Fast Gradient Boosting Decision Trees with Bit-Level Data Structures

Laurens Devos (✉), Wannes Meert, and Jesse Davis

Department of Computer Science, KU Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract. A gradient boosting decision tree model is a powerful machine learning method that iteratively constructs decision trees to form an additive ensemble model. The method uses the gradient of the loss function to improve the model at each iteration step. Inspired by the database literature, we exploit bitset and bitslice data structures in order to improve the run time efficiency of learning the trees. We can use these structures in two ways. First, they can represent the input data itself. Second, they can store the discretized gradient values used by the learning algorithm to construct the trees in the boosting model. Using these bit-level data structures reduces the problem of finding the best split, which involves counting of instances and summing gradient values, to counting one-bits in bit strings. Modern CPUs can efficiently count one-bits using AVX2 SIMD instructions. Empirically, our proposed improvements can result in speed-ups of 2 to up to 10 times on datasets with a large number of categorical feature without sacrificing predictive performance.

Keywords: Gradient boosting · Decision tree · Bitset · Bitslice

1 Introduction

Gradient boosting decision trees (GBDTs) are a powerful and theoretically elegant machine learning method that constructs an additive ensemble of trees. GBDT methods employ an iterative procedure, where the gradient of the loss function guides learning a new tree such that adding the new tree to the model improves its predictive performance. GBDTs are widely used in practice due the availability of high quality and performant systems such as XGBoost [3], LightGBM [7] and CatBoost [10]. These have been successfully applied to many real-world datasets, and cope particularly well with heterogeneous and noisy data.

This paper explores how to more efficiently learn gradient boosting decision tree models without sacrificing accuracy. When learning a GBDT model, the vast majority of time is spent evaluating candidate splits when learning a single tree. This involves counting instances and summing gradients. State-of-the-art GBDT implementations use full 32- or 64-bit integers or floats to represent the data and the gradients. We propose the BitBoost algorithm which represents the data and gradients using bitsets and bitslices, two data structures originating from database literature. This allows BitBoost to exploit the bit-level parallelism

enabled by modern CPUs. However, these data structures impose strict limitations on how and which numbers can be expressed. This necessitates adapting three operations in the standard GBDT learning algorithm: summing gradients, performing (in)equality checks, and counting. Empirically, BitBoost achieves competitive predictive performance while reducing the runtime by a large margin. Moreover, BitBoost is a publicly available package.[1]

2 Background and Related Work

2.1 Gradient Boosting

Schapire [12] proposed the theoretical idea of boosting, which was implemented in practice by AdaBoost [5, 13]. This idea was generalized by the generic *gradient boosting* algorithm, which works with any differentiable loss function [6, 9]. Given N input instances $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ and a differentiable loss function $\mathcal{L}$, gradient boosting models iteratively improve the predictions of y from $\mathbf{x}$ with respect to $\mathcal{L}$ by adding new weak learners that improve upon the previous ones, forming an additive ensemble model. The additive nature of the model can be expressed by:

$$F_0(\mathbf{x}) = c, \qquad F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + h_{\theta,m}(\mathbf{x}), \tag{1}$$

where m is the iteration count, c is an initial guess that minimizes $\mathcal{L}$, and $h_{\theta,m}(\mathbf{x})$ is some weak learner parametrized by θ such as a linear model or a decision tree.

In this paper, we focus on *gradient boosting decision trees* or *GBDTs*, which are summarized in Algorithm 1. Gradient boosting systems minimize $\mathcal{L}$ by gradually taking steps in the direction of the negative gradient, just as numerical gradient-descent methods do. In GBDTs, such a step is a single tree constructed to fit the negative gradients. One can use a least-squares approach to find a tree $h_{\theta^*,m}$ that tries to achieve this goal:

$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \left[-g_m(\mathbf{x}_i, y_i) - h_{\theta,m}(\mathbf{x}_i)\right]^2, \tag{2}$$

where $g_m(\mathbf{x}, y) = \partial_{\hat{y}} \mathcal{L}(y, \hat{y})|_{\hat{y}=F_{m-1}(\mathbf{x})}$ is the gradient of the loss function $\mathcal{L}$.

$F_0(\mathbf{x}) = \arg\min_c \sum_{i=1}^N \mathcal{L}(y_i, c)$
for $m \leftarrow 1$ *to* M **do**
  $g_{m,i} = \partial_{\hat{y}} \mathcal{L}(y_i, \hat{y}) \mid_{\hat{y}=F_{m-1}(\mathbf{x}_i)}$
  $h_{\theta,m}$ = a tree that optimizes Equation 2
  $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \rho_m \, h_{\theta,m}(\mathbf{x})$
end

Alg. 1: The gradient boosting algorithm.

[1] BitBoost is hosted on GitHub: `https://github.com/laudv/bitboost`

In practice, each individual tree is typically built in a greedy top-down manner: the tree learning algorithm loops over all internal nodes in a depth-first order starting at the root, and splits each node according to the best split condition. The best split condition is found by identifying the feature with the best split candidate. For example, the best split candidate for a numerical feature is a value s that partitions the instance set – the data instances that sort to the current node – into a left and right subset according to the condition $x < s$ with maximal gain, with x the feature value. Gain is defined as the difference in the model's loss before and after the split.

Competitive gradient boosting systems use a histogram-based approach to generate a limited number of split candidates (e.g., 256). This means that for each feature, the learner iterates over all remaining instances in the instance set of the node to be split, and fills a histogram by collecting statistics about the instances that go left given a split condition. The statistics include, depending on the system, the sums of the first and optionally the second gradient, and usually also an instance count. The aggregated statistics contain all the necessary information to calculate the best split out of the (limited number of) candidate splits. The complexity of this approach is $\mathcal{O}(F \cdot (n + S))$, with n the instance set size, S the histogram size, and F the feature count.

We will compare our finding with three such competitive systems. XGBoost [3] was introduced in 2016 and improved the scalability of learning by introducing sparsity-aware split finding, a novel parallel architecture, and the ability to take on out-of-core learning. LightGBM [7] was presented one year later and proposed a new gradient weighted sampling technique (GOSS), and a technique to reduce the number features by combining features that have no non-zero values for the same instance (EFB). Lastly, CatBoost [10] improved the accuracy of learning from high-cardinality categorical data by identifying the problem of *prediction shift* and resolving it with a new method called *ordered boosting*. All these systems are efficient, complete, and open-source libraries.

2.2 Bitsets and Bitslices

Bitsets and bitslices are two data structures that originated in the database literature [11, 2]. We wish to utilize these structures in the gradient boosting algorithm with the goal of improving learning times.

A *bitset* is a simple yet effective bit-level data structure used to represent a set of items. A bitset is a string of bits of length N. The ith bit is 1 if the ith item is present in the set and 0 if it is not.

A *bitslice* is a data structure used to represent a sequence of unsigned integers. In a typical array of integers, all bits of an individual number are stored consecutively: $x_1, x_2, \ldots, x_N$, where each x_i is made up of B bits $x_i^B \cdots x_i^2 x_i^1$, with x_i^B the most significant bit (MSB) and x_i^1 is the least significant bit (LSB). A bitslice transposes this structure; instead of consecutively storing all bits of a single number, it groups bits with the same significance:

$$\underbrace{x_1^B x_2^B \cdots x_N^B}_{\text{MSBs}}, \quad \ldots, \quad x_1^2 x_2^2 \cdots x_N^2, \quad \underbrace{x_1^1 x_2^1 \cdots x_N^1}_{\text{LSBs}},$$

There are two main advantages of using bitslices for small integers. First, bitslices can efficiently store integers smaller than the minimally addressable unit – a single byte or 8 bits on modern systems – because of its transposed storage format. For example, naively storing 1000 3-bit integers requires 1000 bytes of storage. A bitslice only needs $3 \times 125 = 375$ bytes. Second, elements in a bitslice can be efficiently summed. To sum up values in a bitslice, one adds up the contributions of each bit group, optionally masking values with a bitset:

$$\sum_{b=1}^{B} 2^{b-1} \times \texttt{CountOnebits}(x_1^b x_2^b \cdots x_N^b \ \wedge \ \underbrace{s_1 s_2 \cdots s_N}_{\text{bitset mask}}) \tag{3}$$

The `CountOneBits` operation, also known as the population count or *popcount* operation, counts the number of one-bits in a bit string. This can be done very efficiently using the vectorized Harley Seal algorithm [8], taking advantage of AVX2 SIMD instructions operating on 32-byte wide registers.

3 BitBoost Algorithm

When learning a decision tree, the central subroutine is selecting which feature to split on in a node. This entails evaluating the gain of all potential variable-value splits and selecting the one with the highest gain. As in Equation 2, a single tree fits the negative gradients g_i of $\mathcal{L}$ using least-squares. A split's gain is defined as the difference in the tree's squared loss before and after the split. Splitting a parent node p into a left and a right child l and r results in the following gain:

$$\begin{aligned} \text{gain}(p, l, r) &= \sum_{i \in I_p} (-g_i + \Sigma_p/|I_p|)^2 - \sum_{i \in I_l} (-g_i + \Sigma_l/|I_l|)^2 - \sum_{i \in I_r} (-g_i + \Sigma_r/|I_r|)^2 \\ &= -\Sigma_p^2/|I_p| + \Sigma_l^2/|I_l| + \Sigma_r^2/|I_r|, \end{aligned} \tag{4}$$

where I_* gives the set of instances that are sorted by the tree to node $*$ and $\Sigma_* = \sum_{i \in I_*} g_i$ is the sum of the gradients for node $*$. In other words, computing the gain of a split requires three operations:

1. *summing* the gradients in both leaves;
2. *performing (in)equality* checks to partition the node's instance sets based on the split condition being evaluated; and
3. *counting* the number of examples in each node's instance set.

Current implementations use 32- or 64-bit integers or floats to represent the gradients, data, and example IDs in instance sets. Hence, the relevant quantities in Equation 4 are all computed using standard mathematical or logical operations.

Our hypothesis is that employing native data types uses more precision than is necessary to represent each of these quantities. That is, by possibly making some approximations, we can much more compactly represent the gradients, data, and instance sets using bitslice and bitset data structures. The primary

advantage of the bit-level representations is speed: we can train a tree much faster by exploiting systems-level optimizations. First, we exploit instruction-level parallelism to very efficiently compute the relevant statistics used in Equation 4. Second, by representing the data and instance sets using bitsets, important operations such as partitioning the data at a node can be translated to vectorized bitwise logical operations, which are much faster to perform than comparison operators on floats or integers.

3.1 Representing the Gradients using Bitslices

When learning a GBDT model, each instance is associated with a real-valued gradient value, which is updated after each iteration in the boosting process. Existing gradient boosting systems typically use 32-bit, or even 64-bit floats to represent these gradients. However, the individual trees tend to be shallow to combat over-fitting. Consequently, rather than precisely partitioning data instances into fine-grained subsets, a tree loosely groups instances with similar gradient values. Intuitively, a gradient value represents an instance's *optimization need*, which is a measure of how much an instance requires a prediction update, and the trees categorize instances according to this need.

Based on this observation, our insight is that storing the precise gradient values may be superfluous, and it may be possible to represent these values using fewer bits without affecting the categorization of the data instances. Therefore, *we explore discretizing the gradient values and storing the values in a bitslice*. A bitslice of width k can represent values $0, \dots, 2^k - 1$. To map the gradient values to the values that can be represented by the bitslice, the outer bounds $b_{\min}$ and $b_{\max}$, corresponding to the bitslice values 0 and $2^k - 1$ respectively, need to be chosen first. Then, gradient values g can be mapped to bitslice values using a simple linear transformation:

$$g \leftarrow \operatorname{round}\left(\frac{\min(b_{\max}, \max(b_{\min}, g)) - b_{\min}}{b_{\max} - b_{\min}}\right). \quad (5)$$

The gradient values are thus mapped to a set of 2^k linearly spaced points. An example of this can be seen in Fig. 1.

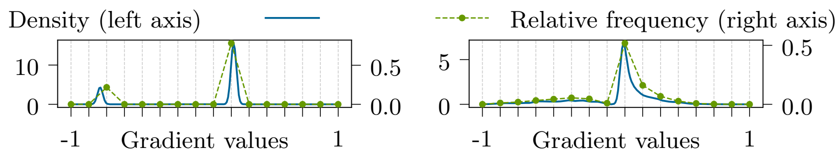

Fig. 1. Gradient value densities and the relative frequency of the 4-bit discretized values of the first and the last iterations. The values were produced by an unbalanced (1/5 positive) binary classification problem using binary log-loss. The gray vertical dotted lines indicate which 16 values can be represented with the 4-bit discretization.

Hence, the key question is how to select the boundaries for the discretization. This is loss-function dependent, and we consider five of the most commonly used loss functions:

- *Least absolute deviation (LAD)*: This loss function is widely used in practice due its better handling of outliers compared to least-squares. The LAD is:

$$\mathcal{L}_1(F(\mathbf{x}), y) = |y - F(\mathbf{x})|, \tag{6}$$

  and its gradients are either -1, indicating that an estimation for a particular instance should increase, or 1, indicating that an estimation should decrease. This information can be expressed with a single bit in a bitslice of width 1, yet existing systems use a full float to represent this.
- *Least-squared loss*: The gradient values do not have naturally defined boundaries, and the magnitude of the extreme values depends on the targets of the regression problem. Interestingly, choosing boundary values on the gradients of the least-squared loss makes it equivalent to Huber loss. For that reason, we look at Huber loss for inspiration when choosing boundaries.
- *Huber loss*: Huber loss is often used instead of squared-loss to combine the faster convergence of least-squared loss with the resilience to outliers of LAD. It has a single parameter δ:

$$\mathcal{L}_{H,\delta}(F(\mathbf{x}), y) = \begin{cases} \frac{1}{2}(y - F(\mathbf{x}))^2 & \text{if } |y - F(\mathbf{x})| \leq \delta, \\ \delta(|y - F(\mathbf{x})| - \frac{1}{2}\delta) & \text{otherwise.} \end{cases} \tag{7}$$

  The boundaries of the gradient values are naturally defined by the parameter δ: $b_{\text{min}} = -\delta$, the most negative gradient values, and $b_{\text{max}} = \delta$, the most positive gradient value. The value of δ is often taken to be the α quantile of the residuals at iteration m, i.e., $\delta_m = \text{quantile}_\alpha\{|y_i - F_{m-1}(\mathbf{x}_i)|\}$ [6].
- *Binary log-loss*: Given labels y_i in $\{-1, 1\}$, binary log-loss is used for binary classification problems and is defined as:

$$\mathcal{L}_{\log}(F(\mathbf{x}), y) = \log(1 + \exp(-2yF(\mathbf{x}))). \tag{8}$$

  This function's gradient values are naturally confined to $[-2, 2]$, so we choose the boundary values accordingly. However, we have found that choosing more *aggressive* boundaries (e.g., -1.25, 1.25) can speed up convergence.
- *Hinge loss*: Like binary log-loss, hinge loss is used for binary classification problems. It is defined as:

$$\mathcal{L}_{\text{hinge}}(F(\mathbf{x}), y) = \max(0, 1 - y\,F(\mathbf{x})). \tag{9}$$

  The possible (sub-)gradient values are -1, 0, and 1.

3.2 Representing the Data using Bitsets

The standard way to represent the data in GBDT implementations is to use arrays of integers or floats. In contrast, we propose encoding the data using bitsets.

How this is done depends on the feature. We distinguish among three feature types: low-cardinality categorical, high-cardinality categorical, and numerical features. To differentiate between low-cardinality and high-cardinality categorical features, we define an upper limit K on a feature's cardinality. A good value of K is dataset specific, but we typically choose K between 8 and 32.

Low-cardinality categorical features. We use a one-hot encoding scheme: given a feature f with r possible values $v_0, \ldots, v_r$, we create r length-N bitsets. The ith position of the kth bitset is set to one if the ith value of f equals v_k. The resulting bitsets can be used to evaluate all possible equality splits: for any equality split $f = v_j$, we compute the instance set of the left subtree by applying the logical `AND` to the current instance set and bitset j (see Fig. 2, right).

High-cardinality features. This requires more work as considering all possible equality splits has two main downsides. First, creating a bitset for each value would negate both the space and computational efficiency of using this representation. Second, because we consider binary trees, splitting based on an equality against a single attribute-value would tend to result in an unbalanced partition of the instance set, which may reduce the quality of the trees. Therefore, we pre-process these features and group together feature values with similar mean gradient statistics and construct one bitset per group. The mean gradient statistic s_j of a categorical value v_j is defined as the summed gradient values g_i for all instances i that have value v_j. Then, we compute K quantiles q_k of s_j and use these to partition the categorical feature values. A bitset is generated for each quantile q_k. The kth bitset has a one for each instance i that has value v_j and $s_j < q_k$. Because the gradient values change at every iteration, we repeat this grouping procedure every t iterations of the boosting procedure, where t is a user-defined parameter. We refer to this parameter t as the *pre-processing rate*. We found that a value of 20 is a reasonable default.

Numerical features. We treat these in an analogous manner as the high-cardinality categorical features. We estimate K weighted quantiles q_k using the absolute gradient values as weights. We use the quantiles as ordered split candidates, and generate K bitsets such that bitset k has a one in the ith position if instance i's value is less than q_k. Like in the high-cardinality case, we perform this transformation and reconstruct the bitsets every t iterations of the boosting procedure, where t is the pre-processing rate.

3.3 Representing an Instance Set using a Bitset

The instance set I_p of a node p in a tree contains the training data instances that satisfy the split conditions on the path to node p from the root. In existing systems, these instance sets are stored as an array of indexes into the input data, i.e., they are *instance lists*. We introduce the use of *instance bitsets* for this purpose. An instance bitset represents the instance set by having a 1 for instances that are in the instance set, and a zero for others. Fig. 2 illustrates the difference on the left.

We have to contend with one subtlety when using a bitset to represent the instance set: the length of the bitset is always N, which is the number of training

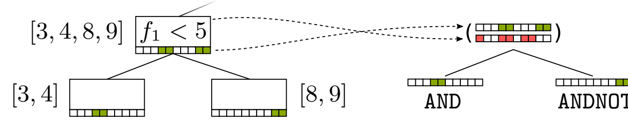

Fig. 2. An illustration of instance lists versus instance bitsets. The dataset comprises 10 instances indexed by $0, 1, \ldots, 9$. Values of feature f_1 are $2, 8, 5, \underline{4}, \underline{3}, 6, 1, 2, \underline{6}, \underline{8}$. The bitset representations of the instance sets are indicated below the node boxes, dark squares are 1s, others are 0s. The instance list $[3, 4, 8, 9]$ using 64-bit integers requires 256 bits, whereas the bitset representation uses only 10 bits.

examples. This is problematic when partitioning the data at each node. Assuming a perfectly balanced tree, the size of an instance set halves at each level of the tree. The length of the bitset remains fixed, but the number of zeros, which represent examples not filtered to the node in question, in the bitset increases. Constructing all nodes of a tree at depth d requires passing over 2^d length-N bitsets, for a total cost of $\mathcal{O}(2^d N)$. In contrast, when using a list-based representation of the instance set, the length of the list halves at each level of the tree. Hence, constructing all nodes at depth d has a total cost of $\mathcal{O}(N)$. Computationally, it is much faster to process a bitset than the list-based representation, even if the list is much shorter than the bitset. The trade-off is that we have to process more bitsets. As the tree gets deeper, the number of bitsets and the fact that each one's length is equal to the number of training examples will eventually make this representation slower than using a list-based representation.

Fortunately, we can exploit the fact that as the depth of the tree increases, the bitsets become progressively sparser by applying a compression scheme to the bitsets. Many compression schemes exist for bitsets, such as Concise [4] and Roaring Bitmaps [1], but most of these schemes are optimized for the general use-case. We have a specific use case and wish to optimize for speed more than for storage efficiency. Therefore, we apply the following simple compression scheme.

We view each bitset as a series of 32-bit blocks. An uncompressed bitset consists of a single array of 32-bit blocks which explicitly stores all blocks in the bitset. A compressed bitset comprises two arrays: Identifiers, which stores sorted 32-bit identifiers of the non-zero blocks, and Blocks, which stores the bit-values of the non-zero 32-bit blocks. For any block i in the bitset, either there exists a j such that $\text{Identifiers}(j) = i$ and the bits of the ith block are $\text{Blocks}(j)$, or the ith block consists of zero-bits. There are two main reasons why we choose 32-bit blocks: (1) 64-bit blocks are too unlikely to be all-zero, and (2) it is hard to efficiently integrate smaller 8- or 16-bit blocks into the `CountOneBits` routine, which heavily relies on SIMD vectorization.

When constructing a tree, after having split a node, we compare the ratio of the number of zero blocks and total number of blocks with a configurable threshold. If the threshold is exceeded, the instance set of the child node is compressed. The threshold determines how aggressively compression is applied.

3.4 Finding the Best Split

Algorithm 2 shows our split finding algorithm based on bitsets and bitslices. As is also the case in XGBoost, LightGBM, and CatBoost, it only considers binary splits. Analogous to the classical histogram-based version, it loops over all features f and initializes a new histogram H_f with bins for each candidate split. The main difference is the inner for-loop. The classical algorithm loops over all instances in the instance set individually, and accumulates the gradient values and instance counts in the histogram bins. In contrast, our algorithm loops over the split candidates which are the bitset representations we generated (see Subsection 3.2). In the body of the loop, the statistics required to evaluate the gain are computed: the sum of the discretized gradients Σ_l of the instances going left is computed using Equation 3, and the number of instances $|I_l|$ going left is computed using the fast `CountOneBits` procedure.

Input: Instance set I_p and gradient sum Σ_p of node p, and gradient bitslice G.
for *all features* f **do**
  H_f = `InitializeNewHist(`f`)`
  for *all candidate splits* s *of* f *with accompanying bitset* B_s **do**
    Σ_l = `BitsliceSum(`$G, B_s \wedge I_p$`)` `/* Equation 3 */`
    $|I_l|$ = `CountOneBits(`$B_s \wedge I_p$`)`
    $H_f[s] = (\Sigma_l, |I_l|)$ `/* Store relevant statistics of` s `in` H_f `*/`
  end
end
Find split s^* with maximum gain in all H_f by evaluating $\frac{\Sigma_l^2}{|I_l|} + \frac{(\Sigma_p - \Sigma_l)^2}{|I_p| - |I_l|} - \frac{\Sigma_p^2}{|I_p|}$.
return $(s^*, B_{s^*}, \Sigma_{l^*}, \Sigma_p - \Sigma_{l^*})$

Alg. 2: Bit-level split finding.

The nested loops fill the histogram H_f for each feature f. Once all histograms are constructed, the best split s^* is determined by comparing the gain of each split. Because we only consider binary splits, we use the property that $I_r = I_p - I_l$, and thus $|I_r| = |I_p| - |I_l|$, and $\Sigma_r = \Sigma_p - \Sigma_l$.

For each feature and each feature split candidate, we perform the `BitsliceSum` and `CountOneBits` operations which are linear in the number of instances. The complexity of the classical algorithm does not include the number of split candidates. This discrepancy is mitigated in two ways. First, the `CountOneBits` operation, which also forms the basis for `BitsliceSum`, is much faster than iterating over the instances individually. Second, we only consider a small number of candidate splits by pre-processing the numerical and high-cardinality categorical features. Note that the classical algorithm does not benefit from a small number of candidate splits, as it can fill the histograms from Algorithm 2 in a single pass over the data.

3.5 Overview of the BitBoost algorithm

Algorithm 1 gave an overview of the generic boosting algorithm. It minimizes the loss function $\mathcal{L}$ by iteratively constructing trees that update the predictions in the direction of the negative gradient. Bitboost adds three additional steps to this algorithm. First, the bitsets for the low-cardinality categorical features are generated once at the start of the learning process and reused thereafter. Second, at the beginning of each iteration the gradients are discretized using a certain number of *discretization bits* and stored in a bitslice. Third, every t iterations, the data bitsets of the high-cardinality categorical and numerical features are regenerated. The parameter t expresses the *pre-processing rate*.

Input: Gradient bitslice G
stack $= \{(I_1, \sum_{i=1}^{N} g_i)\}$ /* Root node; I_1 is all-ones instance bitset */
while *popping* (I_p, Σ_p) *from* stack *succeeds* **do**
  if p *is at maximum depth* **then**
    `ChooseLeafValue(`p`)`
  else
    $(s^*, B_{s^*}, \Sigma_l, \Sigma_r)$ = `FindBestSplit(`I_p, Σ_p, G`)` /* Algorithm 2 */
    $I_l = I_p \wedge B_{s^*}$ and $I_r = I_p \wedge \neg B_{s^*}$ /* Instance sets of children */
    Apply compression to I_l and/or I_r if threshold exceeded.
    `Push(stack,` (I_r, Σ_r)`)` and `Push(stack,` (I_l, Σ_l)`)`
  endif
end

Alg. 3: BitBoost tree construction algorithm.

The individual trees are built in a standard, greedy, top-down manner, as shown in Algorithm 3. The algorithm maintains a stack of nodes to split, which initially only contains the root node. These nodes are split by the best split condition, which is found using Algorithm 2, and the instance sets of the children are generated using simple logical operations. Compression is applied before child nodes are pushed onto the stack when the ratio of zero blocks over the total number of blocks exceeds the configurable threshold.

Leaf nodes hold the final prediction values. To pick the best leaf values, as `ChooseLeafValue` does in Algorithm 3, we use the same strategy as Friedman [6]. For LAD, we use the median gradient value of the instances in the instance set. For least-squares, the values $\Sigma_p/|I_p|$ would be optimal *if the gradient values* g_i *were exact*. This is not the case, so we recompute the mean using the exact gradients. We refer to Friedman's work for the Huber loss and binary log-loss.

4 Experiments

First, we empirically benchmark BitBoost against state-of-the-art systems. Second, we analyze the effect of the four BitBoost-specific parameters on performance.

4.1 Comparison with State-of-the-Art Systems

In this first experiment we compare our system, BitBoost, to three state-of-the-art systems: XGBoost, CatBoost and LightGBM.

Datasets and tasks. We show results for four benchmark datasets that have different characteristics: (1) *Allstate*,[2] an insurance dataset containing 188k instances, each with 116 categorical and 14 continuous features. (2) *Bin-MNIST*,[3], a binary derivative of the famous MNIST dataset of 70k handwritten digits. We converted its 784 features into black and white pixels and predict if a number is less than 5. This dataset represents BitBoost's best case scenario, as all features are binary. (3) *Covertype*,[4] a forestry dataset with 581k instances, each with 44 categorical and 10 continuous features. We consider two classification tasks: for CovType_1, we predict lodgepole-pine versus all (balanced, 48.8% positive); for CovType_2, we predict broadleaf trees versus rest (unbalanced, 2.1% positive). (4) *YouTube*,[5] a dataset with YouTube videos that are trending over a period of time. Numerical features are log-transformed, date and time features are converted into numerical features, and textual features are converted into 373 bag-of-word features (e.g. title and description). This results in a dataset with 121k instances, each with 399 features. We predict the 10-log-transformed view count, i.e., we predict whether a video gets thousands, millions, or billions of views. We use 5-fold cross validation, and average times and accuracies over all folds.

Settings. Most of the parameters are shared across the four systems and only a few of them are system-specific: (1) *Loss function*, we use binary log-loss and hinge loss for classification, and least-squares, Huber loss and LAD for regression. Not all systems support all loss functions: XGBoost does not support Huber loss or LAD. CatBoost does not support Huber loss or hinge loss. LightGBM does not support hinge loss. (2) *Learning rate* is a constant factor that scales the predictions of individual trees to distribute learning over multiple trees. We choose a single learning rate per problem and use the same value for all systems. (3) *Maximum tree depth* limits the depth of the individual trees. We use depth 5 for Allstate, 6 for Covertype and Bin-MNIST, and 9 for YouTube. (4) *Bagging fraction* defines the fraction of instances we use for individual trees. By applying bagging, better generalizing models are build faster. (5) *Feature fraction* defines the fraction of features we use per tree. CatBoost only supports feature selection per tree level. (6) *Minimum split gain* reduces over-fitting by avoiding splits that are not promising. We use a value of 10^{-5} for all systems. (7) *Maximum cardinality K* sets the maximum cardinality of low-cardinality categorical features (BitBoost only). (8) *Pre-processing rate t* sets the rate at which we execute the pre-processing procedure. This is used to avoid pre-processing numerical and high-cardinality features at each iteration (BitBoost only).

[2] `https://www.kaggle.com/c/allstate-claims-severity`
[3] `http://yann.lecun.com/exdb/mnist`
[4] `https://archive.ics.uci.edu/ml/datasets/covertype`
[5] `https://www.kaggle.com/datasnaek/youtube-new`

For XGBoost, we use the histogram tree method since it is faster; for LightGBM, we use the GBDT boosting type; and for CatBoost we use the plain boosting type. We disable XGBoost's and LightGBM's support for sparse features to avoid unrelated differences between the systems. To measure the efficiency of the tree learning algorithms rather than the multi-threading capabilities of the systems, we disable multi-threading and run all experiments on a single core.

Parameter sets were chosen per dataset based on the performance on a validation set and the reported accuracies were evaluated on a separate test set. We provide results for two different parameter sets for BitBoost. BitBoost_A aims to achieve the best possible accuracies, whereas BitBoost_S prioritizes speed. The number of discretization bits, i.e., the precision with which we discretize the gradients, is chosen depending on the problem. Binary log-loss tends to require at least 4 or 8 bits. Hinge loss requires only 2 bits, but is less accurate. We use 4 or 8 bits for least-squares, 2 or 4 for Huber loss. Both least-squares and Huber behave like LAD when using a single bit, the only difference being the lack of resilience to outliers for least-squares. More extensive results and the specific parameter values can be found in the BitBoost repository.

Table 1. Comparison of BitBoost with three state-of-the-art systems. Time is in seconds. Loss is expressed in binary error for classification and mean absolute error (MAE) for regression. The BitBoost_A row shows the results when choosing accuracy over speed. The BitBoost_S row shows the results when choosing speed over accuracy, staying within reasonable accuracy boundaries.

	Allstate		**Covtype_1**		**Covtype_2**		**Bin-MNIST**		**YouTube**	
	Time	Loss	Time	Loss	Time	Loss	Time	Loss	Time	Loss
BitBoost_A	4.8	1159	17.1	12.0	10.7	0.79	4.5	2.78	14.3	0.07
BitBoost_S	1.0	1194	5.4	14.9	7.2	1.02	1.9	3.52	2.5	0.12
LightGBM	12.3	1156	24.1	11.9	21.0	0.71	24.8	2.86	35.0	0.07
XGBoost	11.5	1157	37.0	10.8	35.3	0.63	24.7	2.66	24.9	0.07
CatBoost	82.6	1167	58.1	13.1	52.9	0.91	16.5	3.23	33.6	0.11

Results. Table 1 shows a comparison of the training times and accuracies. On the Allstate dataset, we perform two to over ten times faster than LightGBM and XGBoost, while still achieving accuracies that are comparable to the state of the art. The results for the Covertype dataset show that BitBoost can also handle numerical and high-cardinality features effectively. The Bin-MNIST dataset illustrates that BitBoost is able to construct competitively accurate models using only 20% of the time, and, when giving up some accuracy, can achieve speed-ups of a factor 10. The YouTube dataset requires deeper trees because of its sparse bag-of-word features only splitting off small chunks of instances. The results show that BitBoost also performs well in this case.

In general, BitBoost is able to effectively explore the trade-off between accuracy and speed. Besides the usual parameters like bagging fraction and feature sampling fraction, BitBoost provides one main additional parameter unavailable in other systems: *the number of discretization bits*. This parameter controls the precision at which BitBoost operates, and together with a suitable loss function like LAD or hinge loss, enables trading accuracy for speed in a novel way.

4.2 Effect of the Number of Bits Used to Discretize the Gradient

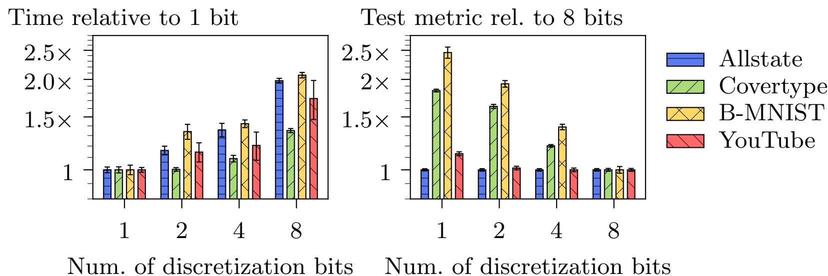

Fig. 3. The effect of the width of bitslice used to discretize the gradient on the model construction time (left) and the performance metric (right). The run times are relative to the fastest option (i.e., one-bit discretization), meaning higher is slower. The performance metric values are relative to the best performing option (i.e., eight-bit discretization), meaning higher is less accurate.

The number of discretization bits used in the bitslice that stores the gradient values will affect BitBoost's performance. Fig. 3 shows the effect of using 1, 2, 4 and 8 bits on run time and predictive performance. The trade-off is clear: using fewer bits decreases the training time, but using more bits improves the predictive performance. Bin-MNIST has the largest effect in terms of run time because it only contains binary features. Hence, there is no work associated with repeatedly converting high-cardinality or real-valued features into bitsets, meaning that the extra computational demand arising from the bigger bitslice has a larger percentage effect on the run time. Note that Allstate's accuracy is unaffected. This is due to the use of LAD loss whose gradient values can be stored with a single bit. We used Huber loss for YouTube, which was able to guide the optimization process effectively using only 2 bits.

4.3 Effect of the Low-Cardinality Boundary K

The parameter K determines which categorical features are considered to be low-cardinality. It will affect run time as higher values of K require considering more split candidates. However, smaller values of K also introduce overhead in terms of the pre-processing needed to cope with high-cardinality attributes,

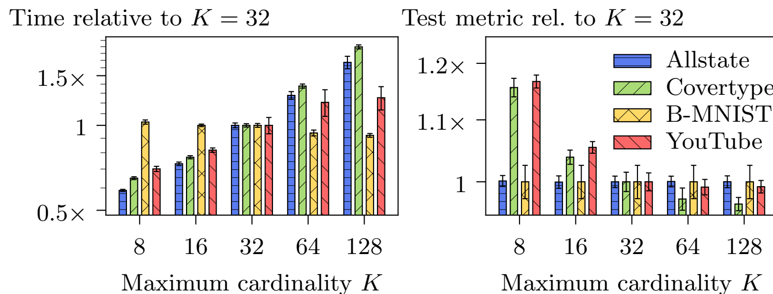

Fig. 4. The effect of K on the run time times relative to $K = 32$ (left); and the accuracies, also relative to $K = 32$ (right).

that is, grouping together similar values for an attribute and generating the associated bitsets. K may also affect predictive performance in two ways: (1) some high-categorical features may have more natural groupings of values than others, and (2) it controls the number of split candidates considered for numerical features.

Fig. 4 shows the effect of varying K on the run time and predictive performance. As K increases, so does the run time, indicating that the extra effort associated with considering more split candidates is more costly than the additional pre-processing necessary to group together similar feature values. Interestingly, the value of K seems to have little effect on the predictive performance for the Allstate dataset, meaning that considering *more fine-grained splits* does not produce better results. The Covertype and YouTube datasets seem to benefit from a higher K value, but the increase in accuracy diminishes for values larger than 32. As the Bin-MNIST dataset only has binary features and does not require any pre-processing, the run time and predictive performance is unaffected by K. In terms of accuracy, a good value of K is likely to be problem specific.

4.4 Effect of Compressing the Instance Bitset on Run Time

Compressing the instance bitset affects run time and involves a trade-off. Always compressing will introduce unnecessary overhead at shallow levels of the tree where there is little sparsity in the instance bitsets. Conversely, never compressing will adversely affect run time when the tree is deeper. To quantify its effect, we measure model construction time as a function of the compression threshold. Fig. 5 plots the run times for compression thresholds ranging from 0.0 (always apply compression) to 1.0 (never apply compression). The trade-off is clearly visible for Covertype and Bin-MNIST (left). Allstate only considers trees of depth 5, which causes instance sets to be more dense on average, making compression less effective. This is confirmed by the results for YouTube that are plotted for different tree depths (right). Shallower trees do not benefit from compression, whereas deeper trees with sparser instance sets do.

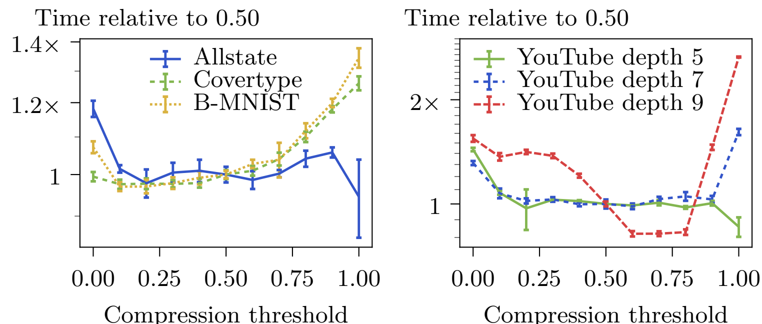

Fig. 5. The model construction time as a function of the compression threshold, which varies from 0.0 (always apply compression) to 1.0 (never apply compression). The time is relative to a threshold of 0.5, which was used in all experiments.

4.5 Effect of Pre-Processing Rate

BitBoost reconverts high-cardinality and continuous features into bitmaps every t iterations. Fig. 6 shows how the model construction time and predictive performance vary as a function of t on the Covertype, Allstate and YouTube datasets. Bin-MNIST is not included as it has neither high-cardinality nor continuous features and thus requires no pre-processing. As expected, run time drops as t increases. Interestingly, it plateaus for values of $t \geq 10$. For Allstate and YouTube, predictive performance is unaffected by this parameter. However, performance slightly degrades for the Covertype dataset for higher values of t.

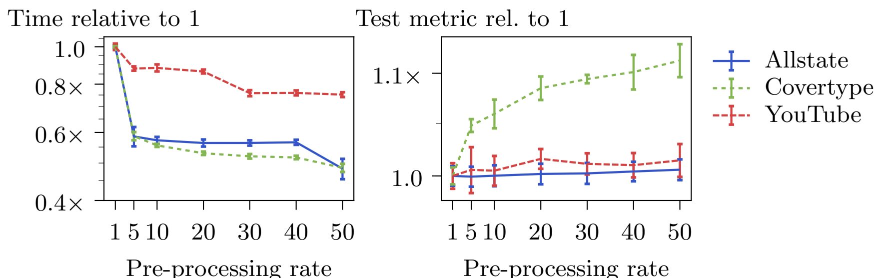

Fig. 6. The effect of the pre-processing rate t on model construction time (left) and predictive performance (right). The results are shown relative to $t = 1$.

5 Conclusion

We have introduced BitBoost, a novel way of integrating the bitset and bitslice data structures into the gradient boosting decision tree algorithm. These data

structure can benefit from bit-level optimizations in modern CPUs to speed up the computation. However, bitslices cannot be used as is in existing gradient boosting decision trees. BitBoost discretizes the gradients such that it has only a limited effect on the predictive performance. The combination of using a bitslice to store the gradient values and representing the data and the instance sets as bitsets reduces the core problem of learning a single tree to summing masked gradients, which can be solved very efficiently. We have empirically shown that this approach can speed up model construction 2 to 10 times compared to state-of-the-art systems without harming predictive performance.

Acknowledgments

LD is supported by the KU Leuven Research Fund (C14/17/070), WM by VLAIO-SBO grant HYMOP (150033), and JD by KU Leuven Research Fund (C14/17/07, C32/17/036), Research Foundation - Flanders (EOS No. 30992574, G0D8819N) and VLAIO-SBO grant HYMOP (150033).

References

1. Chambi, S., Lemire, D., Kaser, O., Godin, R.: Better bitmap performance with roaring bitmaps. Software: Practice and Experience **46**(5), 709–719 (2016)
2. Chan, C.Y., Ioannidis, Y.E.: Bitmap index design and evaluation. In: ACM SIGMOD Record. vol. 27, pp. 355–366. ACM (1998)
3. Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: Proceedings of ACM SIGKDD. pp. 785–794 (2016)
4. Colantonio, A., Pietro, R.D.: Concise: Compressed 'n' composable integer set. Information Processing Letters **110**(16), 644 – 650 (2010)
5. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences **55**(1), 119 – 139 (1997)
6. Friedman, J.H.: Greedy function approximation: A gradient boosting machine. The Annals of Statistics **29**(5), 1189–1232 (2001)
7. Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., Liu, T.Y.: LightGBM: A highly efficient gradient boosting decision tree. In: Advances in Neural Information Processing Systems 30, pp. 3146–3154 (2017)
8. Kurz, N., Muła, W., Lemire, D.: Faster Population Counts Using AVX2 Instructions. The Computer Journal **61**(1), 111–120 (05 2017)
9. Mason, L., Baxter, J., Bartlett, P.L., Frean, M.R.: Boosting algorithms as gradient descent. In: Advances in neural information processing systems. pp. 512–518 (2000)
10. Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A.V., Gulin, A.: CatBoost: unbiased boosting with categorical features. In: Advances in Neural Information Processing Systems 31, pp. 6638–6648 (2018)
11. Rinfret, D., O'Neil, P., O'Neil, E.: Bit-sliced index arithmetic. In: Acm sigmod record. vol. 30, pp. 47–57. ACM (2001)
12. Schapire, R.E.: The strength of weak learnability. Machine Learning **5**(2), 197–227 (1990)
13. Schapire, R.E., Freund, Y., Bartlett, P., Lee, W.S.: Boosting the margin: a new explanation for the effectiveness of voting methods. The Annals of Statistics **26**(5), 1651–1686 (1998)