Mach Learn (2014) 97:295–326
DOI 10.1007/s10994-014-5434-3

An improved multiclass LogitBoost using adaptive-one-vs-one

Peng Sun · Mark D. Reid · Jie Zhou

Received: 25 April 2013 / Accepted: 15 January 2014 / Published online: 28 February 2014

Abstract LogitBoost is a popular Boosting variant that can be applied to either binary or multi-class classification. From a statistical viewpoint LogitBoost can be seen as additive tree regression by minimizing the Logistic loss. Following this setting, it is still non-trivial to devise a sound multi-class LogitBoost compared with to devise its binary counterpart. The difficulties are due to two important factors arising in multiclass Logistic loss. The first is the invariant property implied by the Logistic loss, causing the optimal classifier output being not unique, i.e. adding a constant to each component of the output vector won't change the loss value. The second is the density of the Hessian matrices that arise when computing tree node split gain and node value fittings. Oversimplification of this learning problem can lead to degraded performance. For example, the original LogitBoost algorithm is outperformed by ABC-LogitBoost thanks to the latter's more careful treatment of the above two factors. In this paper we propose new techniques to address the two main difficulties in multiclass LogitBoost setting: (1) we adopt a vector tree model (i.e. each node value is vector) where the unique classifier output is guaranteed by adding a sum-to-zero constraint, and (2) we use an adaptive block coordinate descent that exploits the dense Hessian when computing tree split gain and node values. Higher classification accuracy and faster convergence rates are observed for a range of public data sets when compared to both the original and the ABC-LogitBoost implementations. We also discuss another possibility to cope with LogitBoost's dense Hessian matrix. We derive a loss similar to the multi-class Logistic loss but which

Editor: Shai Shalev-Shwartz.

P. Sun (✉) · J. Zhou
Tsinghua National Laboratory for Information Science and Technology(TNList),
Department of Automation, Tsinghua University, Beijing100084, China
e-mail: sunp08@mails.tsinghua.edu.cn

J. Zhou
e-mail: jzhou@tsinghua.edu.cn

M. D. Reid
Research School of Computer Science,
The Australian National University and NICTA, Canberra, ACT, Australia
e-mail: mark.reid@anu.edu.au

guarantees a diagonal Hessian matrix. While this makes the optimization (by Newton descent) easier we unfortunately observe degraded performance for this modification. We argue that working with the dense Hessian is likely unavoidable, therefore making techniques like those proposed in this paper necessary for efficient implementations.

Keywords LogitBoost · Boosting · Ensemble · Supervised learning · Convex optimization

1 Introduction

Boosting is a successful technique for training classifier for both binary and multi-class classification (Freund and Schapire 1995; Schapire and Singer 1999). In this paper, our focus is on multiclass LogitBoost (Friedman et al. 1998), one of the popular boosting variants. Originally, LogitBoost was motivated by a statistical perspective (Friedman et al. 1998), where boosting algorithm consists of three key components: the loss, the function model, and the optimization algorithm. In the case of LogitBoost, these are the multi-class Logistic loss, the use of additive tree models, and a stage-wise optimization, respectively. For K-nary classification, LogitBoost directly learns a K dimensional vector as the classifier output, each component representing the confidence of predicting the corresponding class.

There are two important factors in LogitBoost's settings. Firstly, an "invariant property" is implied by the Logistic loss, i.e., adding a constant to each component of the classifier output won't change the loss value. Therefore, the classifier output minimizing the total loss is not unique, making the optimization procedure a bit difficult. Secondly, the Logistic loss produces a dense Hessian matrix, causing troubles when deriving the tree node split gain and node value fitting. It is challenging to design a tractable optimization algorithm that fully handles both these factors. Consequently, some simplification and/or approximation is needed.

In Friedman et al. (1998) the Hessian is diagonally approximated. In this way, the minimizer becomes unique, while the optimization—essentially a quadratic problem when using one step Newton—is substantially simplified. Consequently, at each Boosting iteration the tree model updating collapses to K independent Weighted Regression Tree fittings, each tree outputting a scalar.

Unfortunately, Friedman's prescription turns out to have some drawbacks. The over simplified quadratic loss even doesn't satisfy the invariant property, and is thus a very crude approximation. A later improvement, ABC-LogitBoost, is shown to outperform LogitBoost in terms of both classification accuracy and convergence rate (Li 2008, 2010b). This is due to ABC-LogitBoost's careful handling of the above key problems of the LogitBoost setting. The invariant property is addressed by adding a sum-to-zero constraint to the output vector. To make the minimizer unique, at each iteration one variable is eliminated while the other $(K-1)$ are free, so that the loss is re-written with $(K-1)$ variables. At this point, the diagonal Hessian approximation is, again, adopted to permit $K-1$ scalar trees being independently fitted for each of the $K-1$ classes. The remaining class—called the base class—is recovered by taking minus the sum of the other $K-1$ classes. The base class—i.e., the variable to be eliminated—is selected adaptively per iteration (or every several iterations), hence the acronym ABC (Adaptive Base Class). Note the diagonal Hessian approximation in ABC-LogitBoost is taken for the $(K-1)$ dimensional problem yielded by eliminating a redundant variable. Li (2008, 2010b) considers it a more refined approximation than that of original LogitBoost (Friedman et al. 1998).

In this paper, we propose two novel techniques to address the challenging aspects of the LogitBoost setting. In our approach, one vector tree is added per iteration. To make the minimizer unique, we allow a K dimensional sum-to-zero vector to be fitted for each tree

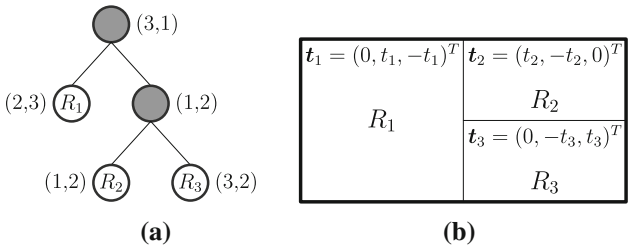


Fig. 1 A newly added tree at some boosting iteration for a 3-class problem. **a** A class pair (shown in *brackets*) is selected for each tree node. For each internal node (*filled*), the pair is for computing split gain; For terminal nodes (*unfilled*), it is for node vector updating. **b** The feature space (*the outer black box*) is partitioned by the tree in (**a**) into regions $\{R_1, R_2, R_3\}$. On each region only two coordinates are updated based on the corresponding class pair shown in (**a**)

node. This permits us to explicitly formulate the computation for both node split gain and node value fitting as a K dimensional constrained quadratic optimization, which arises as a subproblem in the inner loop for split seeking when fitting a new tree. To avoid the difficulty of a dense Hessian, we propose that for **each** of these subproblems, only two coordinates (i.e., two classes or a class pair) are adaptively selected for updating, hence we call the modified algorithm AOSO-LogitBoost (Adaptive One vS One). Figure 1 gives an overview of our approach. In Sect 4.4 we show that first order and second order approximation of loss reduction can be a good measure for the quality of selected class pair.

Following the above formulation, ABC-LogitBoost (derived using a somewhat different framework than Li 2010b) can thus be shown to be a special case of AOSO-LogitBoost with a less flexible tree model. In Sect. 5 we compare the differences between the two approaches in detail and provide some intuition for AOSO's improvement over ABC.

Both ABC or AOSO are carefully devised to address the difficulties of dense Hessian matrix arising in Logistic loss. In other words, the tree model learning would be easy if we were encountering diagonal Hessian matrix. Based on loss design in Primal/Dual view (Masnadi-Shirazi and Vasconcelos 2010; Reid and Williamson 2010), we show that the dense Hessian matrix essentially results from the sum-to-one constraint of class probability in Logistic loss. We thus investigate the possibility of diagonal Hessian matrix by removing such a sum-to-one constraint. The LogitBoost variant we produce does work. However, it is still inferior to AOSO LogitBoost (see Sect. 6 for detailed discussion). We therefore conclude that modifications of the original LogitBoost, such as AOSO and ABC, are necessary for efficient model learning since the dense Hessian matrix seems unavoidable.

The rest of this paper is organised as follows: In Sect. 2 we first formulate the problem setting for LogitBoost. In Sect. 3 we briefly review/discuss original/ABC-/AOSO- LogitBoost in regards of the interplay between the tree model and the optimisation procedure. In Sect. 4 we give the details of our approach. In Sect. 5 we compare our approach to (ABC-)LogitBoost. In Sect. 6 we show how to design a loss that produces diagonal Hessian matrices and assess its implications for model accuracy. In Sect. 7, experimental results in terms of classification errors and convergence rates are reported on a range of public datasets.

2 The problem setup

For K-class classification ($K \geq 2$), consider an N example training set $\{\boldsymbol{x}_i, y_i\}_{i=1}^N$ where $\boldsymbol{x}_i$ denotes a feature value and $y_i \in \{1, \ldots, K\}$ denotes a class label. From the training

set a prediction function $\boldsymbol{F}(\boldsymbol{x}) \in \mathbb{R}^K$ is learned. When clear from context, we omit the dependence on $\boldsymbol{x}$ and simply denote $\boldsymbol{F}(\boldsymbol{x})$ by $\boldsymbol{F}$ (We will do the same to other related variables.). Given a test example with known $\boldsymbol{x}$ and unknown y, we predict a class label by taking $\hat{y} = \arg\max_k F_k, k = 1, \ldots, K$, where F_k is the k-th component of $\boldsymbol{F}$.

The function $\boldsymbol{F}$ is obtained by minimizing a target function on training data:

$$\text{Loss} = \sum_{i=1}^{N} L(y_i, \boldsymbol{F}_i). \tag{1}$$

Each $L(y, \boldsymbol{F})$ term is a *loss* for a single training example $(\boldsymbol{x}, y)$. We will make the loss concrete shortly.

To make the optimization of (1) feasible, a *function model* is needed to describe how $\boldsymbol{F}$ depends on $\boldsymbol{x}$. For example, linear model $\boldsymbol{F} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$ is used in traditional Logistic regression, while Generalized Additive Model is adopted in LogitBoost where

$$\boldsymbol{F}(\boldsymbol{x}) = \sum_{m=1}^{M} \boldsymbol{f}^{(m)}(\boldsymbol{x}), \tag{2}$$

where $\boldsymbol{f}^{(m)}(\boldsymbol{x}) \in \mathbb{R}^K$ is further expressed by tree(s), as will be seen shortly.

Each $\boldsymbol{f}^{(m)}(\boldsymbol{x})$, or simply $\boldsymbol{f}$, is learned by the so-called greedy stage-wise *optimization*. That is, at each Boosting iteration $\boldsymbol{f}^{(m)}$ is added only based on $\boldsymbol{F} = \sum_{j=1}^{m-1} \boldsymbol{f}^{(j)}$.

In summary, the learning procedure, called training, consists of three ingredients: the *loss*, the *function model* and the *optimization* algorithm. In what follows we discuss them in greater details.

2.1 The logistic loss

LogitBoost adopts the Logistic loss (Friedman et al. 1998), which is an implicit function of y and $\boldsymbol{F}$:

$$L(y, \boldsymbol{F}) = -\sum_{k=1}^{K} r_k \log(p_k), \tag{3}$$

where $r_k = 1$ if $k = y$ and 0 otherwise, $\{p_k\}_{k=1}^K$ is connected to $\boldsymbol{F}$ via the so-called Link function:

$$p_k = \frac{\exp(F_k)}{\sum_{j=1}^{K} \exp(F_j)}. \tag{4}$$

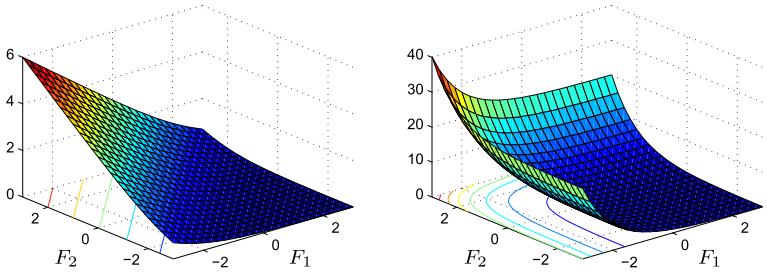
Therefore, $\boldsymbol{p} = (p_1, \ldots, p_K)^T$ can be seen as probability estimate since $p_k \geqslant 0$ and $\sum_{k=1}^K p_k = 1$.

2.1.1 The invariant property

It is easy to verify that an "invariant property" holds for Logistic loss:

$$L(y, \boldsymbol{F}) = L(y, \boldsymbol{F} + c\mathbf{1}), \tag{5}$$

where c is an arbitrary constant, $\mathbf{1}$ is K-dimensional all 1 vector. That is to say, the Logistic loss won't change its value when moving $\boldsymbol{F}$ along a all-1 vector $\mathbf{1}$, which implies that $\boldsymbol{F}$ is "equivalent to" $\boldsymbol{F} + c\mathbf{1}$ because they both lead to the same loss (see Fig. 2b). Such an invariant property is, in turn, consistent with the multi-class prediction rule $\arg\max_k F_k, k = 1, \ldots, K$ which says adding a constant to each of F_k does not alter the prediction.



(a) Logistic Loss **(b)** Exponential Loss

Fig. 2 Multiclass loss $L(y = 1, \boldsymbol{F})$ for K-nary problem. In this figure $K = 2$, thus $\boldsymbol{F} \in \mathbb{R}^2$. The Logistic loss is defined in (3); The Exponential loss is defined in (10). Note the parallel contour plots in the case of Logistic loss

2.1.2 Derivative and quadratic approximation

It is difficult to directly optimize the Logistic loss. To permit numeric method, we need the first order and second order derivatives of example wise loss (3). Simple matrix calculus shows that the gradient $\boldsymbol{g} \in \mathbb{R}^K$ and Hessian $\boldsymbol{H} \in \mathbb{R}^{K \times K}$ w.r.t. $\boldsymbol{F}$ are:

$$\boldsymbol{g} = -(\boldsymbol{r} - \boldsymbol{p}), \tag{6}$$

$$\boldsymbol{H} = \hat{\boldsymbol{P}} - \boldsymbol{p}\boldsymbol{p}^T, \tag{7}$$

where the response $\boldsymbol{r} = (r_1, \ldots, r_K)^T$ and probability estimate $\boldsymbol{p} = (p_1, \ldots, p_K)^T$ are as defined before, the diagonal matrix $\hat{\boldsymbol{P}} = \text{diag}(p_1, \ldots, p_K)$.

With $\boldsymbol{g}$ and $\boldsymbol{H}$ given above, we can write down the Taylor expansion of (3) at $\boldsymbol{F}$ up to second order:

$$q(\boldsymbol{f}|\boldsymbol{F}) = L(y, \boldsymbol{F}) + \boldsymbol{g}^T \boldsymbol{f} + \frac{1}{2} \boldsymbol{f}^T \boldsymbol{H} \boldsymbol{f}, \tag{8}$$

which locally approximates $L(y, \boldsymbol{F} + \boldsymbol{f})$ in quadratic sense. Note that in (8) $\boldsymbol{f}$ is the variable, while the "constants" $\boldsymbol{g}$ and $\boldsymbol{H}$ depend on $\boldsymbol{F}$. When clear from context, we also omit the dependence on $\boldsymbol{F}$ and simply write $q(\boldsymbol{f}|\boldsymbol{F})$ as $q(\boldsymbol{f})$.

It is easy to verify that the invariant property carries over to $q(\boldsymbol{f})$:

$$q(\boldsymbol{f}) = q(\boldsymbol{f} + c\mathbf{1}) \tag{9}$$

by noting $\mathbf{1}^T \boldsymbol{g} = 0$ and $\mathbf{1}^T \boldsymbol{H} = \boldsymbol{H}\mathbf{1} = 0$.

2.1.3 $(K - 1)$ Degrees of freedom and the sum-to-zero constraint

Due to the invariant property, the minimizer $\boldsymbol{f}^*$ of (8) is not unique since any $\boldsymbol{f}^* = c\mathbf{1}$ would be a minimizer. To pin-down the value, we can add a constraint $\mathbf{1}^T \boldsymbol{f} = 0$, which in effect restricts $\boldsymbol{f}$ to vary just in the linear subspace defined by $\mathbf{1}^T \boldsymbol{f} = 0$. Obviously, now we need only $K - 1$ coordinates to express the vector living in the subspace, i.e., the degrees of freedom is $K - 1$.

In Sect. 4.2 we will discuss the rank of the Hessian matrix $\boldsymbol{H}$, which provides another perspective to why the minimizer of (8) is not unique.

Conceptually, the invariant property is primary, causing the minimizer being not unique; The sum-to-zero constraint is secondary, serving as a mathematical tool to make the minimizer unique.

2.1.4 More on $(K-1)$ degrees of freedom

We don't claim that the invariant property be a necessity. In the literature, there exists other multiclass loss not satisfying this condition. In this case, it doesn't need a sum-to-zero constraint and $\boldsymbol{f}$ has K degrees of freedom. For instance, consider the K-nary exponential loss for AdaBoost.MH:

$$L(y, \boldsymbol{F}) = \sum_{k=1}^{K} exp(-y_k^* \boldsymbol{F}_k), \tag{10}$$

where $y_k^* = +1$ if $y_k = k$ and -1 otherwise. See Fig. 2(b) for its graph. In general, $L(y, \boldsymbol{F}) \neq L(y, \boldsymbol{F} + c\mathbf{1})$ for Exponential loss.

It is out of this paper's scope to compare Exponential loss and Logistic loss in multiclass case. Instead, we focus on the Logistic loss and show how it is applied in LogitBoost. Also, in Sect. 6 we discuss a modified Logistic loss not satisfying the invariant property (but simplifying the Hessian and easing the quadratic solver) and show its degraded performance when comparing with original Logistic loss.

2.2 The tree model

As mentioned previously, $\boldsymbol{F}(\boldsymbol{x}) = \sum_{m=1}^{M} \boldsymbol{f}^{(m)}(\boldsymbol{x})$ is additive tree model. However, the way each $\boldsymbol{f}(\boldsymbol{x})$ (we have omitted the subscript m for simplicity and without confusion) being expressed by tree model is not unique.

In this paper, we adopt a single vector-valued tree. Further, we let it be a binary tree (i.e., only binary splits on internal node are allowed) with splits that are vertical to coordinate axis, as in Friedman et al. (1998) and Li (2010b). Formally,

$$\boldsymbol{f}(\boldsymbol{x}) = \sum_{j=1}^{J} \boldsymbol{t}_j I(\boldsymbol{x} \in R_j) \tag{11}$$

where $\{R_j\}_{j=1}^{J}$ are a rectangular partition of the feature space, $I(P)$ is the indicator function which is 1 when the predicate P is true and 0 otherwise, and each $\boldsymbol{t}_j \in \mathbb{R}^K$ is the node value associated with R_j. See Fig. 1 for an illustration. The vector-valued tree model is also adopted in other Boosting implementation, say, Real AdaBoost.MH described in Kégl and Busa-Fekete (2009).

$\boldsymbol{f}(\boldsymbol{x})$ can be also represented by K scalar regression trees as in the original LogitBoost of Friedman et al. (1998) and the Real AdaBoost.MH implementation of Friedman et al. (1998):

$$\boldsymbol{f}(\boldsymbol{x}) = \sum_{k=1}^{K} \sum_{j=1}^{J} t_{k,j} I(\boldsymbol{x} \in R_{k,j}), \tag{12}$$

where $t_{k,j}$ is a scalar and each tree can have its own partition $\{R_{k,j}\}_{j=1}^{J}$.

Finally, it is possible to express $fv(\boldsymbol{x})$ with just $K-1$ trees by adding to $\boldsymbol{f}(\boldsymbol{x})$ a sum-to-zero constraint, as is adopted in Li's series work (Li 2008, 2009, 2010b).

2.3 Stage wise optimization

Formally, using $\boldsymbol{F}_i$ and $\boldsymbol{f}_i$ as shorthand for $\boldsymbol{F}(\boldsymbol{x}_i)$ and $\boldsymbol{f}(\boldsymbol{x}_i)$ respectively, the stage wise optimization is:

$$\boldsymbol{f}^{(m)}(\boldsymbol{x}) = \arg\min_{\boldsymbol{f}(\boldsymbol{x})} \sum_{i=1}^{N} L(y_i, \boldsymbol{F}_i + \boldsymbol{f}_i), \tag{13}$$

where $\boldsymbol{f}(\boldsymbol{x})$ is expressed by tree model as explained in last subsection. This procedure repeats M times with initial condition $F = 0$. Owing to its iterative nature, we only need to know how to solve (13) in order to implement the optimization.

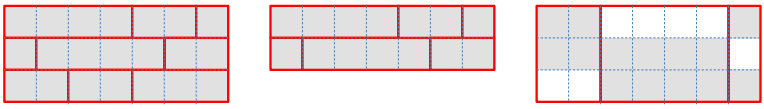
In (13) it is difficult to directly work on the Logistic loss; numeric method must be relied on. Friedman et al. (1998) suggests to use one step Newton, i.e., in (13) replace example wise Logistic loss $L()$ with its corresponding second order Taylor expansion:

$$\boldsymbol{f}^{(m)}(\boldsymbol{x}) = \arg\min_{\boldsymbol{f}(\boldsymbol{x})} \sum_{i=1}^{N} q(\boldsymbol{f}_i | \boldsymbol{F}_i). \tag{14}$$

3 Interplay between tree model and optimization

In last section we have reviewed the problem setup. In particular, we made concrete the three ingredients in LogitBoost: the loss, the function model and the optimization algorithm are the Logistic loss, the additive tree model and the stage wise optimization, respectively. To train a LogitBoost classifier, now the only thing left unexplained is how to optimize the quadratic optimization (14), which is still a bit complicated.

Basically, $\boldsymbol{f}(\boldsymbol{x})$ on a training set $\{\boldsymbol{x}_i, y_i\}_{i=1}^{N}$ can be seen as an $K \times N$ matrix, as in Fig. 3, where the i-th column represents the K values of $\boldsymbol{f}(\boldsymbol{x}_i)$. However, the values in the matrix can not vary independently. Instead, the values must be subject to a tree model, which determines a type of partition on the matrix such that the elements falling into the same cell take a common, unique value. Thus the tree model has an impact on the optimization procedure. Things become even subtle when considering the invariant property of the Logistic loss. The original/ABC-/AOSO-LogitBoost makes different choices to address this tree model-optimization interplay, as will be briefly discussed in the rest of this section. The details are deferred to next section.



(a) original LogitBoost **(b)** ABC-LogitBoost **(c)** AOSO-LogitBoost

Fig. 3 Viewing $\boldsymbol{f}(\boldsymbol{x})$ a $K \times N$ matrix on a training dataset. A J-leaf tree corresponds a J-cell partition on the matrix such that the values falling into the same cell must have a common value. In this figure, $K = 3$, $N = 7$ and $J = 3$. The *squares* in *dashed lines* denote the matrix elements, while the *solid lines* denote the partitions. Note that each cell needs not be continuous, although we intentionally plot it a continuous one for illustrative purpose. **a** K scalar trees. Each tree fits a class (a row) and has its own partition. **b** $(K - 1)$ scalar trees. Each tree fits a class (a row) and has its own partition. The values on the remaining class (i.e. the base class) is recovered by the other classes. **c** only one vector-valued tree. Any column belongs to only one cell. At each cell, only two classes (two rows) can vary (*shaded squares*) and the others keep zero values (*blank squares*)

3.1 Original LogitBoost

To solve problem (14), LogitBoost (Friedman et al. 1998) takes further simplification. In the example wise quadratic loss (8), the Hessian matrix is diagonally approximated. Thus (8) collapses into K independent 1-dimensional loss, i.e., (8) is written as the sum of K terms each involving just one component of $\boldsymbol{f}$:

$$\begin{aligned} q(\boldsymbol{f}|\boldsymbol{F}) &= \sum_{k=1}^{K} q_k(f_k|\boldsymbol{F}), \\ q_k(f_k|\boldsymbol{F}) &= g_k f_k + \frac{1}{2} h_{kk} f_k^2, \end{aligned} \tag{15}$$

where h_{kk} is the k-th diagonal elements in $\boldsymbol{H}$ and we have without confusion omitted the constants that are irrelevant to $\boldsymbol{f}$. Noting the N-additive form and substituting back (15), we can also collapse (14) into K 1-dimensional, independent minimization problems:

$$f_k^{(m)}(\boldsymbol{x}) = \arg\min_{f_k(\boldsymbol{x})} \sum_{i=1}^{N} q_k(\boldsymbol{f}_{i,k}|\boldsymbol{F}_i), \quad k = 1, \ldots, K, \tag{16}$$

where $\boldsymbol{f}_{i,k} = f_k(\boldsymbol{x}_i)$ means the f_k value corresponding to the i-th training examples.

Then K scalar trees are grown, the k-th tree approximately minimizing the k-th problem. Note that each tree can have its own partition, as in Fig. 3(a).

To grow the scalar tree, Friedman et al. (1998) borrowed a traditional regression model in statistics, namely the Weighted Regression Tree. The formulation becomes: for the k-th problem, fit a Weighted Regression Tree on the training examples $\{\boldsymbol{x}_i, y_i\}_{i=1}^N$ with targets $\{-g_{i,k}/h_{i,kk}\}_{i=1}^N$ and weights $\{h_{i,kk}\}_{i=1}^N$, where (in a slightly abuse of notation) we use an additional subscript i for g_k and h_{kk} to denote they correspond to the i-th training example.

3.2 ABC-LogitBoost

LogitBoost adopts a rather crude approximation to (8), where the Hessian matrix $\boldsymbol{H} \in \mathbb{R}^{K\times K}$ is diagonally approximated. ABC-LogitBoost (Li 2010b) considers an intuitively more refined way. Recall that $(K-1)$ degrees of freedom suffices to express (8), i.e., (8) can be re-written with just $(K-1)$ variables by eliminating one redundant variable. Then the new $(K-1)\times(K-1)$ Hessian matrix is, again, diagonally approximated. Li (2010b) shows that this does make a difference, as what follows.

In Sect. 2.1.3 we have explained that adding a sum to zero constraint to (8) doesn't change the problem and actually makes the minimizer unique. We can thus restrict $\boldsymbol{f}$ to vary in the subspace defined by $\mathbf{1}^T\boldsymbol{f} = 0$. Consequently, it is convenient to use a new coordinate system $\widetilde{\boldsymbol{f}} \in \mathbb{R}^{K-1}$. In order to do this, Li (2010b) introduces a concept "base class". Suppose the base class $b = K$, then the new coordinates are that $f_1 = \widetilde{f}_1, \ldots, f_{K-1} = \widetilde{f}_{K-1}$, $f_K = -(\widetilde{f}_1 + \cdots + \widetilde{f}_{K-1})$. Writing them in compact matrix notation, we have:

$$\begin{aligned} \boldsymbol{f} &= \boldsymbol{A}\widetilde{\boldsymbol{f}} \\ \boldsymbol{A} &= \begin{bmatrix} \boldsymbol{I} \\ -\mathbf{1} \end{bmatrix}, \end{aligned} \tag{17}$$

where $\boldsymbol{A} \in \mathbb{R}^{K\times(K-1)}$, $\boldsymbol{I} \in \mathbb{R}^{(K-1)\times(K-1)}$ and $\mathbf{1}$ is $(K-1)$-dim all one vector. Substituting it back to (8) we have:

$$\begin{aligned}\tilde{q}(\widetilde{\boldsymbol{f}}) &\triangleq q(\boldsymbol{A}\widetilde{\boldsymbol{f}}) \\ &= (\boldsymbol{A}^T\boldsymbol{g})^T\widetilde{\boldsymbol{f}} + \frac{1}{2}\widetilde{\boldsymbol{f}}^T\boldsymbol{A}^T\boldsymbol{H}\boldsymbol{A}\widetilde{\boldsymbol{f}} + C \\ &= \widetilde{\boldsymbol{g}}^T\widetilde{\boldsymbol{f}} + \frac{1}{2}\widetilde{\boldsymbol{f}}^T\widetilde{\boldsymbol{H}}\widetilde{\boldsymbol{f}} + C \end{aligned} \tag{18}$$

where C is a irrelevant constant and we have let

$$\begin{aligned}\widetilde{\boldsymbol{g}} &= \boldsymbol{A}^T\boldsymbol{g} \\ \widetilde{\boldsymbol{H}} &= \boldsymbol{A}^T\boldsymbol{H}\boldsymbol{A}. \end{aligned} \tag{19}$$

Li (2010b) then diagonally approximates the $(K-1)\times(K-1)$ matrix $\widetilde{\boldsymbol{H}} = \boldsymbol{A}^T\boldsymbol{H}\boldsymbol{A}$. After doing this, the rest optimization procedure is essentially the same with original LogitBoost: the loss (14) collapses to $(K-1)$ 1-dimensional, independent minimization problems; the k-th tree is fitted by a Weighted Regression Tree using the new targets and weights derived from $\widetilde{\boldsymbol{g}}$ and $\widetilde{\boldsymbol{H}}$ as are defined in (19).

Li (2010b) shows that the choice of b impacts on how well the diagonal approximation is. Two intuitive methods are proposed to select b: (1) The "worst class", i.e., the b having the biggest loss (before minimizing (14)) on that class is selected. (2) The "best class", i.e., all possible b, up to K choices, are tried and the one leading to lowest loss (after minimizing (14)) is selected.

3.3 AOSO-LogitBoost

In AOSO-LogitBoost, we also adopts the loss (18) with $K-1$ free coordinates. However, we update only one coordinate a time. An equivalent formulation goes in the following. We add a sum-to-zero constraint $\mathbf{1}^T\boldsymbol{f} = 0$ to the K-dim loss (8) and let only two coordinates of $\boldsymbol{f}$, say, f_r and f_s, vary and the other $K-2$ keep zero. Due to the sum-to-zero constraint, we further let $f_r = +t$ and $f_s = -t$ where t is a real number.

Obviously, the choice of (r, s) impacts on the goodness of approximation. However, it is unlikely to select the best class pair (r, s) for each training example. To permit the class pair selection as adaptive as possible, we adopt vector-valued tree model in AOSO-LogitBoost, i.e., any column in Fig. 3(c) can not be assigned to two cells. Then, for each node (i.e., each cell in the matrix as in Fig. 3c) we adaptively select the class pair (r, s).

Superficially, AOSO is inferior to original/ABC- LogitBoost, because many of the $\boldsymbol{f}$ values in the matrix are untouched (zeros), as shown in Fig. 3(c). However, we should recall the "big picture": the untouched values might hopefully receive better updating in later Boosting iterations. Consequently, AOSO is still "on average" more efficient than original/ABC-LogitBoost. We will further discuss this issue in Sect. 5.

4 The AOSO-LogitBoost algorithm

In this section we describe the details of AOSO-LogitBoost. Specifically, we will focus on how to build a tree in Boosting iteration. Some key ingredients of tree building improving previous techniques were firstly introduced by Li. However, we will re-derive them in our own language. The credits will be made clear when they are explained in the following.

4.1 Details of tree building

Solving (14) with the tree model (11) is equivalent to determining the parameters $\{\boldsymbol{t}_j, R_j\}_{j=1}^J$ at the m-th iteration. In this subsection we will show how this problem reduces to solving a collection of quadratic subproblems for which we can use standard numerical methods-Block Coordinate Descent.[1] Also, we will show how the gradient and Hessian can be computed incrementally.

We begin by defining some notation for the *node loss*:

$$NodeLoss(\boldsymbol{t}; \mathcal{I}) = \sum_{i \in \mathcal{I}} q(\boldsymbol{t}|\boldsymbol{F}_i) \tag{20}$$

where $\mathcal{I}$ denotes the index set of the training examples on some either internal or terminal node (i.e., those falling into the corresponding region). Minimizing (20) allows us to obtain a set of nodes $\{\boldsymbol{t}_j, R_j\}_{j=1}^J$ using the following procedures:

1. To obtain the values $\boldsymbol{t}_j$ for a given R_j, we simply take the minimizer of (20):
$$\boldsymbol{t}_j = \arg\min_{\boldsymbol{t}} NodeLoss(\boldsymbol{t}; \mathcal{I}_j), \tag{21}$$
where $\mathcal{I}_j$ denotes the index set for R_j.
2. To obtain the partition $\{R_j\}_{j=1}^J$, we recursively perform binary splitting until there are J-terminal nodes.

The key to the second procedure is the computation of the node split gain. Suppose an internal node with n training examples ($n = N$ for the root node), we fix on some feature and re-index all the n examples according to their sorted feature values. Now we need to find the index n' with $1 < n' < n$ that maximizes the node gain defined as loss reduction after a division between the n'-th and $(n'+1)$-th examples:

$$\begin{aligned} NodeGain(n') =& NodeLoss(\boldsymbol{t}^*; \mathcal{I}) \\ &- (NodeLoss(\boldsymbol{t}_L^*; \mathcal{I}_L) + NodeLoss(\boldsymbol{t}_R^*; \mathcal{I}_R)) \end{aligned} \tag{22}$$

where $\mathcal{I} = \{1, \ldots, n\}$, $\mathcal{I}_L = \{1, \ldots, n'\}$ and $\mathcal{I}_R = \{n'+1, \ldots, n\}$; $\boldsymbol{t}^*$, $\boldsymbol{t}_L^*$ and $\boldsymbol{t}_R^*$ are the minimizers of (20) with index sets $\mathcal{I}$, $\mathcal{I}_L$ and $\mathcal{I}_R$, respectively. Generally, this search applies to all features. The division yielding the largest value of (22) is then recorded to perform the actual node split.

Note that (22) arises in the context of an $O(N \times D)$ outer loop, where D is number of features. However, a naïve summing of the losses for (20) incurs an additional $O(N)$ factor in complexity, which finally results in an unacceptable $O(N^2 D)$ complexity for a single boosting iteration.

Fortunately, the gradient and Hessian can be incrementally computed. To see why, simply rewrite the (20) in standard quadratic form:

$$loss(\boldsymbol{t}; \mathcal{I}) = \overline{\boldsymbol{g}}^T \boldsymbol{t} + \frac{1}{2} \boldsymbol{t}^T \overline{\boldsymbol{H}} \boldsymbol{t}, \tag{23}$$

where $\overline{\boldsymbol{g}} = -\sum_{i \in \mathcal{I}} \boldsymbol{g}_i$, $\overline{\boldsymbol{H}} = \sum_{i \in \mathcal{I}} \boldsymbol{H}_i$. Thanks to the additive form, both $\overline{\boldsymbol{g}}$ and $\overline{\boldsymbol{H}}$ can be incrementally/decrementally computed in constant time when the split searching proceeds

[1] We use Newton descent as there is evidence in Li (2010b) that gradient descent, i.e., in Friedmans's MART (Friedman 2001), leads to decreased classification accuracy.

from one training example to the next. Therefore, the computation of (23) eliminates the $O(N)$ complexity in the naïve summing of losses.[2]

4.2 Properties of approximated node loss

To minimise (23), we make use of some properties for (23) that can be exploited when finding a solution. First, the invariant property carries over to the node loss (23):

Property 1 $loss(\boldsymbol{t}; \mathcal{I}) = loss(\boldsymbol{t} + c\mathbf{1}; \mathcal{I})$.

Proof This is obvious by noting the additive form. □

For the Hessian $\boldsymbol{H}$, we have $\text{rank}(\boldsymbol{H}) \leq \text{rank}(\boldsymbol{H}_i)$ by noting the additive form in (23). In Li (2010b) it is shown that $\det \boldsymbol{H}_i = 0$ by brute-force determinant expansion. Here we give a stronger property:

Property 2 Each $\boldsymbol{H}_i$ is a positive semi-definite matrix such that (1) $\text{rank}(\boldsymbol{H}_i) = \kappa - 1$, where κ is the number of non-zero elements in $\boldsymbol{p}_i$; (2) $\mathbf{1}$ is the eigenvector for eigenvalue 0.

The proof can be found in Appendix 1.

The properties shown above indicate that (1) $\boldsymbol{H}$ is singular so that unconstrained Newton descent is not applicable here and (2) $\text{rank}(H)$ could be as high as $K - 1$, which prohibits the application of the standard fast quadratic solver designed for low rank Hessian. In the following we propose to address this problem via block coordinate descent, a technique that has been successfully used in training SVMs (Bottou and Lin 2007).

4.3 Block coordinate descent

For the variable $\boldsymbol{t}$ in (23), we only choose two coordinates, i.e., a class pair, to update while keeping the others fixed. We note that single coordinates cannot be updated independently due to the sum-to-zero constraint. Suppose we have chosen the r-th and the s-th coordinate (we explain precisely how to do so in the next section). Let $t_r = t$ and $t_s = -t$ be the free variables (such that $t_r + t_s = 0$) and $t_k = 0$ for $k \neq r, k \neq s$. Plugging these into (23) yields an unconstrained one dimensional quadratic problem with regards to the scalar variable t:

$$loss(t) = \bar{g}^T t + \frac{1}{2}\bar{h}t^2 \tag{24}$$

where the gradient and Hessian collapse to scalars:

$$\bar{g} = -\sum_{i \in I} \left((r_{i,r} - p_{i,r}) - (r_{i,s} - p_{i,s}) \right) \tag{25}$$

$$\bar{h} = \sum_{i \in I} \left(p_{i,r}(1 - p_{i,r}) + p_{i,s}(1 - p_{i,s}) + 2p_{i,r}p_{i,s} \right), \tag{26}$$

The derivatives (25) and (26) involving two classes were firstly given in Li (2008), where the class subscript r, s are fixed for the nodes in a tree. In this work, we allow r, s vary from node to node.

[2] In Real AdaBoost.MH, such a second order approximation is not necessary (although possible, cf. Zou et al. 2008). Due to the special form of the exponential loss and the absence of a sum-to-zero constraint, there exists analytical solution for the node loss (20) by simply setting the derivative to $\mathbf{0}$. Here also, the computation can be incremental/decremental. Since the loss design and AdaBoost.MH are not our main interests, we do not discuss this further.

To this extent, we are able to obtain the analytical expression for the minimizer and minimum of (24):

$$t^* = \arg\min_t loss(t) = -\frac{\bar{g}}{\bar{h}} \tag{27}$$

$$loss(t^*) = -\frac{\bar{g}^2}{2\bar{h}} \tag{28}$$

by noting the non-negativity of (26).

Based on (27), node vector (21) can be approximated by $\boldsymbol{t}_j$ with components

$$t_{j,k} = \begin{cases} +(-\bar{g}/\bar{h}) & k = r \\ -(-\bar{g}/\bar{h}) & k = s \\ 0 & \text{otherwise} \end{cases} \tag{29}$$

where g and h are computed using (25) and (26), respectively, with index set $\mathcal{I}_j$. Based on (28), the node gain (22) can be approximated as

$$NodeGain(n') = \frac{\bar{g}_L^2}{2\bar{h}_L} + \frac{\bar{g}_R^2}{2\bar{h}_R} - \frac{\bar{g}^2}{2\bar{h}}, \tag{30}$$

where $\bar{g}$ (or $\bar{g}_L$, $\bar{g}_R$) and $\bar{h}$ (or $\bar{h}_L$, $\bar{h}_R$) are computed by using (25) and (26) with index set $\mathcal{I}$ (or $\mathcal{I}_L$, $\mathcal{I}_R$). We note that (30) was firstly derived in Li (2010b) in a slightly different way.

4.4 Class pair selection

Bottou and Lin (2007) provide two methods for selecting (r, s) for an SVM solver that we consider for our purposes. The first is based on a first-order approximation: let t_r and t_s be the free variables and fix the rest to 0. For a $\boldsymbol{t}$ with sufficiently small norm, let $t_r = \epsilon$ and $t_s = -\epsilon$ where $\epsilon > 0$ is some small enough constant. The first order approximation of (23) is then:

$$loss(\boldsymbol{t}) \approx loss(\boldsymbol{0}) + \overline{\boldsymbol{g}}^T \boldsymbol{t} = loss(\boldsymbol{0}) - \epsilon(-\bar{g}_r - (-\bar{g}_s)), \tag{31}$$

where we have denoted the vector $\overline{\boldsymbol{g}} = (\bar{g}_1, \ldots, \bar{g}_r, \ldots, \bar{g}_s, \ldots, \bar{g}_K)^T$. It follows that the indices r, s resulting in largest decrement to (31) are:

$$\begin{aligned} r &= \arg\max_k \{-\bar{g}_k\} \\ s &= \arg\min_k \{-\bar{g}_k\}. \end{aligned} \tag{32}$$

The second method, derived in a similar way, takes into account the second-order information:

$$\begin{aligned} r &= \arg\max_k \{-\bar{g}_k\} \\ s &= \arg\max_k \left\{ \frac{(\bar{g}_r - \bar{g}_k)^2}{\bar{h}_{rr} + \bar{h}_{kk} - 2\bar{h}_{rk}} \right\}, \end{aligned} \tag{33}$$

where we have denoted the matrix $\overline{\boldsymbol{H}} = \{\bar{h}_{ij}\}$.

Both methods are $O(K)$ procedures that are better than the $K \times (K-1)/2$ naïve enumeration of all possible pairs. However, in our implementation we find that (33) achieves better results for AOSO-LogitBoost.

The selection of a class pair (r, s) here is somewhat similar to the selection of base class b in ABC Boost. Actually, Li proposed in Li (2008) that the "worst class" with the largest loss be selected as b. Clearly, the max derivative is another indicator for how "worst" it is. In

this sense, our class pair selection extends the base class idea and can be seen as a concrete implementation of the general "worst class" idea.

The pseudocode for AOSO-LogitBoost is given in Algorithm 1 and shows how all the above approximations are used together to iteratively find a model. The algorithm runs on the sample $\{\boldsymbol{x}_i, y_i\}_{i=1}^N$ for M rounds. In each round m it updates the values of $\boldsymbol{F}_i = \boldsymbol{F}(\boldsymbol{x}_i)$ for each of the instances $\boldsymbol{x}_i$ by first computing a rectangular partition of the feature space $\{R_j^m\}_{j=1}^J$ and the corresponding node values $\{\boldsymbol{t}_j^m\}_{j=1}^J$, and then incrementing $\boldsymbol{F}_i$ by $\boldsymbol{t}_j^m$ where j is the index of the region R_j^m such that $\boldsymbol{x}_i \in R_j^m$.

Algorithm 1 AOSO-LogitBoost for a sample $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$. Parameters: v = shrinkage factor (controls learning rate); J = number of terminal nodes; M = number of rounds

1: $\boldsymbol{F}_i = \boldsymbol{0}, i = 1, \ldots, N$
2: **for** $m = 1$ to M **do**
3: $p_{i,k} = \frac{\exp(F_{i,k})}{\sum_{j=1}^K \exp(F_{i,j})}, k = 1, \ldots, K, i = 1, \ldots, N.$
4: Obtain $\{R_j^m\}_{j=1}^J$ by recursive region partition. Node split gain is computed as (30), where the class pair (r, s) is selected using (33) .
5: Compute $\{\boldsymbol{t}_j^m\}_{j=1}^J$ by (29), where the class pair (r, s) is selected using (33) .
6: $\boldsymbol{F_i} = \boldsymbol{F_i} + v \sum_{j=1}^J \boldsymbol{t}_j^m I(\boldsymbol{x}_i \in R_j^m), \quad i = 1, \ldots, N.$
7: **end for**

5 Comparison to (ABC-)LogitBoost

In this section we compare the derivations of LogitBoost and ABC-LogitBoost and provide some intuition for observed behaviours in the experiments in Sect. 7.

To solve (14), ABC-LogitBoost uses $K - 1$ independent trees defined by

$$f_k(x) = \begin{cases} \sum_j t_{j,k} I(x \in R_{j,k}) & , k \neq b \\ -\sum_{l \neq b} f_l(x) & , k = b \end{cases} \tag{34}$$

where b is the *base class*. As explained in Sect. 3.2, b can be selected as either the "worst class" or the "best class".

Li (2010b) gives how to compute the node value and the node split gain for building the k-th tree ($k \neq b$). Although derived from different motivation as ours, they are actually the same with (29) and (30) in this paper, where the class pair (r, s) is replaced with (k, b). We should not be surprised at this coincidence by noting that AOSO's vector tree has only one freely altering coordinate on each node and thus "behaves" like a scalar tree. In this sense, AOSO and ABC is comparable. Actually, ABC can be viewed as a special form of AOSO with two differences: (1) For each tree, the class pair is fixed for every node in ABC, while it is selected adaptively in AOSO, and (2) $K - 1$ trees are added per iteration in ABC, while only one tree is added per iteration by AOSO.

It seems unappealing to add just one tree as in AOSO, since many $\boldsymbol{f}(\boldsymbol{x})$ values are untouched (i.e.,, set to zeros, as illustrated in Fig. 3c); In the meanwhile, ABC would be better since it updates all the $\boldsymbol{f}(\boldsymbol{x})$ values. Considering the Boosting context, we argue, however, that AOSO should still be preferred in an "on average" sense. After adding one tree in AOSO, the $\boldsymbol{F}(\boldsymbol{x})$ values are updated and the gradient/Hessian is immediately recomputed

for each training example, which impacts on how to build the tree at next iteration. Thus the $\boldsymbol{F}(\boldsymbol{x})$ values might still receive good enough updating after several iterations, due to the adaptive class pair selection for every node at current iteration. In contrast, the $K-1$ trees in ABC use the same set of gradients and Hessians, which are not recomputed until adding all the $K-1$ trees.

Therefore, it is fair to compare ABC and AOSO in regards of number of trees, rather than number of iterations. AOSO's "on average" better performance is confirmed by the experiments in Sect. 7.2.

To evaluate whether the adaptive class pair selection is critical, we considered a variant of AOSO-LogitBoost that adopts a *fixed* class pair selection. Specifically, we still add one tree per iteration, but select a single class pair root node and let it be fixed for all other nodes, which is very similar to ABC's choice. This variant was tried but unfortunately, **degraded performance** was observed so the results are not reported here.

From the above analysis, we believe that AOSO-LogitBoost's more flexible model obtained from the adaptive split selection (as well as its immediate model updating after adding one tree per iteration) is what contributes to its improvement over ABC.

6 Sum-to-one probability and dense Hessian matrix

As in previous discussion, both ABC or AOSO improve on the original LogitBoost method by dealing with dense Hessian matrix due to the Logistic loss as given in (3). An immediate question is that whether we can derive an effective alternative surrogate loss that has a diagonal Hessian matrix "by design"—i.e., can we define a modified Logistic loss that guarantees a diagonal Hessian matrix? In this section, we show how the original multi-class Logistic loss can be derived from a maximum entropy argument via convex duality, in a manner similar to derivations of boosting updates by Shen and Li (2010), Shen and Hao (2011), Lafferty (1999), and Kivinen and Warmuth (1999) and results connecting entropy and loss by Masnadi-Shirazi and Vasconcelos (2010) and Reid and Williamson (2010).

In contrast to earlier work, our analysis focuses on the role of the constraint in defining the loss and the effect it has on the form of its gradient and Hessian. In particular, we'll see that the original Logistic loss's dense Hessian matrix essentially results from sum-to-one constraint on class probabilities. Moreover, we are able to obtain a diagonal Hessian matrix by dropping this constraint when deriving an alternative to the Logistic loss from the same maximum entropy formulation. By doing so we show that the optimization (i.e., via Newton descent) becomes straightforward, however lower classification accuracy and slower convergence rate are observed for the new loss. Therefore, we argue the techniques used by ABC/AOSO seem necessary for dealing with the dense Hessian matrix for the original, more effective, Logistic loss.

6.1 Logistic loss in primal/dual view

We now examine the duality between entropy and loss in a manner similar to that of the more general treatment of Masnadi-Shirazi and Vasconcelos (2010); Reid and Williamson (2010). By starting with a "trivial" maximum entropy problem, we show how consideration of its dual problem recovers a "composite representation" Reid and Williamson (2010) of a loss function in terms of a loss defined on the simplex and corresponding link function. In case of Shannon entropy, we show that such a construction results in the original Logistic loss

function. Appendix 2 describes the matrix calculus definitions and conventions we adopt, mainly from Magnus and Neudecker (2007).

For each feature vector $\boldsymbol{x}$, let $\boldsymbol{\eta}(\boldsymbol{x}) = Pr(\boldsymbol{y}|\boldsymbol{x}) \in \Delta^K$ be the true conditional probability for the K classes where $\Delta^K = \{\boldsymbol{p} \in [0,1]^K : \sum_{k=1}^K p_k = 1\}$ denotes the K-simplex. Let $\mathsf{H} : \Delta^K \to \mathbb{R}$ be an *entropy function*—a concave function such that $\mathsf{H}(\mathbf{e}^k) = 0$ for each vertex $\mathbf{e}^k$ of Δ^K.

Given some set of probabilities $S \subset \Delta^K$—typically defined by a set of constraints—the MaxEnt criterion (Jaynes 1957) advises choosing the $\boldsymbol{p} \in S$ such that $\mathsf{H}(\boldsymbol{p})$ is maximized. We consider a "trivial" instance of the MaxEnt problem in which the set S contains only the single point $\boldsymbol{\eta}$, i.e., $S = \{\boldsymbol{\eta}\}$, yielding the following problem:

$$\begin{aligned} \max \quad & \mathsf{H}(\boldsymbol{p}) \\ s.t. \quad & \boldsymbol{p} - \boldsymbol{\eta} = 0 \\ & \mathbf{1}^T \boldsymbol{p} - \mathbf{1} = 0, \end{aligned} \tag{35}$$

where the argument $\boldsymbol{x}$ to $\boldsymbol{\eta}()$, $\boldsymbol{p}()$ has been omitted. Despite its apparent triviality, the Lagrangian of this problem has an interesting form:

$$\mathcal{L}(\boldsymbol{p}, \boldsymbol{F}, \lambda) = \mathsf{H}(\boldsymbol{p}) + \boldsymbol{F}^T(\boldsymbol{p} - \boldsymbol{\eta}) + \lambda(\mathbf{1}^T \boldsymbol{p} - \mathbf{1}), \tag{36}$$

where $\boldsymbol{F} \in \mathbb{R}^K$, $\lambda \in \mathbb{R}$ are the so-called dual variables. The dual function is obtained by maximizing the Lagrangian over the domain of the primal variable $\boldsymbol{p}$. By setting to zero the derivative of (36) w.r.t. $\boldsymbol{p}$, we can implicitly express $\boldsymbol{p}$ as a function of $\boldsymbol{F}$ and λ via

$$\nabla \mathsf{H}(\boldsymbol{p}) + \lambda \mathbf{1} + \boldsymbol{F} = 0. \tag{37}$$

This gives us the dual function g which depends only on $\boldsymbol{F}$, λ:

$$g(\boldsymbol{F}, \lambda) = \mathsf{H}(\boldsymbol{p}) + \boldsymbol{F}^T(\boldsymbol{p} - \boldsymbol{\eta}) + \lambda(\mathbf{1}^T \boldsymbol{p} - \mathbf{1}), \tag{38}$$

where $\boldsymbol{p} = \boldsymbol{p}(\boldsymbol{F}, \lambda)$ as per (37), and the unconstrained convex dual problem

$$\min \quad g(\boldsymbol{F}, \lambda).$$

Now further eliminating the dual variable λ in (38) by taking derivative of (38) w.r.t. λ as

$$\begin{aligned} \mathsf{D}_\lambda g(\boldsymbol{F}, \lambda) &= (\mathsf{D}_\lambda \mathsf{H}(\boldsymbol{p}))(\mathsf{D}_\lambda \boldsymbol{p}) + \boldsymbol{F}^T(\mathsf{D}_\lambda \boldsymbol{p}) + \lambda \mathbf{1}^T(\mathsf{D}_\lambda \boldsymbol{p}) + (\mathbf{1}^T \boldsymbol{p} - \mathbf{1}) \\ &= (\mathsf{D}_\lambda \mathsf{H}(\boldsymbol{p}) + \lambda \mathbf{1}^T + \boldsymbol{F}^T)(\mathsf{D}_\lambda \boldsymbol{p}) + \mathbf{1}^T \boldsymbol{p} - 1 \\ &= \mathbf{1}^T \boldsymbol{p} - 1. \end{aligned}$$

and setting to zero, we obtain the "partial" dual function:

$$\ell(\boldsymbol{F}) = \mathsf{H}(\boldsymbol{p}) + \boldsymbol{F}^T(\boldsymbol{p} - \boldsymbol{\eta}), \tag{39}$$

where $\boldsymbol{p} = \boldsymbol{p}(\boldsymbol{F}) \in \mathbb{R}^K$, $\lambda = \lambda(\boldsymbol{F}) \in \mathbb{R}$ are given by the $(K+1) \times (K+1)$ equation array

$$\begin{cases} \nabla \mathsf{H}(\boldsymbol{p}) + \lambda \mathbf{1} + \boldsymbol{F} & = 0 \\ \mathbf{1}^T \boldsymbol{p} - 1 & = 0. \end{cases} \tag{40}$$

In Machine Learning, (39) and (40) are precisely the "loss function" and the "link function", respectively (e.g., the Logistic loss (3) and Logistic link (4)). Training boosting classifier is essentially a numerical optimization procedure, where the gradient and Hessian of the loss (39) are needed. We give them as follows:

Theorem 1 *The gradient of (39) is* $\nabla \ell(\boldsymbol{F}) = \boldsymbol{p} - \boldsymbol{\eta}$.

See Appendix 2 for its proof. Note that the gradient is precisely the left-hand-side of the constraint in primal (35), which is a common result in primal-dual theory in convex optimization (Bertsekas 1982).

Theorem 2 *The Hessian of (39) is* $\nabla^2 \ell(\boldsymbol{F}) = -A^{-1} + \frac{1}{\mathbf{1}^T A^{-1} \mathbf{1}} A^{-1} \mathbf{1}\mathbf{1}^T A^{-1}$ *, where the shorthand* $A = \nabla^2 \mathsf{H}(\boldsymbol{p})$.

See Appendix 2 for its proof.

In both of the above two theorems, the dependence on $\boldsymbol{F}$ is implicit where p is given by the Link function (40).

To this extent, we can show some concrete results in case of Logistic settings. The starting point is that we adopt Shannon entropy in (35) such that $\mathsf{H}(\boldsymbol{p}) = -\sum_{k=1}^{K} p_k \log p_k$. Thus its gradient and Hessian are $\nabla \mathsf{H}(\boldsymbol{p}) = -(1 + \log p_1, \ldots, 1 + \log p_K)^T$, $\nabla^2 \mathsf{H}(\boldsymbol{p}) = -\text{diag}(1/p_1, \ldots, 1/p_K)$, respectively. With these two expressions, it follows that the link function by solving equations (40) is precisely the Logistic link in (4). Furthermore, according to Theorem 1 and 2 the gradient and Hessian for the loss (39) are $\nabla \ell = \boldsymbol{p} - \boldsymbol{\eta}$ and $\nabla^2 \ell = \text{diag}(p_1, \ldots, p_K) - \boldsymbol{p}\boldsymbol{p}^T$, which are respectively no more than (6) and (7) by noting that $\boldsymbol{\eta}$ reduces to 1-of-K response $\boldsymbol{r}$ when taking value on the vertex of the probability simplex. These are shown in the left half of Table 1.

6.2 Diagonal Hessian matrix by dropping the sum-to-one constraint

In last subsection we derived the LogitBoost via a duality argument. An immediate observation is that the sum-to-one constraint $\mathbf{1}^T \boldsymbol{p} - 1 = 0$ in (35) seems redundant, since it is already guaranteed by the constraint $\boldsymbol{p} - \boldsymbol{\eta} = 0$ where $\boldsymbol{\eta}$ itself is sum-to-one. This means that the primal problem can be rewritten as:

$$\max \quad \mathsf{H}(\boldsymbol{p}) \tag{41}$$

$$s.t. \quad \boldsymbol{p} - \boldsymbol{\eta} = 0 \tag{42}$$

Deriving its dual form, we obtain the corresponding loss

$$\ell(\boldsymbol{F}) = \mathsf{H}(\boldsymbol{p}) + \boldsymbol{F}^T(\boldsymbol{p} - \boldsymbol{\eta}), \tag{43}$$

and link

$$\nabla \mathsf{H}(\boldsymbol{p}) + \boldsymbol{F} = 0 \tag{44}$$

Still, with $\mathsf{H}(\boldsymbol{p}) = -\sum_{k=1}^{K} p_k \log p_k$ we can obtain the link, the gradient and Hessian for the loss, as shown in the right half of Table 1. One attractive feature of this alternative derivation is the diagonal Hessian matrix that is yielded. When calculating the node value or node gain, we can obtain the precise Newton Step and Newton Decrement since the inversion of Hessian is much easier to compute. Consequently, no effort (e.g., base class selection in ABC or class pair selection in AOSO) is needed to deal with the difficulty of dense Hessian and all classes can be updated for each terminal node.

6.3 Degraded performance

We refer to the modified LogitBoost without sum-to-one constraint as "K-LogitBoost", where all the K classes are updated at each terminal node. We compare K-LogitBoost with AOSO-LogitBoost as in Fig. 4. Noting that only 2 classes are updated at each terminal node in

Table 1 The Link, gradient/Hessian for the loss with or without the sum-to-one constraint

	with sum-to-one	without sum-to-one
Link $\boldsymbol{p} = \boldsymbol{p}(\boldsymbol{F})$	$p_k = \frac{e^{F_k}}{\sum_j^K e^{F_j}}, \quad k = 1, \ldots, K$	$p_k = e^{F_k - 1}, \quad k = 1, \ldots, K$
Gradient $\nabla \ell(\boldsymbol{F})$	$\boldsymbol{p} - \boldsymbol{\eta}$	$\boldsymbol{p} - \boldsymbol{\eta}$
Hessian $\nabla^2 \ell(\boldsymbol{F})$	$\text{diag}(p_1, \ldots, p_K) - \boldsymbol{p}\boldsymbol{p}^T$	$\text{diag}(p_1, \ldots, p_K)$

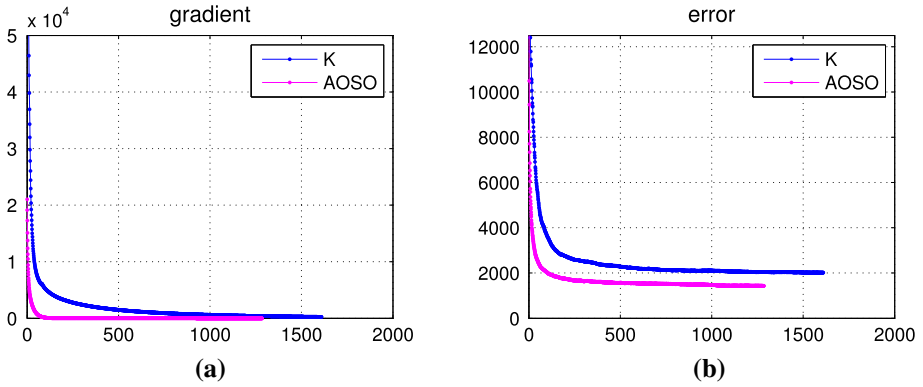


Fig. 4 Comparison between K LogitBoost and AOSO LogitBoost on dataset `M-Basic`. **a** How the L2 norm of gradient $||\boldsymbol{p} - \boldsymbol{\eta}||_2^2$ decreases when iteration proceeds. **b** Test error v.s. iteration

AOSO, we scale the horizontal axis (i.e., the number of iterations) by a factor $2/K$ for AOSO to make it a fair comparison. As can be seen in the results, a slower convergence than AOSO is incurred by K LogitBoost with regard to both test error and L2 norm of gradient, although K LogitBoost is still competitive with AOSO LogitBoost. More comparative results are provided in Sect. 7.3.

We provide an intuitive explanation for this phenomenon. At convergence, the gradient $\boldsymbol{p} - \boldsymbol{\eta}$ vanishes, and consequently, $\boldsymbol{p}$ satisfies the sum-to-one constraint since $\boldsymbol{\eta}$ is a probability. For the original Logistic loss setting, the apparently redundant sum-to-one constraint for $\boldsymbol{p}$ in (35) enforces that the primal variable $\boldsymbol{p}$ approaches $\boldsymbol{\eta}$ on the plane $\mathbf{1}^T \boldsymbol{p} - 1 = 0$ containing the simplex Δ^K. In contrast, for K LogitBoost $\boldsymbol{p}$ may reside outside that plane during the optimization procedure. The latter optimization is intuitively slower since the L2 norm $||\boldsymbol{p} - \boldsymbol{\eta}||_2^2$ never increases when $\boldsymbol{p} - \boldsymbol{\eta}$ is projected onto a plane.

7 Experiments

In this section we compare AOSO-LogitBoost with ABC-LogitBoost, which was shown to outperform original LogitBoost in Li's experiments (Li 2010b, 2009a). We test AOSO on all the datasets used in Li (2010b, 2009a), as listed in Table 2. In the top section are UCI datasets and in the bottom are Mnist datasets with many variations (see Li (2010a) for detailed descriptions).[3]

[3] Code and data are available at http://ivg.au.tsinghua.edu.cn/index.php?n=People.PengSun.

Table 2 Datasets used in our experiments

Datasets	K	#features	#training	#test
Poker525k	10	25	5,25,010	5,00,000
Poker275k	10	25	2,75,010	5,00,000
Poker150k	10	25	1,50,010	5,00,000
Poker100k	10	25	1,00,010	5,00,000
Poker25kT1	10	25	25,010	5,00,000
Poker25kT2	10	25	25,010	5,00,000
Covertype290k	7	54	2,90,506	2,90,506
Covertype145k	7	54	1,45,253	2,90,506
Letter	26	16	16,000	4,000
Letter15k	26	16	15,000	5,000
Letter2k	26	16	2,000	18,000
Letter4K	26	16	4,000	16,000
Pendigits	10	16	7,494	3,498
Zipcode (a.k.a. USPS)	10	256	7,291	2,007
Isolet	26	617	6,238	1,559
Optdigits	10	64	3,823	1,797
Mnist10k	10	784	10,000	60,000
M-Basic	10	784	12,000	50,000
M-Image	10	784	12,000	50,000
M-Rand	10	784	12,000	50,000
M-Noise1	10	784	10,000	2,000
M-Noise2	10	784	10,000	2,000
M-Noise3	10	784	10,000	2,000
M-Noise4	10	784	10,000	2,000
M-Noise5	10	784	10,000	2,000
M-Noise6	10	784	10,000	2,000

To exhaust the learning ability of (ABC-)LogitBoost, Li let the boosting stop when either the training loss is small (implemented as $\leq 10^{-16}$) or a maximum number of iterations, M, is reached. Test errors at last iteration are simply reported since no obvious over-fitting is observed. By default, $M = 10,000$, while for those large datasets (`Covertype290k`, `Poker525k`, `Pokder275k`, `Poker150k`, `Poker100k`) $M = 5,000$. We adopt the same criteria, except that our maximum iterations $M_{AOSO} = (K-1) \times M_{ABC}$, where K is the number of classes. Note that only one tree is added at each iteration in AOSO, while $K-1$ are added in ABC. Thus, this correction compares the same maximum number of trees for both AOSO and ABC.

The most important tuning parameters in LogitBoost are the number of terminal nodes J, and the shrinkage factor v. In (Li 2010b, 2009a), Li reported results of (ABC-)LogitBoost for a number of J-v combinations. We report the corresponding results for AOSO-LogitBoost for the same combinations. In the following, we intend to show that **for nearly all J–v combinations, AOSO-LogitBoost has lower classification error and faster convergence rates than ABC-LogitBoost**.

Table 3 Summary of test classification errors

Datasets	#tests	ABC	AOSO	R	pv	ABC*	AOSO*	R	pv
Poker525k	500000	1736	**1537**	0.1146	0.0002	–	–	–	–
Poker275k	500000	2727	**2624**	0.0378	0.0790	–	–	–	–
Poker150k	500000	5104	**3951**	0.2259	0.0000	–	–	–	–
Poker100k	500000	13707	**7558**	0.4486	0.0000	–	–	–	–
Poker25kT1	500000	37345	**31399**	0.1592	0.0000	37345	**31399**	0.1592	0.0000
Poker25kT2	500000	36731	**31645**	0.1385	0.0000	36731	**31645**	0.1385	0.0000
Covertype290k	290506	9727	**9586**	0.0145	0.1511	–	–	–	–
Covertype145k	290506	13986	**13712**	0.0196	0.0458	–	–	–	–
Letter	4000	**89**	92	−0.0337	0.5892	89	**88**	0.0112	0.4697
Letter15k	5000	**109**	116	−0.0642	0.6815	–	–	–	–
Letter4k	16000	1055	**991**	0.0607	0.0718	1034	**961**	0.0706	0.0457
Letter2k	18000	2034	**1862**	0.0846	0.0018	1991	**1851**	0.0703	0.0084
Pendigits	3498	100	**83**	0.1700	0.1014	90	**81**	0.1000	0.2430
Zipcode	2007	**96**	99	−0.0313	0.5872	**92**	94	−0.0217	0.5597
Isolet	1559	65	**55**	0.1538	0.1759	55	**50**	0.0909	0.3039
Optdigits	1797	55	**38**	0.3091	0.0370	38	**34**	0.1053	0.3170
Mnist10k	60000	2102	**1948**	0.0733	0.0069	2050	**1885**	0.0805	0.0037
M-Basic	50000	1602	**1434**	0.1049	0.0010	–	–	–	–
M-Rotate	50000	5959	**5729**	0.0386	0.0118	–	–	–	–
M-Image	50000	4268	**4167**	0.0237	0.1252	4214	**4002**	0.0503	0.0073
M-Rand	50000	4725	**4588**	0.0290	0.0680	–	–	–	–
M-Noise1	2000	234	**228**	0.0256	0.3833	–	–	–	–
M-Noise2	2000	237	**233**	0.0169	0.4221	–	–	–	–
M-Noise3	2000	238	**233**	0.0210	0.4031	–	–	–	–
M-Noise4	2000	238	**233**	0.0210	0.4031	–	–	–	–
M-Noise5	2000	227	**214**	0.0573	0.2558	–	–	–	–
M-Noise6	2000	201	**191**	0.0498	0.2974	–	–	–	–

Lower one is in bold. Middle panel $J = 20$,$v = 0.1$ except for `Poker25kT1` and `Poker25kT2` on which J, v are chosen by validation (See the text in 7.1); Right panel the overall best. Dash means unavailable in Li (2010b, 2009a). Relative improvements (R) and p values (pv) are given

7.1 Classification errors

7.1.1 Summary

In Table 3 we summarize the results for all datasets. Li (2010b) reports that ABC-LogitBoost is insensitive to J, v on all datasets except for `Poker25kT1` and `Poker25kT2`. Therefore, Li reports classification errors for ABC simply with $J = 20$ and $v = 0.1$, except that on `Poker25kT1` and `Poker25kT2` errors are reported by using each other's test set as a validation set. Based on the same criteria we report AOSO in the middle panel of Table 3 where the test errors as well as the improvement relative to ABC are given. In the right panel

Table 4 Summary of test error rates for (ABC- and AOSO-)LogitBoost and deep learning algorithms on `variants of Mnist`

	M-Basic (%)	M-Rotate (%)	M-Image (%)	M-Rand (%)
SVM-RBF	3.05	11.11	22.61	14.58
SVM-POLY	3.69	15.42	24.01	16.62
NNET	4.69	18.11	27.41	20.04
DBN-3	3.11	**10.30**	16.31	**6.73**
SAA-3	3.46	**10.30**	23.00	11.28
DBN-1	3.94	14.69	16.15	9.80
ABC	3.20	11.92	8.54	9.45
AOSO	***2.87***	*11.46*	***8.33***	*9.18*

SVM-RBF SVM with RBF kernel, *SVM-POLY* SVM with polynomial kernel, *DBN-1(-3)* Deep Belief Network with 1(3) hidden layers, *NNET* Neural Network with 1 hidden layer, *SAA-3* Stacked Auto Associator with 3 hidden layers. See (Larochelle et al 2007) for details. For each dataset, the lowest error among all algorithms is in bold, while the lower one between ABC and AOSO is in italic. Bold and italic result means both

Table 5 Test classification errors on `Mnist10k`

	$v = 0.04$	$v = 0.06$	$v = 0.08$	$v = 0.1$
$J = 4$	2,630 **2,515**	2,600 **2,414**	2,535 **2,414**	2,522 **2,392**
$J = 6$	2,263 **2,133**	2,252 **2,146**	2,226 **2,146**	2,223 **2,134**
$J = 8$	2,159 **2,055**	2,138 **2,046**	2,120 **2,046**	2,143 **2,055**
$J = 10$	2,122 **2,010**	2,118 **1,980**	2,091 **1,980**	2,097 **2,014**
$J = 12$	2,084 **1,968**	2,090 **1,965**	2,090 **1,965**	2,095 **1,995**
$J = 14$	2,083 **1,945**	2,094 **1,938**	2,063 **1,938**	2,050 **1,935**
$J = 16$	2,111 **1,941**	2,114 **1,928**	2,097 **1,928**	2,082 **1,966**
$J = 18$	2,088 **1,925**	2,087 **1,916**	2,088 **1,916**	2,097 **1,920**
$J = 20$	2,128 **1,930**	2,112 **1,917**	2,095 **1,917**	2,102 **1,948**
$J = 24$	2,174 **1,901**	2,147 **1,920**	2,129 **1,920**	2,138 **1,903**
$J = 30$	2,235 **1,887**	2,237 **1,885**	2,221 **1,885**	2,177 **1,917**
$J = 40$	2,310 **1,923**	2,284 **1,890**	2,257 **1,890**	2,260 **1,912**
$J = 50$	2,353 **1,958**	2,359 **1,910**	2,332 **1,910**	2,341 **1,934**

In each J–v entry, the first is for ABC-LogitBoost and the second for AOSO-LogitBoost. Lower one is in bold

of Table 3 we provide the comparison for the best results achieved over all J–v combinations when the corresponding results for ABC are available in Li (2010b) or Li (2009a).

We also tested the statistical significance between AOSO and ABC. We assume the classification error rate is subject to some Binomial distribution. Let z denote the number of errors and n the number of tests, then the estimate of error rate $\hat{p} = z/n$ and its variance is $\hat{p}(1 - \hat{p})/n$. Subsequently, we approximate the Binomial distribution by a Gaussian distribution and perform a hypothesis test. The p values are reported in Table 3.

Table 6 Test classification errors on `M-Image`

	$v = 0.04$	$v = 0.06$	$v = 0.08$	$v = 0.1$
$J = 4$	5,539 **5,445**	5,480 **5,404**	5,408 **5,387**	5,430 **5,310**
$J = 6$	5,076 **4,909**	4,925 **4,851**	4,950 **4,801**	4,919 **4,822**
$J = 8$	4,756 **4,583**	4,748 **4,587**	4,678 **4,599**	4,670 **4,594**
$J = 10$	4,597 **4,436**	4,572 **4,445**	4,524 **4,448**	4,537 **4,467**
$J = 12$	4,432 **4,332**	4,455 **4,350**	4,416 **4,331**	4,389 **4,332**
$J = 14$	4,378 **4,264**	4,338 **4,277**	4,356 **4,240**	4,299 **4,245**
$J = 16$	4,317 **4,163**	4,307 **4,184**	4,279 **4,202**	4,313 **4,258**
$J = 18$	4,301 **4,119**	4,255 **4,176**	4,230 **4,167**	4,287 **4,188**
$J = 20$	4,251 **4,084**	4,231 **4,101**	4,214 **4,093**	4,268 **4,167**
$J = 24$	4,242 **4,049**	4,298 **4,077**	4,226 **4,067**	4,250 **4,120**
$J = 30$	4,351 **4,022**	4,307 **4,025**	4,311 **4,026**	4,286 **4,099**
$J = 40$	4,434 **4,002**	4,426 **4,033**	4,439 **4,047**	4,388 **4,090**
$J = 50$	4,502 **4,067**	4,534 **4,077**	4,487 **4,064**	4,479 **4,121**

In each J–v entry, the first is for ABC-LogitBoost and the second for AOSO-LogitBoost. Lower one is in bold

Table 7 Test classification errors on `Letter4k`

	$v = 0.04$	$v = 0.06$	$v = 0.08$	$v = 0.1$
$J = 4$	2,630 **2,515**	2,600 **2,414**	2,535 **2,414**	2,522 **2,392**
$J = 6$	2,263 **2,133**	2,252 **2,146**	2,226 **2,146**	2,223 **2,134**
$J = 8$	2,159 **2,055**	2,138 **2,046**	2,120 **2,046**	2,143 **2,055**
$J = 10$	2,122 **2,010**	2,118 **1,980**	2,091 **1,980**	2,097 **2,014**
$J = 12$	2,084 **1,968**	2,090 **1,965**	2,090 **1,965**	2,095 **1,995**
$J = 14$	2,083 **1,945**	2,094 **1,938**	2,063 **1,938**	2,050 **1,935**
$J = 16$	2,111 **1,941**	2,114 **1,928**	2,097 **1,928**	2,082 **1,966**
$J = 18$	2,088 **1,925**	2,087 **1,916**	2,088 **1,916**	2,097 **1,920**
$J = 20$	2,128 **1,930**	2,112 **1,917**	2,095 **1,917**	2,102 **1,948**
$J = 24$	2,174 **1,901**	2,147 **1,920**	2,129 **1,920**	2,138 **1,903**
$J = 30$	2,235 **1,887**	2,237 **1,885**	2,221 **1,885**	2,177 **1,917**
$J = 40$	2,310 **1,923**	2,284 **1,890**	2,257 **1,890**	2,260 **1,912**
$J = 50$	2,353 **1,958**	2,359 **1,910**	2,332 **1,910**	2,341 **1,934**

In each J–v entry, the first is for ABC-LogitBoost and the second for AOSO-LogitBoost. Lower one is in bold

7.1.2 Comparisons with SVM and deep learning

For some problems, we note LogitBoost (both ABC and AOSO) outperforms other state-of-the-art classifier such as SVM or Deep Learning.

On dataset `Poker`, Li (2009a) reports that linear SVM works poorly (the test error rate is about 40 %), while ABC-LogitBoost performs far better (i.e. <10 % on Poker25kT1 and Poker25kT2). AOSO-LogitBoost proposed in this paper has even lower test error than ABC-LogitBoost, see Table 3.

Table 8 Test classification errors on `Letter2k`

	$v = 0.04$	$v = 0.06$	$v = 0.08$	$v = 0.1$
$J = 4$	2,347 **2,178**	2,299 **2,162**	2,256 **2,179**	2,231 **2,189**
$J = 6$	2,136 **2,049**	2,120 **2,058**	2,072 **2,021**	2,077 **2,005**
$J = 8$	2,080 **1,975**	2,049 **1,977**	2,035 **1,985**	2,037 **1,997**
$J = 10$	2,044 **1,954**	2,003 **1,935**	2,021 **1,926**	2,002 **1,936**
$J = 12$	2,024 **1,924**	1,992 **1,905**	2,018 **1,916**	2,018 **1,906**
$J = 14$	2,022 **1,912**	2,004 **1,874**	2,006 **1,888**	2,030 **1,881**
$J = 16$	2,024 **1,889**	2,004 **1,899**	2,005 **1,888**	1,999 **1,899**
$J = 18$	2,044 **1,873**	2,021 **1,882**	1,991 **1,862**	2,034 **1,894**
$J = 20$	2,049 **1,870**	2,021 **1,875**	2,024 **1,884**	2,034 **1,862**
$J = 24$	2,060 **1,854**	2,037 **1,865**	2,021 **1,860**	2,047 **1,860**
$J = 30$	2,078 **1,851**	2,057 **1,865**	2,041 **1,860**	2,045 **1,867**
$J = 40$	2,121 **1,882**	2,079 **1,870**	2,090 **1,879**	2,110 **1,861**
$J = 50$	2,174 **1,915**	2,155 **1,915**	2,133 **1,851**	2,150 **1,868**

In each J–v entry, the first is for ABC-LogitBoost and the second for AOSO-LogitBoost. Lower one is in bold

Table 9 Test classification errors on `Poker25kT1`

	$v = 0.04$	$v = 0.06$	$v = 0.08$	$v = 0.1$
$J = 4$	90,323 1,02,905 **89,561**	**67,417** 71,450 69,674	**49,403** 51,226 56,702	**42,126** 42,140 47,165
$J = 6$	38,017 43,156 **34,460**	36,839 39,164 **32,716**	35,467 37,954 **31,714**	34,879 37,546 **31,399**
$J = 8$	39,220 46,076 **33,188**	37,112 40,162 **32,566**	36,407 38,422 **32,031**	35,777 37,345 **31,562**
$J = 10$	39,661 44,830 **34,203**	38,547 40,754 **33,251**	36,990 40,486 **32,873**	36,647 38,141 **32,316**
$J = 12$	41,362 48,412 **35,448**	39,221 44,886 **34489**	37,723 42,100 **33,641**	37,345 39,798 **33,241**
$J = 14$	42,764 52,479 **36,495**	40,993 48,093 **37,122**	40,155 44,688 **35,465**	37,780 43,048 **34,840**
$J = 16$	44,386 53,363 **39,087**	43,360 51,308 **38,111**	41,952 47,831 **37,766**	40,050 46,968 **37,275**
$J = 18$	46,463 57,147 **42,047**	45,607 55,468 **40,906**	45,838 50,292 **41,227**	43,040 47,986 **41,859**
$J = 20$	49,577 62,345 **43,394**	47,901 57,677 **44,614**	45,725 53,696 **43,065**	**44,295** 49,864 46,479

In each J–v entry, the first is for ABC-MART, the second for ABC-LogitBoost and the third for AOSO-LogitBoost. Lower one is in bold

On those `variants of Mnist` specially designed for testing various Deep Learning algorithms, Li reported ABC's results by simply setting $J = 20$, $v = 0.1$ and concluded that ABC are comparable to or better than Deep Learning. We provide AOSO's results in Table 4.

7.1.3 Detailed results

We provide one-on-one comparison between ABC and AOSO over a number of J–v combinations, as follows.

`Mnist10k`, `M-Image`, `Letter4k` and `Letter2k`: For these four datasets, classification errors are reported in Li (2010b) with every combination of $J \in \{4, 6, 8, 10, 12, 14, 16,$

Table 10 Test classification errors on `Poker25kT2`

	$\upsilon = 0.04$	$\upsilon = 0.06$	$\upsilon = 0.08$	$\upsilon = 0.1$
$J = 4$	**89,608** 1,02,014 90,120	**67,071** 70,886 69,921	**48,855** 50,783 57,028	**41,688** 41,551 47,447
$J = 6$	37,567 42,699 **34,602**	36,345 38,592 **32,982**	34,920 37,397 **31,840**	34,326 36,914 **31,645**
$J = 8$	38,703 45,737 **33,,456**	36,586 39,648 **32,883**	35,836 37,935 **32,288**	35,129 36,731 **31,790**
$J = 10$	39,078 44,517 **34,429**	38,025 40,286 **33,570**	36,455 40,044 **33,125**	36,076 37,504 **32,547**
$J = 12$	40,834 47,948 **35,725**	38,657 44,602 **34,773**	37,203 41,582 **33,930**	36,781 39,378 **33,417**
$J = 14$	42,348 52,063 **36,666**	40,363 47,642 **37,371**	39,613 44,296 **35,711**	37,243 42,720 **35,021**
$J = 16$	44,067 52,937 **39,238**	42,973 50,842 **38,208**	41,485 47,578 **37,851**	39,446 46,635 **37,454**
$J = 18$	46,050 56,803 **42,233**	45,133 55,166 **40,978**	45,308 49,956 **41,252**	42,485 47,707 **42,047**
$J = 20$	49,046 61,980 **43,506**	47,430 57,383 **44,754**	45,390 53,364 **43,317**	**43,888** 49,506 46,666

In each J–υ entry, the first is for ABC-MART, the second for LogitBoost and the third for AOSO-LogitBoost. Lower one is in bold

Table 11 Test classification errors on `Letter`

	$\upsilon = 0.04$	$\upsilon = 0.06$	$\upsilon = 0.08$	$\upsilon = 0.1$
$J = 4$	**125** 133	**121** 134	122 **121**	119 **114**
$J = 6$	112 **103**	107 **104**	**101** 106	**102** 109
$J = 8$	104 **98**	102 **100**	**93** 96	**95** 99
$J = 10$	101 **94**	100 **93**	**96** 98	**93** 88
$J = 12$	**96** 88	100 **91**	95 **92**	95 **92**
$J = 14$	**96** 88	98 **89**	**94** 88	89 **88**
$J = 16$	97 **88**	94 **93**	93 **90**	**95** 96
$J = 18$	95 **91**	**92** 95	96 **91**	93 **91**
$J = 20$	95 **90**	97 **89**	93 **87**	89 **92**

In each J–υ entry, the first is for ABC-LogitBoost and the second for AOSO-LogitBoost. Lower one is in bold

Table 12 Test classification errors on `Pendigits`

	$\upsilon = 0.04$	$\upsilon = 0.06$	$\upsilon = 0.08$	$\upsilon = 0.1$
$J = 4$	92 **86**	93 **85**	90 **84**	92 **85**
$J = 6$	98 **86**	97 **85**	96 **84**	93 **85**
$J = 8$	97 **84**	94 **84**	95 **84**	93 **86**
$J = 10$	100 **84**	98 **83**	97 **84**	97 **83**
$J = 12$	98 **84**	98 **82**	98 **83**	98 **83**
$J = 14$	100 **84**	101 **81**	99 **84**	98 **83**
$J = 16$	100 **83**	97 **84**	98 **82**	96 **82**
$J = 18$	102 **88**	97 **83**	99 **83**	97 **81**
$J = 20$	106 **83**	102 **82**	100 **83**	100 **83**

In each J–υ entry, the first is for ABC-LogitBoost and the second for AOSO-LogitBoost. Lower one is in bold

18, 20, 24, 30, 40, 50} and $\upsilon \in \{0.04, 0.06, 0.08, 0.1\}$. The comparison with AOSO-LogitBoost is listed in Tables 5, 6, 7 and 8.

Table 13 Test classification errors on `Zipcode`

	$\nu = 0.04$	$\nu = 0.06$	$\nu = 0.08$	$\nu = 0.1$
$J = 4$	**111** 104	108 **105**	114 **104**	107 **101**
$J = 6$	101 **99**	102 **100**	**98** 106	**99** 104
$J = 8$	**99 99**	**95** 98	**96** 102	98 **95**
$J = 10$	**97** 98	**94** 99	97 **96**	**94** 102
$J = 12$	**98** 99	98 **97**	**99** 100	**93** 100
$J = 14$	100 **96**	99 **98**	**97** 100	**92** 94
$J = 16$	98 **97**	**95** 97	99 **96**	**98** 103
$J = 18$	**96 96**	**98** 100	**101** 104	**8** 100
$J = 20$	**97** 98	**96** 97	100 **95**	**96** 99

In each J–ν entry, the first is for ABC-LogitBoost and the second for AOSO-LogitBoost. Lower one is in bold

Table 14 Test classification errors on `Isolet`

	$\nu = 0.04$	$\nu = 0.06$	$\nu = 0.08$	$\nu = 0.1$
$J = 4$	– 56	55 **54**	– 56	**55** 57
$J = 6$	– 55	59 **56**	– 54	58 **54**
$J = 8$	– 54	57 **54**	– 53	60 **55**
$J = 10$	– 54	61 **54**	– 55	62 **56**
$J = 12$	– 52	63 **54**	– 54	64 **50**
$J = 14$	– 48	65 **51**	– 54	60 **55**
$J = 16$	– 55	64 **57**	– 57	62 **58**
$J = 18$	– 55	67 **57**	– 53	62 **57**
$J = 20$	– 51	63 **57**	– 56	65 **55**

In each J–ν entry, the first is for ABC-LogitBoost and the second for AOSO-LogitBoost. Lower one is in bold. Dash means unavailable

Table 15 Test classification errors on `Optdigits`

	$\nu = 0.04$	$\nu = 0.06$	$\nu = 0.08$	$\nu = 0.1$
$J = 4$	**41** 43	42 **41**	**40 40**	41 **39**
$J = 6$	43 **40**	45 **39**	44 **39**	**38** 39
$J = 8$	44 **38**	44 **37**	45 **41**	45 **41**
$J = 10$	50 **37**	50 **38**	46 **36**	42 **38**
$J = 12$	50 **37**	48 **38**	47 **35**	46 **41**
$J = 14$	48 **38**	46 **41**	51 **40**	48 **40**
$J = 16$	54 **39**	51 **37**	49 **38**	46 **38**
$J = 18$	54 **41**	55 **38**	53 **40**	51 **41**
$J = 20$	61 **37**	56 **34**	55 **36**	55 **38**

In each J–ν entry, the first is for ABC-LogitBoost and the second for AOSO-LogitBoost. Lower one is in bold

`Poker25kT1 and Poker25kT2:` These are the only two datasets on which ABC-MART[4] outperforms ABC-LogitBoost in experiments by Li (2010b). Thus we cite the results for both ABC-MART and ABC-LogitBoost in Tables 9 and 10, with

[4] To put it simple, one gets MART (Multiple Additive Regression Trees Friedman 2001) by replacing the Hessian (23) with identity matrix. ABC-MART is its adaptive base class version (Li 2009).

Table 16 #trees added when convergence on selected datasets. R stands for the ratio AOSO/ABC

	Mnist10k	M-Rand	M-Image	Letter15k	Letter4k	Letter2k
ABC	7,092	15,255	14,958	45,000	20,900	13,275
R	0.7689	0.7763	0.8101	0.5512	0.5587	0.5424

Table 17 #trees added when convergence on `Mnist10k` for a number of J–v combinations

	$v = 0.04$	$v = 0.06$	$v = 0.1$
$J = 4$	90,000 1.0	90,000 1.0	90,000 1.0
$J = 6$	90,000 0.7740	63,531 0.7249	38,223 0.7175
$J = 8$	55,989 0.7962	38,223 0.7788	22,482 0.7915
$J = 10$	39,780 0.8103	27,135 0.7973	16,227 0.8000
$J = 12$	31,653 0.8109	20,997 0.8074	12,501 0.8269
$J = 14$	26,694 0.7854	17,397 0.8047	10,449 0.8160
$J = 16$	22,671 0.7832	11,704 1.0290	8,910 0.8063
$J = 18$	19,602 0.7805	13,104 0.7888	7,803 0.7933
$J = 20$	17,910 0.7706	11,970 0.7683	7,092 0.7689
$J = 24$	14,895 0.7514	9,999 0.7567	6,012 0.7596
$J = 30$	12,168 0.7333	8,028 0.7272	4,761 0.7524
$J = 40$	9,846 0.6750	6,498 0.6853	3,870 0.6917
$J = 50$	8,505 0.6420	5,571 0.6448	3,348 0.6589

For each J–v entry, the first number is for ABC, the second for the ratio AOSO/ABC

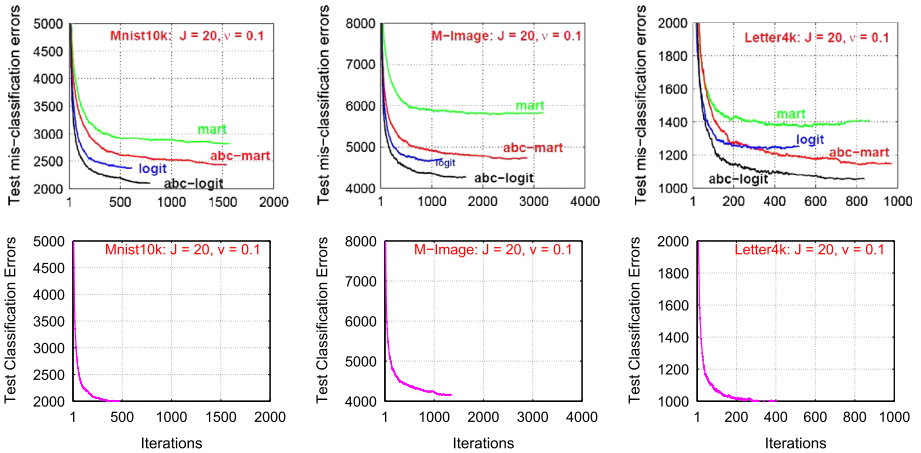


Fig. 5 Errors versus iterations on selected datasets and parameters. *Top row* ABC [copied from Li (2010b)]; *Bottom row* AOSO (*horizontal axis* scaled to compensate the $K - 1$ factor)

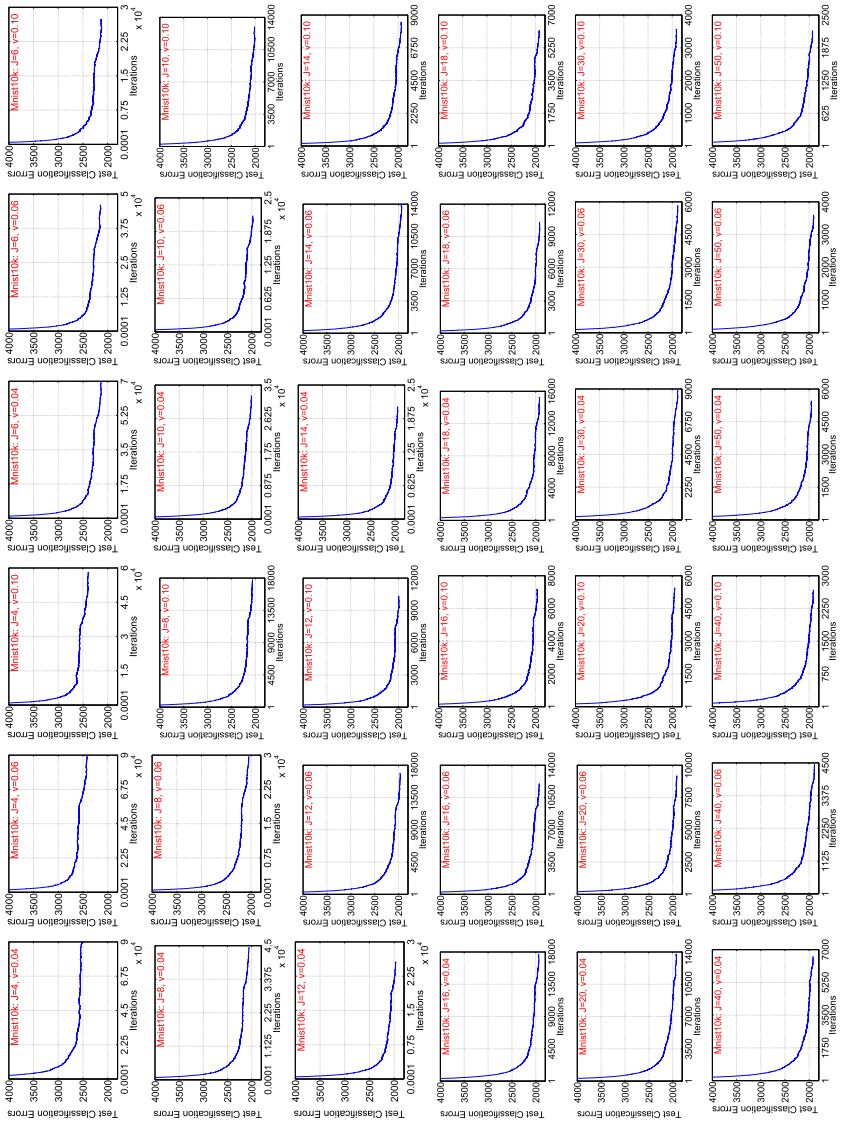


Fig. 6 Errors versus iterations on `Mnist10k`. $J \in \{4, 6, 8, 10, 12, 14, 16, 18, 24, 30, 40, 50\}$

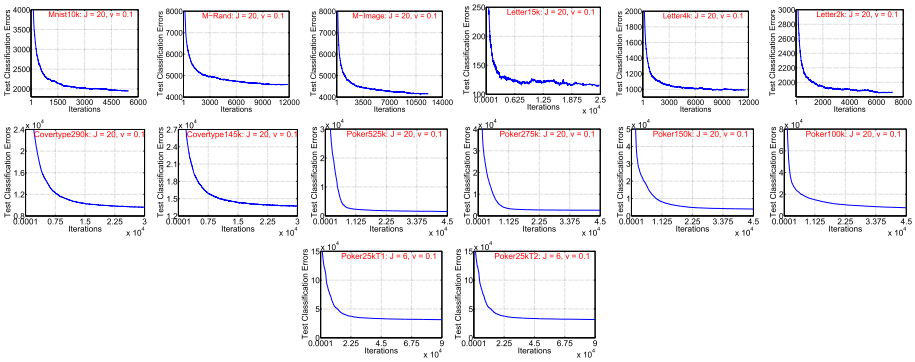


Fig. 7 Errors versus iterations on selected datasets and parameters

$J \in \{4, 6, 8, 10, 12, 14, 16, 18, 20\}$ and $v \in \{0.04, 0.06, 0.08, 0.1\}$. The comparison with AOSO-LogitBoost is also listed. Unlike on previous datasets, AOSO-LogitBoost is a bit sensitive to parameters, which is also observed for ABC-MART and ABC-LogitBoost by Li (2010b).

`Letter`, `Pendigits`, `Zipcode`, ***Isolet*** and ***Optdigits***: For these five datasets, classification errors are reported by Li (2010b) with every combination of $J \in \{4, 6, 8, 10, 12, 14, 16, 18, 20\}$ and $v \in \{0.04, 0.06, 0.08, 0.1\}$ (except that $v \in \{0.06, 0.1\}$ for `Isolet`). The comparison with AOSO-LogitBoost is listed in Tables 11, 12, 13, 14 and 15.

7.2 Convergence rate

Recall that we stop the boosting procedure if either the maximum number of iterations is reached or the training loss is small (i.e. the loss (1) $\leq 10^{-16}$). The fewer trees added when boosting stops, the faster the convergence and the lower the time cost for either training or testing. We compare AOSO with ABC in terms of the number of trees added when boosting stops for the results of ABC available in Li (2010b, 2009a). Note that simply comparing number of boosting iterations is unfair to AOSO, since at each iteration only one tree is added in AOSO and $K - 1$ in ABC.

Results are shown in Table 16 and Table 17. Except for when J–v is too small, or particularly difficult datasets where both ABC and AOSO reach maximum iterations, we found that trees needed in AOSO are typically only 50–80% of those in ABC.

Figure 5 shows plots for test classification error vs. iterations in both ABC and AOSO and show that AOSO's test error decreases faster. More plots for AOSO are given in Figs. 6 and 7. We only choose the datasets and parameters on which the plots of ABC are provided in Li (2010b).

7.3 Comparison between K-LogitBoost and AOSO-LogitBoost

In Fig. 8 we provide comparison between K-LogitBoost and AOSO-LogitBoost on several datasets. The parameters for both algorithms are $J = 20$, $v = 0.1$. As can be seen, K-LogitBoost converges slower than AOSO-LogitBoost, confirming our arguments in Sect. 6.3.

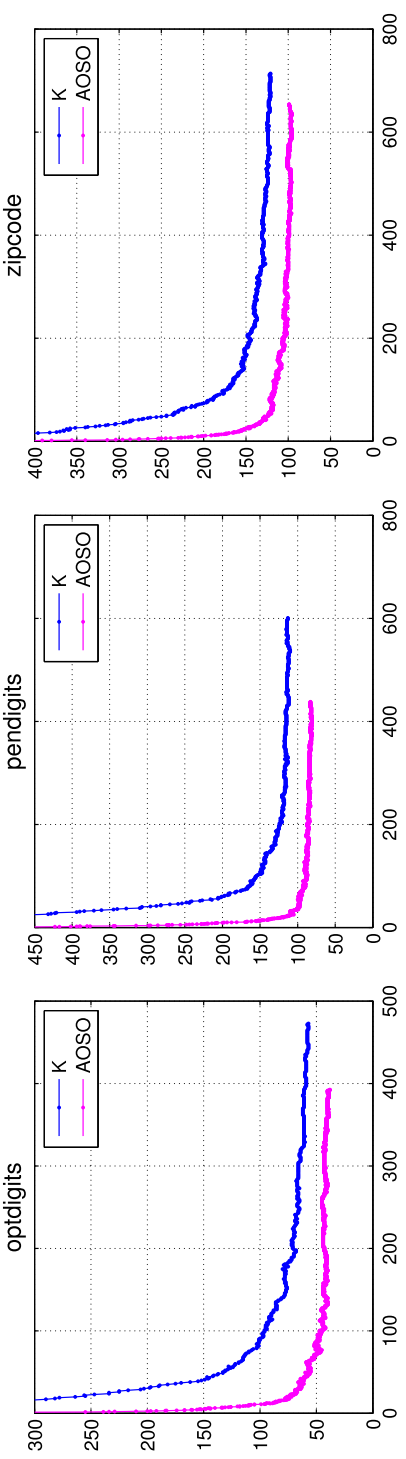


Fig. 8 Comparison between K-LogitBoost and AOSO-LogitBoost on datasets `Optdigits`, ***Pendigits***, ***Zipcode***, ***Letter2k***, ***Mnist10k***, ***M-Image*** with parameters $J = 20$ and $\upsilon = 0.1$: how the testing error decreases with iterations

8 Conclusions

We present an improved LogitBoost, namely AOSO-LogitBoost, for multi-class classification. Compared with ABC-LogitBoost, our experiments suggest that our adaptive class pair selection technique results in lower classification error and faster convergence rates.

Acknowledgments We appreciate Ping Li's inspiring discussion and generous encouragement. We also thank the MLJ anonymous reviewers and the action editor for their valuable comments. This work was supported by National Natural Science Foundation of China (61020106004, 61225008) and an Australian Research Council Discovery Early Career Research Award (DE130101605). NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

Appendix 1: Proof for property 2

First, we give without proof the lemma below that is well known in Matrix Analysis:

Lemma 1 *Let $\boldsymbol{x}$, $\boldsymbol{y}$ be K dimensional vector, $\boldsymbol{A} = \boldsymbol{x}\boldsymbol{y}^T$ be their out-product. Then the K eigen values for $\boldsymbol{A}$ are* $\{\boldsymbol{y}^T\boldsymbol{x}, \underbrace{0, \ldots, 0}_{K-1}\}$.

For positive semi definiteness

Let $\hat{\boldsymbol{Q}} = \text{diag}(\sqrt{p_{i,1}}, \ldots, \sqrt{p_{i,K}}), \boldsymbol{q} = (\sqrt{p_{i,1}}, \ldots, \sqrt{p_{i,K}})^T$. For any $\boldsymbol{x} \in \mathbb{R}^K$, let $\boldsymbol{y} = \hat{\boldsymbol{Q}}\boldsymbol{x}$. Then

$$\begin{aligned} \boldsymbol{x}^T \boldsymbol{H}_i \boldsymbol{x} &= \underbrace{\boldsymbol{x}^T \hat{\boldsymbol{P}} \boldsymbol{x}}_{\boldsymbol{y}^T\boldsymbol{y}} - \underbrace{\boldsymbol{x}^T \boldsymbol{p}\boldsymbol{p}^T \boldsymbol{x}}_{\boldsymbol{y}^T\boldsymbol{q}\boldsymbol{q}^T\boldsymbol{y}} \\ &= \boldsymbol{y}^T (I - \boldsymbol{q}\boldsymbol{q}^T)\boldsymbol{y}. \end{aligned} \tag{45}$$

From Lemma 1, we have that the eigen values for $\boldsymbol{q}\boldsymbol{q}^T$ are $\{1, \underbrace{0, \ldots, 0}_{K-1}\}$ by noting $\boldsymbol{q}^T\boldsymbol{q} = 1$. Then the eigen values for its first order matrix polynomial $\boldsymbol{I} - \boldsymbol{q}\boldsymbol{q}^T$ is $\{0, \underbrace{1, \ldots, 1}_{K-1}\}$, i.e. all the eigen values are non-negative. So it follows that the quadratic $\boldsymbol{y}^T(\boldsymbol{I} - \boldsymbol{q}\boldsymbol{q}^T)\boldsymbol{y} \geq 0$.

For the rank

First, assume the number of non-zero elements in $\boldsymbol{p}_i$ equals the number of classes: $\kappa = K$. We write $\boldsymbol{H}_i$ as

$$\boldsymbol{H}_i = \hat{\boldsymbol{P}}(\boldsymbol{I} - \boldsymbol{1}\boldsymbol{p}_i^T) \tag{46}$$

where $\hat{\boldsymbol{P}} = \text{diag}(p_{i,1}, \ldots, p_{i,K})$ is full rank. So we have

$$rank(\boldsymbol{H}_i) = rank(\boldsymbol{I} - \boldsymbol{1}\boldsymbol{p}_i^T) \tag{47}$$

Noting Lemma 1,the eigenvalues for $\boldsymbol{A} = \mathbf{1}\boldsymbol{p}_i^T$ are $\{1, \underbrace{0, \ldots, 0}_{K-1}\}$. It follows that the eigenvalues for its matrix (first order) polynomial $\boldsymbol{I} - \boldsymbol{A}$ is $\{0, \underbrace{1, \ldots, 1}_{K-1}\}$, which proves that $rank(\hat{\boldsymbol{P}} - \boldsymbol{p}_i \boldsymbol{p}_i^T) = K - 1$.

Then we assume $\kappa < K$. We collect the non-zero elements in $p_{i,k}$ to the upper left corner in matrix $\boldsymbol{H}_i$ and drop the corresponding zero elements by permutation. Repeat the procedure for $\kappa = K$ and we obtain that $rank(\boldsymbol{H}_i) = \kappa - 1$. Note that since $\boldsymbol{p}_i$ is a probability, $\kappa \geq 1$.

For eigenvector

It's straightforward by noting that $\boldsymbol{p}_i$ is a probability and thereby satisfies the sum-to-one constraint.

Appendix 2: Proof for Theorem 1 and 2

Conventions for matrix calculus

Before the proof, we introduce our conventions for matrix calculus, which are borrowed from Magnus and Neudecker (2007).

Let $\phi(\boldsymbol{x}) : \mathbb{R}^K \to \mathbb{R}$ be a scalar function defined for K-dim vector $\boldsymbol{x}$, its gradient is a K-dim vector $\nabla_{\boldsymbol{x}}\phi(\boldsymbol{x})$. In particular, we use the symbol D to represent the derivative, which is no more than the transpose of gradient $\mathsf{D}_{\boldsymbol{x}}\phi(\boldsymbol{x}) = (\nabla_{\boldsymbol{x}}\phi(\boldsymbol{x}))^T$ and thus a K-dim row vector. Without confusion from context, we sometimes abbreviate them as $\nabla\phi$ and $\mathsf{D}\phi$.

For the vector function $\xi(\boldsymbol{x}) : \mathbb{R}^K \to \mathbb{R}^K$, the matrix $\mathsf{D}\xi \in \mathbb{R}^{K\times K}$ is precisely the Jacobian matrix. Therefore, the Hessian matrix of scalar function $\phi()$ can be expressed by the Jacobian of its gradient $\nabla^2\phi = \mathsf{D}(\nabla\phi)$.

For Theorem 1

From first equation of (40) we have

$$\mathsf{D}\mathsf{H}(\boldsymbol{p}) + \boldsymbol{F}^T = -\lambda\mathbf{1}. \tag{48}$$

By applying derivative w.r.t $\boldsymbol{F}$ on both sides of the second equation of (40) we have

$$\mathbf{1}^T(\mathsf{D}\boldsymbol{p}) = 0. \tag{49}$$

By noting both (48) and (49), it follows that

$$\mathsf{D}\ell(\boldsymbol{F}) = (\mathsf{D}\mathsf{H}(\boldsymbol{p}))(\mathsf{D}\boldsymbol{p}) + (\boldsymbol{p} - \boldsymbol{\eta})^T(\mathsf{D}\boldsymbol{F}) + \boldsymbol{F}^T(\mathsf{D}\boldsymbol{p}) \tag{50}$$
$$\mathsf{D}\ell(\boldsymbol{F}) = -\lambda\mathbf{1}^T(\mathsf{D}\boldsymbol{p}) + (\boldsymbol{p} - \boldsymbol{\eta})^T \tag{51}$$
$$\mathsf{D}\ell(\boldsymbol{F}) = (\boldsymbol{p} - \boldsymbol{\eta})^T, \tag{52}$$

which completes the proof.

For Theorem 2

The $(K+1)\times(K+1)$ equations (40) defines an implicit function $\boldsymbol{p}=\boldsymbol{p}(\boldsymbol{F})\in\mathbb{R}^K$, $\lambda=\lambda(\boldsymbol{F})\in\mathbb{R}$. We apply to it the derivative w.r.t. $\boldsymbol{F}$ on both sides and have

$$\begin{cases}(\mathsf{D}\boldsymbol{F})+(\nabla^2\mathsf{H}(\boldsymbol{p}))(\mathsf{D}\boldsymbol{p})+\mathbf{1}(\mathsf{D}\lambda) & =0\\ \mathbf{1}^T(\mathsf{D}\boldsymbol{p}) & =0\end{cases} \tag{53}$$

Noting $\mathsf{D}\boldsymbol{F}=\boldsymbol{I}$ as well as the formula for inverse of block matrix, we obtain from (53) the following equation:

$$\begin{aligned}\begin{bmatrix}\mathsf{D}\boldsymbol{p}\\ \mathsf{D}\lambda\end{bmatrix}&=-\begin{bmatrix}\nabla^2\mathsf{H}(\boldsymbol{p}) & \mathbf{1}\\ \mathbf{1}^T & 0\end{bmatrix}^{-1}\begin{bmatrix}\boldsymbol{I}\\ 0\end{bmatrix}\\ &=-\begin{bmatrix}A^{-1}+A^{-1}\mathbf{1}(-\mathbf{1}^TA^{-1}\mathbf{1})\mathbf{1}^TA^{-1} & \times\\ \times & \times\end{bmatrix}\begin{bmatrix}\boldsymbol{I}\\ 0\end{bmatrix}\end{aligned} \tag{54}$$

where the abbreviation $A=\nabla^2\mathsf{H}(\boldsymbol{p})$ and the symbol "$\times$" means formula we don't care. It thus follows from above formula

$$\mathsf{D}\boldsymbol{p}=-(A^{-1}+A^{-1}\mathbf{1}(-\mathbf{1}^TA^{-1}\mathbf{1})\mathbf{1}^TA^{-1}). \tag{55}$$

In the meanwhile, by applying derivative to $\nabla\ell(\boldsymbol{F})$ in Theorem 1 we precisely have $\nabla^2\ell(\boldsymbol{F})=(\mathsf{D}\boldsymbol{p})^T$. This completes the proof.

References

Bertsekas, D. P. (1982). *Constrained optimization and Lagrange multiplier methods*. Boston: Academic Press.

Bottou, L., & Lin, C. J. (2007). Support vector machine solvers. In L. Bottou, O. Chapelle, D. DeCoste, & J. Weston (Eds.), *Large scale Kernel machines* (pp. 301–320). Cambridge: MIT Press. http://leon.bottou.org/papers/bottou-lin-2006

Freund, Y., & Schapire, R. (1995). A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory* (pp. 23–37). New York: Springer.

Friedman, J. (2001). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, *29*(5), 1189–1232.

Friedman, J., Hastie, T., & Tibshirani, R. (1998). Additive logistic regression: A statistical view of boosting. *Annals of Statistics*, *28*(2), 337–407.

Jaynes, E. (1957). Information theory and statistical mechanics. *The Physical Review*, *106*(4), 620–630.

Kégl, B., & Busa-Fekete, R. (2009). Boosting products of base classifiers. In *Proceedings of the 26th Annual International Conference on Machine Learning* (pp. 497–504). New York: ACM.

Kivinen. J., & Warmuth, M. K. (1999). Boosting as entropy projection. In *Proceedings of the Twelfth Annual Conference on Computational Learning Theory* (pp. 134–144). New York: ACM.

Lafferty, J. (1999). Additive models, boosting, and inference for generalized divergences. In *Proceedings of the Twelfth Annual Conference on Computational Learning Theory* (pp. 125–133).

Larochelle, H., Erhan, D., Courville, A., Bergstra, J., & Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th International Conference on Machine learning* (pp. 473–480). New York: ACM.

Li, P. (2008). Adaptive base class boost for multi-class classification. *Arxiv* preprint arXiv:08111250.

Li, P. (2009a). Abc-boost: Adaptive base class boost for multi-class classification. In *Proceedings of the 26th Annual International Conference on Machine Learning* (pp. 625–632). New York: ACM.

Li, P. (2009b). Abc-logitboost for multi-class classification. *Arxiv* preprint arXiv:09084144.

Li, P. (2010a). An empirical evaluation of four algorithms for multi-class classification: Mart, abc-mart, robust logitboost, and abc-logitboost. *Arxiv* preprint arXiv:10011020.

Li, P. (2010b). Robust logitboost and adaptive base class (abc) logitboost. In *Conference on Uncertainty in Artificial Intelligence*.

Magnus, J. R., & Neudecker, H. (2007). Matrix differential calculus with applications in statistics and econometrics (3rd ed.). New York: Wiley.

Masnadi-Shirazi, H., & Vasconcelos, N. (2010). Risk minimization, probability elicitation, and cost-sensitive svms. In *Proceedings of the International Conference on Machine Learning* (pp. 204–213).

Reid, M. D., & Williamson, R. C. (2010). Composite binary losses. *The Journal of Machine Learning Research*, *11*, 2387–2422.

Schapire, R., & Singer, Y. (1999). Improved boosting algorithms using confidence-rated predictions. *Machine learning*, *37*(3), 297–336.

Shen, C., & Hao, Z. (2011). A direct formulation for totally corrective multi-class boosting. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 2585–2592).

Shen, C., & Li, H. (2010). On the dual formulation of boosting algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *32*(12), 2216–2231.

Zou, H., Zhu, J., & Hastie, T. (2008). New multicategory boosting algorithms based on multicategory fisher-consistent losses. *The Annals of Applied Statistics*, *2*(4), 1290–1306.