MF-Tree: Matrix Factorization Tree for Large Multi-Class Learning

Lei Liu  
Hewlett Packard Laboratories,  
Palo Alto, CA, USA 94304  
lei.liu2@hp.com

Pang-Ning Tan  
Michigan State University,  
East Lansing, MI, USA 48824  
ptan@cse.msu.edu

Xi Liu  
Michigan State University,  
East Lansing, MI, USA 48824  
liuxi4@cse.msu.edu

ABSTRACT

Many big data applications require accurate classification of objects into one of possibly thousands or millions of categories. Such classification tasks are challenging due to issues such as class imbalance, high testing cost, and model interpretability problems. To overcome these challenges, we propose a novel hierarchical learning method known as MF-Tree to efficiently classify data sets with large numbers of classes, which simultaneously induces a taxonomy structure that captures relationships among the classes. Unlike many other existing hierarchical learning methods, our approach is designed to optimize a global objective function. We demonstrate the equivalence between our proposed regularized loss function and the Hilbert-Schmidt Independence Criterion (HSIC). The latter has a nice additive property, which allows us to decompose the multi-class learning problem into hierarchical binary classification tasks. To improve its training efficiency, an approximate algorithm for inducing MF-Tree is also proposed. We performed extensive experiments to compare MF-Tree against several state-of-the-art algorithms and showed both its effectiveness and efficiency when applied to real-world data sets.

1. INTRODUCTION

Multi-class learning is a pervasive problem in the current age of big data, cutting across a broad range of applications, including text categorization [26, 19], image recognition [11, 24, 14], video classification [18, 23], and malware detection [9, 8, 25, 7]. In addition to the massive volume of data that must be handled, the classification tasks in these applications involve thousands or possibly millions of classes. Traditional multi-class learning methods, based on one-versus-one or one-versus-all strategies, require building \(O(c)\) or \(O(c^2)\) binary classifiers, where \(c\) is the number of classes. Such methods are computationally expensive during testing since one must apply a large number of classifiers to predict the class of a test instance.

Hierarchical learning methods [1, 2, 10, 28], which organize the binary classifiers into a tree-like structure, have been developed to overcome this limitation. The advantages to these methods are three-fold. First, they can reduce model testing to sublinear time in \(c\) since the test complexity depends on the maximum depth of the tree. Second, they could help alleviate the class imbalanced problem by learning to discriminate groups of related classes instead of individual classes (except at the leaf nodes of the tree). Finally, the hierarchical structure induced by such methods may offer interesting insights into the semantic relationships among the classes.

Numerous hierarchical learning methods have been developed in recent years, from greedy top-down decision tree classifiers that use simple attribute test conditions as internal nodes of the tree, to more advanced methods that employ binary classifiers such as SVM as their internal nodes. Each of these methods has its own pros and cons. While decision tree classifiers are easy to construct, they are susceptible to the class imbalanced and data fragmentation problems. In addition, a class can be assigned to multiple leaf nodes of the tree, resulting in an unnecessarily large and complex tree. This affects both its runtime for testing as well as interpretability of its hierarchical structure as a class taxonomy. Alternative methods, such as filter tree [2] and its variants, impose a random tree structure on the classes and learn a set of binary classifiers to route the classes to their appropriate leaf nodes in the tree. Because of the arbitrariness in which the classes are assigned to the leaf nodes, the resulting tree might be suboptimal and loses descriptive information about the class relationships. More importantly, both types of methods were not designed to optimize a global objective function. To address these limitations, we propose a novel method known as matrix factorization tree (MF-Tree, in short) to organize the classes into a binary tree by solving a global objective function based on a regularized loss function.

We provide a theoretical proof to demonstrate the equivalence between the proposed loss function and the Hilbert-Schmidt Independence Criterion (HSIC) [13][3][30]. HSIC is a recently proposed criterion for taxonomy learning in an unsupervised learning setting. In this paper, we demonstrate how HSIC can be applied to a supervised multi-class learning problem. We also showed the additive property of the objective function, which allows us to decompose the problem into a hierarchical matrix factorization task that can be represented as a binary tree. Since the tree is constructed by partitioning the classes into disjoint subclasses, the resulting tree provides a good representation of a class taxonomy as each class resides in exactly one leaf node of the tree.

For massive classification problems involving thousands of classes or more, factorizing the matrices is computationally expensive especially at the top levels of the tree. To address this problem, we present an approximate algorithm for inducing MF-Tree, where the partitioning of classes at the top level of the hierarchy is performed using a fast method instead of solving a matrix factorization problem involving large numbers of classes. The more expensive matrix factorization can be applied at the lower levels of the tree when
there are fewer number of classes and training examples to be dealt with. We showed that the accuracy of the approximate MF-Tree is quite comparable to that of MF-Tree, but with a significant improvement in its training time. In short, the main contributions of this paper are as follows:

- We introduce a novel hierarchical classification method for large multi-class learning problems known as MF-Tree. The tree is designed to optimize a global loss function.
- We provide a theoretical proof showing the equivalence between optimizing our regularized loss function and the Hilbert-Schmidt Independence Criterion (HSIC) by setting appropriate constraints on the latent factors of the factorization. We demonstrate the additive property of HSIC that enables the design of a hierarchical tree learning framework.
- We present an approximate algorithm for learning MF-Tree to speed up the tree induction process.
- We provide extensive experiments to evaluate and compare the performance of MF-Tree against several state-of-the-art baselines.

2. RELATED WORK

Classification with large number of classes is an important but challenging problem, and has received increasing attention in recent years [10][18][12]. Existing methods for multi-class learning employ a collection of binary classifiers, organized either in a non-hierarchical (flat) structure or a hierarchical tree architecture.

Flat (Non-hierarchical) classification methods, including one-versus-all [29], one-versus-one [15], and error-correcting output coding (ECOC) [21], require invocation of all the binary classifiers during testing time in order to determine the final prediction. Both one-versus-one and one-versus-all methods are inefficient when the number of classes is large due to the linear and quadratic number of binary classifiers that must be invoked during testing. The ECOC method assigns a unique m bit codeword to each class label (where $m > \log_2 c$). Given a test instance, we must generate an m-bit vector by applying m binary classifiers and then compute its distance to the codewords of all c classes, which is an expensive operation.

Hierarchical methods [28, 2, 1, 10] help to reduce the number of classifiers that need to be invoked during the testing phase by organizing the classifiers in a hierarchical structure. Their runtime for testing depends on the depth of the structure. For example, the Decision Directed Acyclic Graph (DDAG) [28] approach arranges the one-versus-one classifiers in a rooted binary DAG in order to reduce the runtime complexity from $O(c(c - 1)/2)$ to $O(c)$. In fact, it is possible to achieve sublinear complexity (e.g., $O(\log c)$) by assigning each class to a single leaf node of the hierarchy. Such a strategy was adopted by label tree learning methods such as [11, 1, 10], which constructs a linear classifier at each node to assign training instances to their corresponding child nodes. With this approach, only the binary classifiers located along the path from the root node to one of the leaf nodes are invoked when determining the class label of a test instance.

Hierarchical methods vary in terms of how their structure and the associated classes of each leaf node are learned from training data. For example, Bengio et al. [1] developed a label embedding tree approach by training c one-vs-all classifiers to obtain an initial confusion matrix, which is then used as the affinity matrix for applying spectral clustering to partition the classes into smaller subgroups.

Deng et al. [10] presented an alternative strategy that simultaneously learns the class partitioning and the weights of the associated linear classifiers by optimizing their joint objective function. However, the approach requires solving an integer programming problem, which is NP-hard. Instead they relaxed the optimization problem into a linear program in order to find a polynomial time solution. Shi et al. [11] also introduced a relaxed hierarchical learning approach for large-scale visual recognition. At each node, the classes are colored into positive and negative labels by a binary classifier while a subset of confusing classes are ignored. The coloring of classes and training of binary classifier is done simultaneously using a max-margin optimization approach. Lei et al. [22] proposed a label tree learning approach that allows for soft partitioning of the classes by minimizing an information theoretic loss function. Unlike other previous works, the leaf node of the tree may contain one or more one-class SVM models to distinguish the instances from different classes. Despite the extensive work, the main drawbacks of existing hierarchical methods is that they either assume the hierarchical structure is given [2] or they allow a class to be assigned to multiple leaf nodes of the tree. As noted in the previous section, the latter produces trees that are larger than necessary and affects its interpretability.

Hierarchical methods have also been widely used for unsupervised learning, starting with classical methods such as agglomerative hierarchical clustering. Kuan et al. [20] developed a hierarchical clustering method based on rank-2 nonnegative matrix factorization. The authors proposed a fast algorithm based on the active set method to solve their optimization problem. Their method also spent considerable amount of time detecting outliers to avoid generating too many leaf nodes with few instances. However, similar to classical hierarchical clustering, there is no global objective function solved by their algorithm. Taxonomy learning is another related area of our work. A family of unsupervised learning algorithms for taxonomy learning [13][3][30][4] have been proposed to cluster data into a hierarchical structure. Our work is inspired by the Hilbert-Schmidt Independence Criterion (HSIC) metric employed by these works. We showed how HSIC is related to our global objective function for matrix factorization, which enables us to decompose the problem into a hierarchical tree learning task.

3. PRELIMINARIES

This section presents a formal definition of the multi-class learning problem. We also introduce the concepts of Hilbert-Schmidt Independence Criterion and the tree-structured covariance matrix.

The connections between these concepts and the proposed MF-tree method will be shown in Section 4.

3.1 Problem Definition

Let $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ denote a set of training instances sampled from an unknown distribution $P$ over $X \times Y$, where $X \subseteq \mathbb{R}^d$ is the d-dimensional feature space and $Y = \{1, 2, \ldots, c\}$ is the set of permissible categories. For large multi-class learning problems, we assume $c \gg 2$. The goal of multi-class learning is to infer a target function $f$ that minimizes the expected risk of misclassifying any instance drawn from $P$, i.e., $\min_f \mathbb{E}_{(x,y) \in P}[\mathcal{L}(f(x), y)]$, where $\mathcal{L}$ is known as the loss function.

A standard approach for multi-class learning is to reduce the problem into multiple binary classification tasks. However, when the number of classes $c$ is too large, applying all the binary classifiers to predict the class of a test instance is computationally infeasible. To overcome this problem, hierarchical labeled tree algorithms have been developed for large multi-class learning problems. A hi-
erarchical labeled tree is a 4-tuple $T = < V, E, \Phi, \Lambda >$, where $V$ is the set of nodes, $E$ is the set of edges connecting parent nodes to their corresponding child nodes, $\Phi = \{ f_v(x) | v \in V \}$ is the set of classifiers associated with the internal nodes of $T$, and $\Lambda = \{ y_v | y_v \in \mathcal{Y}, \hat{v} \in V \}$ is the set of labels associated with the leaf nodes. In this paper, we restrict our framework to binary rooted trees only, even though, in principle, it is applicable to rooted trees with a branching factor greater than 2. Our objective in this work is to learn an optimal tree $T$ for large, multi-class learning problems.

### 3.2 Hilbert-Schmidt Independence Criterion (HSIC)

This section introduces the Hilbert-Schmidt Independence Criterion. Our presentation follows closely the definitions given in [13][30][3]. Let $(X, Y)$ be a pair of random variables drawn from a joint distribution $P_{X,Y}$. The Hilbert-Schmidt Independence Criterion (HSIC) [13] measures the dependence between $X$ and $Y$ by calculating the norm of its cross-covariance operator $C_{xy}$ in Hilbert space. It has been shown that this norm vanishes if and only if $X$ and $Y$ are independent variables. A larger value of HSIC also indicates a stronger dependence between the two variables, with respect to the choice of kernels [30].

Let $\mathcal{F}$ be a Reproducing Kernel Hilbert Space (RKHS) of functions from $\mathcal{X} \to \mathcal{R}$ with a feature map $\phi$, such that $(\phi(x), \phi(x'))_X = k_X(x, x')$, where $k_X : \mathcal{X} \times \mathcal{X} \to \mathcal{R}$ is a positive definite kernel. Likewise, let $\mathcal{G}$ be the RKHS on $\mathcal{Y}$ with a feature map $\psi$, such that $(\psi(y), \psi(y'))_Y = k_Y(y, y')$, where $k_Y : \mathcal{Y} \times \mathcal{Y} \to \mathcal{R}$ is the corresponding kernel for the metric space $\mathcal{Y}$. The covariance operator $C_{xy}$ is defined as

$$C_{xy} = E_{x,y}[(\phi(x) - \mu_x) \otimes (\psi(y) - \mu_y)]_{HS}$$

where $\mu_x = E[\phi(x)]$, $\mu_y = E[\psi(y)]$ and $\otimes$ denote a tensor product. The Hilbert-Schmidt Independence Criterion is defined as the squared Hilbert-Schmidt norm of the cross-covariance operator $C_{xy}$, which can be written as follows:

$$HSIC_{X,Y} = \|C_{xy}\|^2_{HS} = E_{x,x',y,y'}[k_X(x, x')k_Y(y, y')] + E_{x,y,y'}[k_X(x, x')E_{y,y'}[k_Y(y, y')]] - 2E_{x,y}[E_{x'}[k_X(x, x')]E_{y'}[k_Y(y, y')]]$$

Let $\{x_1, y_1\}, \{x_2, y_2\}, \ldots, \{x_n, y_n\}$ be the set of training instances drawn from $P_{X,Y}$. HSIC can be empirically estimated from the training data as follows [13, 3]:

$$HSIC = \frac{1}{(n-1)^2} tr \left(H_nKH_nL\right)$$

where $tr[\cdot]$ denote the trace of a matrix. $H_n = I - \frac{1}{n}1_n1_n^T$ is a centering matrix, $I$ is the identity matrix, $1_n$ is a column vector of all ones, $K$ is an $n \times n$ uncentered kernel matrix for $x$, and $L$ is an $n \times n$ kernel matrix for the class labels $y$. The matrix product $H_nKH_n$ corresponds to a centered kernel matrix. If $K$ denote the centered kernel matrix, then the computation of HSIC simplifies to

$$\frac{(n-1)^{-2} tr[KL]}{(n-1)^2}$$

### 3.3 Tree-Structured Covariance Matrix

Given a rooted binary tree $T$ on the labeled set $\mathcal{Y}$, we can represent it as a tree-structured covariance matrix [3] as follows.

**Definition 1. (Tree-Structured Covariance Matrix)**

A matrix $C \in \mathcal{R}^{c \times c}$ is a covariance matrix associated with the label tree structure $T$, where $C_{ij}$ is a measure of similarity between the $i$-th and $j$-th classes computed based on the path length from the root node to the closest common ancestor between the leaf nodes for $i$ and $j$ in $T$. The larger the value of $C_{ij}$, the closer is the relationship between the two classes.

Figure 1 shows an example of a labeled tree structure for 4 classes, $c_1, c_2, c_3, c_4$. Let $a, b, c, d, e, f$ denote the edge weights between the nodes. Based on the structure, the tree-structured covariance matrix is a $4 \times 4$ matrix shown on the right hand side of the diagram. For example, $C_{1,2} = a$ since the closest common ancestor between the classes has a path length equals to $a$ from the root node. Similarly, $C_{3,4} = b + e$, which is the path length from the root to the leaf node for $c_3$. Note that the edge weight $a, b, c, d, e, f$ determines the similarity between the different categories. For brevity, the edge weights are assumed to be 1 in the examples presented in the next section.

### 4. MF-TREE: PROPOSED METHOD

This section presents an overview of the proposed method. We first introduce the global objective function to be optimized by MF-Tree. We then show its connection to the HSIC measure described in the previous section. We also illustrate the additive property of HSIC, which allows us to develop a hierarchical tree-based induction algorithm.

#### 4.1 Global Objective Function of MF-Tree

Let $X$ denote an $n \times d$ centered design matrix, whose rows correspond to the data instances and columns correspond to the features. An uncentered design matrix $\tilde{X}$ can always be centered by applying the following operation, $H_n \tilde{X}$, where $H_n$ is the centering matrix defined in Section 3.2.

MF-Tree is designed to minimize the following regularized squared error loss function.

$$\mathcal{J} = \|X - ZW^T\|^2_F + \lambda tr[AWAT]$$

$$= tr[XX^T - XZW^T - ZW^TX^T + ZW^TWZ^T] + \lambda tr[AWAT]$$

where $Z$ is an $n \times c$ matrix, which represents the class membership of each data instance, $W$ is a $d \times c$ matrix, which represents the feature vector of each class, $A$ is a symmetric, positive-definite, $c \times c$ class covariance matrix, and $\lambda$ is the regularization parameter. $\| \cdot \|_F$ denote the Frobenius norm of a matrix and the superscript $T$ denote the transpose operator. The intuition behind the objective function given in Equation (3) is to factorize the centered design matrix $X$ into a product of latent factors $Z$ and $W$.

#### 4.2 Relationship between Global Objective Function and HSIC

To demonstrate the relationship between the objective function and HSIC, let us take the partial derivative of $\mathcal{J}$ given in Equation
(3) with respect to \( W \) and set it to zero:

\[
\frac{\partial J}{\partial W} = -2X^T Z + 2WZ^T Z + 2\lambda WA = 0
\]

\[
W = X^T Z [Z^T Z + \lambda A]^{-1}
\]

Replacing this back into Equation (3) yields the following:

\[
J = \text{tr} \left\{ XX^T - 2XX^T Z [Z^T Z + \lambda A]^{-1} Z^T \right\}
\]

\[
+ \lambda \text{tr} \left\{ X^T Z [Z^T Z + \lambda A]^{-1} A [Z^T Z + \lambda A]^{-1} Z^T X \right\}
\]

Thus, finding a solution for \( Z \) that minimizes \( J \) is equivalent to finding one that maximizes the following objective function:

\[
\max Z \text{tr} \left\{ XX^T Z [Z^T Z + \lambda A]^{-1} Z^T \right\}
\]

Let \( \hat{K} = XX^T \) be the centered kernel matrix associated with the training instances \( X \). During training, we enforce the following constraint:

\[
Z_{\text{train}} [Z_{\text{train}} Z_{\text{train}} + \lambda A]^{-1} Z_{\text{train}}^T = YC Y^T,
\]

(4) where \( Y \) is an \( n \times c \) matrix of class labels and \( C \) is the tree-structured covariance matrix. This can be achieved by setting \( Z_{\text{train}} = \hat{K} \).

**Theorem 1.** If \( A \) be a symmetric, positive-definite matrix, and \( \lambda > 0 \), then \( Z_{\text{train}} Z_{\text{train}} + \lambda A \) is also positive-definite.

**Proof.** Given any real vector \( v \),

\[
v^T Z_{\text{train}} Z_{\text{train}} + \lambda A v = \| Z_{\text{train}} v \|^2 + \lambda v^T A v > 0
\]

since \( A \) is a positive definite matrix.

Since \( Z_{\text{train}} Z_{\text{train}} + \lambda A \) is positive-definite, it is invertible. The equality constraint in Equation (4) ensures that an appropriate \( \lambda \) and \( A \) can be chosen to generate a viable tree, in which the entries of the matrix \( C \) must have a specific structure. Furthermore, let \( L = Y C Y^T \) denote the kernel matrix associated with the class labels. The objective function can now be simplified to:

\[
\max C \text{tr} \left[ KL \right] \text{s.t. } L = Y C Y^T,
\]

(5) which is equivalent to a constrained optimization of HSIC.

### 4.3 Additive Property of HSIC

In the previous subsection, we have shown that the global objective function for MF-Tree is equivalent to HSIC by setting appropriate constraints on \( A \) and \( \lambda \). This enables us to define a kernel matrix for the class labels as \( L = Y C Y^T \), where \( C \) is the tree-structured covariance matrix. In this section, we demonstrate the additive property of HSIC, which enables us to decompose the optimization problem into a hierarchical binary classification task.

First, we show that the tree-structured covariance matrix can be written as a sum of block diagonal matrices. We illustrate this with a simple example. Consider the following covariance matrix for a labeled tree that contains 8 leaf nodes (assuming the edge weights are equal to 1):

\[
C_8 = \begin{bmatrix}
3 & 2 & 1 & 1 & 0 & 0 & 0 & 0 \\
2 & 3 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 3 & 2 & 0 & 0 & 0 & 0 \\
1 & 1 & 2 & 3 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 3 & 2 & 1 & 1 \\
0 & 0 & 0 & 0 & 2 & 3 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 3 & 2 \\
0 & 0 & 0 & 0 & 1 & 1 & 2 & 3
\end{bmatrix}
\]

(6)

We can rewrite the matrix as follows:

\[
C_8 = I_2 \otimes 1_4 1_4^T + I_4 \otimes 1_2 1_2^T + I_8 \otimes 1
\]

(7)

where \( \otimes \) denote a Kronecker product and

\[
I_2 \otimes 1_4 1_4^T = \begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{bmatrix}
\]

\[
I_4 \otimes 1_2 1_2^T = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{bmatrix}
\]

\[
I_8 \otimes 1 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

The first term on the right hand side of Equation (7) simply partitions the first 4 rows and columns of the matrix into 1 cluster and the last 4 rows and columns into a second cluster. The second term on the right hand side partitions the first cluster into 2 smaller subclusters, while the third term puts each subcluster as a cluster itself (this is at the leaf node of the tree). In general, we can write

\[
C_{2m} = B_2 + B_4 + \cdots + B_{2m},
\]

(8)

where \( B_{2^i} = I_{2^{i-1}} \otimes 1_{2^{i-1}} \otimes \cdots \otimes 1_{2^{i-1}}, i > 1 \), and \( m \) is the depth of the binary tree. We can further simplify the decomposition to the following:

\[
C_8 = B_2 + B_4 + B_8
\]

= \begin{bmatrix}
B_{2,1} & B_{4,1} \\
B_{4,1} & B_{4,2}
\end{bmatrix} + \begin{bmatrix}
B_{8,1} & 0_{2 \times 2} & 0_{2 \times 2} & 0_{2 \times 2} \\
0_{2 \times 2} & B_{8,2} & 0_{2 \times 2} & 0_{2 \times 2} \\
0_{2 \times 2} & 0_{2 \times 2} & B_{8,3} & 0_{2 \times 2} \\
0_{2 \times 2} & 0_{2 \times 2} & 0_{2 \times 2} & B_{8,4}
\end{bmatrix}
\]

where each diagonal block has the following form

\[
B_{2^i,j} = \begin{bmatrix}
1_{2^i}^T & 0 \\
0 & 1_{2^i}^T
\end{bmatrix}
\]

(9)
and the dimension for each of the submatrices of 0s and $11^T$s in $B_{k,j}$ is $2^{m-i} \times 2^{m-i}$. Furthermore, given a block matrix $B_k$, we have

$$YB_kY^T = Y_1B_{k,1}Y_1^T + Y_2B_{k,2}Y_2^T + \cdots + Y_{k/2}B_{k,k/2}Y_{k/2}^T$$

where

$$Y = \begin{bmatrix} Y_1 & \cdots & Y_{k/2} \end{bmatrix}$$

Equation (10)

$$B_k = \begin{bmatrix} B_{k,1} & 0 & \cdots & 0 & \cdots & 0 \\
0 & B_{k,2} & \cdots & \cdots & \cdots \\
\vdots & \vdots & \ddots & \cdots & \cdots \\
0 & \cdots & \cdots & B_{k,k/2} \\
\end{bmatrix}$$

Equation (11)

$$Y^T = \begin{bmatrix} Y_1^T \\
\vdots \\
Y_{k/2}^T \\
\end{bmatrix}$$

Equation (12)

As each $B_{k,j}$ corresponds to a block-diagonal matrix given by Equation (9), we can write

$$YC_2Y^T = \sum_{i=1}^{\log_2 k} \sum_{j=1}^{\log_2 k} Y_i B_{2^j,2^j} Y_j^T$$

This decomposition enables us to solve the optimization problem with a hierarchical tree-like decomposition. The outer sum corresponds to decomposition at each depth of the tree, while the inner sum corresponds to expanding the internal nodes associated with the particular depth. Instead of learning $C$ directly by maximizing the trace of HSIC (see Equation (5)), we can learn the set of $B_{k,j}$ iteratively, each of which is associated with an internal node in the tree structure.

### 4.4 MF-Tree construction

To construct the tree, we need to solve the objective function given in (5). Based on the discussion in the previous subsection, the objective function can be simplified as follows:

$$\max_{\{ B \}} \sum_{i,j} \text{tr} \left[ K_{i,j} Y_i B_{2^i,2^j} Y_j^T \right]$$

Equation (13)

This allows us to solve the optimization problem one node at a time. Let $D_j = Y_j^T K_{j,j} Y_j$, which is simply a $c \times c$ similarity matrix of the classes, computed based on their attributes $X$ and the known ground truth labels associated with the block $B_{2^i}$. At each node of the tree construction process, we need to find a matrix $B_{2^i,j}$ that maximizes its alignment with $D_j$, i.e.:

$$\max_{B_{2^i,j}} \text{tr} \left[ D_j B_{2^i,j} \right]$$

For notational convenience, we drop the subscripts $i$ and $j$ in the remainder of the discussion. Note that $B$ is simply a co-association matrix, whose element $[B]_{p,q} = \begin{cases} 1, & \text{if classes } p \text{ and } q \text{ are in the same partition;} \\ 0, & \text{otherwise.} \end{cases}$

To create binary partitions, let $G$ be a $c \times 2$ matrix $^1$, where

$$[G]_{i,j} = \begin{cases} 1, & \text{if class } i \text{ belongs to partition } j; \\ 0, & \text{otherwise.} \end{cases}$$

It is easy to show that $B = GG^T$. Thus,

$$\text{tr} \left[ DB \right] = \text{tr} \left[ DGG^T \right] = \text{tr} \left[ G^T DG \right].$$

This objective function is equivalent to finding a clustering of the classes in a way that maximizes the within-cluster similarity.

### 4.5 Optimization

Note that the following optimization problem at each node

$$\max_{G} \text{tr} \left[ G^T DG \right] \text{ s.t. } G^T 1_2 = 1_n$$

is not feasible unless we enforce a non-negative constraint $G \succeq 0$ [6]. Furthermore, if we relax the condition that elements of $G$ must be either 0 or 1 and replace it with a constraint that $G$ must be an orthogonal matrix

$$\max_{G} \text{tr} \left[ G^T DG \right] \text{ s.t. } G^T G = I_2$$

we can solve the problem easily by finding the first two eigenvectors corresponding to the largest eigenvalues of $D$ [27, 17, 5]. Let $G \succeq 0$ be the eigendecomposition of matrix $D$.

By definition, $D = Y^T KY$. Following the eigendecomposition of $D$, we have

$$Y^T KY = GAG^T$$

Since $G^T G = I$:

$$G^T Y^T KY = \Lambda$$

Let $P = YG$, which is an $n \times 2$ matrix that determines the assignment of each data instance to the corresponding child nodes. Thus,

$$P^T KP = \Lambda$$

During the testing phase, let $k$ be an $n$-dimensional column vector that contains similarity between the test instance to all the training instances. We compute the 2-dimensional partition vector $p \in \mathbb{R}^2$ as follows:

$$\hat{p} = G^T Y^T k$$

After computing $\hat{p}$, we assign the test instance to the partition with the larger value, i.e., $\text{arg max}_i (\hat{p}^T k)_i$.

### 4.6 MF-Tree Induction Algorithm

The pseudocode for constructing MF-Tree is summarized in Algorithm 1. Our algorithm recursively partitions each node by calling the TreeGrowth function. Each invocation of the function returns a root node for a subtree that partitions the classes and their associated training instances into 2 groups. Each node is assumed to have a complex data structure, with the following fields: (1) $v.index$ contains the indices of the training instances assigned to the node, (2) $v.class$ represents the class label if $v$ is a leaf node, (3) $v.G$ is the discriminant function if $v$ is an internal node, and (4) $v.left$ and $v.right$ are pointers to its left and right child.

---

$^1$Here, we assume the number of classes associated with the given node is $c$. As the tree grows deeper, the number of classes also decreases, which in turn, reduces the number of rows in $G$. 

885
Algorithm 1 MF-Tree Construction

Input: \( X_{\text{train}} \in \mathbb{R}^{n \times d}, Y_{\text{train}} \in \mathbb{R}^{n \times c} \)
Output: Tree Structure, \( T \)
1. \( \text{root} = \text{TreeGrowth}(X_{\text{train}}, Y_{\text{train}}) \)
2. Insert \( \text{root} \) into \( T \)
3. return \( T \)

function TreeGrowth\((X,Y)\)
1. \( v = \text{createNode}() \)
2. \( v.index = \text{getIndex}(X) \)
3. if number of classes in \( Y < 2 \) then
4. \( v.class = \text{unique}(Y) \)
5. else
6. \((X^{(l)}, Y^{(l)}, X^{(r)}, Y^{(r)}, v, G) = \text{Partition}(X,Y)\)
7. \( v.left = \text{TreeGrowth}(X^{(l)}, Y^{(l)}) \)
8. \( v.right = \text{TreeGrowth}(X^{(r)}, Y^{(r)}) \)
9. end if
10. return \( v \)

function Partition\((X,Y)\)
1. Compute the centered kernel \( K \) from \( X \).
2. Compute \( D = Y^T K Y \)
3. Compute the first two eigenvectors \( g^{(1)}, g^{(2)} \) of \( D \)
4. Let \( G = [g^{(1)}, g^{(2)}] \)
5. \((X^{(l)}, Y^{(l)}) = \{ \langle x_i, y_i \rangle | y_i = k, \text{arg max}_j G_{kj} = 1 \}\)
6. \((X^{(r)}, Y^{(r)}) = \{ \langle x_i, y_i \rangle | y_i = k, \text{arg max}_j G_{kj} = 2 \}\)
7. return \((X^{(l)}, Y^{(l)}, X^{(r)}, Y^{(r)}, G)\)

The Partition function decides how to partition each class and their training examples into different branches of the tree. The partitioning is done based on the magnitude of matrix \( G \) as described in the previous section. If a class \( k \) is assigned to the left child of a node, all the training instances that belong to class \( k \) will also be propagated to the left child. The algorithm continues to expand a node until all the training instances associated with the node belong to the same class.

To determine its time complexity, note that at each node \( v \), we need to compute the kernel matrix \( K \), an operation that requires \( O(n^2 d) \) time. Computing \( D \) from the kernel matrix \( K \) and class label \( Y \) takes \( O(n^2 c + c^2 n) \) time while finding its first two eigenvectors requires, in the worst-case, \( O(c^3) \) time. Thus, the computational cost at each node is \( O(n^2 c + c^2 n + c^3) \). The cost reduces significantly as the tree grows deeper since the size of matrices \( K \) and \( D \) decreases substantially. Once the tree structure is induced from the training set, we can predict the label for any test instance using the method presented in Algorithm 2.

5. APPROXIMATE MF-TREE

As shown in the previous section, the time complexity for constructing MF-Tree can be quite expensive especially at the top levels of the hierarchy. This is due to the large size of matrices that must be constructed and factorized according to their largest two eigenvalues. However, as we traverse down the tree, the number of classes and training instances to be dealt with decreases considerably, which in turn, helps to speed up the computations. Thus, the bottleneck of our computation lies in the first few iterations of the MF-Tree algorithm. In this section, we present an approach to speed up the tree construction process by learning an approximation for the \( G \) matrix during the first few iterations. Once the matrices become small enough, we can proceed with applying the original TreeGrowth function shown in Algorithm 1.

Our proposed method attempts to balance the tradeoff between accuracy and efficiency of the labeled tree construction process. While efficiency is a concern at the top levels of the tree, accuracy is an issue at lower levels of the tree. Even if the class partitioning at the top levels is suboptimal, one might still be able to recover good partitions at subsequent iterations when refining the tree. Based on this rationale, we develop the following two-step approach to improve training efficiency. First, we apply a simple method to calculate \( G \) at the top levels of the tree. When the depth of the tree exceeds some threshold \( \tau \), we revert back to the original TreeGrowth function. We termed this approach as Approximate MF-Tree, which is summarized in Algorithm 3.

Consider the root node of the labeled tree. Let \( n \) be the number of training instances and \( c \) be the number of classes. We first sort the classes in decreasing order of their class size and then assign the largest class \((c_1)\) to the left node of the root and the second largest class \((c_2)\) to the right child. The sorting operations requires only \( O(c \log c) \) computations. For the remaining classes, we compute their corresponding mean vectors, an operation that requires \( O(nd) \) time. We then assign each class to the left or right child based on their distance to the mean vectors of the first two largest classes. For example, if the mean vector for \( c_3 \) is closer to the mean vector for \( c_1 \) than \( c_2 \), we assign \( c_3 \) to the left child of the root node. Otherwise, we assign it to the right child. This process is repeated until depth \( \tau \). This reduces the time complexity of each node from \( O(c^3) \) to \( O(c \log c + nd) \). As will be shown in our experimental results section, substantial improvement in training time can be achieved by setting the depth threshold \( \tau \) to be 2 or 3 without losing significant accuracy.

6. EXPERIMENTAL EVALUATION

In this section, we compared the performance of our proposed MF-Tree algorithm against several state-of-the-art hierarchical learning algorithms\(^2\), including DAGSVM [28], Discriminative Relaxed Hierarchy Learning (DRHL) [11], confusion matrix based label embedding tree learning approach (CMTL) [1] and Recursive Non-negative Matrix Factorization (RNMF) [22]. In addition to these methods, we have also implemented a recursive version of HSIC clustering by dependence maximization (RDM) algorithm [30]. In this approach, the data instances are partitioned into \( p \) groups at each level of the tree until a stopping criterion is met (when no other classes have more than 5 instances at the leaf node). We con-

\(^2\)Even though the accuracy of non-hierarchical methods such as one-versus-one was comparable to hierarchical methods, they are extremely slow when the number of classes is too large.
Algorithm 3 Approximate MF-Tree

Input: $X_{train} \in \mathbb{R}^{n \times d}$, $Y_{train} \in \mathbb{R}^{n \times k}$
Output: Tree structure $T$

1. Set depth, $d = 0$
2. $root = TreeGrowthApprox(X_{train}, Y_{train}, d, \tau)$
3. Insert $root$ into $T$
4. return $T$

function TreeGrowthApprox$(X, Y, d, \tau)$
1. $v = createNode()$
2. $v.index = getIndex(X)$
3. if $d < \tau$ and number of classes in $Y < 2$
4. \hspace{1em} $(X^{(i)}, Y^{(i)}) = Partition2(X, Y)$
5. \hspace{1em} $v.left = TreeGrowthApprox(X^{(i)}, Y^{(i)}, d + 1)$
6. \hspace{1em} $v.right = TreeGrowthApprox(X^{(r)}, Y^{(r)}, d + 1)$
7. else
8. \hspace{1em} $v = TreeGrowth(X, Y)$
9. end if
10. return $v$

function Partition2$(X, Y)$
1. $classes = Sort(Y)$
2. $(X^{(i)}, Y^{(i)}) = \{(x_i, y_i) | y_i = c_1 \text{ (largest class) } \}$
3. $(X^{(r)}, Y^{(r)}) = \{(x_i, y_i) | y_i = c_2 \text{ (second largest class) } \}$
4. $meanvectors = Mean(X, Y)$
5. for each remaining class $c \in classes$
6. \hspace{1em} $m_1 = distance(meanvectors, c, c_1)$
7. \hspace{1em} $m_2 = distance(meanvectors, c, c_2)$
8. if $m_1 \leq m_2$
9. \hspace{1em} $(X^{(i)}, Y^{(i)}) = (X^{(i)}, Y^{(i)}) \cup \{(x_i, y_i) | y_i = c \}$
10. else
11. \hspace{1em} $(X^{(r)}, Y^{(r)}) = (X^{(r)}, Y^{(r)}) \cup \{(x_i, y_i) | y_i = c \}$
12. end if
13. end for
14. return $(X^{(i)}, Y^{(i)}, X^{(r)}, Y^{(r)})$

Struct a binary classifier at each internal node of the tree as well as 1-class SVM classifiers at the leaf nodes to determine the class labels.

6.1 Experimental Setup

We performed our experiments on two real-world data sets: (1) Caltech-256\(^1\), which is a benchmark data set for multi-class learning. The data set contains 30,607 images from 256 classes, with at least 80 images in each class, and (2) Wiki, a collection of Wikipedia articles. We generated four variants of the data sets for our experiments and denote them as Wiki1, Wiki2, Caltech1, and Caltech2, respectively. A summary of the characteristics for each data set is shown in Table 1.

Caltech1 and Caltech2 are subsets of image data from the Caltech-256 collection. Caltech1 is generated using the same approach as described in [11], in which 80 images are randomly sampled from each class to create the data set. We use a repeated holdout method to create the training and test sets. Specifically, half of the sampled images were reserved for training while the remaining half for testing. We repeat the sampling process five times to create five versions of the data set. We then applied all the algorithms on the data sets and compute their average F1-score. The F1-score is defined as the harmonic mean of the precision (positive predictive value) and recall (sensitivity) for each class. Results are reported based on the average F1-score for all the classes over all five versions of the data sets. In addition, we created a larger data set called Caltech2 by randomly choosing 40 images from each class for training and use all the remaining images for testing. The sampling process is again repeated five times to generate five versions of Caltech2 for our experiments. In both Caltech1 and Caltech2 data sets, we extracted SIFT (scale-invariant feature transform) features to represent the images.

Wiki1 is a data set generated using the same approach as described in [22]. Here, we choose articles from the 214 largest categories to obtain a data set that contains 24,378 articles. In addition, we also created a larger sample data set from the Wikipedia dump, which covers the largest 1618 categories with 65,156 articles. Similar to Caltech1, half of the sampled data were used for training and the remaining half for testing. The sampling process is also repeated five times to generate five versions of the data sets.

6.2 Experimental Results

This section presents the results of our experiments. We use average F1-score as our measure of accuracy for each algorithm. In addition, we compared the testing time for each algorithm along with the size of the induced hierarchy.

6.2.1 Comparison against Baseline Algorithms

We first compare the performance of MF-Tree against the five baseline algorithms. The results shown in Table 2 suggest that the proposed MF-Tree algorithm consistently outperforms other baseline algorithms both in terms of its F1-score and testing efficiency. The size of the hierarchy generated by MF-Tree is also comparable to some of the best algorithms. Even though RNMF generates a shorter tree than MF-Tree, its test time is higher because the leaf nodes of RNMF contains multiple 1-class SVM classifiers that must be invoked in order to predict the class. In contrast, the leaf nodes of MF-Tree has a single class label, which allows us to predict the class efficiently. Among all the competing methods, DAGSVM creates the largest tree. This is because it eliminates one class at a time at each level of the hierarchy. So the depth of the tree is equal to the number of classes. In the meantime, DRHL and RDM also create larger trees because they both allow a class to be assigned to more than one leaf nodes. This affects the test time of the algorithms.

6.2.2 Effect of Parameter Tuning

Unlike MF-Tree, which is parameter-free, the performance of several baseline algorithms depends on the values of their parameters. For example, the branching factor $p$ in RNMF method determines the structure and depth of the tree. If $p = 1$, this produces a decision stump consisting of $c$ 1-class SVM models. This is equivalent to implementing a one-versus-all approach. If $p = 2$, then it produces a binary tree. Clearly, the choice of $p$ affects the depth of the tree. As depth increases, the test efficiency reduces significantly at the expense of decreasing F1 values. Similarly, tree construction parameters are also needed in other baseline methods. To provide a fair comparison, we vary the parameters for each method based on the suggestion of their original papers. Specifically, for RNMF [22]

---

\(^1\)http://authors.library.caltech.edu/7694/
and RDM [30], we vary their parameters \( p \in \{2, 3, 4, 5\} \) whereas for DRHL [11], we set \( p \in \{0.5, 0.6, 0.7, 0.8\} \).

We plotted their classification performance in Figure 2, where the horizontal axis corresponds to test time while the vertical axis corresponds to average F1-score. Ideally, we seek for a classifier with lowest test time and highest F1-score (i.e., closest to the upper left corner). As can be seen from these plots, MF-Tree is better than the baseline hierarchical learning methods on all four data sets.

### 6.2.3 Comparison of Taxonomy Structure

In addition to their F1-scores, it is useful to compare the tree generated by the different methods against their ground truth structure. To do this, we need an evaluation measure to compare the similarity between two trees. In this paper, we apply the edit distance measure, which was originally used for string comparison, to compare the ordered labeled trees (see [31] for a review). Ordered labeled trees are trees in which the left-to-right order among siblings is significant. The distance between two trees is computed by considering the optimal mapping between the two trees. Specifically, the distance is given by the minimum cost of elementary operations to convert one tree into the other. An alternative way to map and edit the trees is by using tree alignment [16].

To quantify the difference between two trees based on their pairwise class ordering, we first construct an adjacency matrix \( A \) for a tree structure using the formula: 

\[
A_{ij} = \frac{\text{num\_hops}(c_i \rightarrow c_j)}{\sum_{c_k \in \text{classes}} \text{num\_hops}(c_k \rightarrow \text{root})}
\]

where \( A_{ij} \) is the similarity between classes \( c_i \) and \( c_j \). To measure the mean squared error (MSE), which is given by:

\[
MSE(A, \hat{A}) = \frac{1}{n^2} \sum_{i,j=1}^{n} (A_{ij} - \hat{A}_{ij})^2
\]

To illustrate how the measure works, consider the hypothetical trees shown in Figure 3. Let \( A \) be the adjacency matrix for the actual tree while \( A_{result1} \) and \( A_{result2} \) are the corresponding adjacency matrices for two competing trees, \( result1 \) and \( result2 \). The MSE for \( result2 \) is 0.01383, which is smaller than the MSE for \( result1 \), which is 0.03125. This result resembles the actual tree except for the misplaced class \( c_2 \). In contrast, the classes \( c_2 \) and \( c_3 \) are misplaced in \( result1 \). For example, \( A(c_3, c_4) = 0.5 \) in the actual tree as the number of hops from category \( c_3 \) to \( c_4 \) is equal to two. However, \( \hat{A}_{result1}(c_3, c_4) = 0.25 \) for \( result1 \) since class \( c_3 \) is 4 hops away from \( c_4 \) whereas \( \hat{A}_{result2}(c_3, c_4) = 0.5 \), which is the same as the distance according to the actual tree.

We compared the MSE score of MF-Tree against the confusion matrix based label embedding tree learning (CMTL) method. We choose this baseline method for two reasons. First, the size of the tree is comparable to MF-Tree. Second, similar to MF-Tree, this approach produces a tree in which each class can only reside in a single leaf node of the tree (whereas other methods allow a class to be assigned to multiple leaf nodes). Trees that restrict each class to a single leaf node are more interpretable as they may be used to define a concept hierarchy for the application domain. The experimental results shown in Figure 4 suggest that MF-Tree produces a tree structure that is closer to the ground truth compared to CMTL. Note that Caltech1 and Caltech2 use the same set of classes for training to construct the taxonomy, so their trees are identical. We report their results together as Caltech in Figure 4.

Figure 5 shows a subset of the tree structure generated by MF-Tree on the Caltech1 data set. Observe that the tree was able to capture the relationships among many of the classes quite well. For example, some of the classes of animals (dog, horse, chimp, bear) was assigned to the same branch. Though there were exceptions and misclassifications (e.g., bats and crabs) but the related classes are often quite close together at the lower levels of the tree.

### 6.3 Results for Approximate MF-Tree

Despite its lower test time compared to other approaches, training an MF-Tree is still expensive when the number of classes is large. To overcome this problem, we propose an approximate MF-Tree algorithm, which uses an inexpensive partitioning method to assign classes to their child nodes at the top levels of the hierarchy. The algorithm has a parameter \( \tau \) that determines the maximum depth at which to apply the inexpensive partitioning method.
To justify the effectiveness of the approximate MF-Tree approach, we perform an experiment on the \textsc{Wiki2} data set. We choose this data set because it has the largest number of classes. For this experiment, we vary \( \tau \) from 3 to 6 and compare the average F1-score as well as training time of the algorithms. The results are shown in Table 3. Recall that the average F1-score for MF-Tree on the \textsc{Wiki2} data set is 0.3318. When \( \tau = 3 \), the F1-score reduces slightly to 0.3289. However, there was a 22.6\% training time improvement. As \( \tau \) increases, the F1-score gradually decreases while its training time continues to improve. Note that the size of the tree generated by the original MF-Tree algorithm has a depth equals to 11. When \( \tau = 6 \), the improvement in training time appears to taper off while its F1-score decreases to 0.2868. This clearly shows a trade-off between accuracy and training time efficiency. In practice, we could set the \( \tau \) threshold based on the specific needs of the problem. If our priority is lower training speed, we could set a higher \( \tau \) value. \( \tau \) can also be set based on the computational resources available. If there is limited memory, \( \tau \) can be set to a threshold in such a way that the resulting data matrices can fit into the memory available.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>F1</th>
<th>% improvement in training time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau = 3 )</td>
<td>0.3289</td>
<td>22.6%</td>
</tr>
<tr>
<td>( \tau = 4 )</td>
<td>0.3169</td>
<td>32.3%</td>
</tr>
<tr>
<td>( \tau = 5 )</td>
<td>0.3034</td>
<td>37.8%</td>
</tr>
<tr>
<td>( \tau = 6 )</td>
<td>0.2868</td>
<td>39.5%</td>
</tr>
</tbody>
</table>

7. CONCLUSION

In this paper, we proposed a novel hierarchical method for large-scale multi-class learning called MF-Tree. Our proposed algorithm is driven by a global objective function. We demonstrate the equivalence between the global objective function and the HSIC metric and show it has an additive property that can be exploited to design a hierarchical tree learning algorithm. Experimental results comparing the method against five baseline methods demonstrated both the effectiveness and efficiency of our method.

8. REFERENCES


Figure 5: Illustration of tree structure generated by MF-Tree.


