CatBoost: unbiased boosting with categorical features

Liudmila Prokhorenkova[1,2], **Gleb Gusev**[1,2], **Aleksandr Vorobev**[1],
Anna Veronika Dorogush[1], **Andrey Gulin**[1]
[1]Yandex, Moscow, Russia
[2]Moscow Institute of Physics and Technology, Dolgoprudny, Russia
{ostroumova-la, gleb57, alvor88, annaveronika, gulin}@yandex-team.ru

Abstract

This paper presents the key algorithmic techniques behind CatBoost, a new gradient boosting toolkit. Their combination leads to CatBoost outperforming other publicly available boosting implementations in terms of quality on a variety of datasets. Two critical algorithmic advances introduced in CatBoost are the implementation of *ordered boosting*, a permutation-driven alternative to the classic algorithm, and an innovative algorithm for processing categorical features. Both techniques were created to fight a *prediction shift* caused by a special kind of target leakage present in all currently existing implementations of gradient boosting algorithms. In this paper, we provide a detailed analysis of this problem and demonstrate that proposed algorithms solve it effectively, leading to excellent empirical results.

1 Introduction

Gradient boosting is a powerful machine-learning technique that achieves state-of-the-art results in a variety of practical tasks. For many years, it has remained the primary method for learning problems with heterogeneous features, noisy data, and complex dependencies: web search, recommendation systems, weather forecasting, and many others [5, 26, 29, 32]. Gradient boosting is essentially a process of constructing an ensemble predictor by performing gradient descent in a functional space. It is backed by solid theoretical results that explain how strong predictors can be built by iteratively combining weaker models (*base predictors*) in a greedy manner [17].

We show in this paper that all existing implementations of gradient boosting face the following statistical issue. A prediction model F obtained after several steps of boosting relies on the targets of all training examples. We demonstrate that this actually leads to a shift of the distribution of $F(\mathbf{x}_k) \mid \mathbf{x}_k$ for a training example $\mathbf{x}_k$ from the distribution of $F(\mathbf{x}) \mid \mathbf{x}$ for a test example $\mathbf{x}$. This finally leads to a *prediction shift* of the learned model. We identify this problem as a special kind of target leakage in Section 4. Further, there is a similar issue in standard algorithms of preprocessing categorical features. One of the most effective ways [6, 25] to use them in gradient boosting is converting categories to their target statistics. A target statistic is a simple statistical model itself, and it can also cause target leakage and a prediction shift. We analyze this in Section 3.

In this paper, we propose *ordering principle* to solve both problems. Relying on it, we derive *ordered boosting*, a modification of standard gradient boosting algorithm, which avoids target leakage (Section 4), and a new algorithm for processing categorical features (Section 3). Their combination is implemented as an open-source library[1] called CatBoost (for "Categorical Boosting"), which outperforms the existing state-of-the-art implementations of gradient boosted decision trees — XGBoost [8] and LightGBM [16] — on a diverse set of popular machine learning tasks (see Section 6).

[1]https://github.com/catboost/catboost

Preprint. Work in progress.

2 Background

Assume we observe a dataset of examples $\mathcal{D} = \{(\mathbf{x}_k, y_k)\}_{k=1..n}$, where $\mathbf{x}_k = (x_k^1, \dots, x_k^m)$ is a random vector of m *features* and $y_k \in \mathbb{R}$ is a *target*, which can be either binary or a numerical response. Examples $(\mathbf{x}_k, y_k)$ are independent and identically distributed according to some unknown distribution $P(\cdot, \cdot)$. The goal of a learning task is to train a function $F\colon \mathbb{R}^m \to \mathbb{R}$ which minimizes the expected loss $\mathcal{L}(F) := \mathbb{E}L(y, F(\mathbf{x}))$. Here $L(\cdot, \cdot)$ is a smooth loss function and $(\mathbf{x}, y)$ is a *test example* sampled from P independently of the training set $\mathcal{D}$.

A gradient boosting procedure [12] builds iteratively a sequence of *approximations* $F^t\colon \mathbb{R}^m \to \mathbb{R}$, $t = 0, 1, \dots$ in a greedy fashion. Namely, F^t is obtained from the previous approximation F^{t-1} in an additive manner: $F^t = F^{t-1} + \alpha h^t$, where α is a *step size* and function $h^t\colon \mathbb{R}^m \to \mathbb{R}$ (a *base predictor*) is chosen from a family of functions H in order to minimize the expected loss:

$$h^t = \arg\min_{h \in H} \mathcal{L}(F^{t-1} + h) = \arg\min_{h \in H} \mathbb{E}L(y, F^{t-1}(\mathbf{x}) + h(\mathbf{x})). \tag{1}$$

The minimization problem is usually approached by the *Newton method* using a second–order approximation of $\mathcal{L}(F^{t-1} + h^t)$ at F^{t-1} or by taking a *(negative) gradient step*. Both methods are kinds of functional gradient descent [10, 24]. In particular, the gradient step h^t is chosen in such a way that $h^t(\mathbf{x})$ approximates $-g^t(\mathbf{x}, y)$, where $g^t(\mathbf{x}, y) := \frac{\partial L(y,s)}{\partial s}\big|_{s=F^{t-1}(\mathbf{x})}$. Usually, the least-squares approximation is used:

$$h^t = \arg\min_{h \in H} \mathbb{E}\left(-g^t(\mathbf{x}, y) - h(\mathbf{x})\right)^2. \tag{2}$$

CatBoost is an implementation of gradient boosting, which uses binary decision trees as base predictors. A *decision tree* [4, 10, 27] is a model built by a recursive partition of the feature space $\mathbb{R}^m$ into several disjoint regions (tree nodes) according to the values of some *splitting attributes* a. Attributes are usually binary variables that identify that some feature x^k exceeds some *threshold* t, that is, $a = \mathbb{1}_{\{x^k > t\}}$, where x^k is either numerical or binary feature, in the latter case $t = 0.5$.[2] Each final region (leaf of the tree) is assigned to a value, which is an estimate of the response y in the region for the regression task or the predicted class label in the case of classification problem.[3] In this way, a decision tree h can be written as

$$h(\mathbf{x}) = \sum_{j=1}^{J} b_j \mathbb{1}_{\{\mathbf{x} \in R_j\}}, \tag{3}$$

where R_j are the disjoint regions corresponding to the leaves of the tree.

3 Categorical features

3.1 Related work on categorical features

A categorical feature is one with a discrete set of values called *categories* that are not comparable to each other. One popular technique for dealing with categorical features in boosted trees is *one-hot encoding* [7, 25], i.e., for each category, adding a new binary feature indicating it. However, in the case of high cardinality features (like, e.g., "user ID" feature), such technique leads to infeasibly large number of new features. To address this issue, one can group categories into a limited number of clusters and then apply one-hot encoding. A popular method is to group categories by *target statistics* (TS) that estimate expected target value in each category. Micci-Barreca [25] proposed to consider TS as a new numerical feature instead. Importantly, among all possible partitions of

[2] Alternatively, non-binary splits can be used, e.g., a region can be split according to all values of a categorical feature. However, such splits, compared to binary ones, would lead to either shallow trees (unable to capture complex dependencies) or to very complex trees with exponential number of terminal nodes (having weaker target statistics in each of them). According to [4], the tree complexity has a crucial effect on the accuracy of the model and less complex trees are less prone to overfitting.

[3] In a regression task, splitting attributes and leaf values are usually chosen by the least–squares criterion. Note that, in gradient boosting, a tree is constructed to approximate the negative gradient (see Equation (2)), so it solves a regression problem.

categories into two sets, an optimal split on the training data in terms of logloss, Gini index, MSE can be found among thresholds for the numerical TS feature [4, Section 4.2.2] [11, Section 9.2.4]. In LightGBM [20], categorical features are converted to gradient statistics at each step of gradient boosting. Though providing important information for building a tree, this approach can dramatically increase (i) computation time, since it calculates statistics for each categorical value at each step, and (ii) memory consumption to store which category belongs to which node for each split based on a categorical feature. To overcome this issue, LightGBM groups tail categories into one cluster [21] and thus looses part of information. Besides, the authors claim that it is still better to convert categorical features with high cardinality to numerical features [19]. Note that TS features require calculating and storing only one number per one category.

Thus, using TS as new numerical features seems to be the most efficient method of handling categorical features with minimum information loss. TS are widely-used, e.g., in the click prediction task (click-through rates) [1, 15, 18, 22], where such categorical features as user, region, ad, publisher play a crucial role. We further focus on ways to calculate TS and leave one-hot encoding and gradient statistics out of the scope of the current paper. At the same time, we believe that the ordering principle proposed in this paper is also effective for gradient statistics.

3.2 Target statistics

As discussed in Section 3.1, an effective and efficient way to deal with a categorical feature i is to substitute the category x_k^i of k-th training example with *one* numeric feature equal to some *target statistic* (TS) $\hat{x}_k^i$. Commonly, it estimates the expected target y conditioned by the category: $\hat{x}_k^i \approx \mathbb{E}(y \mid x^i = x_k^i)$.

Greedy TS A straightforward approach is to estimate $\mathbb{E}(y \mid x^i = x_k^i)$ as the average value of y over the training examples with the same category x_k^i [25]. This estimate is noisy for low-frequency categories, and one usually smoothes it by some prior p:

$$\hat{x}_k^i = \frac{\sum_{j=1}^{n} \mathbb{1}_{\{x_j^i = x_k^i\}} \cdot y_j + a\,p}{\sum_{j=1}^{n} \mathbb{1}_{\{x_j^i = x_k^i\}} + a}, \tag{4}$$

where $a > 0$ is a parameter. A common setting for p is the average target value in the dataset [25].

The problem of such *greedy* approach is *target leakage*: feature $\hat{x}_k^i$ is computed using y_k, the target of $\mathbf{x}_k$. This leads to a conditional shift [30]: the distribution of $\hat{x}^i|y$ differs for training and test examples. The following extreme example illustrates how dramatically this may affect the generalization error of the learned model. Assume i-th feature is categorical, all its values are unique, and for each category A, we have $\mathrm{P}(y = 1 \mid x^i = A) = 0.5$ for a classification task. Then, in the training dataset, $\hat{x}_k^i = \frac{y_k + ap}{1+a}$, so it is sufficient to make only one split with threshold $t = \frac{0.5+ap}{1+a}$ to perfectly classify all training examples. However, for all test examples, the value of the greedy TS is p, and the obtained model predicts 0 for all of them if $p < t$ and predicts 1 otherwise, thus having accuracy 0.5 in both cases. To this end, we formulate the following desired property for TS:

P1 $\mathbb{E}(\hat{x}^i \mid y = v) = \mathbb{E}(\hat{x}_k^i \mid y_k = v)$, *where* $(\mathbf{x}_k, y_k)$ *is the* k*-th training example.*

In our example above, $\mathbb{E}(\hat{x}_k^i \mid y_k) = \frac{y_k + ap}{1+a}$ and $\mathbb{E}(\hat{x}^i \mid y) = p$ are different.

There are several ways to avoid this conditional shift. Their general idea is to compute the TS for $\mathbf{x}_k$ on a subset of examples $\mathcal{D}_k \subset \mathcal{D} \setminus \{\mathbf{x}_k\}$ excluding $\mathbf{x}_k$:

$$\hat{x}_k^i = \frac{\sum_{\mathbf{x}_j \in \mathcal{D}_k} \mathbb{1}_{\{x_j^i = x_k^i\}} \cdot y_j + a\,p}{\sum_{\mathbf{x}_j \in \mathcal{D}_k} \mathbb{1}_{\{x_j^i = x_k^i\}} + a}. \tag{5}$$

Holdout TS One way is to partition the training dataset into two parts $\mathcal{D} = \hat{\mathcal{D}}_0 \sqcup \hat{\mathcal{D}}_1$ and use $\mathcal{D}_k = \hat{\mathcal{D}}_0$ for calculating the TS according to (5) and $\hat{\mathcal{D}}_1$ for training (e.g., applied in [8] for Criteo dataset). Though such *holdout* TS satisfies P1, this approach significantly reduces the amount of data used both for training the model and calculating the TS. So, it violates the following desired property:

P2 *Effective usage of all training data for calculating TS features and for learning a model.*

Leave-one-out TS At first glance, a *leave-one-out* technique might work well: take $\mathcal{D}_k = \mathcal{D} \setminus \mathbf{x}_k$ for training examples $\mathbf{x}_k$ and $\mathcal{D}_k = \mathcal{D}$ for test ones [31]. Surprisingly, it does not prevent target leakage. Indeed, consider a constant categorical feature: $x_k^i = A$ for all examples. Let n^+ be the number of examples with $y = 1$, then $\hat{x}_k^i = \frac{n^+ - y_k + a\,p}{n - 1 + a}$ and one can perfectly classify the training dataset by making a split with threshold $t = \frac{n^+ - 0.5 + a\,p}{n - 1 + a}$.

Ordered TS CatBoost uses a more effective strategy. It relies on the ordering principle, the central idea of the paper, and is inspired by online learning algorithms which get training examples sequentially in time [1, 15, 18, 22]). Clearly, the values of TS for each example rely only on the observed history. To adapt this idea to standard offline setting, we introduce an artificial "time", i.e., a random permutation σ of the training examples. Then, for each example, we use all the available "history" to compute its TS, i.e., take $\mathcal{D}_k = \{\mathbf{x}_j : \sigma(j) < \sigma(k)\}$ in Equation (5) for a training example and $\mathcal{D}_k = \mathcal{D}$ for a test one. The obtained *ordered* TS satisfies the requirement P1 and allows to use all training data for learning the model (P2). Note that, if we use only one random permutation, then preceding examples have TS with much higher variance than subsequent ones. To this end, CatBoost uses different permutations for different steps of gradient boosting, see details in Section 5.

4 Prediction shift and ordered boosting

4.1 Prediction shift

In this section, we reveal the problem of prediction shift in gradient boosting, which was neither recognized nor previously addressed. Like in case of TS, prediction shift is caused by a special kind of target leakage. Our solution is called *ordered boosting* and resembles the ordered TS method.

Let us go back to the gradient boosting procedure described in Section 2. In practice, the expectation in (2) is unknown and is usually approximated using the same dataset $\mathcal{D}$:

$$h^t = \arg\min_{h \in H} \frac{1}{n} \sum_{k=1}^{n} \left(-g^t(\mathbf{x}_k, y_k) - h(\mathbf{x}_k)\right)^2. \tag{6}$$

Now we describe and analyze the following chain of shifts:

1. the conditional distribution of the gradient $g^t(\mathbf{x}_k, y_k) \mid \mathbf{x}_k$ (accounting for randomness of $\mathcal{D} \setminus \{\mathbf{x}_k\}$) is shifted from that distribution on a test example $g^t(\mathbf{x}, y) \mid \mathbf{x}$;
2. in turn, base predictor h^t defined by Equation (6) is biased from the solution of Equation (2);
3. this, finally, affects the generalization ability of the trained model F^t.

As in the case of TS, these problems are caused by the target leakage. Indeed, gradients used at each step are estimated using the target values of the same data points the current model F^{t-1} was built on. However, the conditional distribution $F^{t-1}(\mathbf{x}_k) \mid \mathbf{x}_k$ for a training example $\mathbf{x}_k$ is shifted, in general, from the distribution $F^{t-1}(\mathbf{x}) \mid \mathbf{x}$ for a test example $\mathbf{x}$. We call this a *prediction shift*.

Related work on prediction shift The shift of gradient conditional distribution $g^t(\mathbf{x}_k, y_k) \mid \mathbf{x}_k$ was previously mentioned in papers on boosting [3, 13] but was not formally defined. Moreover, even the existence of non-zero shift was not proved theoretically. Based on the out-of-bag estimation [2], Breiman proposed *iterated bagging* [3] which constructs a bagged weak learner at each iteration on the basis of "out-of-bag" residual estimates. However, as we formally show in Appendix E, such residual estimates are still shifted. Besides, the bagging scheme increases learning time by factor of the number of data buckets. Subsampling of the dataset at each iteration proposed by Friedman [13] addresses the problem much more heuristically and also only alleviates it.

Analysis of prediction shift We formally analyze the problem of prediction shift in a simple case of a regression task with the quadratic loss function $L(y, \hat{y}) = (y - \hat{y})^2$.[4] In this case, the negative gradient $-g^t(\mathbf{x}_k, y_k)$ in Equation (6) can be substituted by the residual function $r^{t-1}(\mathbf{x}_k, y_k) := y_k - F^{t-1}(\mathbf{x}_k)$.[5] Assume we have $m = 2$ features x^1, x^2 that are i.i.d. Bernoulli random variables

[4] We restrict the rest of Section 4 to this case, but the approaches of Section 4.2 are applicable to other tasks.

[5] Here we removed the multiplier 2, what does not matter for further analysis.

with $p = 1/2$ and $y = f^*(\mathbf{x}) = c_1 x^1 + c_2 x^2$. Assume we make $N = 2$ steps of gradient boosting with decision stumps (trees of depth 1) and step size $\alpha = 1$. We obtain a model $F = F^2 = h^1 + h^2$. W.l.o.g., we assume that h^1 is based on x^1 and h^2 is based on x^2, what is typical for $|c_1| > |c_2|$ (here we set some asymmetry between x^1 and x^2).

Theorem 1 *1. If two independent samples $\mathcal{D}_1$ and $\mathcal{D}_2$ of size n are used to estimate h^1 and h^2, respectively, using Equation* (6)*, then $\mathbb{E}_{\mathcal{D}_1,\mathcal{D}_2} F^2(\mathbf{x}) = f^*(\mathbf{x}) + O(1/2^n)$ for any $\mathbf{x} \in \{0,1\}^2$.*
2. If the same dataset $\mathcal{D} = \mathcal{D}_1 = \mathcal{D}_2$ is used in Equation (6) *for both h^1 and h^2, then $\mathbb{E}_{\mathcal{D}} F^2(\mathbf{x}) = f^*(\mathbf{x}) - \frac{1}{n-1} c_2 (x^2 - \frac{1}{2}) + O(1/2^n)$.*

This theorem means that the trained model is an unbiased estimate of the true dependence $y = f^*(\mathbf{x})$, when we use independent datasets at each gradient step.[6] On the other hand, if we use the same dataset at each step, we suffer from a bias $-\frac{1}{n-1} c_2 (x^2 - \frac{1}{2})$, which is inversely proportional to the data size n. Also, the value of the bias can depend on the relation f^*: in our example, it is proportional to c_2. We track the chain of shifts for the second part of Theorem 1 in a sketch of the proof below, while the full proof of Theorem 1 is available in Appendix A.

Sketch of the proof. Denote by ξ_{st}, $s, t \in \{0, 1\}$, the number of examples $(\mathbf{x}_k, y_k) \in \mathcal{D}$ with $\mathbf{x}_k = (s, t)$. We have $h^1(s, t) = c_1 s + \frac{c_2 \xi_{s1}}{\xi_{s0} + \xi_{s1}}$. Its expectation $\mathbb{E}(h^1(\mathbf{x}))$ on a test example $\mathbf{x}$ equals $c_1 x^1 + \frac{c_2}{2}$. At the same time, the expectation $\mathbb{E}(h^1(\mathbf{x}_k))$ on a training example $\mathbf{x}_k$ is different and equals $(c_1 x^1 + \frac{c_2}{2}) - c_2 (\frac{2x^2 - 1}{n}) + O(2^{-n})$. That is, we experience a prediction shift of h^1. As a consequence, the expected value of $h^2(\mathbf{x})$ is $\mathbb{E}(h^2(\mathbf{x})) = c_2 (x^2 - \frac{1}{2})(1 - \frac{1}{n-1}) + O(2^{-n})$ on a test example $\mathbf{x}$ and $\mathbb{E}(h^1(\mathbf{x}) + h^2(\mathbf{x})) = f^*(\mathbf{x}) - \frac{1}{n-1} c_2 (x^2 - \frac{1}{2}) + O(1/2^n)$. □

Finally, recall that greedy TS $\hat{x}^i$ can be considered as a simple statistical model predicting the target y and it suffers from a similar problem, conditional shift of $\hat{x}^i_k \mid y_k$, caused by the target leakage, i.e., using y_k to compute $\hat{x}^i_k$.

4.2 Ordered boosting

Here we propose a boosting algorithm which does not suffer from the prediction shift problem described in Section 4.1. Assuming access to an unlimited amount of training data, we can easily construct such an algorithm. At each step of boosting, we sample a new dataset $\mathcal{D}_t$ independently and obtain unshifted residuals by applying the current model to new training examples. In practice, however, labeled data is limited. Assume that we learn a model with I trees. To make the residual $r^{I-1}(\mathbf{x}_k, y_k)$ unshifted, we need to have F^{I-1} trained without the example $\mathbf{x}_k$. Since we need unbiased residuals for all training examples, no examples may be used for training F^{I-1}, which at first glance makes the training process impossible. However, it is possible to maintain a set of models differing by examples used for their training. Then, for calculating the residual on an example, we use a model trained without it. In order to construct such a set of models, we can use the ordering principle previously applied to TS in Section 3.2. To illustrate the idea, assume that we take one random permutation σ of the training examples and maintain n different *supporting* models $M_1, \ldots, M_n$ such that the model M_i is learned using only the first i examples in the permutation. At each step, in order to obtain the residual for j-th sample, we use the model M_{j-1} (see Figure 1). The resulting Algorithm 1 is called *ordered boosting* below. Unfortunately, this algorithm is not feasible in most practical tasks due to the need of training n different models, what increase the complexity and memory requirements by n times. In CatBoost, we implemented a modification of this algorithm on the basis of the gradient boosting algorithm with decision trees as base predictors (GBDT) described in Section 5.

Ordered boosting with categorical features In Sections 3.2 and 4.2 we proposed to use random permutations σ_{cat} and σ_{boost} of training examples for the TS calculation and for ordered boosting, respectively. Combining them in one algorithm, we should take $\sigma_{cat} = \sigma_{boost}$ to avoid prediction shift. This guarantees that target y_i is not used for training M_i (neither for the TS calculation, nor for the gradient estimation). See Appendix F for theoretical guarantees. Empirical results confirming the importance of having $\sigma_{cat} = \sigma_{boost}$ are presented in Appendix G.

[6] Up to an exponentially small term, which occurs for a technical reason.

5 Practical implementation of ordered boosting

CatBoost has two boosting modes, *Ordered* and *Plain*. The latter mode is the standard GBDT algorithm with inbuilt ordered TS. The former mode presents an efficient modification of Algorithm 1. A formal description of the algorithm is included in Appendix B. In this section, we overview the most important implementation details.

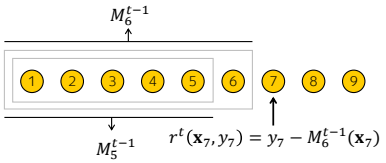

Figure 1: Ordered boosting principle, examples are ordered according to σ.

Algorithm 1: Ordered boosting

input : $\{(\mathbf{x}_k, y_k)\}_{k=1}^n$, I;

$\sigma \leftarrow$ random permutation of $[1, n]$;
$M_i \leftarrow 0$ for $i = 1..n$;
for $t \leftarrow 1$ **to** I **do**
  for $i \leftarrow 1$ **to** n **do**
    $r_i \leftarrow y_i - M_{\sigma(i)-1}(\mathbf{x}_i)$;
  for $i \leftarrow 1$ **to** n **do**
    $\Delta M \leftarrow$ $LearnModel((\mathbf{x}_j, r_j) : \sigma(j) \leq i)$;
    $M_i \leftarrow M_i + \Delta M$;
return M_n

Algorithm 2: Building a tree in CatBoost

input : M, $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, α, L, $\{\sigma_i\}_{i=1}^s$, $Mode$

$grad \leftarrow CalcGradient(L, M, y)$;
$r \leftarrow random(1, s)$;
if $Mode = Plain$ **then**
  $G \leftarrow (grad_r(i)$ for $i = 1..n)$;
if $Mode = Ordered$ **then**
  $G \leftarrow (grad_{r,\sigma_r(i)-1}(i)$ for $i = 1..n)$;
$T \leftarrow$ empty tree;
foreach *step of top-down procedure* **do**
  foreach *candidate split* c **do**
    $T_c \leftarrow$ add split c to T;
    if $Mode = Plain$ **then**
      $\Delta(i) \leftarrow \text{avg}(grad_r(p)$ for $p : leaf_r(p) = leaf_r(i))$ for $i = 1..n$;
    if $Mode = Ordered$ **then**
      $\Delta(i) \leftarrow \text{avg}(grad_{r,\sigma_r(i)-1}(p)$ for $p : leaf_r(p) = leaf_r(i), \sigma_r(p) < \sigma_r(i))$ for $i = 1..n$;
    $loss(T_c) \leftarrow \cos(\Delta, G)$
  $T \leftarrow \arg\min_{T_c}(loss(T_c))$
if $Mode = Plain$ **then**
  $M_{r'}(i) \leftarrow M_{r'}(i) - \alpha\,\text{avg}(grad_{r'}(p)$ for $p : leaf_{r'}(p) = leaf_{r'}(i))$ for $r' = 1..s$, $i = 1..n$;
if $Mode = Ordered$ **then**
  $M_{r',j}(i) \leftarrow M_{r',j}(i) - \alpha\,\text{avg}(grad_{r',j}(p)$ for $p : leaf_{r'}(p) = leaf_{r'}(i), \sigma_{r'}(p) \leq j)$ for $r' = 1..s$, $i = 1..n$, $j \geq \sigma_{r'}(i) - 1$;
return T, M

At the start, CatBoost generates $s + 1$ independent random permutations of the training dataset. The permutations $\sigma_1, \ldots, \sigma_s$ are used for evaluation of splits that define tree structures (i.e., the internal nodes), while σ_0 serves for choosing the leaf values b_j of the obtained trees (see Equation (3)). For examples with short history in a given permutation, both TS and predictions used by ordered boosting ($M_{\sigma(i)-1}(\mathbf{x}_i)$ in Algorithm 1) have a high variance. Therefore, using only one permutation may increase the variance of the final model predictions, while several permutations allow us to reduce this effect in a way we further describe. The advantage of several permutations is confirmed by our experiments in Section 6.

Building a tree In CatBoost, base predictors are oblivious decision trees [9, 14] also called decision tables [23]. Term oblivious means that the same splitting criterion is used across an entire level of the tree. Such trees are balanced, less prone to overfitting, and allow speeding up execution at testing time significantly. The procedure of building a tree in CatBoost is described in Algorithm 2.

In the Ordered boosting mode, during the learning process, we maintain the supporting models $M_{r,j}$, where $M_{r,j}(i)$ is the current prediction for the i-th example based on the first j examples in the permutation σ_r. At each iteration t of the algorithm, we sample a random permutation σ_r from $\{\sigma_1, \ldots, \sigma_s\}$ and construct a tree T_t on the basis of it. First, for categorical features, all TS are computed according to this permutation. Second, the permutation affects the tree learning procedure.

Table 1: Computational complexity.

Procedure	CalcGradient	Build T	Calc all b_j^t	Update M	Calc ordered TS
Complexity for iteration t	$O(s \cdot n)$	$O(\|C\| \cdot n)$	$O(n)$	$O(s \cdot n)$	$O(N_{TS,t} \cdot n)$

Namely, based on $M_{r,j}(i)$, we compute the corresponding gradients $grad_{r,j}(i) = \frac{\partial L(y_i,s)}{\partial s}\big|_{s=M_{r,j}(i)}$. Then, while constructing a tree, we approximate the gradient G in terms of the cosine similarity $\cos(\cdot,\cdot)$, where, for each example i, we take the gradient $grad_{r,\sigma(i)-1}(i)$ (it is based only on the previous examples in σ_r). At the candidate splits evaluation step, the leaf value $\Delta(i)$ for example i is obtained individually by averaging the gradients $grad_{r,\sigma_r(i)-1}$ of the preceding examples p lying in the same leaf $leaf_r(i)$ the example i belongs to. Note that $leaf_r(i)$ depends on the chosen permutation σ_r, because σ_r can influence the values of ordered TS for example i. When the tree structure T_t (i.e., the sequence of splitting attributes) is built, we use it to boost all the models $M_{r',j}$. Let us stress that *one common* tree structure T_t is used for all the models, but this tree is added to different $M_{r',j}$ with different sets of leaf values depending on r' and j, as described in Algorithm 2.

The Plain boosting mode works similarly to a standard GBDT procedure, but, if categorical features are present, it maintains s supporting models M_r corresponding to TS based on $\sigma_1, \ldots, \sigma_s$.

Choosing leaf values Given all the trees constructed, the leaf values of the final model F are calculated by the standard gradient boosting procedure equally for both modes. Training examples i are matched to leaves $leaf_0(i)$, i.e., we use permutation σ_0 to calculate TS here. When the final model F is applied to a new example at testing time, we use TS calculated on the whole training data according to Section 3.2.

Complexity In our practical implementation, we use one important trick, which significantly reduces the computational complexity of the algorithm. Namely, in the Ordered mode, instead of $O(s\,n^2)$ values $M_{r,j}(i)$, we store and update only the values $M'_{r,j}(i) := M_{r,2^j}(i)$ for $j = 1, \ldots, \lceil \log_2 n \rceil$ and all i with $\sigma_r(i) \leq 2^{j+1}$, what reduces the number of maintained supporting predictions to $O(s\,n)$. See Appendix B for the pseudocode of this modification of Algorithm 2.

In Table 1, we present the computational complexity of different components of both CatBoost modes per one iteration (see Appendix C.1 for the proof). Here $N_{TS,t}$ is the number of TS to be calculated at the iteration t and C is the set of candidate splits to be considered at the given iteration. It follows that our implementation of ordered boosting with decision trees has the same asymptotic complexity as the standard GBDT with ordered TS. In comparison with other types of TS (Section 3.2), ordered TS slow down by s times the procedures $CalcGradient$, updating supporting models M, and computation of TS.

Feature combinations Another important detail of CatBoost is using combinations of categorical features as additional categorical features which capture high-order dependencies like joint information of user ID and ad topic in the task of ad click prediction. The number of possible combinations grows exponentially with the number of categorical features in the dataset, and it is infeasible to process all of them. CatBoost constructs combinations in a greedy way. Namely, for each split of a tree, CatBoost combines (concatenates) all categorical features (and their combinations) already used for previous splits in the current tree with all categorical features in the dataset. Combinations are converted to TS on the fly.

Other important details Finally, let us discuss two options of the CatBoost algorithm not covered above. The first one is subsampling of the dataset at each iteration of boosting procedure, as proposed by Friedman [13]. We claimed earlier in Section 4.1 that this approach alone cannot fully avoid the problem of prediction shift. However, since it has proved effective, we implemented it in both modes of CatBoost as a Bayesian bootstrap procedure. Specifically, before training a tree according to Algorithm 2, we assign a weight $w_i = a_i^t$ to each example i, where a_i^t are generated according to the Bayesian bootstrap procedure (see [28, Section 2]). These weights are used as multipliers for gradients $grad_r(i)$ and $grad_{r,j}(i)$, when we calculate $\Delta(i)$ and the components of the vector $\Delta - G$ to define $loss(T_c)$.

The second option deals with first several examples in a permutation. For examples i with small values $\sigma_r(i)$, the variance of $grad_{r,\sigma_r(i)-1}(i)$ can be high. Therefore, we discard $\Delta(i)$ from the beginning of the permutation, when we calculate $loss(T_c)$ in Algorithm 2. Particularly, we eliminate the corresponding components of vectors G and Δ when calculating the cosine similarity between them.

6 Experiments

Comparison with baselines We compare our algorithm with the most popular open-source libraries — XGBoost and LightGBM — on several well-known machine learning tasks. The detailed description of the experimental setup together with dataset descriptions is available in Appendix D. The source code of the experiment is available, and the results can be reproduced.[7] For all learning algorithms, we preprocess categorical features using the ordered TS method described in Section 3.2. The parameter tuning and training were performed on 4/5 of the data and the testing was performed on the remaining 1/5.[8] The results measured by logloss and zero-one loss are presented in Table 2 (the absolute values for the baselines are in Appendix G). For CatBoost, we used Ordered boosting mode in this experiment.[9] One can see that CatBoost outperforms other algorithms on all the considered datasets. We also measured statistical significance of improvements presented in Table 2: except three datasets (Appetency, Churn and Upselling) the improvements are statistically significant with p-value $\ll 0.01$ measured by the paired one-tailed t-test.

To demonstrate that our implementation of plain boosting is an appropriate baseline for our research, we show that a *raw setting* of CatBoost provides state-of-the-art quality. Particularly, we take a setting of CatBoost, which is close to classical GBDT [12], and compare it with the baseline boosting implementations in Appendix G. Experiments show that this raw setting differs from the baselines insignificantly.

Table 2: Comparison with baselines: logloss / zero-one loss (relative increase for baselines).

	CatBoost	LightGBM	XGBoost
Adult	**0.270 / 0.127**	+2.4% / +1.9%	+2.2% / +1.0%
Amazon	**0.139 / 0.044**	+17% / +21%	+17% / +21%
Click	**0.392 / 0.156**	+1.2% / +1.2%	+1.2% / +1.2%
Epsilon	**0.265 / 0.109**	+1.5% / +4.1%	+11% / +12%
Appetency	**0.072 / 0.018**	+0.4% / +0.2%	+0.4% / +0.7%
Churn	**0.232 / 0.072**	+0.1% / +0.6%	+0.5% / +1.6%
Internet	**0.209 / 0.094**	+6.8% / +8.6%	+7.9% / +8.0%
Upselling	**0.166 / 0.049**	+0.3% / +0.1%	+0.04% / +0.3%
Kick	**0.286 / 0.095**	+3.5% / +4.4%	+3.2% / +4.1%

Table 3: Plain boosting mode: logloss, zero-one loss and their change relative to Ordered boosting mode.

	Logloss	Zero-one loss
Adult	0.272 (+1.1%)	0.127 (-0.1%)
Amazon	0.139 (-0.6%)	0.044 (-1.5%)
Click	0.392 (-0.05%)	0.156 (+0.19%)
Epsilon	0.266 (+0.6%)	0.110 (+0.9%)
Appetency	0.072 (+0.5%)	0.018 (+1.5%)
Churn	0.232 (-0.06%)	0.072 (-0.17%)
Internet	0.217 (+3.9%)	0.099 (+5.4%)
Upselling	0.166 (+0.1%)	0.049 (+0.4%)
Kick	0.285 (-0.2%)	0.095 (-0.1%)

We also empirically analyzed the running times of the algorithms on Epsilon dataset. The details of the comparison can be found in Appendix C.2. To summarize, we obtained that CatBoost Plain and LightGBM are the fastest ones followed by Ordered mode, which is about 1.7 times slower.

Ordered and Plain modes In this section, we compare two essential boosting modes of CatBoost: Plain and Ordered. First, we compared their performance on all the considered datasets, the results are presented in Table 3. It can be clearly seen that Ordered mode is particularly useful on small datasets. Indeed, the largest benefit from Ordered is observed on Adult and Internet datasets, which are relatively small (less than 40K training examples), which supports our hypothesis that a higher bias negatively affects the performance. Indeed, according to Theorem 1 and our reasoning in Section 4.1, bias is expected to be larger for smaller datasets (however, it can also depend on other properties of the dataset, e.g., on the dependency between features and target). In order to further validate this hypothesis, we make the following experiment: we train CatBoost in Ordered and Plain modes on randomly filtered datasets and compare the obtained losses, see Figure 2. As we expected,

[7] `https://github.com/catboost/benchmarks/tree/master/quality_benchmarks`

[8] For Epsilon, we use default parameters instead of parameter tuning due to large running time for all algorithms. We tune only the number of trees to avoid overfitting.

[9] The numbers for CatBoost in Table 2 may slightly differ from the corresponding numbers in our GitHub repository, since we use another version of CatBoost with all the discussed features implemented.

for smaller datasets the relative performance of Plain mode becomes worse. To save space, here we present the results only for logloss; the figure for zero-one loss is similar.

We also compare Ordered and Plain modes in the above-mentioned raw setting of CatBoost in Appendix G and conclude that the advantage of Ordered mode is not caused by interaction with specific CatBoost options.

Table 4: Comparison of target statistics, relative change in logloss / zero-one loss compared to ordered TS.

	Greedy	Holdout	Leave-one-out
Adult	+1.1% / +0.8%	+2.1% / +2.0%	+5.5% / +3.7%
Amazon	+40% / +32%	+8.3% / +8.3%	+4.5% / +5.6%
Click	+13% / +6.7%	+1.5% / +0.5%	+2.7% / +0.9%
Appetency	+24% / +0.7%	+1.6% / -0.5%	+8.5% / +0.7%
Churn	+12% / +2.1%	+0.9% / +1.3%	+1.6% / +1.8%
Internet	+33% / +22%	+2.6% / +1.8%	+27% / +19%
Upselling	+57% / +50%	+1.6% / +0.9%	+3.9% / +2.9%
Kick	+22% / +28%	+1.3% / +0.32%	+3.7% / +3.3%

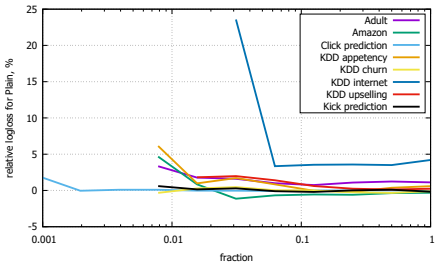

Figure 2: Relative error of Plain boosting mode compared to Ordered boosting mode depending on the fraction of the dataset.

Analysis of target statistics We compare different TSs introduced in Section 3.2 as options of CatBoost in Ordered boosting mode keeping all other algorithmic details the same; the results can be found in Table 4. Here, to save space, we present only relative increase in loss functions for each algorithm compared to CatBoost with ordered TS. Note that the ordered TS used in CatBoost significantly outperform all other approaches. Also, among the baselines, the holdout TS is the best for most of the datasets since it does not suffer from conditional shift discussed in Section 3.2 (P1); still, it is worse than CatBoost due to less effective usage of training data (P2). Leave-one-out is usually better than the greedy TS, but it can be much worse on some datasets, e.g., on Adult. The reason is that the greedy TS suffer from low-frequency categories, while the leave-one-out TS suffer also from high-frequency ones, and on Adult all the features have high frequency.

Finally, let us note that in Table 4 we combine Ordered mode of CatBoost with different TSs. To generalize these results, we also made a similar experiment by combining different TS with Plain mode, used in standard gradient boosting. The obtained results and conclusions turned out to be very similar to the ones discussed above.

Feature combinations The effect of feature combinations discussed in Section 5 is demonstrated in Figure 3 in Appendix G. In average, changing the number c_{max} of features allowed to be combined from 1 to 2 provides an outstanding improvement of logloss by 1.86% (reaching 11.3%), changing from 1 to 3 yields 2.04%, and further increase of c_{max} does not influence the performance significantly.

Number of permutations The effect of the number s of permutations on the performance of CatBoost is presented in Figure 4 in Appendix G. In average, increasing s slightly decreases logloss, e.g., by 0.19% for $s = 3$ and by 0.38% for $s = 9$ compared to $s = 1$.

7 Conclusion

In this paper, we identify and analyze the problem of prediction shifts present in all existing implementations of gradient boosting. We propose a general solution, ordered boosting with ordered TS, which solves the problem. This idea is implemented in CatBoost, which is a new gradient boosting library. Empirical results demonstrate that CatBoost outperforms leading GBDT packages and leads to new state-of-the-art results on common benchmarks.

Acknowledgments

We are very grateful to Mikhail Bilenko for important references and advices that lead to theoretical analysis of this paper, as well as suggestions on the presentation. We also thank Pavel Serdyukov for

many helpful discussions and valuable links, Nikita Kazeev, Nikita Dmitriev, Stanislav Kirillov and Victor Omelyanenko for help with experiments.

References

[1] L. Bottou and Y. L. Cun. Large scale online learning. In *Advances in neural information processing systems*, pages 217–224, 2004.

[2] L. Breiman. Out-of-bag estimation, 1996.

[3] L. Breiman. Using iterated bagging to debias regressions. *Machine Learning*, 45(3):261–277, 2001.

[4] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

[5] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168. ACM, 2006.

[6] B. Cestnik et al. Estimating probabilities: a crucial task in machine learning. In *ECAI*, volume 90, pages 147–149, 1990.

[7] O. Chapelle, E. Manavoglu, and R. Rosales. Simple and scalable response prediction for display advertising. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 5(4):61, 2015.

[8] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM, 2016.

[9] M. Ferov and M. Modrỳ. Enhancing lambdamart using oblivious trees. *arXiv preprint arXiv:1609.05610*, 2016.

[10] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. *The annals of statistics*, 28(2):337–407, 2000.

[11] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[12] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[13] J. H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002.

[14] A. Gulin, I. Kuralenok, and D. Pavlov. Winning the transfer learning track of yahoo!'s learning to rank challenge with yetirank. In *Yahoo! Learning to Rank Challenge*, pages 63–76, 2011.

[15] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9. ACM, 2014.

[16] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3149–3157, 2017.

[17] M. Kearns and L. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM (JACM)*, 41(1):67–95, 1994.

[18] J. Langford, L. Li, and T. Zhang. Sparse online learning via truncated gradient. *Journal of Machine Learning Research*, 10(Mar):777–801, 2009.

[19] LightGBM. Categorical feature support. `http://lightgbm.readthedocs.io/en/latest/Advanced-Topics.html#categorical-feature-support`, 2017.

[20] LightGBM. Optimal split for categorical features. http://lightgbm.readthedocs.io/en/latest/Features.html#optimal-split-for-categorical-features, 2017.

[21] LightGBM. feature_histogram.cpp. https://github.com/Microsoft/LightGBM/blob/master/src/treelearner/feature_histogram.hpp, 2018.

[22] X. Ling, W. Deng, C. Gu, H. Zhou, C. Li, and F. Sun. Model ensemble for click prediction in bing search ads. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 689–698. International World Wide Web Conferences Steering Committee, 2017.

[23] Y. Lou and M. Obukhov. Bdt: Gradient boosted decision tables for high accuracy and scoring efficiency. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1893–1901. ACM, 2017.

[24] L. Mason, J. Baxter, P. L. Bartlett, and M. R. Frean. Boosting algorithms as gradient descent. In *Advances in neural information processing systems*, pages 512–518, 2000.

[25] D. Micci-Barreca. A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems. *ACM SIGKDD Explorations Newsletter*, 3(1):27–32, 2001.

[26] B. P. Roe, H.-J. Yang, J. Zhu, Y. Liu, I. Stancu, and G. McGregor. Boosted decision trees as an alternative to artificial neural networks for particle identification. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 543(2):577–584, 2005.

[27] L. Rokach and O. Maimon. Top–down induction of decision trees classifiers — a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(4):476–487, 2005.

[28] D. B. Rubin. The bayesian bootstrap. *The annals of statistics*, pages 130–134, 1981.

[29] Q. Wu, C. J. Burges, K. M. Svore, and J. Gao. Adapting boosting for information retrieval measures. *Information Retrieval*, 13(3):254–270, 2010.

[30] K. Zhang, B. Schölkopf, K. Muandet, and Z. Wang. Domain adaptation under target and conditional shift. In *International Conference on Machine Learning*, pages 819–827, 2013.

[31] O. Zhang. Winning data science competitions. https://www.slideshare.net/ShangxuanZhang/winning-data-science-competitions-presented-by-owen-zhang, 2015.

[32] Y. Zhang and A. Haghani. A gradient boosting method to improve travel time prediction. *Transportation Research Part C: Emerging Technologies*, 58:308–324, 2015.

Appendices

A Proof of Theorem 1

A.1 Proof for the case $\mathcal{D}_1 = \mathcal{D}_2$

Let us denote by A the event that each leaf in both stumps h^1 and h^2 contains at least one example, i.e., there exists at least one $\mathbf{x} \in \mathcal{D}$ with $\mathbf{x}^i = s$ for all $i \in \{1, 2\}$, $s \in \{0, 1\}$. All further reasonings are given conditioning on A. Note that the probability of A is $1 - O\left(2^{-n}\right)$, therefore we can assign an arbitrary value to any empty leaf during the learning process, and the choice of the value will affect all expectations we calculate below by $O\left(2^{-n}\right)$.

Denote by ξ_{st}, $s,t \in \{0,1\}$, the number of examples $\mathbf{x}_k \in \mathcal{D}$ with $\mathbf{x}_k = (s,t)$. The value of the first stump h^1 in the region $\{x^1 = s\}$ is the average value of y_k over examples from $\mathcal{D}$ belonging to this region. That is,

$$h^1(0,t) = \frac{\sum_{j=1}^n c_2 \mathbb{1}_{\{x_j=(0,1)\}}}{\sum_{j=1}^n \mathbb{1}_{\{x_j^1=0\}}} = \frac{c_2\xi_{01}}{\xi_{00}+\xi_{01}}\,,$$

$$h^1(1,t) = \frac{\sum_{j=1}^n c_1 \mathbb{1}_{\{x_j^1=1\}} + c_2 \mathbb{1}_{\{x_j=(1,1)\}}}{\sum_{j=1}^n \mathbb{1}_{\{x_j^1=1\}}} = c_1 + \frac{c_2\xi_{11}}{\xi_{10}+\xi_{11}}\,.$$

Summarizing, we obtain

$$h^1(s,t) = c_1 s + \frac{c_2\xi_{s1}}{\xi_{s0}+\xi_{s1}}. \tag{7}$$

Note that, by conditioning on A, we guarantee that the denominator $\xi_{s0}+\xi_{s1}$ is not equal to zero.

Now we derive the expectation $\mathbb{E}(h^1(\mathbf{x}))$ of prediction h^1 for a test example $\mathbf{x} = (s,t)$.

It is easy to show that $\mathbb{E}\left(\frac{\xi_{s1}}{\xi_{s0}+\xi_{s1}} \mid A\right) = \frac{1}{2}$. Indeed, due to the symmetry we have $\mathbb{E}\left(\frac{\xi_{s1}}{\xi_{s0}+\xi_{s1}} \mid A\right) = \mathbb{E}\left(\frac{\xi_{s0}}{\xi_{s0}+\xi_{s1}} \mid A\right)$ and the sum of these expectations is $\mathbb{E}\left(\frac{\xi_{s0}+\xi_{s1}}{\xi_{s0}+\xi_{s1}} \mid A\right) = 1$. So, by taking the expectation of (7), we obtain the following proposition.

Proposition 1 *We have* $\mathbb{E}(h^1(s,t) \mid A) = c_1 s + \frac{c_2}{2}$.

It means that the conditional expectation $\mathbb{E}(h^1(\mathbf{x}) \mid \mathbf{x} = (s,t), A)$ on a test example $\mathbf{x}$ equals $c_1 s + \frac{c_2}{2}$, since $\mathbf{x}$ and h^1 are independent.

Prediction shift of h^1 In this paragraph, we show that the conditional expectation $\mathbb{E}(h^1(\mathbf{x}_l) \mid \mathbf{x}_l = (s,t), A)$ on a training example $\mathbf{x}_l$ is shifted for any $l = 1,\ldots,n$, because the model h^1 is fitted to $\mathbf{x}_l$. This is an auxiliary result, which is not used directly for proving the theorem, but helps to track the chain of obtained shifts.

Proposition 2 *The conditional expectation is*

$$\mathbb{E}(h^1(\mathbf{x}_l) \mid \mathbf{x}_l = (s,t), A) = c_1 s + \frac{c_2}{2} - c_2\left(\frac{2t-1}{n}\right) + O(2^{-n})\,.$$

Proof. Let us introduce the following notation

$$\alpha_{sk} = \frac{\mathbb{1}_{\{x_k=(s,1)\}}}{\xi_{s0}+\xi_{s1}}\,.$$

Then, we can rewrite the conditional expectation as

$$c_1 s + c_2 \sum_{k=1}^n \mathbb{E}(\alpha_{sk} \mid \mathbf{x}_l = (s,t), A)\,.$$

Lemma 1 below implies that $\mathbb{E}(\alpha_{sl} \mid \mathbf{x}_l = (s,t), A) = \frac{2t}{n}$. For $k \neq l$, we have

$$\mathbb{E}(\alpha_{sk} \mid \mathbf{x}_l = (s,t), A) = \frac{1}{4}\,\mathbb{E}\left(\frac{1}{\xi_{s0}+\xi_{s1}} \mid \mathbf{x}_l = (s,t), \mathbf{x}_k = (s,1), A\right)$$
$$= \frac{1}{2n}\left(1 - \frac{1}{n-1} + \frac{n-2}{(2^{n-1}-2)\,(n-1)}\right)$$

due to Lemma 2 below. Finally, we obtain

$$\mathbb{E}(h^1(\mathbf{x}_l) \mid \mathbf{x}_l = (s,t)) = c_1 s + c_2\left(\frac{2t}{n} + (n-1)\frac{1}{2n}\left(1 - \frac{1}{n-1}\right)\right)$$
$$+ O\left(2^{-n}\right) = c_1 s + \frac{c_2}{2} - c_2\left(\frac{2t-1}{n}\right) + O(2^{-n}).$$

□

Lemma 1 $\mathbb{E}\left(\frac{1}{\xi_{s0}+\xi_{s1}} \mid \mathbf{x}_1 = (s,t), A\right) = \frac{2}{n}$.

Proof. Note that given $\mathbf{x}_1 = (s,t)$, A corresponds to the event that there is an example with $x^1 = 1-s$ and (possibly another) example with $x^2 = 1-t$ among $\mathbf{x}_2, \ldots, \mathbf{x}_n$.

Note that $\xi_{s0} + \xi_{s1} = \sum_{j=1}^{n} \mathbb{1}_{\{x_j^1 = s\}}$. For $k = 1, \ldots, n-1$, we have

$$\mathrm{P}(\xi_{s0} + \xi_{s1} = k \mid \mathbf{x}_1 = (s,t), A) = \frac{\mathrm{P}(\xi_{s0} + \xi_{s1} = k, A \mid \mathbf{x}_1 = (s,t))}{\mathrm{P}(A \mid \mathbf{x}_1 = (s,t))} = \frac{\binom{n-1}{k-1}}{2^{n-1}\left(1 - 2^{-(n-1)}\right)},$$

since $\mathbb{1}_{\{x_1^1 = s\}} = 1$ when $\mathbf{x}_1 = (s,t)$ with probability 1, $\sum_{j=2}^{n} \mathbb{1}_{\{x_j^1 = s\}}$ is a binomial variable independent of $\mathbf{x}_1$, and an example with $x^1 = 1-s$ exists whenever $\xi_{s0} + \xi_{s1} = k < n$ and $\mathbf{x}_1 = (s,t)$ (while the existence of one with $x^2 = 1-t$ is an independent event). Therefore, we have

$$\mathbb{E}\left(\frac{1}{\xi_{s0}+\xi_{s1}} \mid \mathbf{x}_1 = (s,t), A\right) = \sum_{k=1}^{n-1} \frac{1}{k} \frac{\binom{n-1}{k-1}}{2^{n-1}-1} = \frac{1}{n\left(2^{n-1}-1\right)} \sum_{k=1}^{n-1} \binom{n}{k} = \frac{2}{n}.$$

□

Lemma 2 *We have*

$$\mathbb{E}\left(\frac{1}{\xi_{s0}+\xi_{s1}} \mid \mathbf{x}_1 = (s,t_1), \mathbf{x}_2 = (s,t_2), A\right) = \frac{2}{n}\left(1 - \frac{1}{n-1} + \frac{n-2}{\left(2^{n-1}-2\right)(n-1)}\right).$$

Proof. Similarly to the previous proof, for $k = 2, \ldots, n-1$, we have

$$\mathrm{P}\left(\xi_{s0} + \xi_{s1} = k \mid \mathbf{x}_1 = (s,t_1), \mathbf{x}_2 = (s,t_2), A\right) = \frac{\binom{n-2}{k-2}}{2^{n-2}\left(1 - 2^{-(n-2)}\right)}.$$

Therefore,

$$\begin{aligned}
\mathbb{E}\left(\frac{1}{\xi_{s0}+\xi_{s1}} \mid \mathbf{x}_1 = (s,t_1), \mathbf{x}_2 = (s,t_2), A\right) &= \frac{1}{2^{n-2}\left(1 - 2^{-(n-1)}\right)} \sum_{k=2}^{n-1} \frac{\binom{n-2}{k-2}}{k} \\
&= \frac{1}{2^{n-2}-1} \sum_{k=2}^{n-1} \binom{n-2}{k-2} \left(\frac{1}{k-1} - \frac{1}{(k-1)k}\right) \\
&= \frac{1}{2^{n-2}-1} \sum_{k=2}^{n-1} \left(\frac{1}{n-1}\binom{n-1}{k-1} - \frac{1}{n(n-1)}\binom{n}{k}\right) = \\
&= \frac{1}{2^{n-2}-1}\left(\frac{1}{n-1}(2^{n-1}-2) - \frac{1}{n(n-1)}(2^n - n - 2)\right) = \\
&= \frac{2}{n}\left(1 - \frac{1}{n-1} + \frac{n-2}{\left(2^{n-1}-2\right)(n-1)}\right).
\end{aligned}$$

□

Bias of the model $h^1 + h^2$ Proposition 2 shows that the values of the model h^1 on training examples are shifted with respect to the ones on test examples. The next step is to show how this can lead to a bias of the trained model, if we use the same dataset for building both h^1 and h^2. Namely, we derive the expected value of $h^1(s,t) + h^2(s,t)$ and obtain a bias according to the following result.

Proposition 3 *If both h^1 and h^2 are built using the same dataset $\mathcal{D}$, then*

$$\mathbb{E}\left(h^1(s,t) + h^2(s,t) \mid A\right) = f^*(s,t) - \frac{1}{n-1} c_2 \left(t - \frac{1}{2}\right) + O(1/2^n).$$

Proof. The residual after the first step is

$$f^*(s,t) - h^1(s,t) = c_2 \left(t - \frac{\xi_{s1}}{\xi_{s0} + \xi_{s1}} \right) .$$

Therefore, we get

$$h^2(s,t) = \frac{c_2}{\xi_{0t} + \xi_{1t}} \left(\left(t - \frac{\xi_{01}}{\xi_{00} + \xi_{01}} \right) \xi_{0t} + \left(t - \frac{\xi_{11}}{\xi_{10} + \xi_{11}} \right) \xi_{1t} \right) ,$$

which is equal to

$$-c_2 \left(\frac{\xi_{00}\xi_{01}}{(\xi_{00} + \xi_{01})(\xi_{00} + \xi_{10})} + \frac{\xi_{10}\xi_{11}}{(\xi_{10} + \xi_{11})(\xi_{00} + \xi_{10})} \right)$$

for $t = 0$ and to

$$c_2 \left(\frac{\xi_{00}\xi_{01}}{(\xi_{00} + \xi_{01})(\xi_{01} + \xi_{11})} + \frac{\xi_{10}\xi_{11}}{(\xi_{10} + \xi_{11})(\xi_{01} + \xi_{11})} \right)$$

for $t = 1$. The expected values of all four ratios are equal due to symmetries, and they are equal to $\frac{1}{4}\left(1 - \frac{1}{n-1}\right) + O(2^{-n})$ according to Lemma 3 below. So, we obtain

$$\mathbb{E}(h^2(s,t) \mid A) = (2t-1)\frac{c_2}{2} \left(1 - \frac{1}{n-1} \right) + O(2^{-n})$$

and

$$\mathbb{E}(h^1(s,t) + h^2(s,t) \mid A) = f^*(s,t) - c_2 \frac{1}{n-1} \left(t - \frac{1}{2} \right) + O(2^{-n}) .$$

□

Lemma 3 *We have*

$$\mathbb{E}\left(\frac{\xi_{00}\xi_{01}}{(\xi_{00} + \xi_{01})(\xi_{01} + \xi_{11})} \mid A \right) = \frac{1}{4} \left(1 - \frac{1}{n-1} \right) + O(2^{-n}) .$$

Proof. First, linearity implies

$$\mathbb{E}\left(\frac{\xi_{00}\xi_{01}}{(\xi_{00} + \xi_{01})(\xi_{01} + \xi_{11})} \mid A \right) = \sum_{i,j} \mathbb{E}\left(\frac{\mathbb{1}_{\mathbf{x}_i = (0,0), \mathbf{x}_j = (0,1)}}{(\xi_{00} + \xi_{01})(\xi_{01} + \xi_{11})} \mid A \right) .$$

Taking into account that all terms are equal, the expectation can be written as $\frac{n(n-1)}{4^2} a$, where

$$a = \mathbb{E}\left(\frac{1}{(\xi_{00} + \xi_{01})(\xi_{01} + \xi_{11})} \mid \mathbf{x}_1 = (0,0), \mathbf{x}_2 = (0,1), A \right) .$$

A key observation is that $\xi_{00} + \xi_{01}$ and $\xi_{01} + \xi_{11}$ are two independent binomial variables: the former one is the number of k such that $x_k^1 = 0$ and the latter one is the number of k such that $x_k^2 = 1$. Moreover, they (and also their inverses) are also conditionally independent given that first two observations of the Bernoulli scheme are known ($\mathbf{x}_1 = (0,0), \mathbf{x}_2 = (0,1)$) and given A. This conditional independence implies that a is the product of $\mathbb{E}\left(\frac{1}{\xi_{00} + \xi_{01}} \mid \mathbf{x}_1 = (0,0), \mathbf{x}_2 = (0,1), A \right)$ and $\mathbb{E}\left(\frac{1}{\xi_{01} + \xi_{11}} \mid \mathbf{x}_1 = (0,0), \mathbf{x}_2 = (0,1), A \right)$. The first factor equals $\frac{2}{n}\left(1 - \frac{1}{n-1} + O(2^{-n})\right)$ according to Lemma 2. The second one is equal to $\mathbb{E}\left(\frac{1}{\xi_{01} + \xi_{11}} \mid \mathbf{x}_1 = (0,0), \mathbf{x}_2 = (0,1) \right)$ since A does not bring any new information about the number of k with $x_k^2 = 1$ given $\mathbf{x}_1 = (0,0), \mathbf{x}_2 = (0,1)$. So, according to Lemma 4 below, the second factor equals $\frac{2}{n-1}(1 + O(2^{-n}))$. Finally, we obtain

$$\mathbb{E}\left(\frac{\xi_{00}\xi_{01}}{(\xi_{00} + \xi_{01})(\xi_{01} + \xi_{11})} \right) = \frac{n(n-1)}{4^2} \frac{4}{n(n-1)} \left(1 - \frac{1}{n-1} \right) + O(2^{-n}) = \frac{1}{4} \left(1 - \frac{1}{n-1} \right) + O(2^{-n}).$$

□

Lemma 4 $\mathbb{E}\left(\frac{1}{\xi_{01}+\xi_{11}} \mid \mathbf{x}_1=(0,0), \mathbf{x}_2=(0,1)\right) = \frac{2}{n-1} - \frac{1}{2^{n-2}(n-1)}$.

Proof. Similarly to the proof of Lemma 2, we have

$$\mathrm{P}(\xi_{01}+\xi_{11}=k \mid \mathbf{x}_1=(0,0), \mathbf{x}_2=(0,1)) = \binom{n-2}{k-1} 2^{-(n-2)}\,.$$

Therefore, we get

$$\begin{aligned}\mathbb{E}\left(\frac{1}{\xi_{01}+\xi_{11}} \mid \mathbf{x}_1=(0,0), \mathbf{x}_2=(0,1)\right) &= \sum_{k=1}^{n-1} \frac{1}{k}\binom{n-2}{k-1} 2^{-(n-2)} \\ &= \frac{2^{-(n-2)}}{n-1}\sum_{k=1}^{n-1}\binom{n-1}{k} = \frac{2}{n-1} - \frac{1}{2^{n-2}(n-1)}\,.\end{aligned}$$

□

A.2 Proof for independently sampled $\mathcal{D}_1$ and $\mathcal{D}_2$

Assume that we have an additional sample $\mathcal{D}_2 = \{\mathbf{x}_{n+k}\}_{k=1..n}$ for building h^2. Now A denotes the event that each leaf in h^1 contains at least one example from $\mathcal{D}_1$ and each leaf in h^2 contains at least one example from $\mathcal{D}_2$.

Proposition 4 *If h^2 is built using dataset $\mathcal{D}_2$, then*

$$\mathbb{E}(h^1(s,t) + h^2(s,t) \mid A) = f^*(s,t)\,.$$

Proof.

Let us denote by ξ'_{st} the number of examples $\mathbf{x}_{n+k}$ that are equal to (s,t), $k=1,\dots,n$.

First, we need to derive the expectation $\mathbb{E}(h^2(s,t))$ of h^2 on a test example $\mathbf{x}=(s,t)$. Similarly to the proof of Proposition 3, we get

$$\begin{aligned}h^2(s,0) &= -c_2\left(\frac{\xi'_{00}\xi_{01}}{(\xi_{00}+\xi_{01})(\xi'_{00}+\xi'_{10})} + \frac{\xi'_{10}\xi_{11}}{(\xi_{10}+\xi_{11})(\xi'_{00}+\xi'_{10})}\right)\,, \\ h^2(s,1) &= c_2\left(\frac{\xi_{00}\xi'_{01}}{(\xi_{00}+\xi_{01})(\xi'_{01}+\xi'_{11})} + \frac{\xi_{10}\xi'_{11}}{(\xi_{10}+\xi_{11})(\xi'_{01}+\xi'_{11})}\right)\,.\end{aligned}$$

Due to the symmetries, the expected values of all four fractions above are equal. Also, due to the independence of ξ_{ij} and ξ'_{kl}, we have

$$\mathbb{E}\left(\frac{\xi'_{00}\xi_{01}}{(\xi_{00}+\xi_{01})(\xi'_{00}+\xi'_{10})} \mid A\right) = \mathbb{E}\left(\frac{\xi_{01}}{\xi_{00}+\xi_{01}} \mid A\right)\mathbb{E}\left(\frac{\xi'_{00}}{\xi'_{00}+\xi'_{10}} \mid A\right) = \frac{1}{4}\,.$$

Therefore, $\mathbb{E}(h^2(s,0) \mid A) = -\frac{c_2}{2}$ and $\mathbb{E}(h^2(s,1) \mid A) = \frac{c_2}{2}$.

Summing up, $\mathbb{E}(h^2(s,t) \mid A) = c_2 t - \frac{c_2}{2}$ and $\mathbb{E}(h^1(s,t) + h^2(s,t) \mid A) = c_1 s + c_2 t$. □

B Formal description of CatBoost algorithm

In this section, we formally describe the CatBoost algorithm introduced in Section 5. In Algorithm 3, we provide more information on particular details including the speeding up trick introduced in paragraph "Complexity". The key step of the CatBoost algorithm is the procedure of building a tree described in detail in Function $BuildTree$. To obtain the formal description of the CatBoost algorithm without the speeding up trick, one should replace $\lceil \log_2 n \rceil$ by n in line 6 of Algorithm 3 and use Algorithm 2 instead of Function $BuildTree$.

We use Function $GetLeaf(\mathbf{x}, T, \sigma_r)$ to describe how examples are matched to leaves $leaf_r(i)$. Given an example with features $\mathbf{x}$, we calculate ordered TS on the basis of the permutation σ_r and then choose the leaf of tree T corresponding to features $\mathbf{x}$ enriched by the obtained ordered TS. Using $ApplyMode$ instead of a permutation in function $GetLeaf$ in line 15 of Algorithm 3 means that we use TS calculated on the whole training data to apply the trained model on a new example.

Algorithm 3: CatBoost

input : $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, I, α, L, s, $Mode$

$\sigma_r \leftarrow$ random permutation of $[1, n]$ for $r = 0..s$;
$M_0(i) \leftarrow 0$ for $i = 1..n$;
if $Mode = Plain$ **then**
  $M_r(i) \leftarrow 0$ for $r = 1..s$, $i : \sigma_r(i) \leq 2^{j+1}$;
if $Mode = Ordered$ **then**
  for $j \leftarrow 1$ **to** $\lceil \log_2 n \rceil$ **do**
    $M_{r,j}(i) \leftarrow 0$ for $r = 1..s$, $i = 1..2^{j+1}$;
for $t \leftarrow 1$ **to** I **do**
  $T_t,\ \{M_r\}_{r=1}^s \leftarrow BuildTree(\{M_r\}_{r=1}^s, \{(\mathbf{x}_i, y_i)\}_{i=1}^n, \alpha, L, \{\sigma_i\}_{i=1}^s, Mode)$;
  $leaf_0(i) \leftarrow GetLeaf(\mathbf{x}_i, T_t, \sigma_0)$ for $i = 1..n$;
  $grad_0 \leftarrow CalcGradient(L, M_0, y)$;
  foreach *leaf* j *in* T_t **do**
    $b_j^t \leftarrow -\text{avg}(grad_0(i)$ for $i :\ leaf_0(i) = j)$;
  $M_0(i) \leftarrow M_0(i) + \alpha b^t_{leaf_0(i)}$ for $i = 1..n$;
return $F(\mathbf{x}) = \sum_{t=1}^I \sum_j \alpha\, b_j^t \mathbb{1}_{\{GetLeaf(\mathbf{x}, T_t, ApplyMode) = j\}}$;

Function $BuildTree$

input : M,$\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, α, L, $\{\sigma_i\}_{i=1}^s$, $Mode$

$grad \leftarrow CalcGradient(L, M, y)$;
$r \leftarrow random(1, s)$;
if $Mode = Plain$ **then**
  $G \leftarrow (grad_r(i)$ for $i = 1..n)$;
if $Mode = Ordered$ **then**
  $G \leftarrow (grad_{r, \lfloor \log_2(\sigma_r(i)-1) \rfloor}(i)$ for $i = 1..n)$;
$T \leftarrow$ empty tree;
foreach *step of top-down procedure* **do**
  foreach *candidate split* c **do**
    $T_c \leftarrow$ add split c to T;
    $leaf_r(i) \leftarrow GetLeaf(\mathbf{x}_i, T_c, \sigma_r)$ for $i = 1..n$;
    if $Mode = Plain$ **then**
      $\Delta(i) \leftarrow \text{avg}(grad_r(p)$ for $p :\ leaf_r(p) = leaf_r(i))$ for $i = 1..n$;
    if $Mode = Ordered$ **then**
      $\Delta(i) \leftarrow \text{avg}(grad_{r, \lfloor \log_2(\sigma_r(i)-1) \rfloor}(p)$ for $p :\ leaf_r(p) = leaf_r(i), \sigma_r(p) < \sigma_r(i))$ for $i = 1..n$;
    $loss(T_c) \leftarrow \cos(\Delta, G)$
  $T \leftarrow \arg\min_{T_c}(loss(T_c))$
$leaf_{r'}(i) \leftarrow GetLeaf(\mathbf{x}_i, T, \sigma_{r'})$ for $r' = 1..s$, $i = 1..n$;
if $Mode = Plain$ **then**
  $M_{r'}(i) \leftarrow M_{r'}(i) - \alpha\, \text{avg}(grad_{r'}(p)$ for $p :\ leaf_{r'}(p) = leaf_{r'}(i))$ for $r' = 1..s$, $i = 1..n$;
if $Mode = Ordered$ **then**
  for $j \leftarrow 1$ **to** $\lceil \log_2 n \rceil$ **do**
    $M_{r',j}(i) \leftarrow M_{r',j}(i) - \alpha\, \text{avg}(grad_{r',j}(p)$ for $p :\ leaf_{r'}(p) = leaf_{r'}(i), \sigma_{r'}(p) \leq 2^j)$ for $r' = 1..s,\ i : \sigma_{r'}(i) \leq 2^{j+1}$;
return T, M

C Time complexity analysis

C.1 Theoretical analysis

We present the computational complexity of different components of any of the two modes of CatBoost per one iteration in Table 5.

Table 5: Computational complexity.

Procedure	CalcGradient	Build T	Calc values b_j^t	Update M	Calc ordered TS
Complexity for iteration t	$O(s \cdot n)$	$O(\|C\| \cdot n)$	$O(n)$	$O(s \cdot n)$	$O(N_{TS,t} \cdot n)$

We first prove these asymptotics for the Ordered mode. For this purpose, we estimate the number N_{pred} of predictions $M_{r,j}(i)$ to be maintained:

$$N_{pred} = (s+1) \cdot \sum_{j=1}^{\lceil \log_2 n \rceil} 2^{j+1} < (s+1) \cdot 2^{\log_2 n + 3} = 8(s+1)n\,.$$

Then, obviously, the complexity of CalcGradient is $O(N_{pred}) = O(s \cdot n)$. The complexity of leaf values calculation is $O(n)$, since each example i is included only in averaging operation in leaf $leaf_0(i)$.

Calculation of the ordered TS for one categorical feature can be performed sequentially in the order of the permutation by n additive operations for calculation of n partial sums and n division operations. Thus, the overall complexity of the procedure is $O(N_{TS,t} \cdot n)$, where $N_{TS,t}$ is the number of TS which were not calculated on the previous iterations. Since the leaf values $\Delta(i)$ calculated in line 15 of Function $BuildTree$ can be considered as ordered TS, where gradients play the role of targets, the complexity of building a tree T is $O(|C| \cdot n)$, where C is the set of candidate splits to be considered at the given iteration. Finally, for updating the supporting models (lines 22-23 in Function $BuildTree$), we need to perform one averaging operation for each $j = 1, \ldots, \lceil \log_2 n \rceil$, and each maintained gradient $grad_{r',j}(p)$ is included in one averaging operation. Thus, the number of operations is bounded by the number of the maintained gradients $grad_{r',j}(p)$, which is equal to $N_{pred} = O(s \cdot n)$.

To finish the proof, note that any component of the Plain mode is not less efficient than the same one of the Ordered mode but, at the same time, cannot be more efficient than corresponding asymptotics from Table 5.

C.2 Empirical analysis

It is quite hard to compare different boosting libraries in terms of training speed. Every algorithm has a vast number of parameters which affect training speed, quality and model size in a non-obvious way. Different libraries have their unique quality/training speed trade-off's and they cannot be compared without domain knowledge (e.g., is 0.5% of quality metric worth it to train a model 3-4 times slower?). Plus for each library it is possible to obtain almost the same quality with different ensemble sizes and parameters. As a result, one cannot compare libraries by time needed to obtain a certain level of quality. As a result, we could give only some insights of how fast our implementation could train a model of a fixed size. We use Epsilon dataset and we measure mean tree construction time one can achieve without using feature subsampling and/or bagging by CatBoost (both Ordered and Plain modes), XGBoost (we use histogram-based version, which is faster) and LightGBM. For XGBoost and CatBoost we use the default tree depth equal to 6, for LightGBM we set leaves count to 64 to have comparable results. We run all experiments on the same machine with Intel Xeon E3-12xx 2.6GHz, 16 cores, 64GB RAM and run all algorithms with 16 threads.

We set such learning rate that algorithms start to overfit approximately after constructing about 7000 trees and measure the average time to train ensembles of 8000 trees. Mean tree construction time is presented in Table 6. Note that CatBoost Plain and LightGBM are the fastest ones followed by Ordered mode, which is about 1.7 times slower, which is expected.

Table 6: Comparison of running times on Epsilon

	time per tree
CatBoost Plain	**1.1 s**
CatBoost Ordered	1.9 s
XGBoost	3.9 s
LightGBM	**1.1 s**

Table 7: Description of the datasets.

Dataset name	Instances	Features	Description
Adult[11]	48842	15	Prediction task is to determine whether a person makes over 50K a year. Extraction was done by Barry Becker from the 1994 Census database. A set of reasonably clean records was extracted using the following conditions: (AAGE>16) and (AGI>100) and (AFNLWGT>1) and (HRSWK>0)
Amazon[12]	32769	10	Data from the Kaggle Amazon Employee challenge.
Click Prediction[13]	399482	12	This data is derived from the 2012 KDD Cup. The data is subsampled to 1% of the original number of instances, downsampling the majority class (click=0) so that the target feature is reasonably balanced (5 to 1). The data is about advertisements shown alongside search results in a search engine, and whether or not people clicked on these ads. The task is to build the best possible model to predict whether a user will click on a given ad.
Epsilon[14]	400000	2000	PASCAL Challenge 2008.
KDD appetency[15]	50000	231	Small version of KDD 2009 Cup data.
KDD churn[16]	50000	231	Small version of KDD 2009 Cup data.
KDD Internet[17]	10108	69	Binarized version of the original dataset. The multi-class target feature is converted to a two-class nominal target feature by re-labeling the majority class as positive ('P') and all others as negative ('N'). Originally converted by Quan Sun.
KDD upselling[18]	50000	231	Small version of KDD 2009 Cup data.
Kick prediction[19]	72983	36	Data from "Don't Get Kicked!" Kaggle challenge.

Finally, let us note that CatBoost has a highly efficient GPU implementation. The detailed description and comparison of the running times are beyond the scope of the current article, but these experiments can be found on the corresponding GitHub page.[10]

D Experimental setup

D.1 Description of the datasets

The datasets used in our experiments are described in Table 7.

[10] `https://github.com/catboost/benchmarks/tree/master/gpu_training`

D.2 Experimental settings

In our experiments, we evaluate different modifications of CatBoost and two popular gradient boosting libraries: LightGBM and XGBoost. All the code needed for reproducing our experiments is published on our GitHub[20].

Train-test splits Each dataset was randomly split into training set (80%) and test set (20%). We denote them as D_{full_train} and D_{test}.

We use 5-fold cross-validation to tune parameters of each model on the training set. Accordingly, D_{full_train} is randomly split into 5 equally sized parts $D_1, \ldots, D_5$ (sampling is stratified by classes). These parts are used to construct 5 training and validation sets: $D_i^{train} = \cup_{j \neq i} D_j$ and $D_i^{val} = D_i$ for $1 \leq i \leq 5$.

Preprocessing We applied the following steps to datasets with missing values:

- For categorical variables, missing values are replaced with a special value, i.e., we treat missing values as a special category;
- For numerical variables, missing values are replaced with zeros, and a binary dummy feature for each imputed feature is added.

For XGBoost, LightGBM and the raw setting of CatBoost (see Appendix G), we perform the following preprocessing of categorical features. For each pair of datasets $(D_i^{train},\ D_i^{val})$, $i = 1, \ldots, 5$, and $(D_{full\_train}, D_{test})$, we preprocess the categorical features by calculating ordered TS (described in Section 3.2) on the basis of a random permutation of the examples of the first (training) dataset. All the permutations are generated independently. The resulting values of TS are considered as numerical features by any algorithm to be evaluated.

Parameter Tuning We tune all the key parameters of each algorithm by 50 steps of the sequential optimization algorithm Tree Parzen Estimator implemented in Hyperopt library[21] (mode *algo=tpe.suggest*) by minimizing logloss. Below is the list of the tuned parameters and their distributions the optimization algorithm started from:

XGBoost:

- 'eta': Log-uniform distribution $[e^{-7}, 1]$
- 'max_depth': Discrete uniform distribution $[2, 10]$
- 'subsample': Uniform $[0.5, 1]$
- 'colsample_bytree': Uniform $[0.5, 1]$
- 'colsample_bylevel': Uniform $[0.5, 1]$
- 'min_child_weight': Log-uniform distribution $[e^{-16}, e^5]$
- 'alpha': Mixed: $0.5 \cdot$ Degenerate at 0 + $0.5 \cdot$ Log-uniform distribution $[e^{-16}, e^2]$
- 'lambda': Mixed: $0.5 \cdot$ Degenerate at 0 + $0.5 \cdot$ Log-uniform distribution $[e^{-16}, e^2]$
- 'gamma': Mixed: $0.5 \cdot$ Degenerate at 0 + $0.5 \cdot$ Log-uniform distribution $[e^{-16}, e^2]$

[11] https://archive.ics.uci.edu/ml/datasets/Adult
[12] https://www.kaggle.com/c/amazon-employee-access-challenge
[13] http://www.kdd.org/kdd-cup/view/kdd-cup-2012-track-2
[14] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html
[15] http://www.kdd.org/kdd-cup/view/kdd-cup-2009/Data
[16] http://www.kdd.org/kdd-cup/view/kdd-cup-2009/Data
[17] https://kdd.ics.uci.edu/databases/internet_usage/internet_usage.html
[18] http://www.kdd.org/kdd-cup/view/kdd-cup-2009/Data
[19] https://www.kaggle.com/c/DontGetKicked
[20] https://github.com/catboost/benchmarks/tree/master/quality_benchmarks
[21] https://github.com/hyperopt/hyperopt

LightGBM:

- 'learning_rate': Log-uniform distribution $[e^{-7}, 1]$
- 'num_leaves' : Discrete log-uniform distribution $[1, e^{7}]$
- 'feature_fraction': Uniform $[0.5, 1]$
- 'bagging_fraction': Uniform $[0.5, 1]$
- 'min_sum_hessian_in_leaf': Log-uniform distribution $[e^{-16}, e^{5}]$
- 'min_data_in_leaf': Discrete log-uniform distribution $[1, e^{6}]$
- 'lambda_l1': Mixed: $0.5 \cdot$ Degenerate at 0 + $0.5 \cdot$ Log-uniform distribution $[e^{-16}, e^{2}]$
- 'lambda_l2': Mixed: $0.5 \cdot$ Degenerate at 0 + $0.5 \cdot$ Log-uniform distribution $[e^{-16}, e^{2}]$

CatBoost:

- 'learning_rate': Log-uniform distribution $[e^{-7}, 1]$
- 'random_strength': Discrete uniform distribution over a set $\{1, 20\}$
- 'one_hot_max_size': Discrete uniform distribution over a set $\{0, 25\}$
- 'l2_leaf_reg': Log-uniform distribution $[1, 10]$
- 'bagging_temperature': Uniform $[0, 1]$
- 'gradient_iterations' : Discrete uniform distribution over a set $\{1, 10\}$

Next, having fixed all other parameters, we perform exhaustive search for the number of trees in the interval $[1, 5000]$. We collect logloss value for each training iteration from 1 to 5000 for each of the 5 folds. Then we choose the iteration with minimum logloss averaged over 5 folds.

For evaluation, each algorithm was run on the preprocessed training data D_{full_train} with the tuned parameters. The resulting model was evaluated on the preprocessed test set D_{test}.

Versions of the libraries

- catboost (0.3)
- xgboost (0.6)
- scikit-learn (0.18.1)
- scipy (0.19.0)
- pandas (0.19.2)
- numpy (1.12.1)
- lightgbm (0.1)
- hyperopt (0.0.2)
- h2o (3.10.4.6)
- R (3.3.3)

E Analysis of iterated bagging

Based on the out-of-bag estimation [2], Breiman proposed *iterated bagging* [3] which simultaneously constructs K models F_i, $i = 1, \ldots, K$, associated with K independently bootstrapped subsamples $\mathcal{D}_i$. At t-th step of the process, models F_i^t are grown from their predecessors F_i^{t-1} as follows. The current estimate M_j^t at example j is obtained as the average of the outputs of all models F_k^{t-1} such that $j \notin \mathcal{D}_k$. The term h_i^t is built as a predictor of the residuals $r_j^t := y_j - M_j^t$ (targets minus current estimates) on $\mathcal{D}_i$. Finally, the models are updated: $F_i^t := F_i^{t-1} + h_i^t$. Unfortunately, the residuals r_j^t used in this procedure are not unshifted (in terms of Section 4.1), or unbiased (in terms of *iterated bagging*), because each model F_i^t depends on each observation $(\mathbf{x}_j, y_j)$ by construction. Indeed,

although h_k^t does not use y_j directly, if $j \notin \mathcal{D}_k$, it still uses $M_{j'}^{t-1}$ for $j' \in \mathcal{D}_k$, which, in turn, can depend on $(\mathbf{x}_j, y_j)$.

Also note that computational complexity of this algorithm exceeds one of classic GBDT by factor of K.

F Ordered boosting with categorical features

In Sections 3.2 and 4.2, we proposed to use some random permutations σ_{cat} and σ_{boost} of training examples for the TS calculation and for ordered boosting, respectively. Now, being combined in one algorithm, should these two permutations be somehow dependent? We argue that they should coincide. Otherwise, there exist examples $\mathbf{x}_i$ and $\mathbf{x}_j$ such that $\sigma_{boost}(i) < \sigma_{boost}(j)$ and $\sigma_{cat}(i) > \sigma_{cat}(j)$. Then, the model $M_{\sigma_{boost}(j)}$ is trained using TS features of, in particular, example $\mathbf{x}_i$, which are calculated using y_j. In general, it may shift the prediction $M_{\sigma_{boost}(j)}(\mathbf{x}_j)$. To avoid such a shift, we set $\sigma_{cat} = \sigma_{boost}$ in CatBoost. In the case of the ordered boosting (Algorithm 1) with sliding window TS[22] it guarantees that the prediction $M_{\sigma(i)-1}(\mathbf{x}_i)$ is not shifted for $i = 1, \dots, n$, since, first, the target y_i was not used for training $M_{\sigma(i)-1}$ (neither for the TS calculation, nor for the gradient estimation) and, second, the distribution of TS $\hat{x}^i$ conditioned by the target value is the same for a training example and a test example with the same value of feature x^i.

G Experimental results

Comparison with baselines In Section 6 we demonstrated that the strong setting of CatBoost, including ordered TS, Ordered mode and feature combinations, outperforms the baselines. Detailed experimental results of that comparison are presented in Table 8.

Table 8: Comparison with baselines: logloss / zero-one loss, relative increase is presented in the brackets.

	CatBoost	LightGBM	XGBoost
Adult	**0.2695 / 0.1267**	0.2760 (+2.4%) / 0.1291 (+1.9%)	0.2754 (+2.2%) / 0.1280 (+1.0%)
Amazon	**0.1394 / 0.0442**	0.1636 (+17%) / 0.0533 (+21%)	0.1633 (+17%) / 0.0532 (+21%)
Click	**0.3917 / 0.1561**	0.3963 (+1.2%) / 0.1580 (+1.2%)	0.3962 (+1.2%) / 0.1581 (+1.2%)
Epsilon	**0.2647 / 0.1086**	0.2703 (+1.5%) / 0.114 (+4.1%)	0.2993 (+11%) / 0.1276 (+12%)
Appetency	**0.0715 / 0.01768**	0.0718 (+0.4%) / 0.01772 (+0.2%)	0.0718 (+0.4%) / 0.01780 (+0.7%)
Churn	**0.2319 / 0.0719**	0.2320 (+0.1%) / 0.0723 (+0.6%)	0.2331 (+0.5%) / 0.0730 (+1.6%)
Internet	**0.2089 / 0.0937**	0.2231 (+6.8%) / 0.1017 (+8.6%)	0.2253 (+7.9%) / 0.1012 (+8.0%)
Upselling	**0.1662 / 0.0490**	0.1668 (+0.3%) / 0.0491 (+0.1%)	0.1663 (+0.04%) / 0.0492 (+0.3%)
Kick	**0.2855 / 0.0949**	0.2957 (+3.5%) / 0.0991 (+4.4%)	0.2946 (+3.2%) / 0.0988 (+4.1%)

In this section, we empirically show that our implementation of GBDT provides state-of-the-art quality and thus is an appropriate basis for building CatBoost by adding different improving options including the above-mentioned ones. For this purpose, we compare with baselines a *raw setting* of CatBoost which is as close to classical GBDT [12] as possible. Namely, we use CatBoost in GPU mode with the following parameters: *−−boosting–type Plain −−border–count 255 −−dev–bootstrap–type DiscreteUniform −−gradient–iterations 1 −−random–strength 0 −−depth 6*. Besides, we tune the parameters *dev–sample–rate, learning–rate, l2–leaf–reg* instead of the parameters described in paragraph "Parameter tuning" of Appendix D.2 by 50 steps of the optimization algorithm. Further, for all the algorithms, all categorical features are transformed to ordered TS on the basis of a random permutation (the same for all algorithms) of training examples at the preprocessing step. The resulting TS are used as numerical features in the training process. Thus, no CatBoost options dealing with categorical features are used. As a result, the main difference of the raw setting of CatBoost compared with XGBoost and LightGBM is using oblivious trees as base predictors.

For the baselines, we take the same results as in Table 8. As we can see from Table 9, in average, the difference between all the algorithms is rather small: the raw setting of CatBoost outperforms the

[22]Ordered TS calculated on the basis of a fixed number of preceding examples (both for training and test examples).

Table 9: Comparison with baselines: logloss / zero-one loss (relative increase for baselines).

	Raw setting of CatBoost	LightGBM	XGBoost
Adult	0.2800 / 0.1288	-1.4% / +0.2%	-1.7% / -0.6%
Amazon	0.1631 / 0.0533	+0.3% / 0%	+0.1% / -0.2%
Click	0.3961 / 0.1581	+0.1% / -0.1%	0% / 0%
Appetency	0.0724 / 0.0179	-0.8% / -1.0%	-0.8% / -0.4%
Churn	0.2316 / 0.0718	+0.2% / +0.7%	+0.6% / +1.6%
Internet	0.2223 / 0.0993	+0.4% / +2.4%	+1.4% / +1.9%
Upselling	0.1679 / 0.0493	-0.7% / -0.4%	-1.0% / -0.2%
Kick	0.2955 / 0.0993	+0.1% / -0.4%	-0.3% / -0.2%
Average		-0.2% / +0.2%	-0.2% / +0.2%

baselines in terms of zero-one loss by 0.2% while they are better in terms of logloss by 0.2%. Thus, taking into account that a GBDT model with oblivious trees can significantly speed up execution at testing time [23], our implementation of GBDT is very reasonable choice to build CatBoost on.

Ordered and Plain modes In Section 6 we showed experimentally that Ordered mode of CatBoost significantly outperforms Plain mode in the strong setting of CatBoost, including ordered TS and feature combinations. In this section, we verify that this advantage is not caused by interaction with these and other specific CatBoost options. For this purpose, we compare Ordered and Plain modes in the raw setting of CatBoost described in the previous paragraph.

In Table 10, we present relative results w.r.t. Plain mode for two modifications of Ordered mode. The first one uses one random permutation σ_{boost} for Ordered mode generated independently from the permutation σ_{cat} used for ordered TS. Clearly, discrepancy between the two permutations provides target leakage, which should be avoided. However, even in this setting Ordered mode considerably outperforms Plain one by 0.5% in terms of logloss and by 0.2% in terms of zero-one loss in average. Thus, advantage of Ordered mode remains strong in the raw setting of CatBoost.

Table 10: Ordered vs Plain modes in raw setting: change of logloss / zero-one loss relative to Plain mode.

	Ordered, σ_{boost} independent of σ_{cat}	Ordered, $\sigma_{boost} = \sigma_{cat}$
Adult	-1.1% / +0.2%	-2.1% / -1.2%
Amazon	+0.9% / +0.9%	+0.8% / -2.2%
Click	0% / 0%	0.1% / 0%
Appetency	-0.2% / 0.2%	-0.5% / -0.3%
Churn	+0.2% / -0.1%	+0.3% / +0.4%
Internet	-3.5% / -3.2%	-2.8% / -3.5%
Upselling	-0.4% / +0.3%	-0.3% / -0.1%
Kick	-0.2% / -0.1%	-0.2% / -0.3%
Average	-0.5% / -0.2%	-0.6% / -0.9%

In the second modification, we set $\sigma_{boost} = \sigma_{cat}$, which remarkably improves both metrics: the relative difference with Plain becomes (in average) 0.6% for logloss and 0.9% for zero-one loss. This result empirically confirms the importance of the correspondence between permutations σ_{boost} and σ_{cat}, which was theoretically motivated in Appendix F.

Feature combinations To demonstrate the effect of feature combinations, in Figure 3 we present the relative change in logloss for different numbers c_{max} of features allowed to be combined (compared to $c_{max} = 1$, where combinations are absent). In average, changing c_{max} from 1 to 2 provides an outstanding improvement of 1.86% (reaching 11.3%), changing from 1 to 3 yields 2.04%, and further increase of c_{max} does not influences the performance significantly.

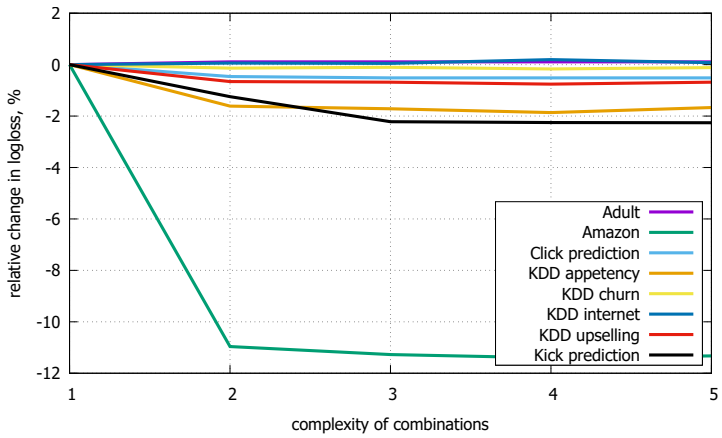

Figure 3: Relative change in logloss for a given allowed complexity compared to the absence of feature combinations.

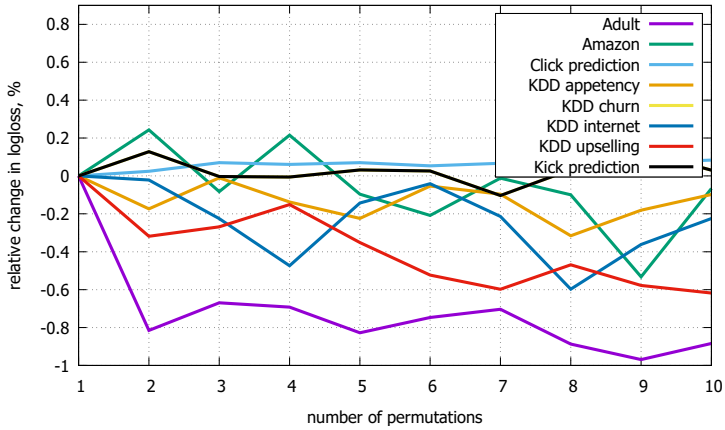

Figure 4: Relative change in logloss for a given number of permutations s compared to $s = 1$,

Number of permutations The effect of the number s of permutations on the performance of CatBoost is presented in Figure 4. In average, increasing s slightly decreases logloss, e.g., by 0.19% for $s = 3$ and by 0.38% for $s = 9$ compared to $s = 1$.