Neural Processing Letters (2019) 50:957–987
https://doi.org/10.1007/s11063-019-09999-3

On Incremental Learning for Gradient Boosting Decision Trees

Chongsheng Zhang[1] · Yuan Zhang[1] · Xianjin Shi[1,2] · George Almpanidis[1] · Gaojuan Fan[1] · Xiajiong Shen[1]

Published online: 13 February 2019

Abstract
Boosting algorithms, as a class of ensemble learning methods, have become very popular in data classification, owing to their strong theoretical guarantees and outstanding prediction performance. However, most of these boosting algorithms were designed for static data, thus they can not be directly applied to on-line learning and incremental learning. In this paper, we propose a novel algorithm that incrementally updates the classification model built upon gradient boosting decision tree (GBDT), namely iGBDT. The main idea of iGBDT is to incrementally learn a new model but without running GBDT from scratch, when new data is dynamically arriving in batch. We conduct large-scale experiments to validate the effectiveness and efficiency of iGBDT. All the experimental results show that, in terms of model building/updating time, iGBDT obtains significantly better performance than the conventional practice that always runs GBDT from scratch when a new batch of data arrives, while still keeping the same classification accuracy. iGBDT can be used in many applications that require in-time analysis of continuously arriving or real-time user-generated data, such as behaviour targeting, Internet advertising, recommender systems, etc.

Keywords Gradient boosting · Gradient boosting decision tree · Incremental learning · Ensemble learning

✉ Xianjin Shi
shixj@haedu.gov.cn

Chongsheng Zhang
chongsheng.zhang@yahoo.com

Yuan Zhang
1124840343@qq.com

George Almpanidis
almpanidis@gmail.com

Gaojuan Fan
fangaojuan@126.com

Xiajiong Shen
shenxj@henu.edu.cn

[1] Henan University, Kaifeng 475001, China

[2] Education Information Centre of Henan Province, Zhengzhou 450018, China

1 Introduction

Boosting [9] is a class of ensemble learning algorithms that combine multiple weak learners, such as simple "IF ELSE" prediction rules, to form a stronger model with better prediction accuracy. In the case of classification, a typical and well-known algorithm is AdaBoost [10], which combines multiple weak learners trained on internally weighted samples. Here, samples are weighted according to the difficulty of being correctly predicted; samples that are correctly predicted will be given decreased weights, while wrongly predicted ones will be assigned increased weights. Gradient Boosting is a well-known Boosting algorithm that sequentially builds a great many decision tree models based on previous models' residuals (i.e. prediction errors, which are measured by certain kind of loss function), then ensembles these models to form a stronger predictor.

Gradient Boosting algorithms have been widely used to solve both regression and classification problems [5], with Gradient Boosting Decision Tree (GBDT) being the main representative. GBDT has shown excellent performance in many real-world applications [23]. However, this algorithm was originally developed for static data, that is, the classification model is trained over a static, fixed size of data. Yet, for many applications that need to model and analyse dynamically arriving data, it would be very time-demanding and impractical to run GBDT from scratch, when a new batch of data comes. Hence, incremental learning techniques for GBDT is indispensable for applications that need to handle/analyse the dynamically arriving data.

The goal of this work is to extend/adapt GBDT to the incremental learning setting, where new samples are continuously arriving in batch. Various investigations on incremental learning have been done in the literature [8,13,14,21]. Depending on the granularities of the newly arrived data to be processed, existing incremental learning methods can be roughly divided into two categories: methods that process new data in batch (a batch may contain many newly arrived samples), and those that process each newly arrived instance. Grbovic and Vucetic [13] proposed the IBoost algorithm, the goal of which is to quickly build a model in an incremental learning manner using the AdaBoost algorithm. The authors in [8] proposed the VFDT algorithm that is an on-line incremental learning algorithm based upon decision trees; it can build an ensemble of decision tree model for streaming data.

In this paper, we propose an incremental learning method for GBDT, namely iGBDT. It handles newly arrived data in batch. It extends the conventional GBDT algorithm to analyse on-the-fly the data that is incrementally arriving in batch. Whenever each new batch of data arrives, iGBDT sequentially checks each decision tree built by GBDT in a top-down manner to see whether the best attribute for each node in the decision tree remains the same, with the arrival of the new batch of data. If the answer is positive, then there is no need to rebuild the current decision tree. This "lazy" updating strategy can incrementally and rapidly update the existing classification model built by GBDT, but without running it from scratch, yet still achieves the same prediction accuracy.

The main contributions of this work are summarized as follows:

– We propose the iGBDT algorithm, which extends the conventional GBDT algorithm to analyse on-the-fly the dynamically arriving data. iGBDT always tries to "lazily" update the classification model which may contain thousands of decision trees, while avoiding rebuilding it from scratch. Our proposal greatly saves the time for rebuilding a new GBDT model whenever new batch of data arrives, it is thus significantly more efficient than the conventional (straightforward) solution for continuously arriving data based on GBDT.

– We conduct experiments on various data sets to validate the effectiveness and efficiency of iGBDT. The results show that iGBDT obtains significantly better performance than the straightforward GBDT method for continuously arriving data, in terms of model building/updating time, but without sacrificing the prediction accuracy.

The remainder of the paper is organised as follows. In Sect. 2, we briefly summarise the state-of-the-art incremental learning methods for classification. In Sect. 3, we present the gradient boosting decision tree, and provide theoretical analysis on it. In Sect. 4, we propose iGBDT, the incremental gradient boosting decision tree. In Sects. 5 and 6, we conduct experiments on iGBDT and the conventional GBDT algorithms, then analyse the results. We draw a conclusion in Sect. 7, where we will also discuss the future work.

2 Related Work

There are many research works on on-line and incremental learning that are based on boosting. Before we introduce the related work in the domain, it should be noted that incremental learning is different from on-line learning, although they share some similarities. Both of them aim at learning (updating) a model when the data comes on the fly to obtain the same model as the one learned in a batch setting (i.e. on static data). The difference is that on-line learning learns a model when the training instances arrive sequentially one by one (1-by-1), whereas incremental learning updates a model when a new batch of data instances arrive. The comparisons between on-line learning and incremental learning are listed in Table 1.

It should be noted that, all the existing on-line learning frameworks can be used for incremental learning as well, because on-line learning algorithms can process the batch of new data 1-by-1.

However, on-line learning methods are usually unable to fully optimise the cost function defined on the training instances, given the processing speed (efficiency) requirements. Given a static data set, conventional classification algorithms (batch algorithms) such as GBDT, Random Forests (RF), Support Vector Machine (SVM) usually achieve better accuracy performance than the on-line learning methods, at the cost of iterative scanning and computation for completely optimizing the cost function defined on the set of training examples.

Therefore, on-line learning methods commonly have very excellent efficiency advantages, while the major batch algorithms (such as GBDT and RF) can commonly achieve better accuracy performance [23].

Boosting is a very useful ensemble framework machine learning. The gradient boosting decision tree (GBDT) is a representative boosting algorithm. It builds multiple decision trees

Table 1 Similarities and differences between on-line learning and incremental learning for boosting algorithms

	Related work	Differences	Similarities
On-line learning	[5–7,9–11,13–15,21]	Update the model whenever a new instance arrives	Learn an update-to-date model on the fly when new instances are constantly arriving
Incremental learning	Our work in this paper	When a batch of new instances arrive, learn/update the existing model rapidly	

sequentially, where each decision tree is built on the residual from the previous decision tree. GBDT has proven to be highly accurate in many real-world data classification applications.

While GBDT is highly accurate in prediction, it suffers from being hard to parallelise. On the other hand, bagging methods can be easily implemented in a parallel manner. To this end, the authors in [19] propose a hybrid algorithm, BagBoo, that combines both boosting and bagging, by building a gradient boost decision tree with 10–20 trees in each bag. This enables BagBoo to benefit from the advantages of both bagging and boosting and yield better prediction results but with less running time.

On-line learning for Boosting The works in [18] and [17] present an on-line boosting method and compares it with the baseline batch algorithm experimentally, in terms of accuracy and running time, showing that this on-line algorithm achieves better efficiency performance than the batch algorithm. The authors in [2] develop a boosting framework that can be used to derive on-line boosting algorithms with different baseline methods, including Logistic Regression, Least Squares Regression, and Multiple Instance Learning. Their experiments show that the corresponding on-line boosting algorithms obtain better performance than the baseline batch algorithm, in terms of accuracy and convergence. The study in [3] adapts the research in [24] to the on-line setting by speeding up the convergence of boosting. The work in [4] proposes two on-line boosting algorithms; Online Boost-By-Majority (BBM), which improves upon the work of Chen et al. in [6], and AdaBoost.OL. Their experiments show that Online BBM is good at picking the optimal number of weak learners for achieving a specified accuracy, while AdaBoost.OL is adaptive and parameter-free, but not optimal. The authors in [20] present a new on-line boosting algorithm for updating the weights of a boosted classifier, which yields a closer approximation to AdaBoost algorithm than previous on-line boosting algorithms. They develop a fast and accurate on-line learning algorithm, which sequentially updates its confidence in a set of weak hypotheses.

On-line learning for Gradient Boosting The work in [15] is an early work for on-line gradient boosting that solves the robustness problem of boosting in on-line settings. The authors in [16] design an on-line boosting method for gradient feature selection for pedestrian detection and tracking.

Difference between on-line learning and data streaming methods Online learning and incremental learning are different from data stream classification in that data streaming techniques often need to "forget" the old data, since they can only process a fixed size of the latest data in order to obtain the analytical results in real-time. Detecting concept drift is often an important issue in data stream mining. But for on-line learning and incremental learning, the new instances are usually considered to be equally important as the old ones, therefore it is essential to keep the "old" instances.

In the literature, there have been extensive studies on data streaming classification [1,12]. In the following, we briefly present two well-known data streaming techniques for classification. Grbovic et al. in [13] present IBoost, which is a boosting algorithm based on AdaBoost that fits to data stream. In order to solve the problem of classifying fast data streams, [8] presents a novel algorithm, named VFDT, that adaptively learns (or updates) a model over the streaming data.

To summarise, a great many research efforts have been put in developing on-line boosting methods, yet there is little study that addresses on-line and incremental gradient boosting. This motivates us to investigate the incremental learning techniques for gradient boosting decision trees.

3 GBDT Algorithm

In this section, we outline the theoretical background of GBDT and demonstrate in detail the steps of the algorithm using a toy example.

3.1 A Brief Introduction to the GBDT Algorithm

Gradient boosting decision tree (GBDT) is a boosting method among the best performers in data classification. In order to understand GBDT, we need to first understand Gradient Boosting (GB).

GB is a framework for boosting. The main idea is to sequentially build each decision tree model on the gradient descent direction of a loss function, based on the residual (the difference between predicted value and the true value of each instance) from previous models. The loss function describes the accuracy of the models, the greater the value of the loss function, the worse the prediction ability of the trained models. If the value of the loss function decreases with the addition of new models, then the prediction power of these models improves. The expected way is to let the value of the loss function decline in the direction of its gradient descent. Different loss functions can be used here, such as the least square loss function, the least absolute deviation and the *Huber* loss functions. GB can be used to solve the regression problem as well as the binary classification problem. The pseudo code of Gradient Boosting is given below [11], in Algorithm 1.

Algorithm 1 Gradient Boosting

1: $F_0(\boldsymbol{x}) = argmin_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$
2: **for** m=1 to M **do**
3: $\quad \widetilde{y}_i = -[\frac{\partial L(y_i, F(\boldsymbol{x_i}))}{\partial F(\boldsymbol{x_i})}]_{F(\boldsymbol{x})=F_{m-1}(\boldsymbol{x})} \qquad i = 1, N$
4: $\quad$ build the m^{th} prediction model $h(x; a_m)$
5: $\quad \boldsymbol{a_m} = argmin_{\boldsymbol{a},\zeta} \sum_{i=1}^{N} [\widetilde{y}_i - \zeta h(\boldsymbol{x_i}; \boldsymbol{a_m})]^2$
6: $\quad \gamma_m = argmin_\gamma \sum_{i=1}^{N} L(y_i, F_{m-1}(\boldsymbol{x_i}) + \gamma h(\boldsymbol{x}; \boldsymbol{a_m}))$
7: $\quad F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + p_r \gamma_m h(\boldsymbol{x}; \boldsymbol{a_m}) \quad 0 < p_r \leq 1$
8: **end for**

In line 1, GB first initialises the model using a very simple method, where, for $F_0(x)$, the minimum or the mean of the class label values of all the instances is commonly used. Next, in M iterations it trains M models (in total) in the *for* loop. M can be considered a hyperparameter of the algorithm; increasing M reduces the error on the training set, but setting it too high can lead to overfitting. For every (imperfect) model m, in line 3, it computes the pseudo-residuals $\widetilde{y}_i$, i.e. the negative direction gradient value through the loss function ($\sum_{i=1}^{N} L(y_i, \gamma)$) according to the $m-1$ models already learned, then builds the mth prediction model (base learner, e.g. a tree) $h(\boldsymbol{x}; \boldsymbol{a_m})$ (line 4) and obtains its parameters ($\boldsymbol{a_m}$) by fitting it to the pseudo-residuals, using the least square method to make sure the new model, $h(\boldsymbol{x_i}; \boldsymbol{a_m})$, can achieve a minimum value in the gradient direction (line 5). Next, it computes the coefficient/multiplier of the new model using the loss function by solving the one-dimensional optimisation problem of line 6. Lastly, with the decision model $h(\boldsymbol{x}; \boldsymbol{a_m})$, GB updates the model ($F_m(\boldsymbol{x})$) through linear superposition (line 7). A hyperparameter p_r (learning rate) in the model update rule of line 7 controls the regularisation by shrinkage of

GB. Small learning rates typically improve the model's generalisation ability over gradient boosting without shrinking at the price of increasing training and querying time.

If decision trees are used as the weak learners for Gradient Boosting, the corresponding algorithm is called Gradient Boosting Decision trees (GBDT). In this paper, we use the least square loss function and the Classification And Regression Tree (CART) to implement the GBDT algorithm. It is worth noting that, here, CART will be used for regression, instead of classification. The pseudo code of GBDT is given below, in Algorithm 2.

Algorithm 2 The GBDT algorithm under the least square loss function

1: $F_0(\boldsymbol{x}) = \overline{y}$
2: **for** m=1 to M **do**
3: $\widetilde{y_i} = y_i - F_{m-1}(\boldsymbol{x_i}), \quad i = 1, N$
4: $(\gamma_m, \boldsymbol{a_m}) = argmin_{\boldsymbol{a},\gamma} \sum_{i=1}^{N} [\widetilde{y_i} - \gamma h(\boldsymbol{x_i}; \boldsymbol{a})]^2$
5: $F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + p_r \gamma_m h(\boldsymbol{x}; \boldsymbol{a_m})$
6: **end for**

In line 1, GBDT first initialises the model with the mean value of the class values of all the instances. Then $M = p_t$ models (trees) of depth $\leq p_d$ are sequentially learned in M steps using the *for* loop. In line 3, under the least square loss function, GBDT computes the residual of the existing $m - 1$ models, which have already been learned so far. With the above residual as the class values of the corresponding instances in the training data, it trains the mth decision tree model using CART. Since this a case of tree models, a different multiplier γ_m is computed for every leaf. Then, by incorporating the new decision tree model, it updates the ensemble model $F_{m-1}(\boldsymbol{x})$ to form the new ensemble model $F_m(\boldsymbol{x})$, through linear superposition (line 5).

For clearness, in Table 2, we summarise the notation used for GB and GBDT (Algorithms 1 and 2 respectively) throughout the paper.

3.2 A Toy Example for Building GBDT

We now give a toy example to demonstrate how GBDT builds the first two decision trees, where the second one is trained on the residual of the first decision tree. Table 3 describes the instances (samples) in the training set. In this table, "online time" denotes the daily time span when an Internet user is online, "play time" represents the amount of time when a user plays games, "income" attribute is the monthly income information of the user. Finally, the age attribute is the target (class) to be predicted.

In Fig. 1, we build the first decision tree using CART, given a user-specified tree depth of 3. In the root node, we first compute the mean class value (the value of the "age" attribute) of all the instances, which is 38. Next, we compute the best split attribute according to the techniques in Sect. 3.3, where we find "online time" to be the best split attribute. So we horizontally split the data into left and right parts, where three instances go to the left child and one instance (i.e. the instance of [1 0 2000 65]) goes to the right child. For the left child (its mean class value is 29), we continue finding the best split attribute, which here is "play time". Then again, we split the data on this node into left and right parts. This process is repeated iteratively until the maximum depth of the tree is reached, or there is only one instance in the node, or all the instances under the current (sub)node have the same value in the target class. The values under each leaf node (e.g. "[0]", "[−2, 2]") represent the residual error of this tree, which is calculated by subtracting the class value of each instance in the

Table 2 List of notations in this work

Symbol	Notation
x	Input data
y	Output data variable
N	Size of the data
M	Number of iteration steps = number of models to be trained. In the case of GBDT $M \equiv p_t$
m	Current iteration/model
F	An approximation to the function (model) that minimizes the expected value of loss function L
F_0	A constant function that corresponds to the starting model, which is incrementally expanded in a greedy fashion
F_m	An imperfect/weak/approximation model at iteration m
a_m	Parameters for model F_m
L	The loss function that describes the accuracy of the ensemble model
$h(x; a_m)$	A base/weak learner (prediction/decision model) at iteration m
γ_m	Step magnitude multiplier/coefficient. Corresponds to the weights of base learners h_m used to approximate the model
Hyperparameters	
p_r	Learning rate; controls shrinkage, $0 < r \leq 1$
p_d	Maximum depth of decision trees in GBDT and iGBDT
p_t	Number of trees in GBDT and iGBDT
p_b	Batch size (transactions per batch) in iGBDT

Table 3 Original instances of the training set

Online time	Play time	Income	Age
2	1	5000	30
5	3	3000	23
1	0	2000	65
3	1	4000	34

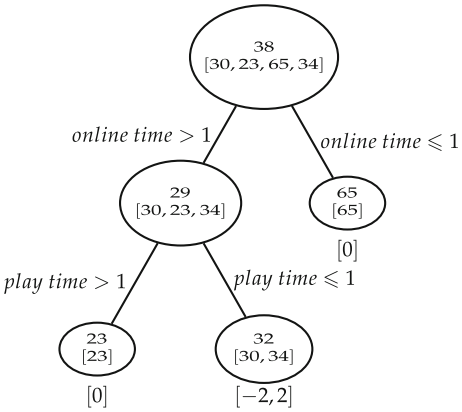

Fig. 1 The process of building the first decision tree

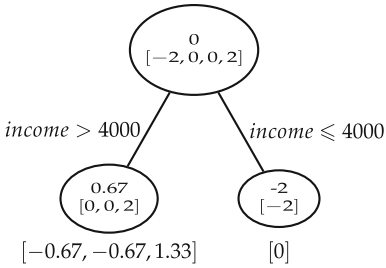

Fig. 2 The second decision tree built by GBDT

Table 4 The updated training set for building the second decision tree

Online time	Play time	Income	Age
2	1	5000	−2
5	3	3000	0
1	0	2000	0
3	1	4000	2

node and the corresponding mean class value. These residual values (which are [−2 0 0 2] in this example) will be used as the target to be predicted by the second decision tree.

The second decision tree is shown in Fig. 2. Here, the class values are [−2 0 0 2], which are the residual error values from the first decision tree. As can be seen in Table 4, we replace the "age" attribute values in Table 3 by [−2 0 0 2], while the values for other attributes remain the same. Upon Table 4, we build the second decision tree. The steps are similar to the first decision tree and the corresponding decision tree is depicted in Fig. 2.

When all the model building steps are completed, we can predict the "age" value of a new instance. Given a new instance [2, 0.8, 3200], we will predict its "age" value using GBDT. We first use the decision tree in Fig. 1 to predict the "age" value of the new instance. Given that "online time" being 2, and "play time" being 0.8, this new instance will be predicted to be 32 years old.

We next use the second decision tree given in Fig. 2; the corresponding value for this instance (note: it is the predicted residual value of the first decision tree, but not the real "age" value) will be predicted as -2, since the income of this test instance is 3200. Given a learning rate of $r = 1$, the final "age" value to be predicted is $32 + 1 \times (-2) = 30$. It should be noted that r is a user-specified value in the range of (0,1]. If the learning rate is 0.3, then predicted "age" value will be 32 (i.e. the ceiling of 31.7).

3.3 Finding the Best Split Attribute

In the process of building the GBDT model, the key point is to find the best split attribute which makes sure the square loss error is minimised. This means that the best splitting attribute and its splitting point (value) should let the gradient decline at the fastest speed. By each splitting, instances are classified into more accurate regions. The effect of prediction is thus gradually optimised. Figure 3 illustrates the process of finding the best split attribute and the corresponding splitting point.

In Fig. 3, let g be the splitting point (value) of an attribute, μ_l (resp. μ_r) be the mean value of the class values of all the instances on the left (resp. right) node, and l (resp. r) be number of instances on the left (resp. right) node after splitting. The definitions of μ_l and μ_r are given below.

$$\mu_l = \frac{1}{l}\sum_{i=1}^{l} y_i \qquad \mu_r = \frac{1}{r}\sum_{j=1}^{r} y_j$$

The sum of square loss error on the left node can be derived using the following formula.

$$s_l = argmin \sum_{i=1}^{l} (y_i - \mu_l)^2 \tag{1}$$

Similarly, the sum of least square loss error on the right node is given in Formula 2.

$$s_r = argmin \sum_{i=1}^{r} (y_i - \mu_r)^2 \tag{2}$$

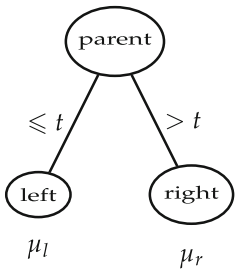

Fig. 3 Finding the best split attribute and point for GBDT

In order to minimise the sum square loss error of a split, the predicted value must be close to the average of the class values of all the instances (on the left or right child). The objective is to minimise the following formula.

$$argmin\left[\sum_{i=1}^{l}(y_i-\mu_l)^2+\sum_{i=1}^{r}(y_i-\mu_r)^2\right] \tag{3}$$

For each attribute, we first sort the instances by the attribute, we next try each value as the split point (value), then use Formula 3 to calculate the corresponding sum square loss error. Finally, we pick the split point that achieves the minimum sum square loss error for this attribute.

After the best split points for all the attributes have been computed, the attribute that has the smallest sum least square loss error will be chosen as the best split attribute, the corresponding best split point (value) will be used to horizontally split the data into the left and right parts. Then the process for finding the best split attribute (and point) for a given node in GBDT finishes.

4 iGBDT Algorithm

When the data instances are continuously arriving in batch, it is clear that the ensemble model built by GBDT needs to be constantly updated to guarantee the performance (accuracy) of the classification model. However, the common practice for real-world applications often combines newly arrived data with the previous training set, then learns a new prediction model from scratch using GBDT. This approach is called *straightforward GBDT* hereafter. It can yield the expected GBDT model but at the cost of overly large amount of computation time when the new batches of data are constantly arriving.

To solve this problem, we propose a new method that extends GBDT to the incremental setting mentioned above. Our method is referred to as *iGBDT*. It is able to learn a GBDT model incrementally when new batches of data instances are arriving, but it does not always require running GBDT from scratch. In a nutshell, the learning-from-scratch straightforward GBDT framework for incremental learning is a brute-force method: whenever a new batch of data instances arrives, this method always merges the new data with the original training set, then rebuilds a new GBDT model from the very beginning.

Unlike the on-line learning techniques, the proposed iGBDT method processes all the new input (the batch of new data) in the same time; it can not afford processing the new input 1-by-1, since this will be very time-demanding for the GBDT (iGBDT) algorithm.

In essence, iGBDT aims at "lazily" updating the old GBDT ensemble model (which may contain thousands of decision trees) to form a new model, instead of building it from scratch. iGBDT includes the following steps, which are given in the following subsections.

4.1 Resorting All the Attributes to be Reused by All the Decision Trees in iGBDT

According to the algorithm of GBDT, when finding the best split attribute and split value, each attribute should be sorted in advance by its values. In iGBDT, the previously sorted data set for each attribute will be reused by all the decision trees, thus this sorting is an once-for-all computation process in iGBDT. This process is shown in Fig. 4.

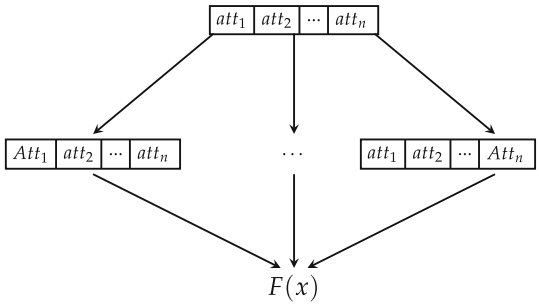

Fig. 4 Resorting the new data according to each attribute in iGBDT

In Fig. 4, $att_1, att_2 \ldots att_n$ are the attributes of training data set. After a new batch of data arrives, $Att_1, att_2 \ldots att_n$ refers to the new training set sorted by att_1. Similarly, $att_1, att_2 \ldots Att_n$ denotes the new training set sorted by att_n.

In the sorting process, we apply the "sort-merge" method to speed up the sorting process, which is described below.

4.2 Using the Sort-Merge Strategy to Merge Two Sorted Sets

Given the original training set of an attribute which has already been sorted ascendingly, iGBDT first sorts the instances in the new batch by the same attribute, then applies the "sort-merge" strategy to merge the two sorted sets into one ordered set.

iGBDT merges the original data D1 and the new batch data D2 (both ordered according to attribute k) to obtain an ordered data set D3. So the number of samples in D3 $=$ D2 $+$ D1

When using the "sort-merge" method, iGBDT sets two pointers, $p1$ which initially points to the first element of the originally sorted set, and $p2$ points to the sorted list of the new batch for the same attribute. Then iGBDT iteratively performs the following process, in each round it compares the two values of the elements which $p1$ and $p2$ refer to:

- if the later is smaller than the former, then it outputs the value of $p2$ and moves $p2$ to the next element in the ordered set for the new batch;
- otherwise, it first outputs $p1$, then moves $p1$ to the next element in the original set.

With the "sort-merge" method, iGBDT only needs to scan two ordered sets once, thus it greatly reduces the amount of time needed in re-sorting the whole data, especially when the original ordered set is much larger than the new batch of data.

4.3 Checking the Best Split Attribute to "Lazily" Update the Ensemble Model

As new batches of data are continuously arriving, the ensemble model built by GBDT which may contain thousands of decision trees should be timely updated so as to accurately model/reflect the complete data which contains the newly arrived instances.

iGBDT first uses the techniques in Sect. 4.2 to promptly reorder the complete data containing the new batch of instances. Next, based on the newly ordered data, iGBDT will check and update the existing GBDT ensemble model.

As described in Sect. 3, the decision trees in GBDT are built in sequence: when building the current decision tree, GBDT needs the residual (prediction error) of the ensemble of all the previous decision trees, as the target (class value) to be predicted. Therefore, in order to

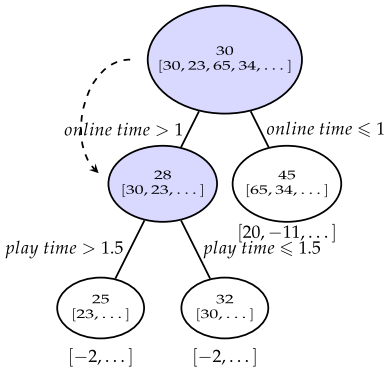

Fig. 5 An example where iGBDT only updates the auxiliary node information, but does not have to rebuild the tree

update the GBDT ensemble model, we will need to sequentially check each decision tree model.

Starting from the first decision tree, iGBDT computes, according to Formula 3, the best split attribute (and the corresponding best split point/value) for the newly ordered data, then checks if this attribute is the same as the one on the root node of the first decision tree. In other words, iGBDT checks if the attribute on the root node remains the best split attribute for the newly ordered data. If yes, then iGBDT only updates the auxiliary information of the node, mainly the mean class value and best split point. Then, it horizontally splits the newly ordered data into left and right parts.

Next, in a depth-first manner, iGBDT recursively checks the child nodes in a way similar to the root node, i.e. computes the best split attribute for the new data allocated to the current node, then checks if this attribute is the same as the one kept on this node. If yes, it then updates the auxiliary information of the current node, then continues checking its child nodes, until all the nodes of this decision tree have been checked/updated. If no, which means the best split attribute has changed, then from the current node, iGBDT rebuilds the sub-tree under this node with the new allocated data. The process of updating the auxiliary information of a node is shown in Fig. 5, using the same toy example of Sect. 3.

Once the checking/updating of the first decision tree is finished, iGBDT uses the decision tree to predict the class values of all the instances and compute the residual, which will be used as the class values to be predicted by the second decision tree. On the new training data containing all the instances and the corresponding residual value as the class value, iGBDT checks from the root node of the second decision tree, to see if the best split attribute remains the same. The remaining steps are the same as in the first decision tree.

Before checking the third decision tree, iGBDT uses the first two decision trees to predict the class values of all the instances, then ensembles the two outputs with a user-specified learning rate. Next, it computes the residuals for the corresponding instances. The steps for checking the third decision tree are similar to how iGBDT checks the first/second decision trees.

iGBDT finishes when all the decision trees have been checked/updated. It should be noted that, if only the best split attribute for an intermediate node has changed, then iGBDT just rebuilds the sub-tree under the node. Figure 6 gives an example of such a case, where iGBDT only rebuilds the sub-tree under the node highlighted (i.e. the left child of the root node), since the best split attribute for this intermediate node has changed from "play time" (in Fig. 5) to "income".

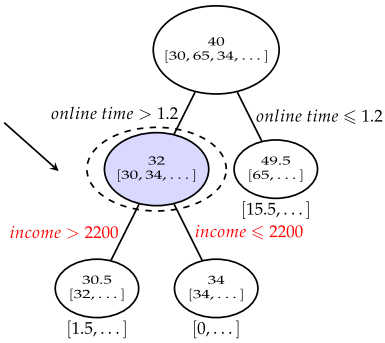

Fig. 6 An example where iGBDT needs to rebuild the sub-tree from highlighted intermediate node

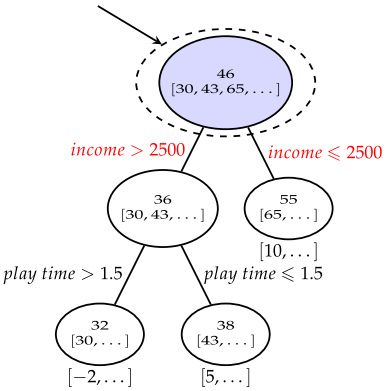

Fig. 7 An example where iGBDT should rebuild the current decision tree and all the following decision trees, since the best split attribute for the root node has changed

However, if the best split attribute for the root node of a decision tree has changed, iGBDT rebuilds the current and all the following decision trees. This can be demonstrated by Fig. 7, where the best split attribute of the root node has changed from "online time" (in Fig. 6) to "income" (in Fig. 7). In this case, iGBDT will rebuild the current and all the following decision trees.

4.4 The Overall iGBDT Algorithm

The overall algorithm for iGBDT is described below, in Algorithms 3 and 4.

Algorithm 3 iGBDT algorithm

input: original training set D, new batch of data B, number of decision trees T, and the original decision trees Tree[1]...Tree[T].
output: new model
1: **for** $m = 1$ to M **do**
2: $S_m \leftarrow sortMerge(D + B)$
3: **end for**
4: **for** $t = 1$ to T **do**
5: $updateTree(Tree[t].rootNode, t, T)$
6: **end for**

Algorithm 4 The updateTree function

```
1: function updateTree(node, t, T)
2:     attr, square loss ← find the best split attribute and its square loss from node.data, i.e. the data allocated
   to this node
3:     if attr = node.attr then
4:         updateNode(node)
5:         updateTree(node.leftNode,t,T)
6:         updateTree(node.rightNode,t,T)
7:     else
8:         if node = root then
9:             tree[t]...tree[T] ← rebuildTree(node.data)
10:            return
11:        else
12:            subTree ← rebuildSubtree(node.data)
13:        end if
14:    end if
15: end function
```

In Algorithm 3, when a new batch of ($p_b = B$) data instances arrives:

1. iGBDT first sort-merges the whole data (containing the new batch) according to each attribute, thus generates a training set for each attribute (lines 1–3). Here, we will use the techniques in Sect. 4.1 to speed up this process.
2. it then sequentially checks/updates each decision tree (line 5), using the methods given in Sect. 4.3. It includes the following steps (described in Algorithm 4):
   (a) computing the best split attribute and the corresponding split point for the data allocated to this node (line 2);
   (b) checking whether the above best split attribute is the same as the node's original best split attribute (line 3).
      – If the answer is yes, then it just updates the node information such as the mean class value (line 4), then continues checking its left and right children (lines 5–6).
      – If the answer is negative, it first checks if it is a root node, if yes, it rebuilds the current and all the following decision trees from scratch (lines 9–10); otherwise, only the sub-tree under the current node is rebuilt (line 12).

Essentially, iGBDT needs to check each decision tree and its sub-trees to figure out whether the corresponding best split attribute has changed, but it saves the huge amount of time spent on rebuilding the thousands of decision trees. Thus, iGBDT is significantly more efficient than the straightforward GBDT approach to incremental learning.

4.5 Space-Complexity of iGBDT

In this subsection, we make analysis of the space-complexity of the proposed iGBDT algorithm.

GBDT is intrinsically an ensemble of sequentially trained (shallow) binary decision trees. Therefore, the building of the subsequent decisions relies on the precedent decision trees. This is very different from RF, in which every decision tree of RF can be trained in parallel (independently).

Nevertheless, the decision trees in GBDT are very shallow, normally with a tree depth of 1–4 (or 1–6). Let p_d be the maximum tree depth, p_t be the number of decision trees. The

maximum number of nodes in one decision tree is 2^{p_d}, so the worst space complexity of iGBDT is $w * |p_t| * 2^{p_d}$, where w is the space that the additional structure (e.g. the splitting feature and value of a node) of one decision tree takes. It is noticeable that, in theory, the worst space complexity of iGBDT is independent of the number of attributes (features) of the data set, while the number of decision trees and tree depth are the main determinants.

5 Experimental Setup

5.1 Data Sets

For fair comparison, we use data sets from the UCI repository for training with different number of attributes, instances and classes. In Table 5, we list the data sets used in our experiments, all of them are used for classification. For instance, *Abalone* is typically used for predicting age from attributes such as sex, length, diameter, height of physical measurements. *Winequality-white* has 11 class labels (0–10) and it is frequently used for classification; it has attributes such as fixed acidity, volatile acidity and citric acid. *Page-blocks-10an-nn* is used for classifying all the blocks of the page layout of a document that have been detected by a segmentation process. *Covertype* is used for predicting forest-cover type using cartographic variables such as aspect and slope. The descriptions of the data sets in Table 5 can be found in the UCI repository [7].

5.2 Parameter Settings

iGBDT and GBDT are both implemented using the Java programming language. In Table 6, we list the hyperparameters of GBDT and iGBDT that will be considered/tested in the experiments, including the number of decision trees, tree depth (depth), and learning rate. We note that initial data set ratio represents the percentage of data used for the initial model building, training set ratio denotes the percentage of data used for the incremental learning,

Table 5 Datasets descriptions

#	Data set name	# instances	# attributes	# classes
1	Abalone	4138	8	18
2	Wall-following	5456	24	4
3	Satimages	6453	24	4
4	Optdigits	5620	60	10
5	Phoneme	5404	5	2
6	Winequality-white	4898	11	7
7	Twonorm-5an-nn	7400	20	2
8	Penbased-5an-nn	10,992	16	10
9	Spambase-5an-nn	4597	57	2
10	Page-blocks-10an-nn	5472	10	5
11	AIDS	36,082	60	3
12	EEG-eye-state	14,980	14	2
13	Magic	19,020	10	2

Table 6 Parameter setting for the experiments on iGBDT and GBDT

#	Data set name	p_t (# of trees)	p_d (max tree depth)	p_r (learning rate)	Initial data set ratio	Training set ratio	Testing set ratio	p_b (batch size)
1	Abalone							
2	Wall-following							
3	Satimages							
4	Optdigits		3	0.1				300
5	Phoneme	1000	4	0.2	0.2	0.6		400
6	Winequality-white	2000	5	0.3	0.3	0.5	0.2	500
7	Twonorm-5an-nn							
8	Penbased-5an-nn							
9	Spambase-5an-nn							
10	Page-blocks-10an-nn							
11	AIDS		3	0.1				1000
12	EEG_eye_state	2000	4	0.2	0.2	0.6		2000
13	Magic	4000	5	0.3	0.3	0.5	0.2	3000

Table 7 The influence of batch size (p_b) on the running time of iGBDT and GBDT

Data sets	Parameters											
	Time (s)											
	$p_t = 1000$ Batch size (p_b), accuracy (Acc)						$p_t = 2000$					
	300	Acc	400	Acc	500	Acc	300	Acc	400	Acc	500	Acc
1. Abalone-iGBDT	271	0.995	284	0.995	184	0.995	580	0.995	622	0.995	281	0.995
1. Abalone-GBDT	1001	0.995	1093	0.995	353	0.995	1254	0.995	1390	0.995	437	0.995
2. Wall-following-iGBDT	973	0.980	1274	0.980	946	0.980	2012	0.980	2511	0.980	2033	0.980
2. Wall-following-GBDT	6294	0.980	5601	0.980	4529	0.980	6180	0.980	5494	0.980	4385	0.980
3. Satimages-iGBDT	1927	0.991	1778	0.991	1780	0.991	4006	0.991	3628	0.991	3788	0.991
3. Satimages-GBDT	27,359	0.991	20,055	0.991	17,535	0.991	28,229	0.991	20,942	0.991	18,455	0.991
4. Optdigits-iGBDT	2497	0.991	1963	0.991	2016	0.991	4612	0.991	3830	0.991	3877	0.991
4. Optdigits-GBDT	22,431	0.991	16,947	0.991	12,739	0.991	19,227	0.991	14,535	0.991	11,137	0.991
5. Phoneme-iGBDT	452	0.815	577	0.815	452	0.815	1086	0.815	1114	0.815	1021	0.815
5. Phoneme-GBDT	3039	0.815	2678	0.815	2145	0.815	2377	0.815	2111	0.815	1677	0.815
6. Winequality-white-iGBDT	463	0.999	482	0.999	446	0.999	869	0.999	1022	0.999	809	0.999
6. Winequality-white-GBDT	1242	0.999	1014	0.999	807	0.999	1246	0.999	1028	0.999	813	0.999
7. Twonorm-5an-nn-iGBDT	1597	0.947	1555	0.947	1439	0.947	3314	0.947	3213	0.947	3133	0.947
7. Twonorm-5an-nn-GBDT	25,412	0.947	22,052	0.947	16,728	0.947	25,547	0.947	21,561	0.947	16,825	0.947
8. Penbased-5an-nn-iGBDT	2433	0.979	2316	0.979	2726	0.979	8302	0.979	7617	0.979	5365	0.979
8. Penbased-5an-nn-GBDT	31,460	0.979	26,102	0.979	21,205	0.979	27,624	0.979	22,457	0.979	19,580	0.979

Table 7 continued

Data sets	Parameters											
	Time (s)											
	$p_t = 1000$ Batch size (p_b), accuracy (Acc)						$p_t = 2000$					
	300	Acc	400	Acc	500	Acc	300	Acc	400	Acc	500	Acc
9. Spambase-5an-nn-iGBDT	2180	0.918	1483	0.918	1454	0.918	2771	0.918	1780	0.918	1618	0.918
9. Spambase-5an-nn-GBDT	2931	0.918	1841	0.918	1694	0.918	3027	0.918	2840	0.918	2224	0.918
10. Page-blocks-10an-nn-iGBDT	549	0.996	639	0.996	739	0.996	1300	0.996	1282	0.996	802	0.996
10. Page-blocks-10an-nn-GBDT	1711	0.996	1530	0.996	956	0.996	1431	0.996	1446	0.996	1753	0.996

	$p_t = 2000$ Batch size (p_b), accuracy (Acc)						$p_t = 4000$					
	1000	Acc	2000	Acc	3000	Acc	1000	Acc	2000	Acc	3000	Acc
11. AIDS-iGBDT	151,888	0.990	82,547	0.990	65,927	0.990	193,196	0.990	106,306	0.990	84,050	0.990
11. AIDS-GBDT	347,570	0.990	183,437	0.990	137,348	0.990	371,531	0.990	212,612	0.990	152,818	0.990
12. EEG_eye_state-iGBDT	8557	0.733	8010	0.733	5069	0.733	16,342	0.733	17,080	0.733	18,143	0.733
12. EEG_eye_state-GBDT	100,812	0.733	53,278	0.733	29,143	0.733	120,693	0.733	58,638	0.733	32,425	0.733
13. Magic-iGBDT	12,972	0.863	10,457	0.863	1814	0.863	41,848	0.863	27,923	0.863	20,547	0.863
13. Magic-GBDT	55,310	0.863	27,888	0.863	20,717	0.863	58,316	0.863	35,273	0.863	32,169	0.863

Learning rate is $p_r = 0.1$. Tree depth is $p_d = 3$

Table 8 The influence of batch size (p_b) on the running time ratio between iGBDT and GBDT

Data sets	Parameters											
	Time (s)											
	$p_t = 1000$ Batch size (p_b), accuracy (Acc)						$p_t = 2000$					
	300	Acc	400	Acc	500	Acc	300	Acc	400	Acc	500	Acc
1. Abalone	3.69	0.995	3.85	0.995	1.92	0.995	2.16	0.995	2.23	0.995	0.88	0.995
2. Wall-following	6.47	0.980	4.40	0.980	4.79	0.980	3.07	0.980	2.19	0.980	2.16	0.980
3. Satimages	14.20	0.991	11.28	0.991	9.85	0.991	7.05	0.991	5.77	0.991	4.87	0.991
4. Optdigits	9.00	0.991	8.63	0.991	6.32	0.991	4.17	0.991	3.80	0.991	2.87	0.991
5. Phoneme	6.72	0.815	4.64	0.815	4.77	0.815	2.19	0.815	1.98	0.815	1.64	0.815
6. Winequality-white	2.68	0.999	2.10	0.999	1.81	0.999	1.43	0.999	1.18	0.999	1.00	0.999
7. Twonorm-5an-nn	15.91	0.947	14.18	0.947	11.62	0.947	7.71	0.947	6.71	0.947	5.37	0.947
8. Penbased-5an-nn	12.93	0.979	11.27	0.979	7.78	0.979	3.33	0.979	2.95	0.979	3.65	0.979
9. Spambase-5an-nn	1.10	0.918	1.24	0.918	1.17	0.918	1.09	0.918	1.60	0.918	1.37	0.918
10. Page-blocks-10an-nn	3.12	0.996	2.39	0.996	1.29	0.996	1.10	0.996	1.13	0.996	2.19	0.996
	$p_t = 2000$ Batch size (p_b), accuracy (Acc)						$p_t = 4000$					
	1000	Acc	2000	Acc	3000	Acc	1000	Acc	2000	Acc	3000	Acc
11. AIDS	2.29	0.990	2.22	0.990	2.08	0.990	1.92	0.990	2.00	0.990	1.82	0.990
12. EEG_eye_state	11.78	0.733	6.65	0.733	5.75	0.733	7.39	0.733	3.43	0.733	1.79	0.733
13. Magic	4.26	0.863	2.67	0.863	11.42	0.863	1.39	0.863	1.26	0.863	1.57	0.863

Learning rate is $p_r = 0.1$. Tree depth is $p_d = 3$

while testing set ratio represents the percentage of data used for testing the accuracy/efficiency of GBDT and iGBDT.

6 Results and Analysis

6.1 Comparison Between iGBDT and Straightforward GBDT

In Tables 7, 8 and Fig. 8, we study the influence of the batch size parameter on the running time efficiency of iGBDT and the straightforward GBDT method using different data sets. For instance, "Abalone-iGBDT" in Table 8 represents the corresponding experimental results for iGBDT on the *Abalone* data set, while "Abalone-GBDT" denotes the corresponding results of GBDT on the same data set. In the first 10 data sets which are relatively smaller than the last three data sets, we set smaller batch size, i.e. 300–500 transactions per batch. Overall, iGBDT is 2–16 times faster than GBDT. In the last three data sets, i.e. *AIDS*, *EEG_eye_state* and *Magic*, we set larger batch sizes, 1000–3000 transactions per batch. We see that iGBDT is 2–12 times faster than GBDT on these three data sets. In general, the running time of the straightforward GBDT method decreases monotonically when batch size increases. It is natural that, when batch size is small, the straightforward GBDT approach needs to rebuild the model more frequently, which is the reason why it needs more time in smaller batch sizes. We note that, the batch size parameter does not influence the final predication accuracy, since the decision trees of both straightforward GBDT and iGBDT will always be identical, given the same tree depth and learning rate parameters.

In Table 9 and Fig. 9, we investigate the influence of the learning rate parameter (p_r) on the running time of iGBDT and GBDT algorithms for $p_t = 1000/2000/4000$ number of trees. It should be noted that, in order to yield better prediction accuracy, users often need to tune heuristically the learning rate (p_r) and tree depth (p_d) hyperparameters simultaneously and to particular data sets. Generally, the smaller the learning rate, the higher the prediction accuracy; but at the same time, a smaller learning rate often requires more time for GBDT to train the model. We see from Fig. 9 that the running time of both GBDT and iGBDT decreases when learning rate increases but the running time of the straightforward GBDT method drops considerably faster than iGBDT. Overall, iGBDT is 1.2–12 times faster than GBDT at different learning rate values, as can be seen in Table 9.

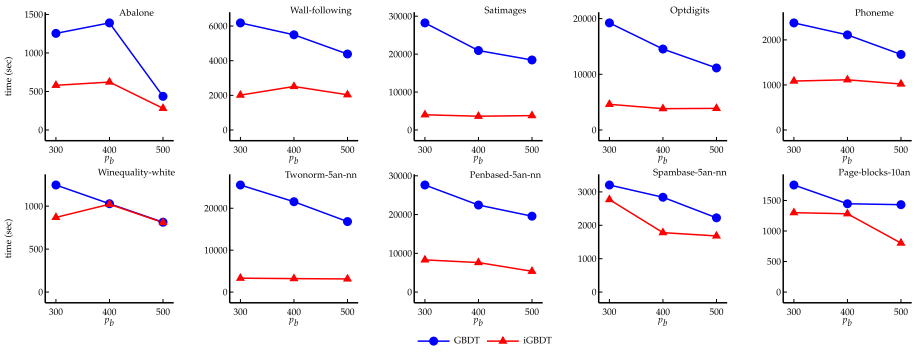

Fig. 8 The influence of batch size (p_b) on the running time of iGBDT and GBDT for $p_t = 2000$ trees

Table 9 The influence of learning rate on the running time of iGBDT and GBDT

Data sets	Parameters											
	Time (s)											
	$p_t = 1000$ Learning rate (p_r), accuracy (Acc)						$p_t = 2000$					
	0.1	Acc	0.2	Acc	0.3	Acc	0.1	Acc	0.2	Acc	0.3	Acc
1. Abalone-iGBDT	184	0.995	251	0.995	185	0.995	434	0.995	310	0.995	227	0.995
1. Abalone-GBDT	354	0.995	534	0.995	555	0.995	1139	0.995	645	0.995	481	0.995
2. Wall-following-iGBDT	1525	0.980	1411	0.986	946	0.991	2182	0.980	2033	0.986	1547	0.991
2. Wall-following-GBDT	4587	0.980	2256	0.986	1599	0.991	4363	0.980	3608	0.986	3430	0.991
3. Satimages-iGBDT	3189	0.991	2426	0.992	1780	0.991	5871	0.991	4252	0.992	3788	0.991
3. Satimages-GBDT	18,362	0.991	10,305	0.992	7185	0.991	20,575	0.991	11,696	0.992	7956	0.991
4. Optdigits-iGBDT	2597	0.991	2150	0.991	2016	0.991	5423	0.991	4694	0.991	3877	0.991
4. Optdigits-GBDT	12,575	0.991	7288	0.991	4954	0.991	11,733	0.991	6829	0.991	5609	0.991
5. Phoneme-iGBDT	661	0.812	617	0.814	452	0.835	1070	0.812	811	0.814	517	0.835
5. Phoneme-GBDT	2144	0.812	1030	0.814	690	0.835	1675	0.812	1053	0.814	1021	0.835
6. Winequality-white-iGBDT	496	0.999	446	0.999	381	0.999	809	0.999	417	0.999	319	0.999
6. Winequality-white-GBDT	970	0.999	606	0.999	509	0.999	1095	0.999	1087	0.999	815	0.999
7. Twonorm-5an-nn-iGBDT	1762	0.947	1716	0.950	1439	0.934	3133	0.947	3064	0.950	3043	0.934
7. Twonorm-5an-nn-GBDT	17,077	0.947	10,611	0.950	7484	0.934	16,228	0.947	9552	0.950	6589	0.934
8. Penbased-5an-nn-iGBDT	2726	0.980	2649	0.980	2596	0.980	8302	0.980	5624	0.980	5324	0.980
8. Penbased-5an-nn-GBDT	21,255	0.980	11,780	0.980	8823	0.980	19,902	0.980	10,915	0.980	8157	0.980

Table 9 continued

Data sets	Parameters											
	Time (s)											
	$p_t = 1000$						$p_t = 2000$					
	Learning rate (p_r), accuracy (Acc)											
	0.1	Acc	0.2	Acc	0.3	Acc	0.1	Acc	0.2	Acc	0.3	Acc
9. Spambase-5an-nn-iGBDT	1454	0.918	894	0.902	594	0.890	1719	0.918	878	0.902	589	0.890
9. Spambase-5an-nn-GBDT	1733	0.918	1083	0.902	1033	0.890	2224	0.918	2170	0.902	2014	0.890
10. Page-blocks-10an-nn-iGBDT	739	0.996	523	0.996	374	0.996	938	0.996	538	0.996	370	0.996
10. Page-blocks-10an-nn-GBDT	954	0.996	546	0.996	535	0.996	1753	0.996	1096	0.996	1009	0.996
	Learning rate (p_r), accuracy (Acc)											
	$p_t = 2000$						$p_t = 4000$					
11. AIDS-iGBDT	62,031	0.990	48,367	0.990	41,039	0.990	64,874	0.990	43,851	0.990	31,330	0.990
11. AIDS-GBDT	124,026	0.999	86,370	0.999	68,398	0.990	135,155	0.990	87,702	0.990	68,109	0.990
12. EEG_eye_state-iGBDT	11,251	0.733	9100	0.724	5069	0.693	22,072	0.733	18,143	0.724	10,548	0.693
12. EEG_eye_state-GBDT	25,643	0.733	20,764	0.724	11,320	0.693	31,193	0.733	24,181	0.724	18,614	0.693
13. Magic-iGBDT	17,153	0.863	8788	0.864	1814	0.863	19,192	0.863	10,113	0.864	7433	0.863
13. Magic-GBDT	18,439	0.863	16,320	0.864	6746	0.863	32,837	0.863	32,597	0.864	32,169	0.863

Tree depth is $p_d = 3$. Batch size is $p_b = 500$ for the first 10 data sets and $p_b = 3000$ for the rest

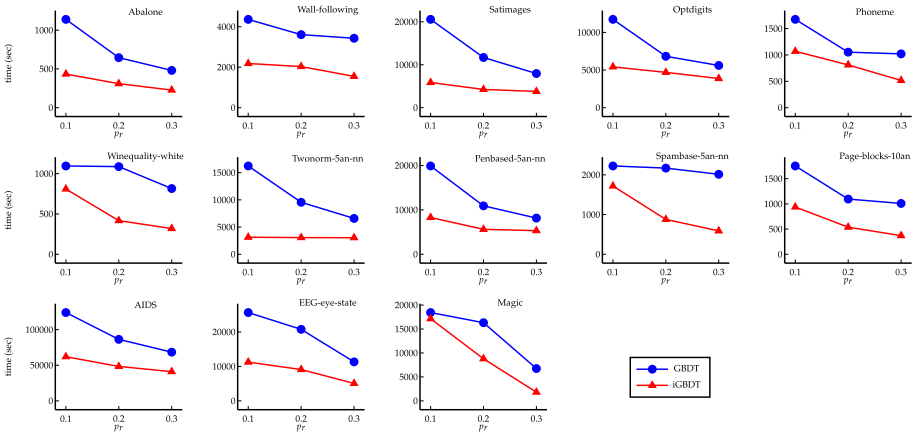

Fig. 9 The influence of learning rate (p_r)on the running time of iGBDT and GBDT for $p_t = 2000$ trees

In Table 10 and Fig. 10, we study the influence of tree depth on the running time efficiency. We fix the other parameters ($p_t = 2000$, $p_b = 500/3000$, $p_r = 0.1$), while varying the tree depth parameter p_d from 3 to 5. We observe that the running time of iGBDT increases monotonically with tree depth. Overall, with varying tree depth, the performance gap between the two approaches is consistently very wide, with iGBDT being 1.2–12 times faster than GBDT.

It must be emphasized that, in all the experiments, we always observe the same accuracy results on the same data set, for iGBDT and the straightforward GBDT. This demonstrates that iGBDT can achieve the same accuracy results as the straightforward GBDT.

In summary, it is evident that iGBDT is significantly faster than the straightforward GBDT approach for incremental learning, regardless of the batch size, the number of trees used, tree depth, and learning rate.

In Table 11, we investigate the reasons of the speed-up of iGBDT over *straightforward* GBDT. We use 4 data sets, *Phoneme*, *Wall_following*, *Page-blocks-10an-nn* and *Magic*, and fix the relevant parameters. One of the acceleration source of iGBDT is its node-checking strategy. When a new batch of data arrives, iGBDT lazily checks if each decision tree remains the same, until a place where the remaining decision trees need to be rebuilt. In the following experiments, for each batch of data, we keep record of the rebuilding positions of iGBDT. Let the $(k+1)$th decision tree be the place where iGBDT starts rebuilding the remaining decision trees for the current batch of data, $R1$ represent the total node checking time of iGBDT in the first k decision trees, $C1$ denote the time spent by GBDT in building the same k decision trees. For each batch of data, we record the position where iGBDT needs to start rebuilding the remaining decision trees to achieve the same model as the *straightforward* GBDT method, and compares the corresponding checking time of iGBDT versus the rebuilding time of GBDT in the first k decision trees. In Table 11, we showcase the rebuilding positions of iGBDT in the first, middle, and the second-to-last batches, and the corresponding ratio between iGBDT's checking time and GBDT's rebuilding time in the first k decision trees. Instead of picking the last batch which may contain inadequate number of transactions, we use the second-to-last batch. We observe that the rebuilding positions generally increase monotonically with the number of batches, especially with large data sets. For instance, in the middle batch of *Magic*, the rebuilding position is 210, which is relatively large. The average ratio between

Table 10 The influence of tree depth (p_d) on the running time of iGBDT and GBDT

Data sets	Parameters											
	Time (s)											
	$p_t = 1000$ Tree depth (p_d), accuracy (Acc)						$p_t = 2000$					
	3	Acc	4	Acc	5	Acc	3	Acc	4	Acc	5	Acc
1. Abalone-iGBDT	184	0.995	261	0.995	258	0.995	429	0.995	430	0.995	510	0.995
1. Abalone-GBDT	354	0.995	356	0.995	419	0.995	481	0.995	535	0.995	552	0.995
2. Wall-following-iGBDT	946	0.980	1319	0.993	1560	0.998	2033	0.980	2753	0.993	3074	0.998
2. Wall-following-GBDT	3437	0.980	3609	0.993	4523	0.998	3371	0.980	3542	0.993	4481	0.998
3. Satimages-iGBDT	1780	0.991	2260	0.990	2661	0.988	3788	0.991	4850	0.990	5443	0.988
3. Satimages-GBDT	18,219	0.991	23,725	0.990	23,806	0.988	20,695	0.991	27,858	0.990	28,108	0.988
4. Optdigits-iGBDT	2016	0.991	2016	0.994	2853	0.990	3877	0.991	4320	0.994	5512	0.990
4. Optdigits-GBDT	7461	0.991	10,353	0.994	12,530	0.990	7060	0.991	10,034	0.994	12,063	0.990
5. Phoneme-iGBDT	432	0.815	452	0.840	458	0.841	1004	0.815	1021	0.840	1233	0.841
5. Phoneme-GBDT	2155	0.815	2230	0.840	4006	0.841	1672	0.815	1727	0.840	3092	0.841
6. Winequality-white-iGBDT	446	0.999	493	0.999	617	0.999	809	0.999	908	0.999	1120	0.999
6. Winequality-white-GBDT	969	0.999	1333	0.999	1765	0.999	813	0.999	1105	0.999	1452	0.999
7. Twonorm-5an-nn-iGBDT	1439	0.947	1859	0.947	2314	0.947	3133	0.947	3952	0.947	4894	0.947
7. Twonorm-5an-nn-GBDT	17,377	0.947	24,207	0.947	35,228	0.947	16,686	0.947	29,860	0.947	49,170	0.947
8. Penbased-5an-nn-iGBDT	2726	0.980	3325	0.980	4306	0.982	8302	0.980	6634	0.980	8606	0.982
8. Penbased-5an-nn-GBDT	21,246	0.980	23,601	0.980	24,539	0.982	20,108	0.980	21,296	0.980	22,146	0.982

Table 10 continued

Data sets	Parameters											
	Time (s)											
	$p_t = 1000$ Tree depth (p_d), accuracy (Acc)						$p_t = 2000$					
	3	Acc	4	Acc	5	Acc	3	Acc	4	Acc	5	Acc
9. Spambase-5an-nn-iGBDT	1038	0.920	1233	0.902	1454	0.840	1070	0.920	1592	0.902	1787	0.840
9. Spambase-5an-nn-GBDT	1312	0.920	1567	0.902	1740	0.840	2224	0.920	2460	0.902	2657	0.840
10. Page-blocks-10an-nn-iGBDT	739	0.996	745	0.996	793	0.995	954	0.996	1123	0.996	1358	0.995
10. Page-blocks-10an-nn-GBDT	930	0.996	1101	0.996	1330	0.995	1418	0.996	1622	0.996	1753	0.995
	Tree depth (p_d), accuracy (Acc)											
	$p_t = 2000$						$p_t = 4000$					
11. AIDS-iGBDT	37,569	0.990	42,736	0.990	58,625	0.990	41,594	0.990	50,090	0.990	62,397	0.990
11. AIDS-GBDT	76,671	0.990	85,472	0.990	136,338	0.990	83,187	0.990	96,327	0.990	141,812	0.990
12. EEG_eye_state-iGBDT	5069	0.733	6707	0.760	8363	0.782	18,143	0.733	13,711	0.760	16,211	0.782
12. EEG_eye_state-GBDT	24,686	0.733	57,500	0.760	66,577	0.782	26,959	0.733	64,970	0.760	126,249	0.782
13. Magic-iGBDT	8016	0.863	12,013	0.860	14,678	0.864	11,620	0.863	15,099	0.860	17,802	0.864
13. Magic-GBDT	16,318	0.863	21,436	0.860	25,847	0.864	32,169	0.863	42,655	0.860	51,814	0.864

Learning rate is $p_r = 0.1$. Batch size is $p_b = 500$ for the first 10 data sets and $p_b = 3000$ for the rest

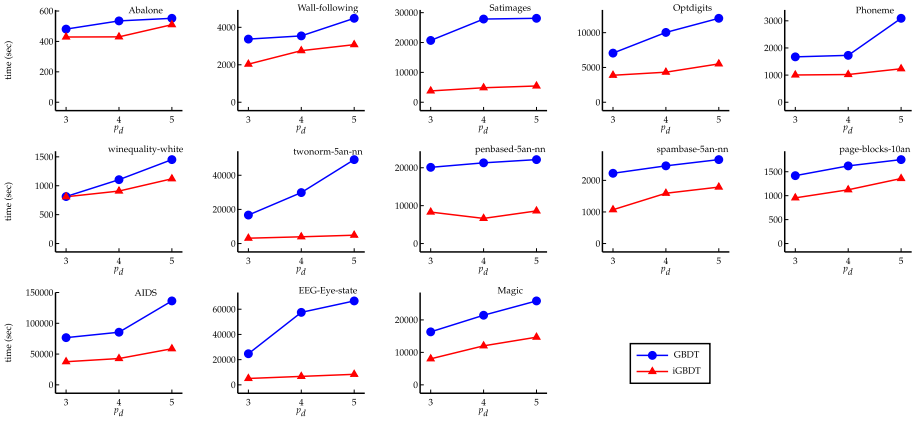

Fig. 10 The influence of tree depth (p_d) on the running time of iGBDT and GBDT for $p_t = 2000$ trees

iGBDT's checking time and GBDT's rebuilding time is approximately 1.85. Therefore, the node-checking strategy in iGBDT is very effective.

In the above experiments, to purely compare iGBDT's checking time and the *straightforward* GBDT's rebuilding time in the first k decision trees, we have not calculated the time spent in resorting when a new batch of data arrives. Yet, the incremental sort-merge strategy is another advantage of iGBDT, which has been addressed in Sect. 4.2. Since new batches of data are dynamically arriving, iGBDT's sort-merge strategy can greatly speed up the sorting procedure before updating the decision trees. Overall, the sort-merge and node-checking strategies are the two important reasons for iGBDT's speed-up over the *straightforward* GBDT.

6.2 Comparison Among iGBDT, Online BBM and VFDT

6.2.1 Comparison Between iGBDT and Online BBM

In Sect. 2, we have introduced the latest Online BBM algorithm which has proven to outperform state-of-the-art online learning algorithms. In this subsection, we compare the performance of iGBDT with Online BBM to further validate its performance. Online BBM needs to tune the number of weak learners parameter. In our experiments, we set this parameter in the range of [20,100]. For each data set, we always pick the best parameter (i.e. the number of weak learners) for Online BBM. The implementations of Online BBM was obtained from [4], which was implemented in C++. We note that iGBDT was implemented in Java, which has a larger runtime overhead than C++ optimised native code and is typically slower. The results are reported in Table 12.

In Table 12, we see that Online BBM is significantly faster than iGBDT in training time. This is because the principles/procedures of iGBDT and Online BBM are very different, Online BBM does not need to keep the past data but updates each weak learner when a new instance arrives, whereas iGBDT needs to keep both the old data and the newly arrived instances to maintain the same prediction accuracy as the straightforward GBDT method and train thousands of decision trees. However, it is interesting to observe that, in terms of testing time, iGBDT is typically 7–10 times faster than Online BBM. More importantly, with the

Table 11 The benefits of the node-checking strategy in iGBDT versus GBDT

Data sets	Stage					
	Time ratio					
	First batch		Middle batch		Second to last batch	
	Rebuild position	Checking time/rebuilding time	Rebuild position	Checking time/rebuilding time	Rebuild position	Checking time/rebuilding time
Phoneme	30	1.132	20	1.078	33	1.163
Wall_following	19	1.807	21	2.058	42	2.368
Page-blocks-10an-nn	5	1.896	17	1.170	49	2.055
Magic	19	2.037	210	2.550	62	2.854

Number of trees is $p_t = 1000$, tree depth is $p_d = 3$, learning rate is $p_r = 0.1$, batch size is $p_b = 300$ for the 4 data sets

exception of *AIDS* data set, iGBDT is significantly more accurate than Online BBM. Their accuracy difference is typically between 5 and 20%, which is significant.

In short, Online BBM outperforms iGBDT in training time, while iGBDT beats Online BBM in both accuracy results of prediction efficiency.

6.2.2 Comparison Between iGBDT and VFDT

In Table 12, we also compare the performance between iGBDT and VFDT which is a popular decision tree algorithm for data stream mining. Overall, VFDT is faster than iGBDT in training time. This is mainly due to the fact that VFDT only builds one decision tree, while iGBDT keeps hundreds or thousands of shallow decision trees. However, in terms of prediction efficiency, iGBDT outperforms VFDT. In fact, VFDT is the slowest in prediction out of the three methods in comparison.

In terms of accuracy performance, when the batch size is small, there is no clear winner between the two methods: on *Phoneme* and *EEG_eye_state* data sets, iGBDT outperforms VFDT, but the reverse situation occurs on the *Mushroom* and *Twonorm-5an-nn* data sets. Nevertheless, on the *Magic*, *Adult*, and *AIDS* data sets, the reported accuracy of iGBDT is significantly better than VFDT. It is interesting to observe that, when the batch size increases from 1000 to 3000, the accuracy performance of iGBDT is very stable.

We notice that the accuracy of VFDT is constant, irrespective of the batch size. This is because in VFDT there is only one decision tree built upon the whole training data and there does exist tree rebuilding; meanwhile, the testing data is always pre-fixed in our experiments.

However, it must be emphasized that, these three algorithms fit to different application scenarios. Online BBM is the fastest algorithm in training. iGBDT needs significantly more training time than the other two algorithms, owing to the fact that it needs to build (update) thousands of decision trees; but it is the fastest in prediction. VFDT is usually used for streaming data. It is notable that iGBDT's accuracy performance can be greatly enhanced by tuning the learning rate and tree-depth parameters. Its overall accuracy performance is the best among the major classification algorithms, as reported in [23]. Owing its superior accuracy performance on static data, it is meaningful to adapt GBDT to the incremental learning settings.

6.3 Discussion

6.3.1 Comparing iGBDT for Incremental Learning with the On-Line Learning Algorithms

Due to the intrinsic algorithm characteristics of GBDT, in particular the sequential manner of building the shallow decision trees, and the dependence between the current and the precedent decision trees, auxiliary data structures are required. These non-partitionable, update-heavy data structures bring additional space (memory) cost and can be expensive to maintain. Therefore, iGBDT can be used up to the point where there is enough memory to store the additional information, which is proportional to the size of all training data.

In comparison, for on-line learning methods such as on-line BBM, their intrinsic algorithm characteristics enable the on-line (incremental) manner of learning; they can update the model on the fly when new data comes, without re-running from scratch. These algorithms are typically light on computational resources, since they usually do not need to keep past data

Table 12 Performance comparison among iGBDT, Online BBM and VFDT

Data sets	Method								
	Time (ms)								
	iGBDT/Online BBM/VFDT								
	p_b (batch size)								
	300			400			500		
	Training	Testing	Acc	Training	Testing	Acc	Training	Testing	Acc
Phoneme-iGBDT	1000	23	0.815	2716	5	0.815	2013	2	0.815
Phoneme-Online BBM	169	20	0.757	136	19	0.763	116	21	0.732
Phoneme-VFDT	1890	110	0.765	13800	110	0.765	1150	110	0.765
Mushroom-iGBDT	26,801	1	0.569	20577	1	0.569	14,965	1	0.569
Mushroom-Online BBM	589	37	0.487	554	36	0.519	522	36	0.519
Mushroom-VFDT	4060	130	0.929	3080	130	0.929	2350	130	0.929
EEG_eye_state-iGBDT	245,734	3	0.733	104,653	2	0.733	369,363	4	0.733
EEG_eye_state-Online BBM	910	48	0.548	797	48	0.548	733	51	0.548
EEG_eye_state-VFDT	8160	140	0.606	6180	140	0.606	4890	140	0.606
Twonorm-5an-nn-iGBDT	14,600	6	0.947	12,565	6	0.947	8867	7	0.947
Twonorm-5an-nn-Online BBM	343	43	0.600	246	44	0.645	245	45	0.610
Twonorm-5an-nn-VFDT	3560	140	0.961	2690	130	0.961	2190	140	0.961
	p_b (batch size)								
	1000			2000			3000		
AIDS-iGBDT	4,924,418	3	0.990	2,440,691	2	0.990	1,795,151	3	0.990
AIDS-Online BBM	713	111	0.991	545	111	0.991	471	111	0.991
AIDS-VFDT	19,300	260	0.966	10,410	270	0.966	707	280	0.966
Magic-iGBDT	47,495	17	0.863	22,494	18	0.863	15,791	17	0.863
Magic-Online BBM	344	43	0.600	246	44	0.645	245	45	0.610
Magic-VFDT	3290	140	0.802	1670	150	0.802	11,310	140	0.802
Adult-iGBDT	2,715,164	11	0.852	1,265,384	11	0.852	862,094	11	0.852
Adult-Online BBM	675	97	0.807	662	107	0.811	611	102	0.813
Adult-VFDT	12,050	140	0.844	6210	180	0.844	4190	180	0.844

(i.e. they do not need additional data structures) to fully optimise the cost function defined on the training instances.

Nevertheless, it should be noted that the overall accuracy performance of the (static) GBDT method is better than the other classification algorithms as well as the on-line learning algorithms; but there is still no incremental learning approach for GBDT in the literature. The need to maintain a high classification accuracy throughout an incremental learning process was the motivation for proposing iGBDT.

6.3.2 Addressing the Total Running Time Issue of iGBDT and GBDT on the Complete Data

Given a data set D, there are two cases to consider: case (a) directly run GBDT on D; or case (b) split D into $|N_b|$ batches and run iGBDT on the consequent batches. If $|N_b|$ is large (e.g. $|N_b| \geq 10$), the running time of case a is usually smaller than that of case b using iGBDT. This is very similar to the case of other classification as well as frequent itemset mining algorithms [22]. For instance, if we split D into 10 batches of equal size, the total running time of SVM on these 10 separate batches is larger than running SVM directly on D. GBDT is composed of hundreds (or thousands) of sequentially trained shallow decision trees and it needs iterative computation to fully optimise the cost function defined on the training instances. If we split the data into $|N_b|$ batches, for each batch, iGBDT would need to check (and update or rebuild the decision tree if the splitting attribute of a node or the root has changed) all the decision trees. Thus, the total running time of iGBDT on $|N_b|$ batches would be significantly larger than that of case a.

However, incremental learning fits to the application scenarios where the batch of new data arrives on the fly, i.e. the subsequent batches of data are unavailable when processing the current batch of new data, and also there are constraints for in-memory processing, i.e. total RAM is inadequate to hold all batches of data. Even so, other solutions could be proposed, like waiting for enough number of batches (until available memory is exhausted), i.e. postpone running GBDT until RAM is full, then run GBDT from scratch on these batches to obtain the same prediction model as iGBDT. We believe that sometimes this solution is reasonable for real-world applications, when the waiting time for the incoming batches plus the above GBDT running time is smaller than the sum of all time costs of iGBDT on the same batches.

6.3.3 Limitations/Drawbacks of this Work

Finally, we would like to point out the limitations (drawbacks) of this work.

- iGBDT needs additional data structures to check whether the splitting features of each decision tree has changed, as mentioned above.
- The performance of iGBDT also depends on the data characteristics; if the concept (the inherent patterns of the data) evolves rapidly, iGBDT's efficiency will drop consequently.
- We have not applied our proposed iGBDT method in real-world settings; all the data sets used for testing are comparatively small.

7 Conclusion and Future Work

In this work, we propose a novel algorithm (iGBDT) which can incrementally update the classification model built upon GBDT. iGBDT does not always require running GBDT from

scratch when new data is dynamically arriving in batch. Experiments show that, in terms of model building/updating time, iGBDT obtains significantly better performance than the straightforward GBDT method, while still keeping the same classification accuracy. In future work, we will test iGBDT in real-world applications. We will adapt it to the Spark distributed computing platform to further improve its efficiency. We will also study the suitability of our method to transfer learning.

References

1. Aggarwal CC (2007) Data streams: models and algorithms, vol 31. Springer, Berlin
2. Babenko B, Yang MH, Belongie S (2009) A family of online boosting algorithms. In: IEEE 12th international conference on computer vision workshops (ICCV workshops). IEEE, pp 1346–1353
3. Beygelzimer A, Hazan E, Kale S, Luo H (2015a) Online gradient boosting. In: NIPS, pp 2458–2466
4. Beygelzimer A, Kale S, Luo H (2015b) Optimal and adaptive algorithms for online boosting. In: ICML
5. Chapelle O, Chang Y (2011) Yahoo! learning to rank challenge overview. In: JMLR proceedings, pp 1–24
6. Chen ST, Lin HT, Lu CJ (2012) An online boosting algorithm with theoretical justifications. In: ICML
7. Dheeru D, Karra Taniskidou E (2017) UCI machine learning repository. http://archive.ics.uci.edu/ml. Accessed 31 Jan 2019
8. Domingos P, Hulten G (2000) Mining high-speed data streams. In: ACM SIGKDD. ACM, pp 71–80
9. Freund Y (1995) Boosting a weak learning algorithm by majority. Inf Comput 121(2):256–285
10. Freund Y, Schapire RE (1997) A decision-theoretic generalization of on-line learning and an application to boosting. J Comput Syst Sci 55:119–139
11. Friedman JH (2001) Greedy function approximation: a gradient boosting machine. Ann Stat 29:1189–1232
12. Gaber MM, Zaslavsky A, Krishnaswamy S (2005) Mining data streams: a review. ACM Sigmod Rec 34(2):18–26
13. Grbovic M, Vucetic S (2011) Tracking concept change with incremental boosting by minimization of the evolving exponential loss. In: PKDD. Springer, Berlin, pp 516–532
14. Hulten G, Spencer L, Domingos PM (2001) Mining time-changing data streams. In: ACM SIGKDD, pp 97–106
15. Leistner C, Saffari A, Roth PM, Bischof H (2009) On robustness of on-line boosting: a competitive study. In: IEEE 12th international conference on computer vision workshops (ICCV workshops), pp 1362–1369
16. Liu X, Yu T (2007) Gradient feature selection for online boosting. In: 2007 IEEE 11th international conference on computer vision (ICCV). IEEE, pp 1–8
17. Oza NC, Russell S (2001) Experimental comparisons of online and batch versions of bagging and boosting. In: ACM SIGKDD. ACM, pp 359–364
18. Oza NC, Russell SJ (2001) Online bagging and boosting. In: Eighth international workshop on artificial intelligence and statistics, pp 105–112
19. Pavlov DY, Gorodilov A, Brunk CA (2010) Bagboo: a scalable hybrid bagging-the-boosting model. In: Proceedings of the 19th ACM conference on information and knowledge management, CIKM 2010, Toronto, Ontario, Canada, October 26–30, 2010, pp 1897–1900
20. Pelossof R, Jones M, Vovsha I, Rudin C (2009) Online coordinate boosting. In: IEEE 12th international conference on computer vision workshops (ICCV workshops). IEEE, pp 1354–1361
21. Perkins S, Lacker K, Theiler J (2003) Grafting: fast, incremental feature selection by gradient descent in function space. J Mach Learn Res 3(Mar):1333–1356
22. Zhang C, Hao Y, Mazuran M, Zaniolo C, Mousavi H, Masseglia F (2013) Mining frequent itemsets over tuple-evolving data streams. In: Proceedings of the 28th annual ACM symposium on applied computing, SAC'13, Coimbra, Portugal, March 18–22, 2013, pp 267–274
23. Zhang C, Liu C, Zhang X, Almpanidis G (2017) An up-to-date comparison of state-of-the-art classification algorithms. Expert Syst Appl 82:128–150. https://doi.org/10.1016/j.eswa.2017.04.003
24. Zhang T, Yu B (2005) Boosting with early stopping: convergence and consistency. Ann Stat 33:1538–1579