Automatic Construction of N-ary Tree Based Taxonomies

Kunal Punera, Suju Rajan, Joydeep Ghosh
Dept. of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78751
kunal,suju,ghosh @ ece.utexas.edu

Abstract

Hierarchies are an intuitive and effective organization paradigm for data. Of late there has been considerable research on automatically learning hierarchical organizations of data. In this paper, we explore the problem of learning n-ary tree based hierarchies of categories with no user-defined parameters. We propose a framework that characterizes a "good" taxonomy and also provide an algorithm to find it. This algorithm works completely automatically (with no user input) and is significantly less greedy than existing algorithms in literature. We evaluate our approach on multiple real life datasets from diverse domains, such as text mining, hyper-spectral analysis, written character recognition etc. Our experimental results show that not only are n-ary trees based taxonomies more "natural", but also the output space decompositions induced by these taxonomies for many datasets yield better classification accuracies as opposed to classification on binary tree based taxonomies.

1 Introduction

Hierarchical taxonomies have become an important tool in the organization of knowledge in many domains. The US Patent Office class codes, the Library of Congress catalog, and even the ACM Computing Classification System are hierarchical in structure. In general, taxonomies structured as hierarchies make it easier to navigate and access the data as well as to maintain and enrich it. This is especially true in the context of the World Wide Web where the amount of available information is overwhelming. Many internet directories such as Yahoo[1] and DMOZ[2] are organized as hierarchies.

AUTOMATIC TAXONOMY CONSTRUCTION.

The taxonomy construction process involves the specification of a hierarchical arrangement of classes as well as placement of data into nodes of this hierarchy. In the past, this has been typically accomplished by hand. For example, both Yahoo and DMOZ taxonomies were created manually by employees and volunteers respectively. This manual labeling process is, however, time-consuming, expensive, and, in the case of an changing and expanding corpus like the World Wide Web, inherently incomplete.

In this paper we tackle the problem of arranging a given set of categories into a hierarchy, specifically as leaves of a **rooted n-ary tree**. Moreover, given the high cost and unscalable nature of manual intervention, we seek to do this completely automatically (parameter-free). There have been several proposed approaches solve parts of this problem, but they all either require significant user input on the structure of the tree [4], or place artificial restrictions on it [6]. Details of these existing approaches and how our work differs from them can be found in [3].

SUMMARY OF CONTRIBUTIONS.

- We present an approach that constructs taxonomies of categories in a completely automated fashion. We introduce a novel constraint on the relationships between categories, and this helps our algorithm learn "good" taxonomies with no user-defined parameters.
- Our approach doesn't place any restrictions on the branching factor of the tree being learned, effectively constructing n-ary trees. This avoids arbitrary groupings of categories at the top levels of the tree.
- In our approach, some greedy decisions made early in the taxonomy construction process are re-evaluated in more specific contexts. This makes our approach significantly less greedy than some existing methods.
- Through experiments on datasets from a variety of domains, we show that taxonomies modeled as n-ary trees are more "natural" and result in better hierarchical classification accuracies than those modeled as binary trees.

Our approach is detailed in Section 2. Then we present an experimental evaluation of our approach on a set of datasets from diverse domains in Section 3.

[1] www.yahoo.com
[2] www.dmoz.org

2 Approach

We begin with a detailed description of the problem of automatic taxonomy construction and then propose a solution to it, which we refer to as Automatic Taxonomy Generator (ATG). Henceforth in this paper, we use the terms "class" and "category" interchangeably. Further, since taxonomies are hierarchical, and trees to be specific, we use the terms "taxonomy", "hierarchy", and "tree" interchangeably too.

2.1 Automatic Taxonomy Construction

We tackle the problem of learning the structure of a rooted n-ary tree with the classes placed at the leaves. The desiderata of a solution are as follows:

A standard requirement of any hierarchical taxonomy employed for classification is that "similar" classes be placed close to each other, say, in terms of tree distance. For example, in the creation of a Shopping taxonomy, the least common ancestor of classes "Car" and "SUV" should be farther from the root than that of classes "Car" and "PowerTool".

In order to increase the interpretability of taxonomies, we require that the content of each internal node be as homogeneous as possible. Taxonomies modeled as binary trees often group classes arbitrarily at the top levels. For example, consider a Shopping taxonomy where the root is associated with the set of classes {"Electronics", "Computers", "Home and Garden", "Clothing and Accessories"}, and where a binary tree would necessarily pair arbitrary classes together. We desire that the taxonomy construction process partition the set of classes at each internal node into as many parts as needed to maintain the homogeneity of the children nodes.

Many existing approaches require the user to specify, for instance, the number of internal nodes in the tree or at each level. These parameters are very difficult to set manually without intimate knowledge of the structure of data. We want our approach to avoid any such parameters.

FORMAL DEFINITION.

We need a few definitions to make the ideas expressed above and the subsequent solution to the problem precise. Let X be the set of data-points, such that each data-point x_i has an associated class label l_i from a set of k classes C. Thus, each class c_j has a set of data-points X_{c_j} associated with it using which its prior π_{c_j} and class-conditional probability density functions $p_{c_j} = p_X(x|c_j)$ can be estimated.

We want an arrangement of the classes C into a taxonomy. Let the taxonomy be represented by a rooted n-ary tree T with k leaves. Let $leaf(T)$ and $root(T)$ represent the set of leaves and the root of the tree T respectively. Each class is placed at exactly one leaf of T so that $leaf(T) = C$. Let w be an internal node of T, and let T_w denote the subtree rooted at w. Each such internal node w is then associated with a set of classes $C_w = leaf(T_w)$. Let X_{C_w} represent the data obtained by putting together the data belonging to all classes in C_w. Using X_{C_w}, each set of classes C_w, and thereby each internal node w, has an associated prior $\pi(C_w)$ and a probability density function p_{C_w}. Note that more sophisticated models for p_{C_w} can be used, such as a mixture model of pdfs associated with classes in C_w. Finally, we note that in this paper a collection of sets of classes such as $\{C_{v_i} : 1 \leq i \leq m\}$ is sometimes shortened to $\{C_{v_i}\}_{i=1}^m$.

2.2 Proposed Solution (ATG)

In this section we first describe a generic algorithm for the construction of a hierarchy and then justify the design choices made in this paper.

A GENERIC TOP-DOWN ALGORITHM.

We adopt a top-down approach to learning the tree structure. We start with T as a single node. Then $root(T)$ is associated with the set of given classes C and the variable $root(T).tosplit$ is set to $true$. At any time during the algorithm's run there are a set of leaves of the tree T that have their $tosplit$ variable set to $true$. We pick one such leaf w for splitting. Let w be associated with the set of classes C_w. We then need to find the m disjoint subsets $\{C_{v_i}\}_{i=1}^m$ into which C_w must be partitioned. This involves both finding the value of m and the subsets themselves. This partitioning is computed by a procedure called `findPartition`, which is described later in this section. Once the C_{v_i} are obtained, we create m new nodes v_i that are assigned as immediate children of w. Each C_{v_i} is then associated with the corresponding leaf v_i. The *tosplit* variable of each v_i whose associated C_{v_i} has more than one class is set to true; $w.tosplit$ is set to $false$. In this fashion we proceed with splitting leaves until all leaves have *tosplit* set to false. In other words, internal nodes are split until the leaves have only one class each. The pseudo-code for this algorithm is shown in Figure 1.

THE PARTITIONING CRITERION.

As mentioned above, at each internal node w of the tree we need to find a partitioning of the set of classes C_w into an appropriate number of subsets. But before we describe our choice of partitioning criterion we need a notion of distance between sets of classes.

We compute the distance between sets of classes using the *Jensen-Shannon (JS) divergence* [1]. The distance between two sets of classes C_1 and C_2 is defined in terms of their associated pdfs p_{C_1} and p_{C_2}, and priors $\pi_i = \pi(C_i)$

$$JS_\pi(\{C_1, C_2\}) = \pi_1 KL(p_{C_1}, \pi_1 p_{C_1} + \pi_2 p_{C_2}) + \pi_2 KL(p_{C_2}, \pi_1 p_{C_1} + \pi_2 p_{C_2})$$

where $\pi_1 + \pi_2 = 1$, $\pi_i \geq 0$, and KL is the Kullback-Leibler divergence. The JS divergence measures how "far" the classes are from their weighted combination, where the π_i assign the contribution of the two distributions. The JS measure is always non-negative, symmetric in its arguments, and, unlike the KL divergence, is bounded. Moreover, it can

Algorithm `constructTaxonomy`
Input: C is the set of all classes
p_{cj} are class-conditional density functions
Output: T is the rooted n-ary tree with $leaf(T) = C$
1. Initialize T as a single node. Set $root(T).classes = C$ and $root(T).tosplit = true$
2. `while` ($w.tosplit == true$), for some node w
3. $\{C_{v_i}\}_{i=1}^{m}$ = `findPartition`($w.classes$)
4. Create m new nodes v_i, set $v_i.tosplit = false$
5. `for-each` v_i
6. set v_i as a child of w
7. $v_i.classes = C_{v_i}$
8. `if` ($|C_{v_i}| > 1$) `then` set $v_i.tosplit = true$
9. `end-for`
10. `end-while`

Algorithm `findPartition`
Input: C_w is the set of n classes to partition
Output: $\{C_{v_i}\}_{i=1}^{m}$ form the partition of C
1. Let each class in C_w be a cluster $\{C_{v_i}\}_{i=1}^{n}$
2. Get JS-divergence among all pairs from C_{v_i}, and also between each C_{v_i} and C_w
3. Find all pairs $P_k = (C_{v_i}, C_{v_j})$ that violate the constraint in Equation (3)
4. From P, select the pair (C_{v_i}, C_{v_j}) which has the lowest value for the expression in Equation (4).
5. `while` (there exists a pair (C_{v_i}, C_{v_j}))
6. Replace C_{v_i} and C_{v_j} with $C_{v_k} = C_{v_i} \cup C_{v_j}$
7. Recompute pairwise JS-divergence as in step 2
8. Pick the pair (C_{v_i}, C_{v_j}) as in step 3 and 4
9. `end-while`

Figure 1. Pseudo-code for the proposed approach

be generalized to more than 2 sets of classes/distributions. The JS divergence between k sets of classes C_i is defined as

$$JS_\pi(\{C_i : 1 \leq i \leq k\}) = \sum_{i=1}^{k} \pi_i KL(p_i, p_m) \qquad (1)$$

where $\sum_i \pi_i = 1, \pi_i \geq 0$, and p_m is the weighted mean probability distribution $p_m = \sum_i \pi_i p_i$ [1]. In this paper we use $JS(\{c_j : c_j \in C_{v_i}\})$ to refer to the JS divergence between the set of distributions p_{c_j}; and $JS(\{C_{v_i}, C_{v_j}\})$ to refer to the JS divergence between the distributions $p_{C_{v_i}}$ and $p_{C_{v_j}}$, though they are sets of classes.

Using this definition of distance we can define a criterion of partitioning C_w. We would like to partition C_w into m disjoint subsets $\{C_{v_i} : 1 \leq i \leq m\}$ so as to minimize

$$JS_\pi\left(C_{v_i}(l), C_{v_i} \setminus C_{v_i}(l)\right) \sum_{i=1}^{m} \pi(C_{v_i}) JS_{\pi'}\left(\{c_j : c_j \in C_{v_i}\}\right) \qquad (2)$$

where $\pi(C_{v_i}) = \sum_{c_j \in C_{v_i}} \pi_{c_j}$, and $\pi'_{c_j} = \pi_{c_j}/\pi(C_{v_i})$, under the constraint that $\forall i, j \neq i$

$$JS_{\pi''}\left(\{C_{v_i}, C_{v_j}\}\right) > min\{JS_{\pi''}\left(\{C_{v_i}, C_w\}\right), JS_{\pi''}\left(\{C_{v_j}, C_w\}\right)\} \qquad (3)$$

where $\pi'' = \{1/2, 1/2\}$.

The objective function in Equation (2) computes the similarity of all the classes to the subset that they end up in. Minimizing this function gives us subsets that are very homogeneous. We would like to minimize this objective function over all possible m sized partitionings of C_w, where m ranges from $2 ... \|C_w\|$. Since the function in Equation 2 would be trivially minimized if C_w was partitioned into $\|C_w\|$ singleton subsets, we need to constraint the solution.

The constraint in Equation (3) ensures that none of the m subsets are closer to each other than to the "parent" C_w. In other words any solution in which there exist at least one pair of subsets that are closer to each other than to the parent is considered invalid. This constraint enforces a distance between sibling nodes, and is natural in the context of a taxonomy. If two sets of classes C_{v_1} and C_{v_2} are closer to each other than each is to their parent C_w, then it can be argued that they should be placed in the same subset, and be separated lower in the tree. Setting priors π'' to uniform in Equation (3) gives equal importance to all distributions, and prevents larger classes from biasing the mean distribution towards themselves.

An attractive feature of this constraint is that the threshold on the distance between subsets is defined by the distances of the subsets from the parent set, and from each other. Hence, a solution with C_{v_1} and C_{v_2} very close to each other will still be considered valid if either one of them is still closer to C_w. On the other hand, another solution in which C_{v_3} and C_{v_4} are far from each other might not be considered valid if both are even further away from the parent C_w. Since the partitioning criterion depends on the distance relationships between classes, the structure of the taxonomy is learned automatically and no parameters need to be set by the user.

As mentioned above, we want to minimize the objective in Equation (2) over all possible partitionings of C_w into m (where $2 \leq m \leq \|C_w\|$) subsets that satisfy the constraint in Equation (3). The optimal solution can be obtained by enumerating all possible solutions which satisfies the constraint, and picking the one which minimizes the objective. The time complexity of this procedure will be exponential in the number of classes in the parent. Hence, we need an algorithm that computes a "good" solution efficiently at the expense of optimality guarantees.

A GREEDY ALGORITHM TO FIND PARTITIONINGS.

In order to find a solution efficiently we devise a greedy agglomerative approach. This is implemented as the procedure `findPartition` in the pseudo-code in Figure 1.

Let the current node w being partitioned have a set of n classes C_w associated with it. We seek to find the partitioning by agglomeratively *clustering* the set of classes. We begin with each class as a separate cluster, $\{C_{v_i}\}_{i=1}^{n}$. We then obtain pair-wise distances (as defined by the JS divergence) between each pair of clusters, and also between each cluster and C_w. We define a *candidate-pair*

for merging as a pair of clusters C_{v_i} and C_{v_j}, such that they violate the constraint in Equation (3). From all such candidate-pairs we pick the one that has the smallest value for $(\pi(C_{v_i}) + \pi(C_{v_j}))JS_\pi(\{C_{v_i}, C_{v_j}\})$ and merge its constituents. The process of merging clusters C_{v_i} and C_{v_j} involves replacing them by another cluster C_{v_k} that includes both their classes ($C_{v_k} = C_{v_i} \cup C_{v_j}$), and then recalculating the pair-wise distances and finding the new set of candidate-pairs. We repeat this process of merging until no candidate-pairs remain. The final set of clusters (each a set of classes) define the partitioning of C_w.

In our algorithm, we start with a presumably *invalid* solution with the lowest possible objective function value; each class forms a singleton cluster $\{C_{v_i}\}_{i=1}^{n}$. We then successively merge clusters (and incur an increase in objective function value) until a valid solution is obtained. The pair of clusters for merging are chosen from the set of clusters that violate the constraint, in such a way so as to minimize the increase in objective function value. After a valid solution has been obtained we stop merging since further merging cannot improve the value of the objective function.

Proposition 1. *At any step in the* `findPartition` *algorithm, merging the candidate-pair (C_{v_i}, C_{v_j}) with lowest value of $\left(\pi(C_{v_i}) + \pi(C_{v_j})\right) JS_\pi \left(\{C_{v_i}, C_{v_j}\}\right)$ results in the least increase in the objective function value*

Corollary 1. *Merging clusters will always result in an increase in the value of the objective function.*

Proposition 1 can be proved by noting that the change in objective value due to merging sets C_{v_i} and C_{v_j} is

$$\begin{aligned}\delta &= \left(\pi(C_{v_i}) + \pi(C_{v_j})\right) JS_{\pi'} \left(\{c : c \in C_{v_i} \cup C_{v_j}\}\right) \\ &\quad - \pi\left(C_{v_i}\right) JS_{\pi'} \left(\{c : c \in C_{v_i}\}\right) \\ &\quad - \pi\left(C_{v_j}\right) JS_{\pi'} \left(\{c : c \in C_{v_j}\}\right) \\ &= (\pi(C_{v_i}) + \pi(C_{v_j}))JS_\pi(\{C_{v_i}, C_{v_j}\}) \qquad (4)\end{aligned}$$

where $\pi' = \pi_c/\pi(C_v)$ are the class priors normalized within each cluster. Equation (4) can be obtained by using Theorem 4 in [1]. Then Corollary 1 follows from the non-negativity of Jensen-Shannon divergence.

Proposition 2. *The* `findPartition` *algorithm in Figure 1 will terminate with at least two clusters.*

This proposition states that when only two clusters are left, each cluster will be closer to the parent than to the other cluster. In other words, a solution with only two clusters is always valid. This follows from the fact that the Jensen-Shannon divergence $JS_\pi(\{p_i\})$ is convex in p_i for a fixed π. This property of our algorithm ensures that the procedure `constructTaxonomy` in Figure 1 always terminates by outputting a tree with classes placed at the leaves.

The merging process in our algorithm is greedy and no guarantee can be given that the objective function will be optimally minimized. However, we note that some of these merges will be re-evaluated when children nodes are further partitioned lower in the tree. The "parent" node during these new partitions will be different and more specific. We claim that this ameliorates some of the effects of greedy merges and helps our algorithm find better hierarchies.

3 Experiments

In this section we present results of our evaluation of n-ary and binary tree based taxonomies generated by ATG and Agglomerative Information Bottleneck respectively.

DATASETS AND IMPLEMENTATION DETAILS.

We experiment on standard datasets that have a possible hierarchical structure over the classes. Further, we ensure that the datasets cover a wide range of domains and applications, such as text-categorization (20-Newsgroups), remote-sensing (KSC, Botswana), real-valued attribute based categorization (Glass, Pendigits, Vowel). For the Text dataset, the class-conditional pdfs at the leaf and the internal nodes are estimated by assuming an independent, multinomial distribution of the words. In the case of real-valued data, the pdfs are modeled as multi-variate Gaussian distributions. While partitioning an internal node, a vocabulary specific to that internal node is generated using the Fisher index criterion. In addition, for the remote sensing data, we make use of a domain-specific feature reduction technique called "best-bases" to account for the high degree of correlation between the different features. Further dataset and implementation details are provided in [3].

AGGLOMERATIVE INFORMATION BOTTLENECK.

Agglomerative Information Bottleneck (AIB) was proposed by Slonim and Tishby in [5] where it was used to hierarchically cluster words in a given dataset. However, this technique can also be applied to classes in order to construct a hierarchical structure over them. The method starts with each class as a separate cluster. The algorithm then produces a binary tree by greedily merging clusters that minimize the loss in mutual information of the intermediate clustering with the category labels. In this respect, this method resembles the way we partition the set of classes at each node in the function `findPartition` in Figure 1. Readers are referred to the original paper [5] for details of the AIB algorithm and to [3] for a comparison with our current approach. In the next section, we will compare the n-ary taxonomies generated by ATG with the binary taxonomies generated by AIB. This evaluation will also serve as a comparison of n-ary tree based taxonomies with binary tree based ones.

N-ARY TAXONOMIES ARE MORE NATURAL.

The taxonomies generated by ATG and AIB are depicted in the Appendix of the technical report [3]. From the taxonomy diagrams, one can see that for all datasets the ATG

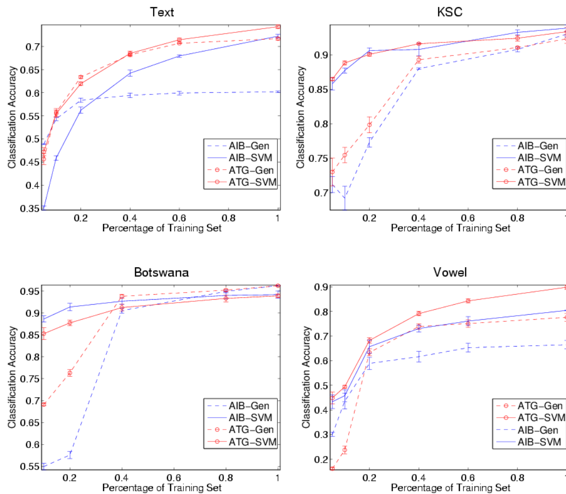

Figure 2. Hierarchies built from all training data.

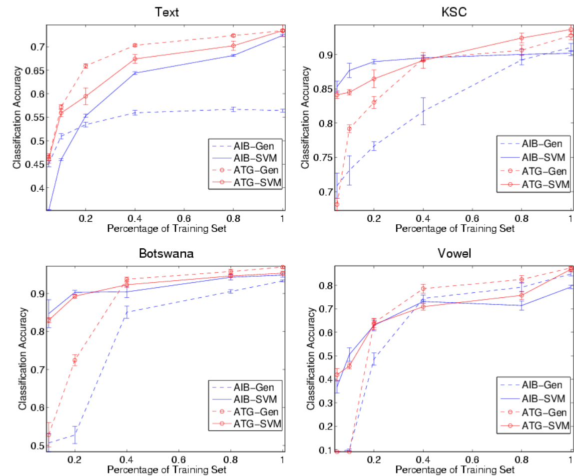

Figure 3. Hierarchies built from limited training data.

method yields n-ary taxonomies that reflects the underlying class affinities well. In particular, for the Glass dataset, we recover the exact hierarchical structure specified in the UCI-ML description of the dataset.

While the taxonomies generated by the AIB also eventually group similar classes together, the meta-classes generated higher up in the tree are a mix of fairly well-separated classes, such as the "autos" and "motorcycles" grouped with that of "politics" and "science" for the 20-Newsgroup dataset. This behavior is all the more striking for the Botswana dataset. The greedy nature of the AIB approach ensures that merge decisions are never re-evaluated, whereas reevaluating the similarities in a node-specific feature space better reveals the inter-class affinities in the ATG technique . For instance, for the 20-Newsgroup dataset, the ATG technique by virtue of reconsidering the Electronics/Computer cluster in a more specialized space is able to correctly group the "comp.os.ms.windows.misc" class with the "comp.windows" and "comp.graphics" classes, unlike AIB that clumps it with the hardware classes during the initial stages of hierarchy creation. Similar observations can be drawn from results on the other datasets.

CLASSIFICATION ACCURACIES.

A hierarchical taxonomy can be used as a classifier in which a multi-class problem can be broken down into a set of simpler problems. If the hierarchies are well-designed, each sub-problem would be simpler than the original one and would also typically require a smaller set of features to resolve it [2]. Figure 2 plots the learning rates of SVM and Bayesian classifiers learned on hierarchies constructed by ATG and AIB. While the classifiers are trained on differing fractions of training data, the hierarchies are constructed using all the training data. The superior classification accuracies of the ATG-Bayesian classifiers, under the limited data conditions, validates our assumption about the utility of using a "more natural" tree to learn a hierarchical classifier. Figure 2(a) shows that for datasets like that of Text, even "powerful" classifiers such as SVMs benefit from the n-ary splits in terms of the classification accuracies. In the set of graphs in Figure 3 we plot the learning rates of classifiers when even the hierarchies are constructed on fractions of the training data. It can be seen that for all datasets using the ATG hierarchy with internal Bayesian classifiers outperforms the AIB based hierarchical classifiers. Using SVM-based ATG classifiers offers comparable, if not better, classification accuracies than using SVMs with the AIB hierarchy. The results show that the proposed ATG method can not only be used to generate meaningful hierarchies, but can also be used as an alternative classifier especially for the low data conditions. More details of the experimental setup as well as additional results are provided in [3].

References

[1] I. S. Dhillon, S. Mallela, and R. Kumar. Enhanced word clustering for hierarchical text classification. In 8th *KDD*, pages 191–200, 2002.

[2] D. Koller and M. Sahami. Hierarchically classifying documents using very few words. In *ICML*, pages 170–178, 1997.

[3] K. Punera, S. Rajan, and J. Ghosh. Automatic construction of n-ary tree based taxonomies. Technical Report IDEAL-2006-TR06, Electrical and Computer Engineering, University of Texas at Austin, [`www.lans.ece.utexas.edu/~kunal/papers/icdm06-nary-long.pdf`], 2006.

[4] E. Segal, D. Koller, and D. Ormoneit. Probabilistic abstraction hierarchies. In 14th *NIPS*, 2001.

[5] N. Slonim and N. Tishby. Agglomerative information bottleneck. In 12th *NIPS*, 1999.

[6] V. Vural and J. G. Dy. A hierarchical method for multi-class support vector machines. In 21st *ICML*, 2004.