek). This *k*NN implementation includes both a Porter and Lovins stemmer (the Lovins stemmer came “for free” with the YASE code base used for inverted file creation, while the Porter stemmer is more frequently used today and was therefore implemented in addition). To do this, I simply modified a freely available C implementation of the Porter stemmer to accept and return string data in the format used internally by the categorizer.<sup>5</sup>

Both stemmers can be chosen at compile time by a compiler flag (preprocessor directive) and are thus modular in the C sense.

## 2.2 Term weighting

In this step, the vectors representing documents in the training set, the *document term weight* vector (*DTW*), are constructed, where for each term *i*,

$$DTW_i = (1 + \log(DTF_i))IDF_i$$

where *DTF<sub>i</sub>* represents the document term frequency (the number of times term *i* appears within the particular document), and *IDF<sub>i</sub>*, or the inverse document frequency, is a weighting which indicates how useful the *i<sup>th</sup>* term is for distinguishing between documents in different classes. This is here calculated as

$$IDF_i = \log\left(1 + \frac{N}{TF_i}\right),$$

where *N* is the total number of documents in the collection and *TF<sub>i</sub>* is the number of documents which term *i* appears at least one time in (the term frequency)—the intuition being that terms which appear in every document (*N/TF<sub>i</sub> = 1*) will not be as useful for classifying as words which appear in only a small subset of the documents (*N/TF<sub>i</sub> > 1*) and that a term which appears twice as often as another may be less useful than the latter, but not half as useful (ie, the weighting is logarithmically damped).

In *k*NN, term weights must also be calculated for documents in the test set (query documents) This follows a similar motivation as in the training vectors, although the weighting is slightly different. This weighting is currently implemented for each *QTW* (query term weight vector) as

$$QTW = \left(\frac{1}{2} + \left(\frac{1}{2} \frac{QTF}{QMF}\right)\right)IDF,$$

where *IDF* is used from the training phase<sup>6</sup>, *QTF* is the term frequency of a given term in the particular query document, and *QMF* is the maximum frequency of any term in the query.

Both the document and query term weighting schemes here described follow the method employed by the YASE project [7] (an open source search engine which I have used as a basis for my code, particularly to construct inverted document files). It is not clear *a priori* whether one term weighting scheme is superior to any other (to make such a decision from the fore, one would need a better understanding of the dataset’s properties). The point here being that the effectiveness of one weighting scheme over another itself carries important information about the dataset’s structure (perhaps, for example, an indication of the importance of the hierarchy). The teams at IBM and at UMD have chosen different weighting schemes (preliminary results indicate the performance of each is differing in significant ways), which will allow this knowledge

---

<sup>5</sup>ANSI C uses a pointer to type `char` and a trailing ‘\0’ delimiter to store text strings; I have chosen instead to implement them as a pointer to *N* + 1 unsigned bytes in sequential memory, where the first byte contains the number of bytes which follow (*N*). This has proven to simplify string manipulation (eg, superfluous calls to `strlen` can be avoided); I have also found it easier to keep track of memory allocation and garbage collection in this way.

<sup>6</sup>Here as always, we assume that the training documents are representative of document in the domain of interest—here the test set.

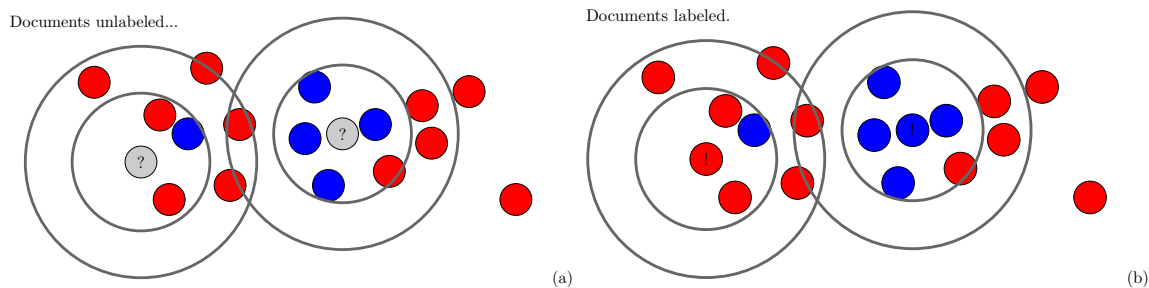


Figure 3: An imagined two dimensional projection of several labeled (red or blue) document vectors in feature space. In (a) the test documents are unlabeled, and in (b) they have taken the labels of their nearest neighbors in the space.

base to grow more quickly (and hopefully, to maximize the effectiveness of a hierarchical classifier). The most significant difference between these  $k$ NN implementations is precisely this choice of term weighting.<sup>7</sup>

### 2.3 How $k$ NN works

With the documents presented as vectors, we are now free to imagine each document as representing a data point in some  $n$  dimensional feature space, where  $n$  is the number of terms across the collection. The very simple idea of the nearest neighbor algorithm is to determine which document in the training set is nearest to the test document in the feature space, and to then assign the test document labels from that nearest neighbor. Figure 3 graphically depicts this idea. The algorithm generalizes as one might expect for  $k > 1$  neighbors.

### 2.4 Similarity Measure

Vectors can be *near* each other in many senses; one could calculate their Euclidian distance, their overlap, or their inner product (perhaps with some normalization). I have chosen to do the latter, and calculate the similarity of a training and test document as

$$S = \frac{(DTW)^T(QTW)}{\|DTW\|_2},$$

so that, within a factor of  $\frac{1}{\|QTW\|_2}$  (which has no effect, since the one query is constant across all the training documents for each categorization attempt),  $S$  is simply the cosine of the angle between  $DTW$  and  $QTW$ . The norm  $\|DTW\|_2$  can be thought of as the length of the particular document vector (which enforces the intuitive constraint that a document with twice as many salient terms as another is not considered more similar than the other if it is simply much longer).

### 2.5 Class labeling

A similarity measure is calculated for a test document's query vector and every document in the training collection that shares at least one term in the query. As similarity scores are calculated, they are inserted into a balanced binary tree (AVL), which removes the need to later sort the scores (cf. Section 2.6.1 below). The  $k$  largest scores are then pulled from the tree and used to determine the test documents labels.

<sup>7</sup>eg, The group at IBM is accounting for document length in the term weighting while I am accounting for it in the similarity measure (by cosine normalization).

### 2.5.1 Training data acquisition and storage

The categorizer must be told which labels are mapped onto which training documents and then provide that data through an interface to other code portions (eg, the labeling code). This happens in the following way: 1) the training labels are available (after some possible preprocessing) in a delimited flat file which contains an entry for each segment ID and it's respective category numbers; this file is called the *ground truth* file. 2) the categorizer parses this file and builds a doubly linked list containing a key for each segment ID, as well as a pointer to a byte containing the number of category numbers (unsigned bytes) that follow in sequential memory.<sup>8</sup> In the immediate future I will be switching this storage into a balanced binary tree (it had been implemented before a tree code), which has an element retrieval time of  $O(\log N)$  (a significant improvement over the worst-case  $O(N)$  of a list traversal). 3) This category data is then provided to the rest of the categorizer by a function call which includes a pointer to the doubly linked list as well as the segment idea (key) for which category data is requested; the function returns with a pointer to the first byte of the requested data (which, recall, stores the number of category numbers, or bytes, which follow in sequential memory).

At this point, we have  $k$  nearest training documents, their similarity scores, and their associated categories ID's.

Labeling now proceeds in the following way: 1) a new singly linked list of all the categories returned for the  $k$  nearest neighbors is created, wherein each list element contains the sum of all similarity scores associated with that element's respective category number. These sums of scores are termed the *category scores*. 2) All categories having a score above a *category score threshold* are taken as labels for the test document<sup>9</sup>.

## 2.6 Data structures

### 2.6.1 AVL-Trees

An AVL (Adelson-Velskii and Landis) Tree[10] is a binary search tree<sup>10</sup> with a balancing condition<sup>11</sup>—that is, as elements are inserted into the tree, the tree is *rotated* to ensure that the node taken as root has (within 1) equal number of descendents on its right and left.

Such a tree is currently being used to store the similarity scores as they are computed; consequently, as we only desire the  $k$  largest scores and the tree is guaranteed to have a height of  $O(\log_2 N)$ , we can find our  $k$  nearest neighbors in only  $O(k \log_2 N)$  time. Alternatively, the scores would have to be tallied in a list and then sorted, but the asymptotic best time for a sort is  $O(N \log_2 N)$ —we can achieve this best possible runtime with less work by simply using a balanced tree.

## 2.7 Complexity Analysis

Yang et. al [11] give the training time complexity for  $k$ NN as  $O(L)$  and the scoring time complexity as  $O(NL_v^2/V)$ , where  $L$  is the total number of word-document co-occurences in the training set,  $N$  is the

---

<sup>8</sup>This step could be done instead during the training phase, storing the category data in the inverted document file for each document ID (and in fact, this is how I originally implemented it); I have chosen rather to do it during categorization because the operation runs in constant time and we may conceivably change training labels between runs (and will then not want to reconstruct the inverted indexes of our training data).

<sup>9</sup>Contrast this to the simpler problem for which only one category is to be assigned per test document, in which case we might only take the category with the highest returned score.

<sup>10</sup>A binary tree is a simple, undirected, connected, acyclic graph with one node denoted the root; every node contains a key and has at most two edges departing from it (termed the *left* and *right* edges). Every descendent along a node's right edge will have only keys larger than the node's key, and likewise, every descendent along a node's left edge will have only keys smaller than the node's key. Elements which have no descendents are termed *leaves*. The *height* of the tree is the number of edges along the longest traversal from root to a leaf.

<sup>11</sup>One can imagine a worst case in which elements were inserted into a tree already sorted, so that the root would have the smallest key and successive elements always descended to the right; in other words, a list rather than tree would be built. A balanced tree then is simply a tree such that the root has an equal number of nodes descending on its right and left (within 1); a balanced tree ensures that the height of a full tree is  $O(\log_2(1 + N))$

number of training documents,  $L_v$  is the average number of unique words in a document, and  $V$  is the total number of unique words (the vocabulary size).

In this implementation, category sorting is done in  $O(k \log_2 N)$  (although Yang et. al. note that this may be slightly improved by clever modification of the quicksort algorithm).

## 3 The HPLC Method

### 3.1 The segment model

Documents are most often modeled using words as features and their frequency as the feature value. In HPLC, however, the modeled features are co-occurrences of words  $j$  in segments  $i$ ,  $(i, j)$ .

Documents are conceived to be probabilistically generated from a mixture of topic conditional word distributions. Figure 16 shows an example hierarchy for the 20 Newsgroups data set. The distribution of words at the leaf nodes (eg, `rec.sport.hockey`) is then some mixture of the distributions which lie in the leaf’s heritage (ie, a mixture of the `sports` and `root` topic conditional word distributions).

More specifically, the model assumes the data has been probabilistically generated in the following way:

1. Pick a segment class  $\alpha$  with probability  $P(\alpha)$ .
2. Choose a segment  $i$  using the class-conditional probability  $P(i|\alpha)$ .
3. Sample a word topic  $\nu$  with the class-conditional probability  $P(\nu|\alpha)$ . Note, topics may only be chosen if they are ancestors of the class already chosen in Step 1.
4. Generate word  $j$  using the topic-conditional distribution  $P(j|\nu)$ .

A segment class  $\alpha$  refers to a set of segments sharing some (here undefined) common thematic features (eg, the segments all refer to “the post war era” or “propaganda”), while a word topic  $\nu$  denotes a semantic field which may be described by a particular word distribution. Class conditional word distributions can be seen then as weighted mixtures of their ancestral topic conditional word distributions.

In this model, segments may be partly relevant to different topics, and segments from different classes may share words from topics in their common ancestry. From our knowledge of the data set, we expect some hierarchical structure to be exhibited (eg, Berlin descends from Germany which descends from World). Therefore, it is hoped such a (hierarchical) model will more accurately categorize the segments.

According to the model, the probability of a particular word/segment co-occurrence,  $P(i, j)$ , is given by:

$$P(i, j) = \sum_{\alpha} P(\alpha) P(i|\alpha) \sum_{\nu \uparrow \alpha} P(\nu|\alpha) P(j|\nu) \tag{1}$$

(where we have used the fact that segments and words are conditionally independent). The notation  $\nu \uparrow \alpha$  signifies we sum over all  $\nu$  which are ancestors (upwards) of  $\alpha$ ; in effect,  $P(\nu|\alpha) = 0$  when  $\nu$  is not an ancestor of  $\alpha$ . The parameters of the model are then simply the discrete probabilities  $P(\alpha)$ ,  $P(i|\alpha)$ ,  $P(\nu|\alpha)$ , and  $P(j|\nu)$ .

## 3.2 Training

### 3.2.1 Maximum-Likelihood Parameter Estimation

Suppose we have a probability density function,  $p(\mathcal{X}|\Theta)$ , governed by a set of parameters,  $\Theta$ . If a sample includes  $N$  data points, where we assume independence, then the probability density for the entire sample will be

$$p(\mathcal{X}|\theta) = \prod_{i=1}^N p(x_i|\theta) = \mathcal{L}(\Theta|\mathcal{X})$$

We can conceive of this both as the likelihood of the data given the parameters,  $P(\mathcal{X}|\Theta)$ , and as the likelihood of the parameters given the data,  $\mathcal{L}(\Theta|\mathcal{X})$ .

Our goal in Maximum-Likelihood Estimation (MLE) is to choose the parameters  $\Theta^*$  which maximize the likelihood of the parameters, ie.,

$$\Theta^* = \arg \max_{\Theta} \mathcal{L}(\Theta|\mathcal{X}). \quad (2)$$

Because the logarithm is a monotonically increasing function, we may maximize  $\log \mathcal{L}(\Theta|\mathcal{X})$  instead, which will often be easier to compute.<sup>12</sup> We call this function the log-likelihood. Depending on the problem, this function may be either trivial or extremely difficult to maximize.

In the training phase of HPLC, we wish to maximize the likelihood function,

$$P(\mathcal{D}) = \prod_{i=1}^L \left( P(\alpha) P(i|\alpha) \sum_{\nu} P(\nu|\alpha) P(j|\nu) \right),$$

or rather, it's log likelihood counterpart,

$$P(\mathcal{D}) = \sum_{i=1}^L \log \left( P(\alpha) P(i|\alpha) \sum_{\nu} P(\nu|\alpha) P(j|\nu) \right).$$

Since each training-data tuple includes class information, we may determine the maximum likelihood estimates of  $P(\alpha)$  and  $P(i|\alpha)$  from relative frequencies. That is,

$$P(\alpha) = \frac{|\{r|c(r) = \alpha\}|}{L} \quad (3)$$

and

$$P(i|\alpha) = \frac{P(i, \alpha)}{P(\alpha)} = \frac{|\{r|i(r) = i \text{ and } c(r) = \alpha\}|}{|\{r|c(r) = \alpha\}|} \quad (4)$$

For the remaining parameters,  $P(\nu|\alpha)$  and  $P(j|\nu)$ , which cannot be maximized analytically, we turn now to the Expectation-Maximization (EM) algorithm.

### 3.2.2 The Expectation-Maximization Algorithm

The EM algorithm is a general method for determining the maximum likelihood estimates for parameters when the data is incomplete, has missing values, or when portions of the data can be considered to be “hidden”.

We will denote the data  $\mathcal{X}$  as the observed data, and will hypothesize that additional (unobserved) data,  $\mathcal{Y}$ , is at work in the system. In any case, we assume a complete set of data exists,  $\mathcal{Z} = (\mathcal{X}, \mathcal{Y})$ , governed by the joint density function

$$P(z|\Theta) = P(x, y|\Theta) = P(y|x, \Theta)P(x|\Theta).$$

We also define a new likelihood function, the complete data likelihood

$$\mathcal{L}(\Theta|\mathcal{Z}) = \mathcal{L}(\Theta|\mathcal{X}, \mathcal{Y}) = P(\mathcal{X}, \mathcal{Y}|\Theta).$$

**E-Step** In the first step of the EM algorithm (the E-step), we calculate the expectation of the complete data log-likelihood with respect to the unobserved data  $\mathcal{Y}$ , given the observed data  $\mathcal{X}$  and the current estimates for the parameters  $\Theta$ . That is, we calculate

$$Q(\Theta, \Theta^{(i-1)}) = E[\log P(\mathcal{X}, \mathcal{Y}|\Theta)|\mathcal{X}, \Theta^{(i-1)}].$$

---

<sup>12</sup>Namely, since we would otherwise be multiplying probabilities (ie, numbers significantly less than 1), this will reduce our risk of underflow error.

**M-step** In the second step of the EM algorithm (the M-step) we set  $\Theta$  equal to the argmax of  $Q(\Theta, \Theta^{(i-1)})$ , which we may rewrite as

$$Q(\Theta, \Theta^{(i-1)}) = \sum_{r=1}^L \int (\log P(x_r, y_r | \Theta)) P(y_r | x_r; \Theta^{(i-1)}) dy_r. \quad (5)$$

Now,

$$P(y_r | x_r; \Theta^{(i-1)}) = \frac{P(x_r, y_r; \Theta^{(i-1)})}{\int P(x_r, y_r; \Theta^{(i-1)}) dy_r}, \quad (6)$$

so we may combine Equations 5 and 6 to obtain

$$Q(\Theta, \Theta^{(i-1)}) = \sum_{r=1}^L \int (\log P(x_r, y_r | \Theta)) \frac{P(x_r, y_r; \Theta^{(i-1)})}{\int P(x_r, y_r; \Theta^{(i-1)}) dy_r} dy_r \quad (7)$$

The deterministic annealing variant of the EM algorithm involves parameterizing the posterior in Equation 6 with a parameter  $\beta$ , so that

$$f(y_r | x_r; \Theta^{(i-1)}) = \frac{P(x_r, y_r; \Theta^{(i-1)})^\beta}{\int P(x_r, y_r; \Theta^{(i-1)})^\beta dy_r}. \quad (8)$$

Clearly, if  $\beta$  is set to 1,  $f(y_r | x_r; \Theta^{(i-1)}) = P(y_r | x_r; \Theta^{(i-1)})$ , and the expectation obtained by using 8 in place of 6 in Equation 7 is simply the expectation desired in the standard (non-annealed) EM algorithm. The deterministic annealing EM algorithm follows from this simple substitution and the intuition that a ‘‘heated’’ posterior (ie,  $\beta < 1$ ) will be smoother and more easily maximized<sup>13</sup>.

### Deterministic Annealing EM

1. Set  $\beta = \beta_0$ ,  $0 < \beta_0 \ll 1$
2. Initialize a guess  $\Theta^{(0)}$ , and set  $i = 1$ .
3. E-step: compute

$$Q_\beta(\Theta, \Theta^{(i-1)}) = \sum_{r=1}^L \int (\log P(x_r, y_r | \Theta)) \frac{P(x_r, y_r; \Theta^{(i-1)})^\beta}{\int P(x_r, y_r; \Theta^{(i-1)})^\beta dy_r} dy_r \quad (9)$$

4. M-step: set  $\Theta^{(i)}$  equal to the argmax of  $Q_\beta(\Theta, \Theta^{(i-1)})$ .
5. If  $\|\Theta^i - \Theta^{(i-1)}\| \geq \tau$ , goto Step 3. Else continue.
6. Increase  $\beta$ .
7. If  $\beta < \beta_{max}$ , increment  $i$  and return to Step 3; else stop.

It can be shown [5] that, for the hierarchical segment model, we may arrive at:

### E-step:

$$\langle T_{\alpha\nu}(r) \rangle^{(t)} = \frac{P(\alpha)P(i(r)|\alpha)P(\nu|\alpha)P(j(r)|\nu)}{\sum_\alpha P(\alpha)P(i(r)|\alpha)\sum_\nu P(\nu|\alpha)P(j(r)|\nu)} \quad (10)$$

<sup>13</sup>The idea comes from a statistical physics analogy, where heated solids will form crystal structures, thus minimizing energy, if cooled slowly.

**M-step:**

$$P^{(t+1)}(\nu|\alpha) = \frac{\sum_r \langle T_{\alpha\nu}(r) \rangle^{(t)}}{\sum_r \sum_\nu \langle T_{\alpha\nu}(r) \rangle^{(t)}} \quad (11)$$

$$P^{(t+1)}(j|\nu) = \frac{\sum_{r,j(r)=j} \sum_\alpha \langle T_{\alpha\nu}(r) \rangle^{(t)}}{\sum_r \sum_\alpha \langle T_{\alpha\nu}(r) \rangle^{(t)}} \quad (12)$$

Together with Equations 3 and 4, we may use Equations 10-12 to find MLE estimates for all of our model parameters during training.

### 3.3 Categorization

A segment  $d$  may now be categorized based on the posterior class probability  $P(\alpha|d) \propto P(d|\alpha)P(\alpha)$ , where the class probability  $P(\alpha)$  is known from the training phase and  $P(d|\alpha)$  may be estimated using EM to maximize the log-likelihood of the segment w.r.t.  $P(d|\alpha)$ :

$$L(d) = \sum_s \log\left(\sum_\alpha P(\alpha)P(d|\alpha) \sum_\nu P(\nu|\alpha)P(j(s)|\nu)\right),$$

where  $s$  is the number of co-occurrences in the segment. This may be accomplished with the following EM iteration,

**E-step:**

$$\langle C_\alpha(s) \rangle^{(i-1)} = \frac{P(\alpha)P(d(s)|\alpha) \sum_\nu P(\nu|\alpha)P(j(s)|\nu)}{\sum_\alpha P^{(i-1)}(\alpha)P(d(s)|\alpha) \sum_\nu P(\nu|\alpha)P(j(s)|\nu)} \quad (13)$$

**M-step:**

$$P^{(i)}(d|\alpha) = \frac{\sum_s \langle C_\alpha(s) \rangle^{(i-1)}}{\#\alpha + \sum_s \langle C_\alpha(s) \rangle^{(i-1)}} \quad (14)$$

Once  $P(d|\alpha)$  is estimated for every segment class  $\alpha$ , we expect the segment to belong to the class for which  $P(\alpha|d)$  is maximized.

## 4 hiercat: an HPLC Implementation

This section is intended to give a brief overview of what the HPLC implementation, **hiercat**, does during categorization. For a more detailed explanation of using **hiercat** and its accompanying scripts, consult the manual [9].

### 4.1 Data preparation and preprocessing

#### 4.1.1 Specifying the (Poly)Hierarchy

Class/topic hierarchies are specified for **hiercat** using a simple XML language, as in Figure 4. This hierarchy descriptor file is then parsed by an included perl script (**parseHierarchy.pl**), which produces

1. A file, **classes.dat**, which provides a mapping between human readable class names (eg, **talk.politics.guns**) and their internal numeric representation.
2. A file, **topics.dat**, which provides a similar mapping between human readable topic names (eg, **politics**) and their internal id number.
3. A binary representation of the hierarchy, to be parsed by **hiercat**, which specifies the ancestral topics for each class.

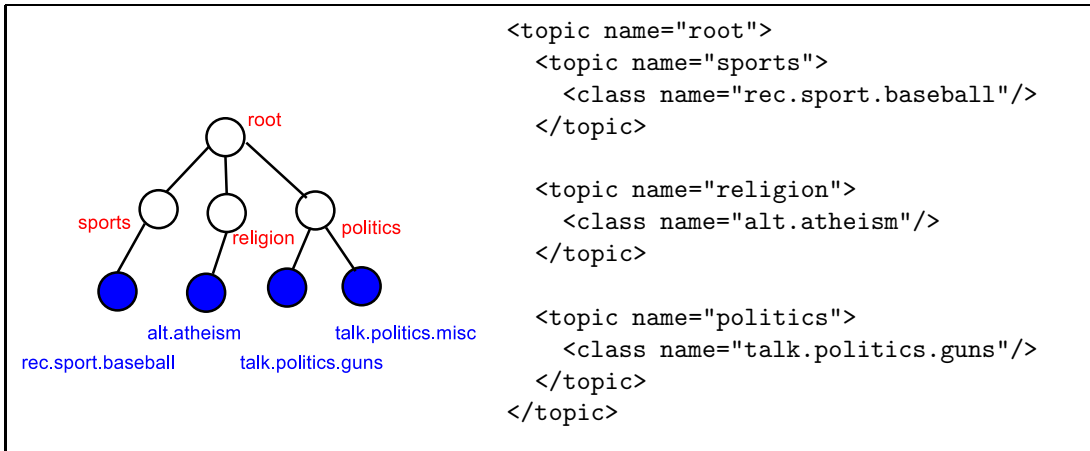


Figure 4: The XML specification of a pruned and thinned 20 Newsgroups hierarchy, containing only three classes.

```
0 13,14,5
1 3,27
...
```

Figure 5: An example training labels data file. For example, line one states that training document 0 is labeled with class numbers 13, 14, and 5. Note that segments can belong to multiples classes in the hierarchy.

#### 4.1.2 Preprocessing Training Data

In this phase of preprocessing (accomplished by the included script, `step1.pl`), a mapping between unique segment token stems (ie, words in the training documents, after stoplisting and stemming) and their internal numeric representation is produced and stored in `wordnums.dat`. The list of training documents and their internal id number is also stored in `docnums.dat`.

Each training data file is then converted (using the word number mapping) to a file `X.sig` (where `X` is the training document number), wherein each document token has been replaced by it's internal id number. This file is then easily parsed by `hiercat`.

#### 4.1.3 Preprocessing Testing Data

The word→wordnumber mapping created in the training data preprocessing is then applied to the documents to be categorized (the testing data). These mappings are likewise stored as `X.test.sig`, where `X` represents the testing document number. A list of testing documents and their internal id number is stored in `docnums.test.dat`.

#### 4.1.4 Specifying Training Labels

Training label data must be available in the `hiercat.training` file before `hiercat` may run. The file has a simple space and comma delimited format, as seen in Figure 5.

### 4.2 Running hiercat

Figure 6 gives a brief layout of the internal structure of `hiercat`.

During a run, logging messages are appended to an optionally specified logfile (default is `hiercat.log`). At program completion, an experiment output file is produced (with the default name `hiercat.output.X`, where `X` is the timestamp). This output file contains

- a complete specification of the experiment, sufficient to produce the exact results again
- the categorization results
- optionally specified session comments
- timing, system, miscellaneous information

The output information is stored in the file using a simple XML language, as in Figure 7. The intent is to

1. allow every run to be later analyzed for any possible reason; ie, a run initially done to measure runtime vs. number of classes will also save output data that could later be used to measure runtime vs. convergence tolerance.
2. ensure that a configuration which produces optimal results can be easily recovered by reviewing previous output files.
3. guarantee that code modifications do not unexpectedly alter categorization output across revisions.

Every completed training phase also stores the entire MLE set of parameters to disk (`params.dump`), to allow `hiercat` to restart in the testing phase. This makes it easy to test on various test sets, while retaining a training set (useful, since the training phase is considerably more costly than testing).

Categorization results are given as the non-normalized “probability” of a class given a document. For example, in the experiment specified in Figure 7, we see that test document 0 had  $P(\alpha_0|d_0) = 0.25$  and  $P(\alpha_1|d_0) = 3.64 \times 10^{-12}$ ; the document would thus be categorized with class 0 (here `english`).

## 4.3 Implementation Considerations

### 4.3.1 Languages

**Perl** All preprocessing in `hiercat` is done in Perl. Perl was chosen for its simplicity in string manipulation, and the wide availability of useful packages (eg, `XML::DOM` for XML parsing).

**ANSI C** The main body of `hiercat` is written entirely in ANSI C, without any third party non-standard packages. Initial versions had utilized some third party libraries (eg., to manipulate sparse matrices), but, to insure portability, it became simpler to implement their functionality in the `hiercat` codebase. For this reason, `hiercat` is extremely portable.

### 4.3.2 Parallelization

`hiercat` has been parallelized using MPI. During the training phase, classes and topics are distributed across processing nodes. Each node then completes its E-step (Equation 10) and advances the parameters using Equations 11 and 12, for all distributions conditioned on each of the nodes assigned classes/topics. Work is similarly distributed during the testing phase (although the testing phase constitutes only a very small portion of the computational cost in comparison to training).

Figure 8 demonstrates positive initial results for speedup across multiple processing nodes. This is to be expected, as the algorithm is easily (“embarrassingly”) parallelizable.

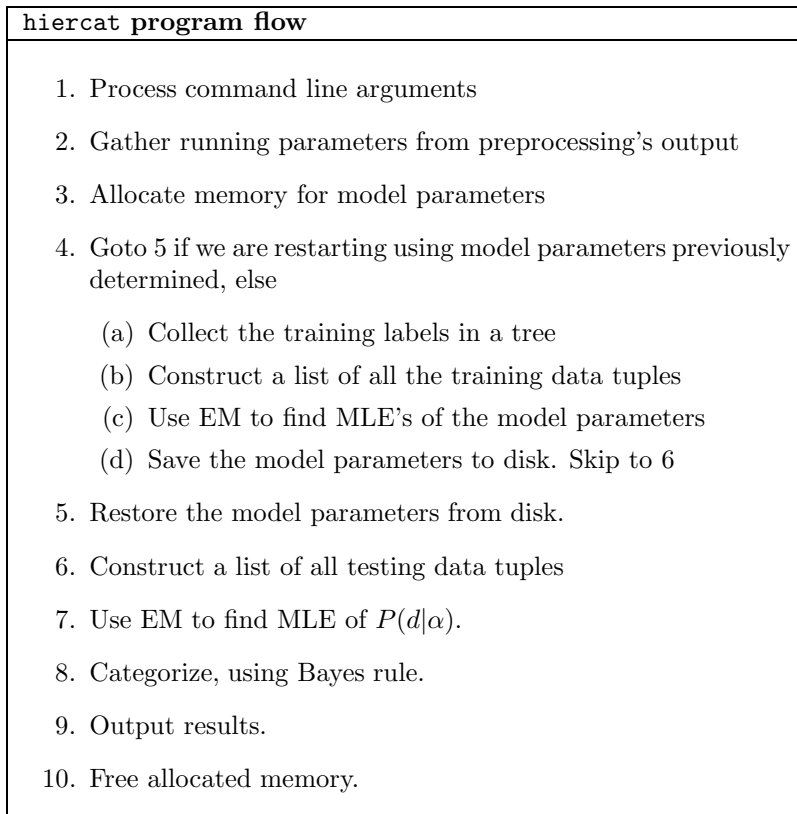


Figure 6: The layout of `hiercat`.

```

<hiercat-session-output>
  <version>1.15</version>

  <session-comment>
  English/Latin language classification on data3.
  </session-comment>

  <start-time>Sat May 1 00:46:55 2004</start-time>
  <end-time>Sat May 1 00:46:55 2004</end-time>
  <run-duration>0.186622</run-duration>
  <training_docs>2</training_docs>
  <testing_docs>1</testing_docs>
  <unique_words>2</unique_words>
  <word_classes>2</word_classes>
  <word_topics>3</word_topics>
  <data_tuples>2</data_tuples>
  <hierarchy>
    <classnums>
0 english
1 latin
    </classnums>
    <topicnums>
0 root
1 english
2 latin
    </topicnums>
  </hierarchy>
  <training>
    <tolerance>1e-05</tolerance>
    <anneal-schedule>0.1 0.3 0.5 0.8 1</anneal-schedule>
    <training-docnums>
0 1
1 2
    </training-docnums>
    <training-labels>
0 0
1 1
    </training-labels>
  </training>
  <testing>
    <testing-docnums>
0 1.test
    </testing-docnums>
    <testing-guesses>
0:0=0.25|1=3.64047e-12
    </testing-guesses>
  </testing>
</hiercat-session-output>

```

Figure 7: An example hiercat output file.

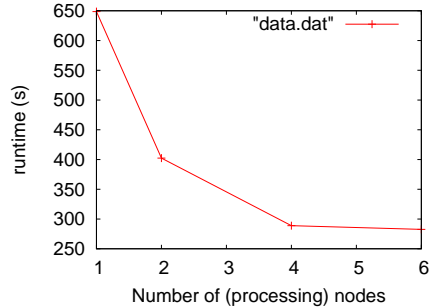


Figure 8: Runtime (s) vs. number of processing nodes for a small test problem on the 20 News data set. Speedup is good until the number of classes/topics distributed to each node is small (ie, it becomes advantageous to simply perform the EM-steps locally rather than to pay communication costs). It is clear that, for experiments with more classes/topics (the 20 News test used only 14 classes), speedup will continue to improve with the number of processing nodes.

### 4.3.3 Data Structures

**3D Sparse Matrices** A critical obstacle to efficiently iterating Equations 11 and 12 is the efficient calculation and retrieval of the  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$  values calculated in Equation 10. The simple (and inefficient) approach is to simply calculate  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$  for every  $(\alpha, \nu, r)$ -tuple summed over in the M-step; however, it is clear from the M-step equations that this would require wasteful recalculations of  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$  for already considered tuples.

A preferable alternative would be to calculate  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$  for each each  $(\alpha, \nu, r)$ -tuple before the M-step, storing them to allow for reuse. Unfortunately, this would require us to store  $N_\alpha \times N_\nu \times L$  real numbers—a prohibitive amount of storage for even a modest problem.<sup>14</sup> Fortunately,  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$  is in fact zero for most of the possible  $(\alpha, \nu, r)$ -tuples—which is clear from Equation 10 if we recall that  $P(\nu|\alpha) = 0$  if  $\nu$  is not an ancestor of  $\alpha$ . Memory problems can naturally be avoided then, by storing the values of  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$  as a 3-dimensional sparse matrix.

A second problem we must face is memory access.

$$P^{(t+1)}(\nu|\alpha) = \frac{\sum_r \langle T_{\alpha\nu}(r) \rangle^{(t)}}{\sum_r \sum_\nu \langle T_{\alpha\nu}(r) \rangle^{(t)}} \quad (15)$$

$$P^{(t+1)}(j|\nu) = \frac{\sum_{r, j(r)=j} \sum_\alpha \langle T_{\alpha\nu}(r) \rangle^{(t)}}{\sum_r \sum_\alpha \langle T_{\alpha\nu}(r) \rangle^{(t)}} \quad (16)$$

Consider Equation 15, the first portion of the M-step (Equation 11 repeated for convenience). We would like our sparse matrix data structure to allow efficient random access to  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$  (for the numerator) while also able to quickly sum over all  $\nu$  (for the denominator).

**hiercat** solves these problems in the following way: During the E-step, in preparation for Equation 15, the values of  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$  are calculated and stored in a  $L \times N_\nu \times N_\alpha$  sparse matrix using an adaptation of the compressed row storage (CRS) sparse matrix format.  $N_\alpha$  arrays of length  $L$  of pointers to AVL trees keyed by  $\nu$  store the values of  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$ . Random access is efficient due to the balanced binary storage of columns, while summing across all  $\nu$  is efficiently done by summing all elements in the appropriate AVL

<sup>14</sup>For example, a pruned 20newsgroups hierarchy example with 3 classes, 7 topics, and 3 training docs per class had  $L = 2287$ . This amounts to 48027 reals to be stored. Assuming 8 byte doubles, we must store 3842116 bytes. Now consider a more realistic example: a pruned 20newsgroups hierarchy with 14 classes, 20 topics, 10 training documents per class, and  $L = 185538$ . This case would require  $4.16 \times 10^8$  bytes of memory. Clearly, the memory requirement of such a solution do not scale favorably.

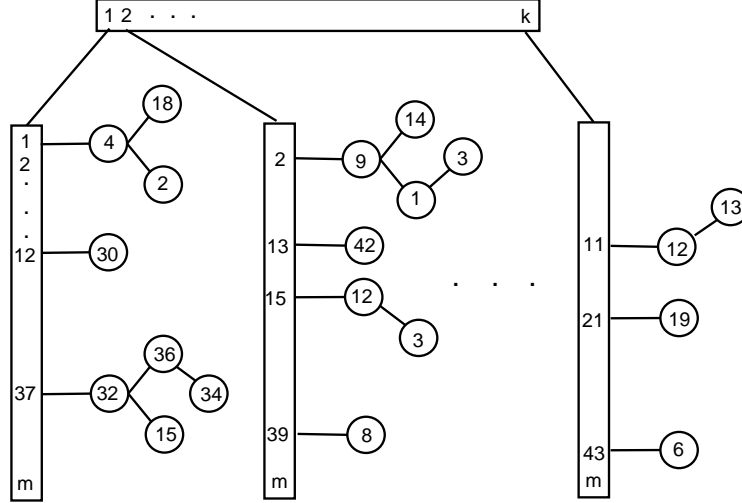


Figure 9: `hiercat`'s CRS sparse matrix implementation, with rows stored as balanced binary (AVL) trees. Each element of the length  $m$  array points to an AVL tree containing that row's elements, keyed in the tree by the element's column number. The length  $k$  array points to each level in the 3-dimensional sparse matrix. This figure depicts a  $m \times n \times k$  sparse matrix, although  $n$  (the row length) is not used and may effectively grow. For example, the matrix has non-zero elements at  $(37, 34, 1)$  and  $(43, 6, k)$ .

tree. This tree summation may be done iteratively using a depth-first (stack based) traversal, or recursively. Figure 9 diagrams the general design of `hiercat`'s sparse matrix data structure.

To provide an efficient memory access path through Equation 16, the E-step also stores the values of  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$  in a second sparse matrix, now of size  $L \times N_{\alpha} \times N_{\nu}$ . This likewise allows for efficient summing over all  $\alpha$  in both the denominator and numerator, by again summing the appropriate tree.

In practice, for smaller test problems (eg, 20 News), we instead use an accumulator to store row sums (as the values are calculated during the E-step). This has the advantage of allowing rowsums to be recalled in  $O(1)$  time, while having the disadvantage of adding  $O(L(N_{\nu} + N_{\alpha}))$  space complexity over a slower, tree summation, method.

#### 4.3.4 Complexity Analysis

**Tree sums** Figure 10 contains a recursive tree summing routine in C. Given a tree containing  $N$  nodes, we may denote the runtime of this routine as  $T(N)$ . From Figure 10 then, it is clear that the complexity is specified by the recurrence relation

$$T(N) = 2T(N/2) + 1. \quad (17)$$

where we note especially that a tree containing only one node may be summed in constant time. That is,  $T(1) = O(1)$ .

Since Equation 17 holds for any  $N$ , we may substitute in  $N/2$ , giving

$$2T(N/2) = 2(2T(N/4) + 1) = 4T(N/4) + 2$$

so that

$$T(N) = 4T(N/4) + 3.$$

Likewise, we may substitute  $N/4$  into Equation 17, giving

$$4(T(N/4)) = 4(2T(N/8) + 1) = 8T(N/8) + 4$$

```

HIER_REAL sumTree(AvlTree T)
/* return the sum of an avl tree (using recursion) */
{
    HIER_REAL sum = 0;

    if(T != NULL){
        sum += sumTree(T->Left);           /* 1 */
        sum += *((HIER_REAL *)T->Payload); /* 2 */
        sum += sumTree(T->Right);         /* 3 */
    }

    return sum;
}

```

Figure 10: C Function to return the sum of a tree using recursion.

so that

$$T(N) = 8T(N/8) + 7.$$

Or in general, after  $k$  many such substitutions, we will have

$$2^k T(N/2^k) + k^2 - 1.$$

Now,  $k$  is simply the height of the tree being summed over; for a balanced tree  $k = \log_2 N$ . Accordingly,

$$T(N) = 2^{\log_2 N} T(N/2^{\log_2 N}) + (\log_2)^2 - 1 = NT(1) + \log_2^2 N - 1.$$

It follows then that tree summation is  $O(N)$ .

Figure 12 shows  $T(N)/f(N)$  vs.  $f(N)$ , where  $f$  denotes the asymptotic worst time complexity we expect from our analysis, and  $T(N)$  now denotes the empirically observed running time. If  $f(N)$  were an overestimate, we would expect the ratio  $T(N)/f(N)$  to converge to zero; if  $f(N)$  were an underestimate, the ratio would diverge; finally, if  $f(N)$  is a tight fit on the running time, then we expect the ratio to converge to some positive constant. Figure 12 indicates the analysis is a tight fit.

**Recalling a value from the 3D Sparse Matrix** It is well known [10] that the time to find a node in a balanced binary tree is  $O(\log_2 N)$ , where  $N$  is the number of nodes in the tree. Obtaining the level and row index in our sparse matrix implementation may be done in constant time, so the worst-case time complexity of recalling a value from the matrix reduces to the time required to find an element in the balanced tree representation of the row.

In **hiercat**, values of  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$  are inserted into the sparse matrix whenever the topic  $\nu$  is an ancestor of the class  $\alpha$ . It follows that there may not be more elements in a row (ancestors of a class) than the height of the topic hierarchy. Therefore, the time complexity for recalling an element from the 3D sparse matrix is (worst-case)  $O(\log_2 h)$ , where  $h$  is the height of the topic hierarchy.

**Time-complexity bottleneck** Timing observations demonstrate that the bulk of **hiercat**'s runtime is consumed in the update of  $P(j|\nu)$ , according to Equation 16. Figure 14 contains the applicable code.

The E-step data,  $\langle T_{\alpha\nu}(r) \rangle^{(t)}$ , is stored in a  $L \times N_\alpha \times N_\nu$  sparse matrix for the updating of  $P(j|\nu)$ . Accordingly, the summation over  $\alpha$  in Equation 16 (and Figure 14, line 1) is simply a tree summation over the AVL tree containing the  $r^{\text{th}}$  row in level  $\nu$ . It is clear that the exterior loops in Figure 14 contribute a factor of  $O(N_j N_\nu L)$  to the time complexity. Now, the number of nodes in the tree representation of row  $r$  at

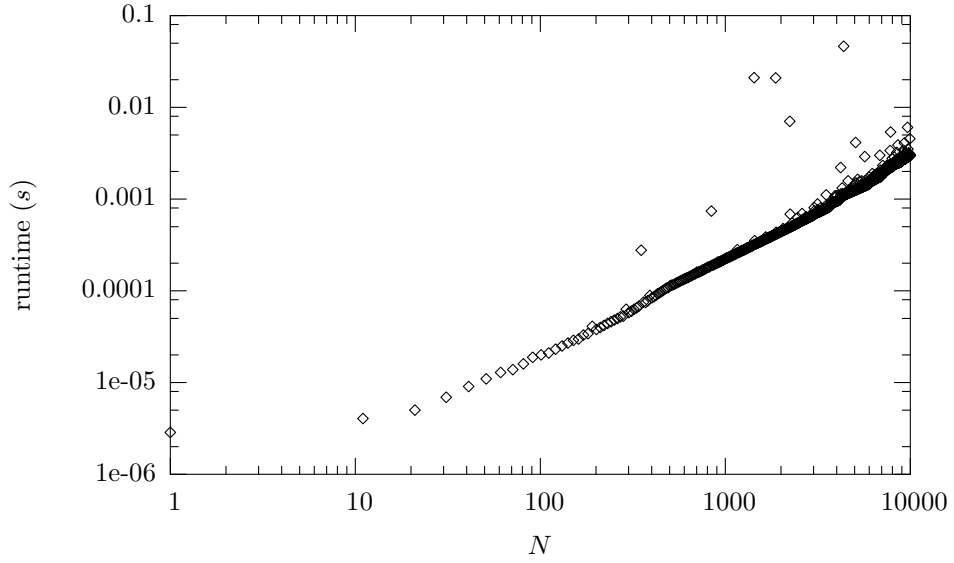


Figure 11: Runtime ( $s$ ) vs. number of nodes ( $N$ ) in a tree summation on `hiercat`'s (balanced) binary tree.

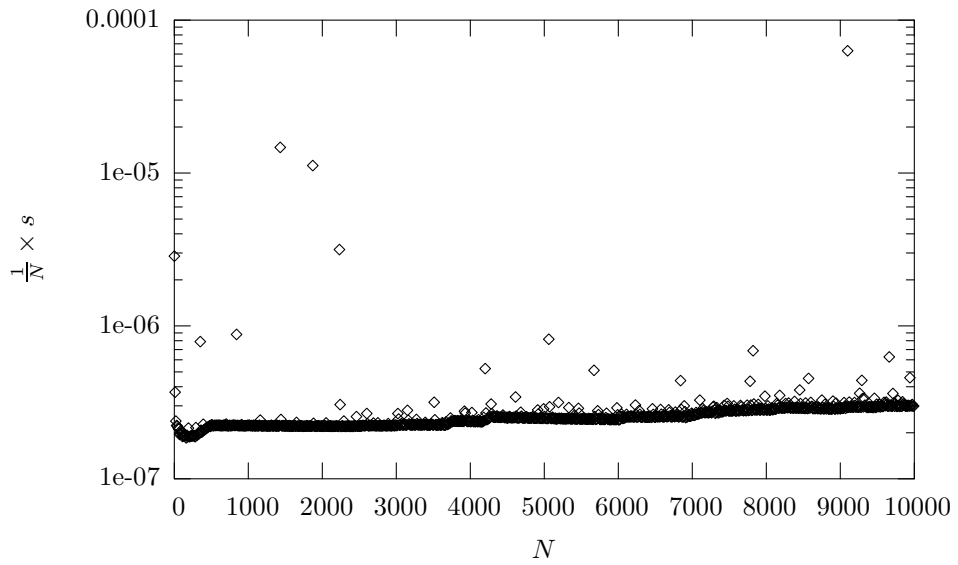


Figure 12: The ratio of runtime ( $s$ ) to nodes  $N$  vs. number of nodes ( $N$ ) in a tree summation on a balanced binary tree. Since the recursive tree summation algorithm is  $O(N)$ , we expect the observed convergence to a constant ratio of runtime to input size  $N$ .

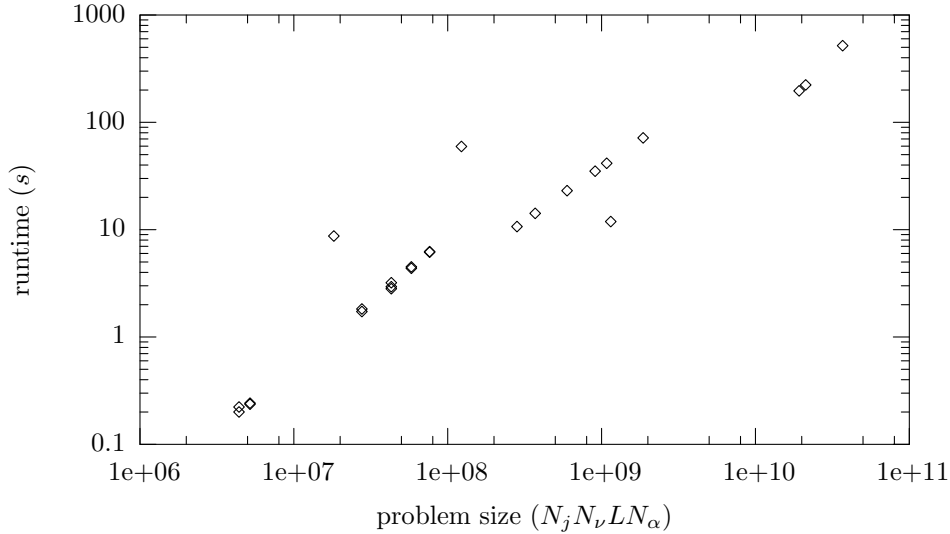


Figure 13: The time to update  $P(j|\nu)$ ,  $T$ , in seconds plotted against algorithmic time complexity  $f$ , for several trials of various input size. Our complexity analysis places  $f = O(N_j N_\nu L \log N_\alpha)$ . The convergence of  $T/f$  to a positive constant indicates  $f$  is a tight fit for the running time.

level  $\nu$  cannot exceed  $N_\alpha$ .<sup>15</sup> Taken together, the time complexity of updating  $P(j|\nu)$  is then  $O(N_j N_\nu L N_\alpha)$ . Figure 13 plots the time to update  $P(j|\nu)$  vs. time complexity. If our analysis represents a good fit for the time complexity, we expect the slope to be approximately linear for large problem size. This is observed.

## 5 Validation/Evaluation

### 5.1 Definitions

**Precision, Recall, and the  $F$ -measure** Categorization attempts will fall into one of four groupings, given a target set (the set of all documents truly in a class) and a set of documents actually returned (ie, the set of documents classified—truly or falsely—as the target class). Accordingly, categorizers may fail in one of two different ways: 1. they may falsely identify a document as positive—believing it to be in the target class (fp), or 2. they may falsely identify a document as being negative—not in the target class (fn). Likewise, they may succeed by both determining a document is not in a class (tn) and by determining that a document actually is in the target set (tp).[3] Figure 15 shows the possible classification results more clearly.

Ultimately, a categorizer is only as good as the end user is happy with it's output. That said, several measurements of quality are typically made for categorization attempts. The naive choice, the ratio of correct to incorrect responses (accuracy), turns out to be somewhat uninformative—as it makes no distinction between types of failures. Precision,

$$P = \frac{tp}{(tp + fp)} = \frac{|A \cap B|}{|A|},$$

(where  $A$  and  $B$  are as in Figure 15), the ratio of classified documents in the target class to all classified documents, also does not tell the entire story—as a categorizer may be made as precise as desired by only

<sup>15</sup>The actual number of elements in row  $r$  at level  $\nu$  will be the number of classes  $\alpha$  which are descendents of the word topic  $\nu$ . Clearly this is an over-estimate, as only one topic node (the root) will in fact have all classes as a descendant.

```

for(J=0; J< unique_words; J++){
  for(nu = mybnu; nu< mytnu; nu++){
    numerator = 0;
    denominator = 0;
    for(r=0; r < L; r++){
      tmp1 = getSpmatRowSum(T_v_ra[nu], r); /* 1 */
      if(j_r[r] == J){
        numerator += tmp1; /* 2 */
      }
      denominator += tmp1; /* 3 */
    }
    P_jGv[J][nu] = numerator/denominator;
  }
}

```

Figure 14: A C implementation of updating  $P(j|\nu)$  according to Equation 16.

categorizing a small number of documents. Recall,

$$R = \frac{tp}{(tp + fn)} = \frac{|A \cap B|}{|B|},$$

the percentage of target documents properly classified, also falls short—simply classifying all documents in the collection as the target class would guarantee perfect recall!

Ideally, then, a system will have both high precision and high recall, although, in practice, one is typically gained at the expense of the other (they are typically roughly inversely proportional). In practice, then, some weighted average of the two measures is used to ascertain the effectiveness of the categorizer. Keep in mind that a given categorizer will generally only perform optimally if appropriate runtime parameters are set (eg, the number of neighbors,  $k$ , or the threshold score to consider in  $k$ NN ), and so a single metric of overall suitability is desirable. The  $F$  measure is often used, and is defined as

$$F = \frac{1}{\alpha \frac{1}{P} + (1 - \alpha) \frac{1}{R}},$$

where  $\alpha$  represents the relative weight value placed on precision and recall; an equal valuing ( $\alpha = .5$ ) reduces the measure to the harmonic mean of  $P$  and  $R$ ,

$$F = \frac{2PR}{(R + P)}.$$

To calculate these measures, the categorizer builds a *contingency table* for each category decision<sup>16</sup> made during the test phase. A contingency table is simply a tabulation of the cardinality of each partition shown in Figure 15, that is,

	yes is <b>correct</b>	no is <b>correct</b>
yes was <b>assigned</b>	#tp	#fp
no was <b>assigned</b>	#fn	#tn

One may then calculate averages for  $P$ ,  $R$ , and  $F$  across categories (*macro-averaging*) or by building a master contingency table of all the decisions made and averaging across every decision (*micro-averaging*). In micro-averaging, the largest categories will dominate, while in macro-averaging, a better picture of categorizer

<sup>16</sup>In other words, the categorizer attempts to answer the question, does this test document belong in the category I am currently considering?

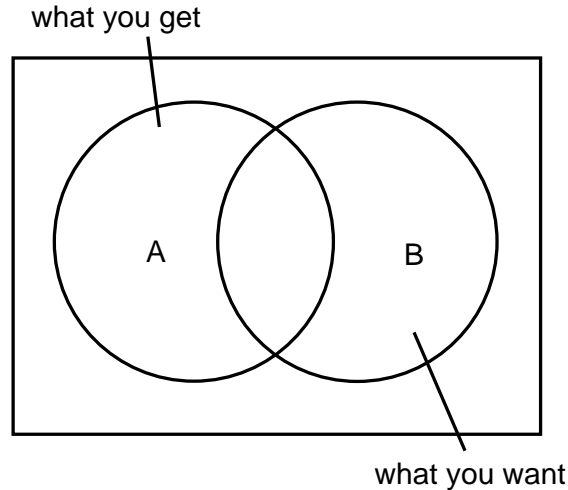


Figure 15: Possible results of classification attempts.

effectiveness is given across all the categories (although one may naturally be principally concerned with performance on the largest category sizes). Neither choice of averaging is alone sufficient;<sup>17</sup> both convey information about the categorizer (and consequently the nature of the dataset).

Remember that categories are assigned as labels only if their category score is above some score threshold. In the  $k$ NN categorizer, if the threshold is taken to be very high then only neighbors which are very near or a preponderance of near like-labeled neighbors will cause the category score to exceed the score threshold; in this case, we naturally expect the precision to be very high and the recall to be very low (ie, the percentage of documents categorized properly of those categorized is high, although only a small percentage of documents in the entire collection were categorized at all). Alternatively, we may take the threshold to be very low, in which case all documents will (for large  $k$ ) have all labels, and so the recall is very high while the precision is very low.

Alternatively, we may measure categorization effectiveness by only considering the highest ranked class for each test document (at least for problems where only one label may properly be assigned to the document). In this one-label case, we also consider classification accuracy (ie, the ratio of correct to incorrect classifications).

**Edge Error** The measures defined so far have only been concerned with comparing performance between multiple categorizers, in terms of *how many documents were properly categorized*. However, since we have knowledge of the topic hierarchy, we can reasonably ask a secondary question: in the case that a categorizer improperly categorizes, *how wrong is it?*

A simple measure of this error is the *edge error*—the number of edges connecting an incorrect guess to the correct answer in the topic hierarchy.

**Confusion Matrices** A confusion matrix is simply a  $n \times n$  sparse matrix, where  $n$  is total number of classes in a categorization attempt. The  $(i, j)$  element of the matrix records the total number of categorization attempts which had correct class  $i$  but were actually assigned class  $j$ . An ideal categorizer will produce a completely diagonal confusion matrix, while in practice, many elements will be off-diagonal.

<sup>17</sup>Suppose for a labelled collection that the labels are sorted in decreasing order of frequency  $f$ , and their position in the sort is denoted by  $r$ . Zipf's law states that  $f \propto \frac{1}{r}$ . Roughly restated, in any collection there will be a few categories with very many documents, and many categories with only very few or only one document. These *singletons* represents a significant problem in categorization as there will typically be little or no training data available for them.

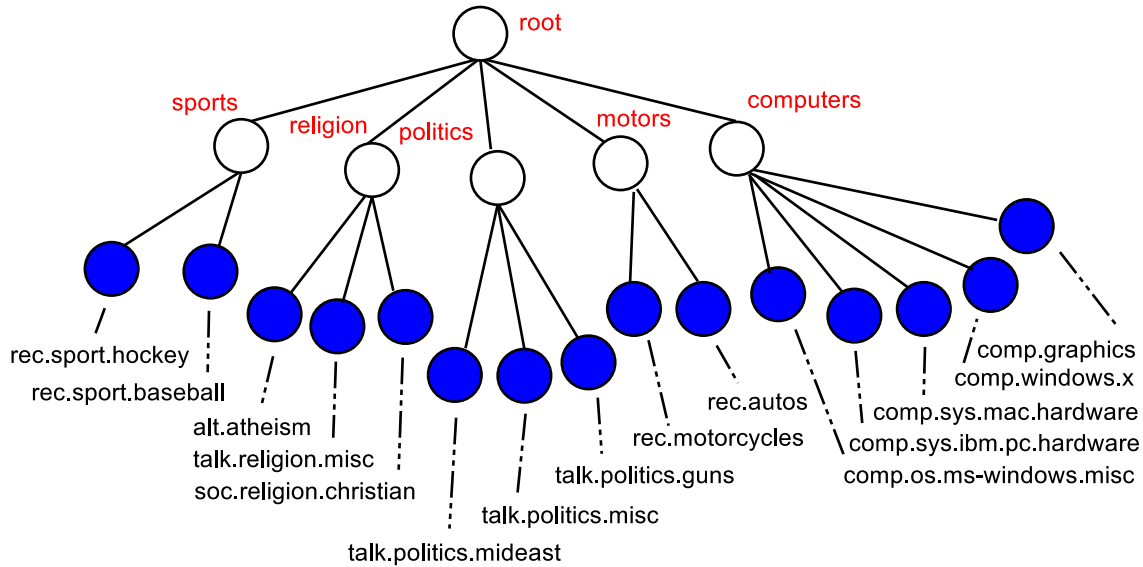


Figure 16: A pruned 20 Newsgroups hierarchy.

Confusion matrices are especially useful for visualizing effectiveness of categorizers across the set of labels. Bands and other prominent features in the visualization can quickly convey the particular strengths or weaknesses of a categorization system (for example, one categorizer may discriminate on geographic classes better than another, which excels at chronological labels).

## 5.2 20 Newsgroups

“The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups.” [1] These newsgroups (our classes) may be organized into a simple hierarchy, as depicted in Figure 16, following the lead of the original experiments conducted by Gaussier, et. al [4]. This same choice of hierarchy was chosen especially for the purpose of validating the categorizer by way of comparison with their initial implementation’s results.

**Precision & Recall** Figure 17 contains micro-averaged precision-recall curves for several categorization experiments (with varying training sizes) using both `hiercat` and 10NN. The dark (hyperbolic) lines represent level curves in the  $F$ -measure. The peak  $F$ -measure for each curve is plotted vs. training size in Figure 18. It is observed that, `hiercat` outperforms  $k$ NN across the bulk of training data sizes. Furthermore, the results are consonant with our reference implementation [4].

Recall that precision and recall are sensible measures for categorizers which return a set of scored classes; a threshold may then be set above which class scores are considered to be label assignments. As this threshold is swept from a very high to very low value, we walk rightward along the precision-recall curve (intuitively, at a high threshold, we expect a high precision, and a low threshold, a high recall).

**Accuracy** For categorization attempts that only consider one (a “best”) label, however, we are naturally concerned with other, simpler, performance measures. An obvious choice is micro-averaged classification accuracy. Figure 19 is a plot of classification accuracy vs. training data size. Accuracy is here viewed as a point estimate of the proportion of all documents which would be properly categorized. Accordingly, error

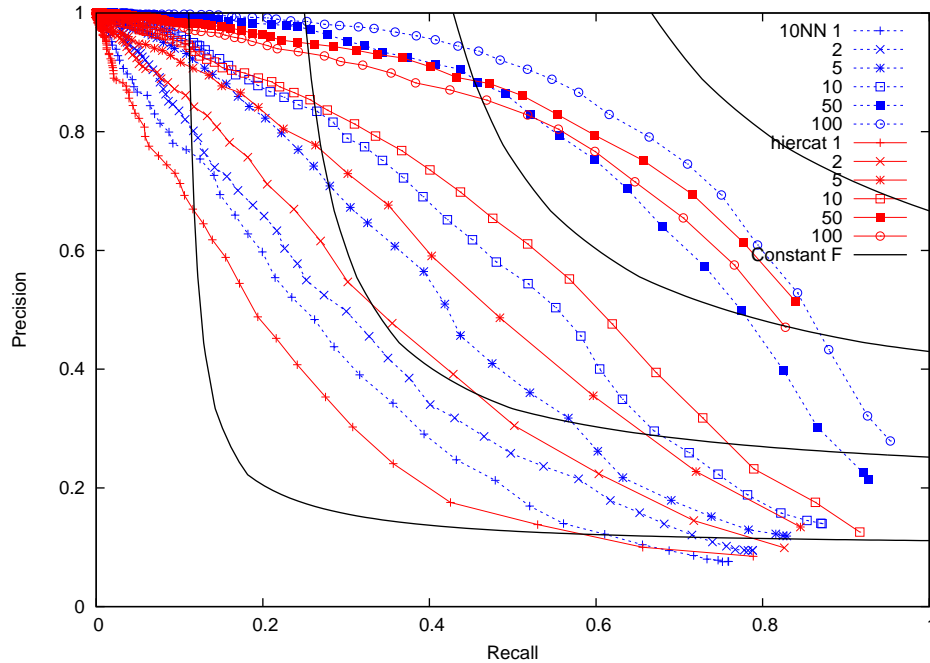


Figure 17: Micro averaged precision and recall for both  $k$ NN and HPLC. Black lines represent level surfaces in the  $F$ -measure at .2, .4, .6, and .8. Note that `hiercat` is in red, 10NN is in blue, while comparable tests use identical symbols.

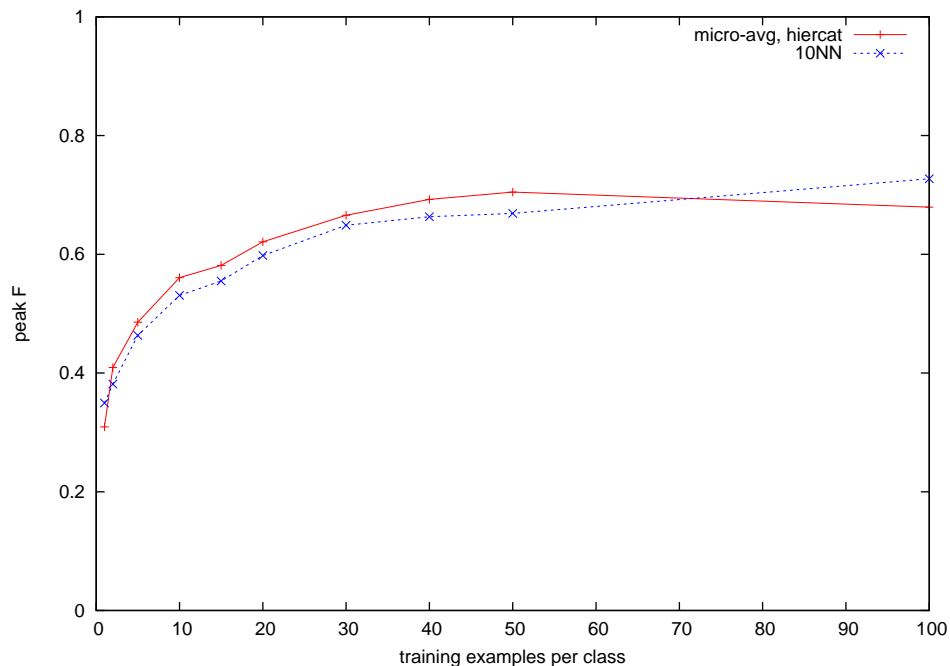


Figure 18: Peak  $F$ -measure vs. number of training examples per class, for both 10NN and `hiercat`.

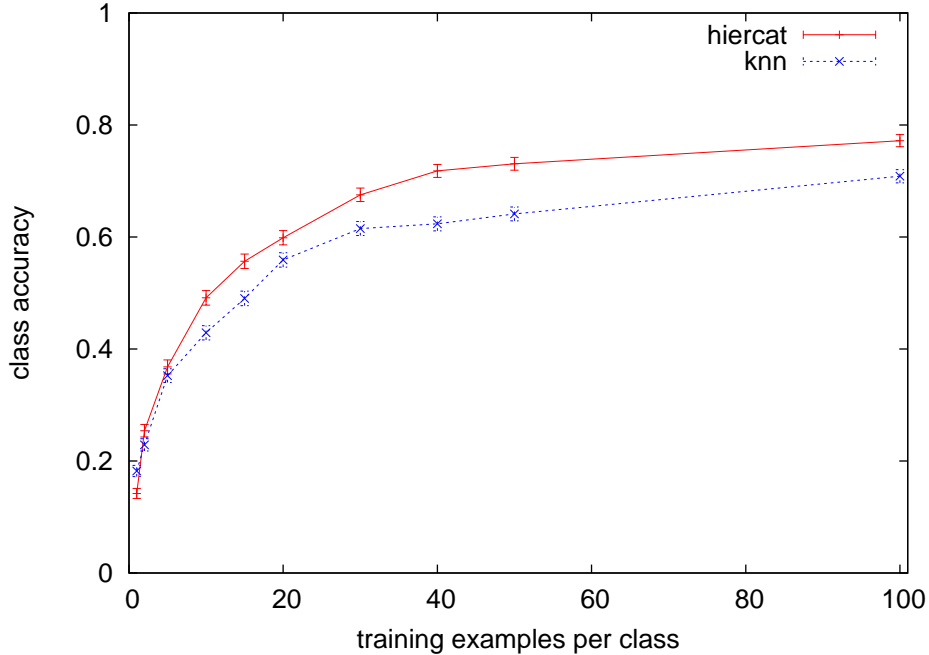


Figure 19: Micro-averaged class accuracy vs. number of training examples per class for both  $k$ NN and **hiercat**'s categorization of 1500 20 Newsgroup messages taken from the topic hierarchy of Figure16.

bars in Figure 19 denote standard error for point estimation ( $\hat{p}$ ) of a population proportion  $p$ . Specifically,

$$\hat{p} = \frac{X}{n}, \quad \text{and estimated S.E.}(\hat{p}) = \sqrt{\frac{\hat{p}\hat{q}}{n}}$$

where  $X$  is a random variable denoting the number of classification attempts having the characteristic that they are properly categorized, and  $n$  is the total number of attempts. It is clear that **hiercat** outperforms  $k$ NN across the bulk of training data sizes.

**Edge Error** For these verification experiments, edge error was also calculated and plotted in Figure 20. We now take the perspective that there is some population of incorrectly assigned classification attempts for which we hope to determine a point estimate of the mean edge error. Our estimator is now simply  $\bar{X}$ , where  $X$  is a random variable for a particular edge error, and our estimated standard error,  $\text{S.E.}(\bar{X}) = \frac{S}{\sqrt{n}}$ . The error bars in Figure 20 denote standard error.

From this data, it is clear that **hiercat** significantly outperforms  $k$ NN, even when it miscategorizes (by producing a guess which is on average closer to the correct answer in the hierarchy).

**Confusion Matrices** Figure 21 contains confusion matrices for both  $k$ NN and **hiercat**, on an identical test problem (50 examples per class, categorizing 1500 uniformly distributed test documents). Several prominent and interesting features are immediately visible:

1. The **hiercat** matrix has many more on-diagonal elements than its  $k$ NN counterpart.
2. Many documents are distributed along guessed class 7, `talk.politics.misc`. This is not surprising, since such newsgroups talk about effectively everything. Hiercat, however, manages to leverage the hierarchy and improve categorization performance (thus, a less-prominent band along guess 7).

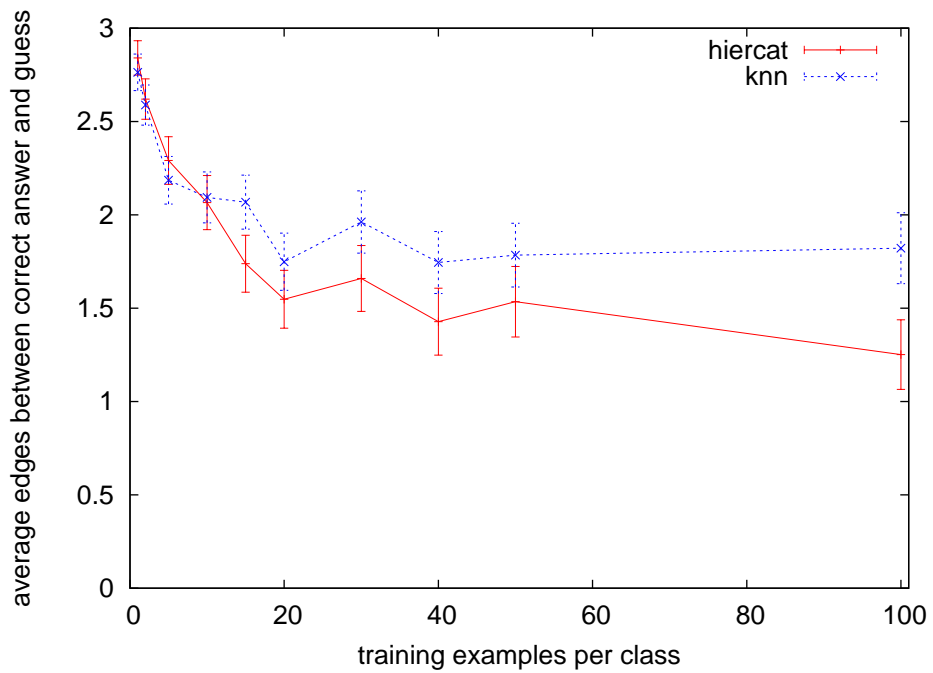


Figure 20: Edge error vs. number of training examples per class for both  $k$ NN and **hiercat**'s categorization of 1500 20 Newsgroup messages taken from the topic hierarchy of Figure16. Note that, for most training sizes, **hiercat** produces a label which is closer to the correct answer, when its guess is in fact incorrect.

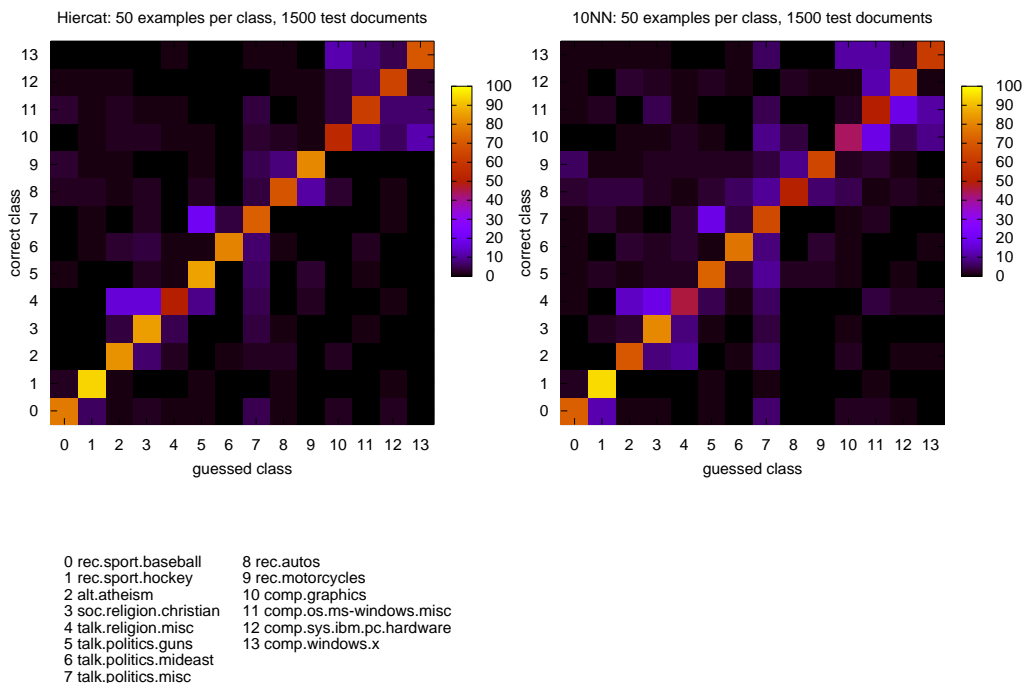


Figure 21: Confusion matrix representation of 1500 categorization attempts using both `hiercat` and 10NN on 15 Newsgroups; this particular test was conducted with 50 training examples per class.

3. Confusion is large around classes 10-13 (computer newsgroups). Again, `hiercat` shows improvement over  $k$ NN.
4. Confusion is notable around classes 2-4 (religious newsgroups). Once again, `hiercat` improves upon the flat categorizer ( $k$ NN).

## 6 Conclusions

Both  $k$ NN and HPLC categorizers were implemented and verified. Empirical evidence strongly suggests that the hierarchy-respecting HPLC model outperforms the (flat)  $k$ NN implementation, both in terms of categorization accuracy and hierarchical measures (eg, edge error).

## 7 Future work

A critical advantage of flat categorizers is their time complexity superiority over their hierarchical counterparts. To overcome these margins, more efficient training algorithms must be produced. This is especially true for EM based methods, such as HPLC. Various methods of improving EM performance have been suggested in the literature (notably, [6]); it is possible that training times might be significantly improved by one or several of these altered EM algorithms.

Our immediate goal is to apply `hiercat` to the categorization of MALACH data.

## 8 Acknowledgements

I would like to thank Dr. Doug Oard for guiding me in this project, as well as Craig Murry for his help at various points along the way. Also thanks to Drs. Bill Dorland and Harland Glaz, the course leaders under whose auspices this work was completed, for their constructive criticism and continued encouragement.

## References

- [1] 20 newsgroups data set homepage. [http://people.csail.mit.edu/u/j/jrennie/public\\_html/20Newsgroups/](http://people.csail.mit.edu/u/j/jrennie/public_html/20Newsgroups/).
- [2] Malach home page, JHU. <http://www.clsp.jhu.edu/research/malach/>.
- [3] H. Schütze C.D. Manning. *Foundations of Statistical Natural Language Processing*, pages 540–544, 575–578, 604–606. MIT Press, 1999.
- [4] K. Popat E. Gaussier, C. Goutte and F. Chen. Advances in information retrieval, 24th bcs-irsg european colloquium on ir research glasgow, uk, march 25-27, 2002 proceedings, *a hierarchical model for clustering and categorizing documents*. In Fabio Crestani, Mark Girolami, and C. J. van Rijsbergen, editors, *ECIR*, volume 2291 of *Lecture Notes in Computer Science*. Springer, 2002.
- [5] E. Gaussier. Derivation of EM formulas for complete hierarchical model. *Unpublished, received via private correspondence*.
- [6] M. Jamshidian and R. I. Jennrich. Conjugate gradient acceleration of the EM algorithm. *Journal of the American Statistical Association*, vol. 88(no. 421)::221–228, Mar. 1993.
- [7] Dibyendu Majumdar. YASE project homepage. [http://www.mazumdar.demon.co.uk/yase\\_index.html](http://www.mazumdar.demon.co.uk/yase_index.html).
- [8] Doug Oard. Malach home page, UMD. <http://raven.umd.edu/dlrg/malach/>.
- [9] J. Scott Olsson. Project home page. <http://www.math.umd.edu/~olsson/amsc663/>.
- [10] M.A. Weiss. *Data Structures and Algorithm Analysis in C*, pages 110–122. Addison-Wesley, 1997.
- [11] Yiming Yang, Jian Zhang, and Bryan Kisiel. A scalability analysis of classifiers in text categorization. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 96–103. ACM Press, 2003.