Saving Evaluation Time for the Decision Function in Boosting: Representation and Reordering Base Learner

Peng Sun SUNP08@MAILS.TSINGHUA.EDU.EDU
Tsinghua National Laboratory for Information Science and Technology(TNList), Department of Automation, Tsinghua University, Beijing 100084, China

Jie Zhou JZHOU@TSINGHUA.EDU.CN
Tsinghua National Laboratory for Information Science and Technology(TNList), Department of Automation, Tsinghua University, Beijing 100084, China

Abstract

For a well trained Boosting classifier, we are interested in how to save the testing time, i.e., to make the decision without evaluating all the base learners. To address this problem, in previous work the base learners are sequentially calculated and early stopping is allowed if the decision function has been confident enough to output its value. In such a chain structure, the order of base learners is critical: better order can lead to less evaluation time.

In this paper, we present a novel method for ordering. We base our discussion on the data structure representing Boosting's decision function. Viewing the decision function a boolean expression, we propose a Binary Valued Tree for its representation. As a secondary contribution, such a representation unifies the work by previous researchers and helps devise new representation. Also, its connection to Binary Decision Diagram(BDD) is discussed.

1. Introduction

We are interested in how to save testing time for a Boosting classifier. Let's assume there are $T \in \mathbb{N}^+$ base learners and unit 1 time cost is incurred whenever a base learner is evaluated at testing. A naive evaluation of the decision function by revealing all the base learners obviously involves time complexity T .

Proceedings of the 30^{th} International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.

In this paper we pursue less-than-T algorithms while preserving the classification accuracy of the original decision function as much as possible. This would be helpful for testing time demanding applications, e.g., object detection in computer vision, trigger design in high energy physics, web page ranking, spam filtering, to name a few (Busa-Fekete et al., 2012; Chen et al., 2012).

The previous work addressing this problem falls in two categories: 1) Intrusive modification on Boosting algorithm, including (Viola & Jones, 2004; Saberian & Vasconcelos, 2010; Reyzin, 2011; Grubb & Bagnell, 2012). In such work, the trade off between classification error and evaluation time cost is considered in the training stage of Boosting. 2) Conversion method, which exploits the structure of the Decision Function learned in original Boosting. Due to the additive form, the output of the Decision Function can be known without evaluating all the base learners. In FastExit (Sochman & Matas, 2005; Kim et al., 2012), evaluation stops if the sign cannot be altered by remaining base learners; In Direct Backward Pruning (Zhang & Viola, 2007), a more greedy stopping strategy considering data distribution is proposed; (Busa-Fekete et al., 2012) models the Decision Function evaluation as a Markov Decision Process in Reinforcement Learning framework; (Kim et al., 2012) views the outputs of each base learner on training dataset as new features and retrain a classification tree on them.

Our work belongs to the latter category. Our hope: given a well trained Boosting classifier on which various techniques have been applied to ensure a low generalization error, we attempt to maximally squeeze out testing time when preserving, exactly or approximately, the learned classification boundary. To do this, we view the Decision Function as a ***Boolean Expres-***

sion (BE) in line of (Kim et al., 2012). Then we represent the BE with appropriate data structure that can be easily manipulated so as to minimise a surrogate target function of evaluation time cost.

Binary Decision Diagram (BDD) (Bryant, 1986), a well known data structure for general BE, of course does the job. However, the BE we are concerning has its own special structure: the additive form. Noting this, we propose a representation tailored for this type of BE, which we refer to as ***Binary Valued Tree (BVT)*** . BVT is just a conceptual tool in the sense that one needs not really implement it in program. Instead, it helps us to unifies previous work and devise new representation (e.g., see section 3.2).

Akin to BDD, BVT is an "ordered representation" where a fixed order of base learners is taken for all possible inputs to the BE. For such a representation, we argue that the natural order, *i.e.*, the as-is one yielded after Boosting finished its training, is usually sub-optimal, while reordering can save more evaluation time. Only a few of researchers seem to have noticed the problem, e.g., (Sochman & Matas, 2005; Kim et al., 2012), (Bourdev & Brandt, 2005) and (Chen et al., 2012). [1]

Other than the above methods, in this paper we propose the use of SIFTING algorithm for reordering. The underlying idea is simple: starting with a natural order and repeatedly swapping two adjacent base learners until some criterion is satisfied. Firstly introduced by (Rudell, 1993), SIFTING has been proved to be an effective algorithm for BDD optimization (Ebendt et al., 2005). We borrow it here since we find it applicable to those "ordered representations" in BVT family. SIFTING performs well in our experiments.

The rest of this paper is organized as follows: in section 2 we discuss the representation of Boosting's decision function; in section 3 we further discuss it when considering data distribution; in section 4 we introduce a measure for evaluation time and show how to minimize it by reordering; in section 5 we review related work; finally we show the experimental results in section 6.

2. Data-Free Representation

In this section we will view Decision Function as BE with special structure, *i.e.*, the ***Pseudo Boolean Expression (PBE)*** . We explain how to represent PBE and show how this representation is helpful to understand the technique for saving evaluation time.

2.1. Decision Function as Pseudo Boolean Expression

Boosting's decision function consists of firstly summing up many base learners and then taking its sign:

$$H(h_1, \ldots, h_T) = sgn(\textstyle\sum_{t=1}^{T} h_t), \tag{1}$$

where $sgn()$: $\mathbb{R} \mapsto \{-1, +1\}$ is sign function and h_t is the scalar output of the t-th base learner. In this preliminary work, we simply assume the base learner is binary valued (e.g., a decision stump):

$$h_t \in \{\alpha_t, \beta_t\}, \tag{2}$$

where $t = 1, \ldots, T$ and α_t, $\beta_t \in \mathbb{R}$. Without loss of generality, we further assume $\alpha_t \leq \beta_t$ always holds.

Expression (1) is essentially boolean in that both the input $h_t \in \{\alpha_t, \beta_t\}$ and output $H \in \{-1, +1\}$ are bi-state. However, the expression involves summation defined for real numbers. We hence refer to (1) as Pseudo Boolean Expression (PBE).

2.2. Graphic Representation of PBE

Due to its finitely binary nature and its additive form, PBE (1)'s all possible states can be "visualized". We begin with a helper function.

Definition 1 (***Accumulative Sum***)**.** For $t = 1, \ldots, T$, we recursively define the accumulative sum function as

$$acc(t) = acc(t-1) + h_t \tag{3}$$

with initial condition $acc(0) = 0$.

Depending on context, $acc(t)$ should be understood as either a set of values for all possible base learner output $h_1, \ldots, h_t$ or a single value for some specific base learner output.

We are now ready to show the following concept:

Definition 2 (***Binary Valued Tree, BVT***)**.** A tree visualizing all possible $acc(t)$, $t = 0, 1, \ldots, T$ of some PBE, where

1: Nodes are organised level by level. Each level is a Real Number Axis. Nodes at the t-th level are posited by corresponding $acc(t)$ values.
2: The edges outgoing from the $(t-1)$-th level and incoming to the t-th level capture all possible ways how $acc(t)$ can be computed from $acc(t-1)$ according to (3).

[1] Unfortunately, we were not aware of (Chen et al., 2012) until a reviewer pointed it to us. It is thus too late to include a thorough comparison with this work in the current paper. Our apology!

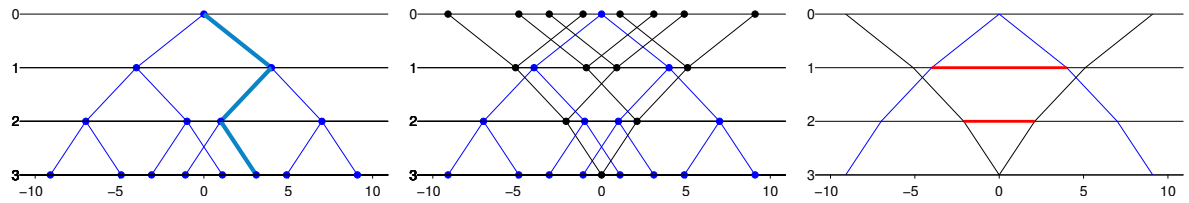

Figure 1. **Left**: a BVT for PBE, see texts in Definition 2 and Example 1; **Middle**: DBVT overlapped on BVT, see texts in Definition 4; **Right**: $\mathcal{PI}$ (red) and $\mathcal{EI}$ (black), see texts in Definition 6 and Definition 5.

3: Each path starting from the root node and ending at the t-th level denotes a particular output for $h_1, \ldots, h_t$.

Example 1. The left graph of Figure 1 show the BVT for 3 base learners $h_1 \in \{-4, +4\}$, $h_2 \in \{-3, +3\}$, $h_3 \in \{-2.1, +2.1\}$, where

1: Values of the 4 nodes at level 2 are -7, -1, +1, +7.
2: The 4 edges between level 2 and level 1 captures the recursive relation $acc(2) = acc(1) + h_2$, where $h_2 \in \{-3, +3\}$.
3: The path for a certain input ($h_1 = +4, h_2 = -3, h_3 = +2.1$) is shown in the figure with thick lines in light blue.

To this extent, we immediately have the following concepts:

Definition 3 (***Dual Accumulative Sum***)**.** For $t = T, T-1, \ldots, 1$, recursively define

$$dacc(t-1) = dacc(t) + (-h_t) \tag{4}$$

with initial condition $dacc(T) = 0$. Note the minus sign before h_t!

Definition 4 (***Dual Binary Value Tree, DBVT***)**.** A BVT tree plotted bottom up, in reverse order for base learners $-h_T, -h_{T-1}, \ldots, -h_1$.

An example of DBVT is shown by black nodes and edges in the middle graph of Figure 1, where we also plot the corresponding BVT for the same Decision Function.

For either BVT or DBVT, we are interested in where the leftmost (rightmost) path intersects the level axis. Formally, we give following definition:

Definition 5 (***Range Interval***)**.** For the t-th level, define the range interval

$$\mathcal{I} = [min\{acc(t)\}, max\{acc(t)\}] \tag{5}$$

for BVT and the dual range interval

$$\mathcal{I}' = [min\{dacc(t)\}, max\{dacc(t)\}] \tag{6}$$

for DBVT. Note $acc(t)$ or $dacc(t)$ denotes a set of values here.

When range interval of BVT meets that of DBVT, it gives rise to the following two concepts:

Definition 6 (***Prison/Escape Interval*** $\mathcal{PI}$ & $\mathcal{EI}$)**.** At the t-th level, the Prison Interval ($\mathcal{PI}$) is defined as

$$\mathcal{PI} = \mathcal{I} \bigcap \mathcal{I}', \tag{7}$$

while the Escape Interval ($\mathcal{EI}$) is defined as the complement of $\mathcal{PI}$

$$\mathcal{EI} = \overline{\mathcal{PI}}, \tag{8}$$

where $\mathcal{I}$ and $\mathcal{I}'$ are defined in Definition 5.

The reason why we call them Prison/Escape is revealed short after. We now demonstrate them in the right graph of Figure 1, where the BVT as well as DBVT is simply the duplicate of that in the middle graph of Figure 1. For clarity the nodes and edges we don't care are omitted: only the leftmost and right most paths are plotted for either BVT (in blue lines) or DBVT (in black lines). At level 1 or level 2, $\mathcal{PI}$ is in red color, while the remaining parts at both sides are $\mathcal{EI}$ in black color. At level 0 or level 4, the cardinality of $\mathcal{PI}$ is zero, while $\mathcal{EI}$ coincides with the whole axis.

2.3. FastExit

The so-called FastExit, proposed in (Sochman & Matas, 2005; Kim et al., 2012), works based on a smart observation: compute $acc(t)$ for $t = 0, 1, ...$ and stop immediately while its sign cannot be altered by remaining base learners.

This algorithm can be interpreted geometrically: starting from the root node of BVT, walk left/right path by revealing the base learners one by one; keep going if arriving at $\mathcal{PI}$ (be prisoned); exit and output the sign as final decision if arriving at $\mathcal{EI}$ (escape).

Such a geometrical explanation, although trivial in itself, can help to understand/develop other more complicated algorithms, as what follows.

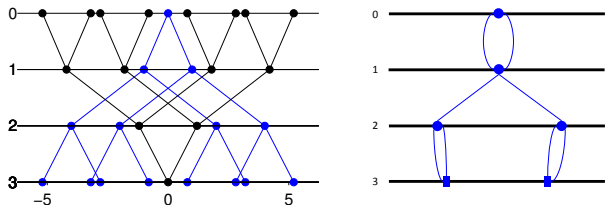

Figure 2. BVT+DBVT (Left) and the BDD (Right). See texts in section 2.4.

2.4. BDD (Binary Decision Diagram)

Since a data structure for any BE, BDD can of course represent expression (1). We refer to (Sanner, 2011) for the formalization of BDD and the general construction algorithm. When considering (1)'s structure (*i.e.*, its additive form), however, the construction can be simplified. This problem is studied under the name of Threshold BDD and strictly formalized in (Behle, 2010; Mayer-Eichberger, 2008), which is too lengthy to be restated here. In what follows we just briefly explain how it works using the geometrical concepts introduced above and point out how BDD helps save evaluation time.

Consider a BVT overlapped by a DBVT. The t-th level is divided into several intervals by DBVT nodes. Each such intervals can contain more than one BVT nodes – in case this happens, merge the nodes (since they are equivalent, see, say, Mayer-Eichberger (2008)) and use a single node as placeholder for the interval. After merging all possible BVT nodes we obtain the corresponding BDD, as is exemplified in Figure 2.

What's interesting is that there may exist BDD nodes whose left and right edges lead to the same child, which means this type of node is redundant such that we can travel to its child without computing the base learner (e.g., check the BDD node at the 0th level in the right graph of Figure 2. Note that such a redundancy cannot be captured by BVT). Theoretically, BDD representation saves more evaluation time than BVT.

3. Data-Dependent Representation

The representations in previous section are data free in that they just depend on the form of $acc()$. To save more evaluation time, in this section we show how to modify the representation when considering data distribution implied in training data set.

3.1. Direct Backwards Pruning

Using the concepts of BVT, we geometrically interpret how the Direct Backwards Pruning (DBP, Zhang & Viola (2007)) algorithm exploits data distribution.

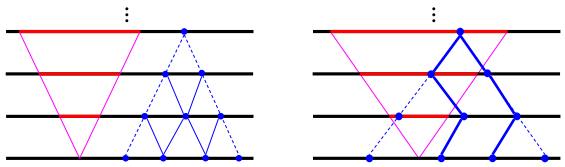

Figure 3. The last 4 levels of some BVT, whose nodes and edges are in blue. $\mathcal{PI}$ in red and $\mathcal{EI}$ in black at each level axis. Border paths for $\mathcal{PI}$ are in magenta. **Left**: In case of FastExit. The sign of the node at top level is known. **Right**: In case of DBP. The node at top level falls in $\mathcal{PI}$, its outgoing paths (blue dashed lines) could reach either $\mathbb{R}^-$ or $\mathbb{R}^+$ at bottom level. However, the example paths (blue solid lines) still lead to only $\mathbb{R}^+$.

Recall that a BVT shows all 2^T possible paths for $acc(T)$. When given N training examples, however, we usually have $N \ll 2^T$ for large enough T, which means there may be many paths carrying no training examples in this case! Suppose a BVT node falling in $\mathcal{PI} \bigcap \mathbb{R}^+$. While all its possible outgoing paths reach either $\mathbb{R}^-$ or $\mathbb{R}^+$ at bottom level, all its paths carrying examples may still lead to only $\mathbb{R}^+$ finally (as illustrated in the right graph of Figure 3). In case this happens, we can reset $\mathcal{PI}$'s upper bound to where the node resides. Similar processing goes for lower bound. After doing this, the modified $\mathcal{PI}$ usually becomes smaller, while the modified $\mathcal{EI}$, complement of the modified $\mathcal{PI}$, becomes larger, as in Figure 4.

When traversing BVT, we may have reached the modified $\mathcal{EI}$ but still stay in original $\mathcal{PI}$. In this sense, DBP theoretically saves more evaluation time than FastExit.

3.2. Interpolation between FastExit and DBP

DBP trusts the training data too much. Its $\mathcal{EI}$ tends to be too wide so as to blow up the testing error. Take the right graph of Figure 3 as an example. The two paths carrying training examples both lead to $\mathbb{R}^+$, but a testing example could walk the leftmost path reaching $\mathbb{R}^-$ finally (dashed blue line). In this case, the pruned $\mathcal{PI}$ is immature and testing error is incurred. To avoid this side effect, we can interpolate between FastExit and DBP (as shown in Figure 4). We propose an algorithm as follows.

DBPSYN: Interpolation by Synthesized Data Since the path walked by an example corresponds to the binary output of each base learner, it can be seen as a sequence of 0/1 code. Thus the difference between training example paths and testing example paths can

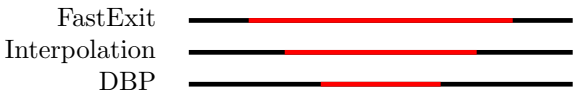

Figure 4. Typical size of $\mathcal{PI}$ (red interval) and $\mathcal{EI}$ (black intervals) at the same level for three methods.

be modeled by error code.

For the 0/1 codes of a training example, we can get a new example by randomly flipping a reasonable portion of the codes, where the portion is determined by 5-fold cross validation of train/test partitions. In this way we are able to obtain a set of synthesized training examples that mimic testing examples (3 times the size of training set in our experiments). This trick is similar to the one proposed in (Kim et al., 2012). Using the augmented examples including both original ones and synthesized ones, we apply DBP method. The $\mathcal{PI}$ and $\mathcal{EI}$ we obtain in effect interpolates between FastExit and DBP.

4. Reordering

In the representation discussed in previous texts, a certain order for base learners is implicitly assumed. However, the evaluation time is actually very sensitive to the order when taking into account data distribution. We provide an intuitive example as follows:

Example 2. For three base learners: $h_1 \in \{-2, +2\}$, $h_2 \in \{-1, +1\}$, $h_3 \in \{-1, +1\}$, consider two different orderings : (1) h_1, h_2, h_3. as in left graph of Figure 5; (2) h_3, h_2, h_1, as in right graph of Figure 5. When calculating the Decision Function for a certain input $(h_1 = +2, h_2 = +1, h_3 = +1)$, ordering (1) is obviously preferred since the path lead to $\mathcal{EI}$ at level 1, while for ordering (2) not until the bottom level does it occur.

In this section we discuss the techniques for evaluation time reduction by adjusting base learner order. To begin with, we review two methods in previous work.

Sorting Letting the weight of a base learner be $|\alpha| + |\beta|$, we can simply sort the weights of each base learner in descending order, as proposed in (Sochman & Matas, 2005; Kim et al., 2012). The underlying concept is as in Example 2 above. This method is data independent.

Top-Down Reordering The method proposed in (Bourdev & Brandt, 2005) can be seen as a "data version" of Sorting method.

In BVT, consider the mean value difference between positive and negative examples distributing on some level axis. Intuitively, larger difference indicates higher possibility for the positive and negative examples to push each other to $\mathcal{EI}$ at both sides of that level. Instead of sorting weight, we can therefore sort the difference.

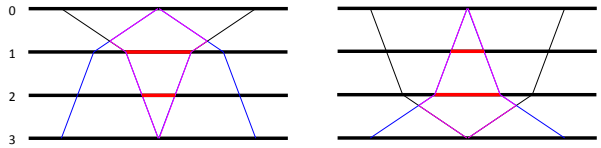

Figure 5. Base learners in descending (left) and ascending(right) order. Only the leftmost and rightmost paths are plotted for BVT and DBVT. $\mathcal{PI}$ in red and $\mathcal{EI}$ in black. Border paths for $\mathcal{PI}$ are in magenta.

Based on above observation, (Bourdev & Brandt, 2005) proposed that the order be determined one by one in a T rounds algorithm. At first round, among the T base learners the best one leading to the largest mean value difference is selected for the level one; At second round, among remaining $T-1$ base learners the best one is selected for the level two, and so on.

Obviously, such a top-down single-pass reordering involves $T + (T-1) + \ldots + 1$ computations. Thus the complexity of the algorithm is $O(T^2)$.

4.1. Expected Path Length as a Measure for Evaluation Time

The above two methods are intuitive in that they try not to explicitly minimize a target function measuring the evaluation time. In this subsection we define such a quantitative measure.

Definition 7 (***Path Length***)**.** The number of non-terminal nodes on a path.

Since computing a base learner incurs unit 1 time cost, the path length measures the time cost of traversing BVT or BDD for a single example. For a set of examples, however, we need the following "average" measure:

Definition 8 (***Expected Path Length***)**.** The expected path length can be defined either path wise:

$$EPL = \textstyle\sum_i p_i l_i, \tag{9}$$

where l_i is the path length and p_i is the probability traversing that path, or be defined level wise:

$$EPL = \textstyle\sum_{t=1}^{T} P_t, \tag{10}$$

where P_t is the portion of examples falling in $\mathcal{PI}$ at t-th level (or falling in redundant BDD nodes).

Now we can explain our technique that minimizes EPL by base learner reordering.

4.2. SIFTING

To reorder base learners, in this paper we propose the use of the so-called SIFTING algorithm (Rudell, 1993; Ebendt et al., 2005), which searches the optimal ordering by repeatedly swapping two adjacent base learners as in following pseudo codes.

Algorithm 1 SIFTING.

Input: An arbitrary base learner order; Maximum number of rounds M.
for $round = 1$ **to** M **do**
  for each base learner **do**
    Swap it with its successive base learner on and on until it sinks to the bottom; then
    Swap it with previous base learner on and on until it floats to the top.
    During the course, the best level, which leads to the smallest EPL, is recorded.
    The base learner being inspected (now at the top level) is then moved to the best level we just found.
  end for
end for

Time Complexity Analysis

Theorem 1 (Square Time Complexity)**.** *The time complexity for SIFTING is* $O(T^2)$.

Proof. In one round, there are T inner loops (for swapping) in each of the T outer loops (for every base learner). The square time complexity assertion holds by noting that swapping base learner is $O(1)$, as in following lemma. □

Lemma 2 (Localized Swapping)**.** *The time complexity for swapping two adjacent base learners is* $O(1)$.

Proof. Let $1 \leq i \leq T-1$. In BVT or BDD, after swapping base learner i with $i+1$, $acc(t)$ and $dacc(t)$ by definition remains for $t \neq i$, while only $acc(i)$ and $dacc(i)$ change. Therefore, only the topology at level i (*i.e.*, the nodes and nodes' incoming/outgoing edges) needs updating, since the topology is absolutely determined by $accv()$ and $dacc()$). That is to say, in the summation in EPL (10), only the portion P_i at level i needs recalculation while other portion P_t stays unchanged for $t \neq i$. □

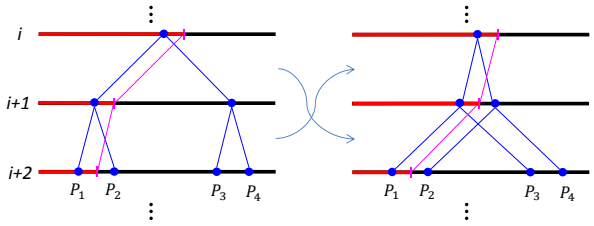

Figure 6. A snippet of BVT before (left) and after (right) swapping two adjacent base learners. $\mathcal{PI}$ in red, border paths in magenta.

Why SIFTING

We provide intuitive for why we prefer SIFTING to Sorting or Top-Down Reordering.

1) Sorting in descending order maybe sub-optimal for some data distribution, as the example in Figure 6. On the left of Figure 6 are three levels for two base learners in some BVT. The base learners are in descending weight order. For the 4 nodes at level $i+2$, the portion of training examples they carry are denoted by p_1 to p_4. After swapping (as shown on the right of the Figure 6), level i and level $i+2$ keep unchanged, while only level $i+1$ needs updating (See proof of Lemma 2). For the node in $\mathcal{PI}$ at level $i+1$, the portion of examples it carries changes from p_1+p_2 to p_1+p_3. Recalling the definition (10), sorting in ascending order even helps to reduce EPL in this case provided $p_3 < p_2$.

2) Top-Down Reordering is essentially a single-pass greedy algorithm. SIFTING may find a better result than it by finely tuning the order, with feasible time complexity.

5. Related Work

5.1. BDD and MDP

(Busa-Fekete et al., 2012) models the Decision Function by Markov Decision Process (MDP) that is represented by a Directed Acyclic Graph (DAG) with precisely T nodes for each of the T base learners. When calculating, the node is considered sequentially following the natural order, *i.e.*, the as-is order in Boosting. At each node, one of three possible Actions, *i.e.*, {Go To Next Node, Quit, Skip}, is taken based on the States consisting of current $acc()$ value and the index of current base learner.

In regards of internal structure, the DAG for MDP is actually equivalent to BDD we explain in section 2.4. "Expanding" the t-th node in DAG yields the t-th level of a BDD. In another words, the BDD nodes at t-th

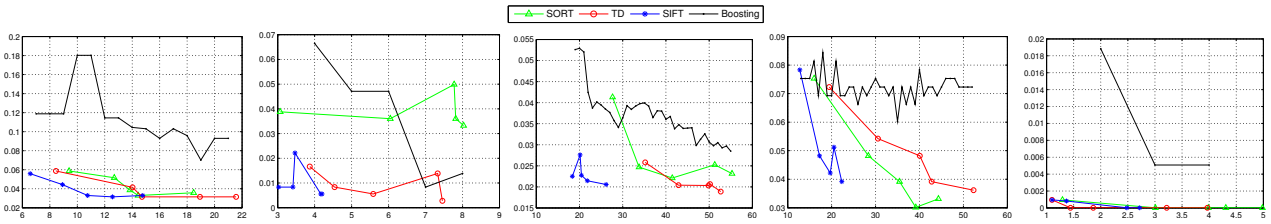

Figure 7. EPL (horizontal axis) v.s. test error (vertical axis) curve, for datasets #2, #3, #6, #8, #10.

level can be seen as the States in MDP. Therefore, taking actions in MDP is to traverse different types of nodes in BDD (*i.e.*, "Skip": traverse a redundant BDD node; "Quit": traverse a BDD node in $\mathcal{EI}$; "Go To Next Node": traverse an ordinary BDD node).

The State-to-Action mapping is learned from a set of training data by trading off error rate and evaluation time via Reinforcement Learning framework in (Busa-Fekete et al., 2012). In contrast, the BDD nodes connection in section 2.4 is derived from BVT, which precisely captures all possible inputs to Decision Function (1).

5.2. BDD and Tree

In view of Boolean Expression, the Decision Function (1) can be represented by either BDD or Tree. In BDD, all paths share a common order of base learner, while in Tree each path can have its own order. Therefore, it is very difficult to minimize EPL for a tree by reordering the base learner.

(Kim et al., 2012) proposed that one simply grows a new tree. The outputs of base learners on training data (including the synthesized data in order not to alter the decision function too much) are seen as new feature vectors and are thus fed into some top-down tree growing procedure. Information gain is used as split criterion. Actually, (Goodman & Smyth, 1988) pointed out that this method is a type of Shannon-Fano coding attempting a minimum average code length, which by definition is precisely the EPL (10) discussed in this paper.

There is other work utilizing tree structure where each node associates a linear combination of the base learners, including (Grossmann, 2004; Tu, 2005; Xu et al., 2013).

5.3. Others

In supplement, we discuss the problem of imbalanced class prior, which is common in object detection in Computer Vision. Also, we briefly review the related work from community other than Machine Learning.

6. Experiments

To achieve small EPL, we compare three reordering methods: 1) SORT(weight sorting, Sochman & Matas (2005); Kim et al. (2012)); 2) TD(top down Reorder, Bourdev & Brandt (2005)); 3) SIFT(SIFTING, advocated in this paper), for either exact or approximate representation of Boosting's decision function.

6.1. Setup

We carry out experiments on the following data sets:

#1:	optdigits_bin05	#2:	pendigits80
#3:	optdigits17	#4:	pendigits49
#5:	mnist10k06	#6:	mnist10k37
#7:	mnist10k49	#8:	zipcode38
#9:	covertype145k06	#10:	poker25kT109

They are adopted from UCI and MNIST, see supplement for details.

LogitBoost (Friedman et al., 2000) with decision stump is used to generate the binary valued base learners (2) and the Decision Function (1). We run Boosting for 1000 iterations and combine those duplicate decision stumps. Therefore, the number of unique base learners $T < T'$.

SORT and TD have no tuning parameters. For SIFT, up to $M = 4$ rounds are tried on training data. During the course, we stop it whenever the EPL increases after some round.

6.2. Results and Analysis

An exact representation precisely reproduces Boosting's classification boundary and thus has the same testing error with Boosting. To tell whether a reordering method is good in this case, we can simply check the EPL. In Table 1 we show the results for an exact representation–FastExit, using three reordering methods. We also include as baseline the results of Boosting, where EPL is the number of its unique base learners T.

Table 1. Results for FastExit with natural order and three reordering methods: SORT, TD and SIFTING. Results for original Boosting, as baseline, is shown at first row. At each entry, the first cell is EPL and the second is error rate, both in testing stage. Lower EPL in **boldface**.

	dataset#1		dataset#2		dataset#3		dataset#4		dataset#5		dataset#6		dataset#7		dataset#8		dataset#9		dataset#10	
Boosting	26	0.83%	136	3.29%	54	0.28%	87	0.29%	220	1.32%	314	1.9%	518	4.02%	256	3.31%	475	4.68%	22	0.00%
fastexit	11.54	0.83%	72.79	3.29%	27.01	0.28%	40.37	0.29%	138.88	1.32%	222.41	1.9%	409.09	4.02%	179.03	3.31%	409.30	4.68%	6.68	0.00%
fastexit-SORT	9.37	0.83%	54.74	3.29%	23.14	0.28%	32.15	0.29%	119.86	1.32%	200.83	1.9%	370.52	4.02%	163.52	3.31%	311.35	4.68%	4.98	0.00%
fastexit-TD	9.56	0.83%	56.27	3.29%	24.31	0.28%	34.73	0.29%	130.60	1.32%	237.48	1.9%	425.62	4.02%	172.82	3.31%	449.77	4.68%	5.18	0.00%
fastexit-SIFT	**9.07**	0.83%	**50.86**	3.29%	**22.25**	0.28%	**31.00**	0.29%	**115.40**	1.32%	**190.16**	1.9%	**349.95**	4.02%	**159.30**	3.31%	**223.02**	4.68%	**4.59**	0.00%

Table 2. Results for FastExit and BDD with reordering by SIFTING. At each entry, the first cell is EPL and the second is error rate, both in testing stage.

	dataset#1		dataset#2		dataset#3		dataset#4		dataset#7		dataset#10	
Boost	23.00	1.11%	110.00	3.43%	64.00	0.28%	86.00	0.29%	92.00	4.9%	6.48	0.00%
fastexit	12.34	1.11%	63.65	3.43%	32.53	0.28%	47.45	0.29%	69.64	4.9%	19.00	0.00%
BDD	12.28	1.11%	63.65	3.43%	32.51	0.28%	46.48	0.29%	62.23	4.93%	5.13	0.00%
fastexit-SIFT	8.67	1.11%	35.96	3.43%	20.20	0.28%	24.64	0.29%	57.96	5.07%	4.59	0.00%
BDD-SIFT	8.58	1.11%	36.06	3.43%	20.44	0.28%	25.19	0.29%	58.02	4.93%	4.67	0.00%

Given a fixed T' for the approximate representation DBPSYN, we find none of the three reordering methods is able to have both lower EPL and lower test error. Therefore, for each reordering method we vary the T' to generate a sequence of (EPL, test error) pairs and connect those points as a curve (*i.e.*, the Pareto front), as is shown in Figure 7 (More results in supplement). The closer to bottom left corner the curve, the better the reordering method.[2]

As can be seen in Table 1 and Figure 7, all three reordering methods yield improved EPL. Besides, we find that:

1) SIFT > SORT > TD on a majority of the datasets for both exact and approximate representation, where > means outperform. That is to say, SIFTING is always a preferred reordering method.

2) Approximate methods lead to significantly smaller EPL than exact methods. However, testing error rate for approximate methods is usually greater. The DBPSYN seems a good trade off.

We also compare two exact representations, *i.e.*, FastExit and BDD, with reordering by SIFTING. Basically, in BDD we need store too many nodes which is beyond our computational resources. So we implement a fixed small step LogitBoost. After training LogitBoost and combining the duplicate decision stumps, the outputs of base learners can thus be quantized to integers. This largely reduces the space complexity for BDD. Even so, it is still space demanding and the base learner swapping is very slow. In Table 2, we report the results for part of the datasets.

From Table 2, it seems that BDD shows little improvements over its close relative FastExit. We believe it's due to the fact that in BDD the redundant nodes within $\mathcal{PI}$ are actually very rare. Recalling the connection between BDD and the work in (Busa-Fekete et al., 2012) (section 5.1), we thus cautiously suggest that the Action "Skip" in MDP (Busa-Fekete et al., 2012) be not necessary provided a proper base learner ordering be found (In (Busa-Fekete et al., 2012) the natural order is taken and no reordering is attempted).

7. Conclusion and Future Work

We discuss the representation of decision function in Boosting. To reduce the evaluation time of decision function, we propose a novel method for base learner reordering. In the future, we will take into account feature cost, general base learner other than decision stump and multi-class classification.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grants 61225008, 61020106004, 61021063, and by the Ministry of Education of China under Grant 20120002110033. Thanks Scott Sanner for helpful discussion and pointing to us the important role of data structure in machine learning. Thanks Tengyu Sun, Han Hu, Jianjiang Feng for reading the manual script. Thanks all anonymous reviewers' comments that significantly improve this paper's quality. Thanks Minmin Chen for clarifying discussion.

[2]Comparison using Pareto front by varing T' is suggested by a reviewer.

References

Behle, Markus. *Binary decision diagrams and integer programming*. PhD thesis, der Universität des Saarlandes, 2010.

Bourdev, Lubomir and Brandt, Jonathan. Robust object detection via soft cascade. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pp. 236–243. IEEE, 2005.

Bryant, Randal E. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

Busa-Fekete, R, Benbouzid, D, Kégl, Balázs, et al. Fast classification using sparse decision dags. In *29th International Conference on Machine Learning (ICML)*, 2012.

Chen, Minmin, Xu, Zhixiang, Weinberger, Kilian Q, Chapelle, Olivier, Kedem, Dor, and Saint Louis, MO. Classifier cascade for minimizing feature evaluation cost. In *AISTATS*, volume 15, pp. 218–226, 2012.

Ebendt, Rudiger, Fey, Görschwin, and Drechsler, Rolf. *Advanced BDD optimization*. Springer, 2005.

Friedman, Jerome, Hastie, Trevor, and Tibshirani, Robert. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

Goodman, Rodney M. and Smyth, Padhraic. Decision tree design from a communication theory standpoint. *Information Theory, IEEE Transactions on*, 34(5):979–994, 1988.

Grossmann, E. Adatree: Boosting a weak classifier into a decision tree. In *Computer Vision and Pattern Recognition Workshop, 2004. CVPRW '04. Conference on*, pp. 105–105, 2004. doi: 10.1109/CVPR.2004.22.

Grubb, Alexander and Bagnell, J Andrew. Speedboost: Anytime prediction with uniform near optimality. In *AISTATS*, volume 15, pp. 458–466, 2012.

Kim, Tae-Kyun, Budvytis, Ignas, and Cipolla, Roberto. Making a shallow network deep: Conversion of a boosting classifier into a decision tree by boolean optimisation. *International Journal of Computer Vision*, pp. 1–13, 2012.

Mayer-Eichberger, Valentin Christian Johannes Kaspar. *Towards solving a system of pseudo boolean constraints with binary decision diagrams*. PhD thesis, Universidade Nova de Lisboa, 2008.

Reyzin, Lev. Boosting on a budget: Sampling for feature-efficient prediction. In *Proceedings of the 28th Internation Conference on Machine Learning (ICML)*, 2011.

Rudell, Richard. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pp. 42–47. IEEE Computer Society Press, 1993.

Saberian, Mohammad J and Vasconcelos, Nuno. Boosting classifier cascades. In *Proceedings of the 24th Annual Conference on Neural Information Processing Systems (NIPS)*, 2010.

Sanner, Scott. Decision diagrams in automated planning and scheduling. Tutorial at ICAPS2011. http://users.cecs.anu.edu.au/~ssanner/Papers/Decision_Diagrams_Tutorial.pdf, 2011.

Sochman, Jan and Matas, Jiri. Waldboost-learning for time constrained sequential detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pp. 150–156. IEEE, 2005.

Tu, Zhuowen. Probabilistic boosting-tree: learning discriminative models for classification, recognition, and clustering. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 2, pp. 1589–1596 Vol. 2, 2005. doi: 10.1109/ICCV.2005.194.

Viola, Paul and Jones, Michael J. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.

Xu, Zhixiang, Kusner, Matt, Weinberger, Kilian, and Chen, Minmin. Cost-sensitive tree of classifiers. In *30th International Conference on Machine Learning (ICML)*, 2013.

Zhang, Cha and Viola, Paul. Multiple-instance pruning for learning efficient cascade detectors. In *Proceedings of the 21th Annual Conference on Neural Information Processing Systems (NIPS)*, 2007.