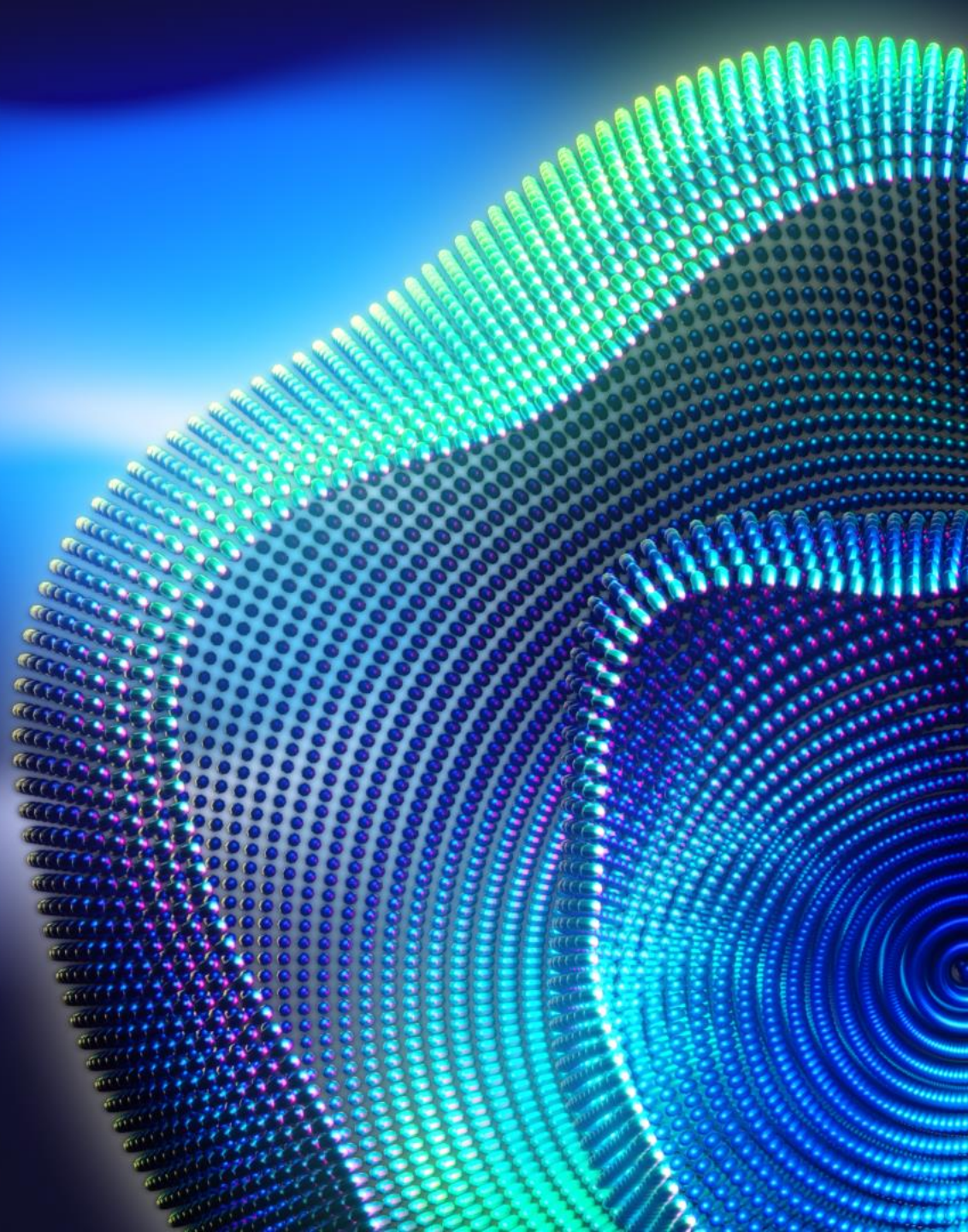
AIJ Artificial Intelligence Journey

SketchBoost

Fast Gradient Boosted Decision Tree for Multioutput Problem

Anton Vakhrushev, Leonid Iosipoi

Sber AI Lab, 2022



AIJ Artificial Intelligence Journey

Gradient Boosting for multioutput tasks

Background on GBDT

Gradient Boosted Decision Tree (GBDT) is one of the most powerful methods for solving prediction problems in both classification and regression domains.

It is a dominant tool today in applications where **tabular data** is abundant, for example, in **e-commerce**, **financial**, and **retail** industries.

It has contributed countless top solutions in **Kaggle** competitions.

Motivation

Our main focus is the scalability of GBDT to multioutput problems:

- **multiclass** classification
- **multilabel** classification
- **multioutput** regression

These problems arise in various areas such as **Finance**, **Multivariate Time Series Forecasting**, **Recommender Systems**, and others.

Motivation

There are several extremely efficient, open-source, and production-ready implementations of GBDT such as

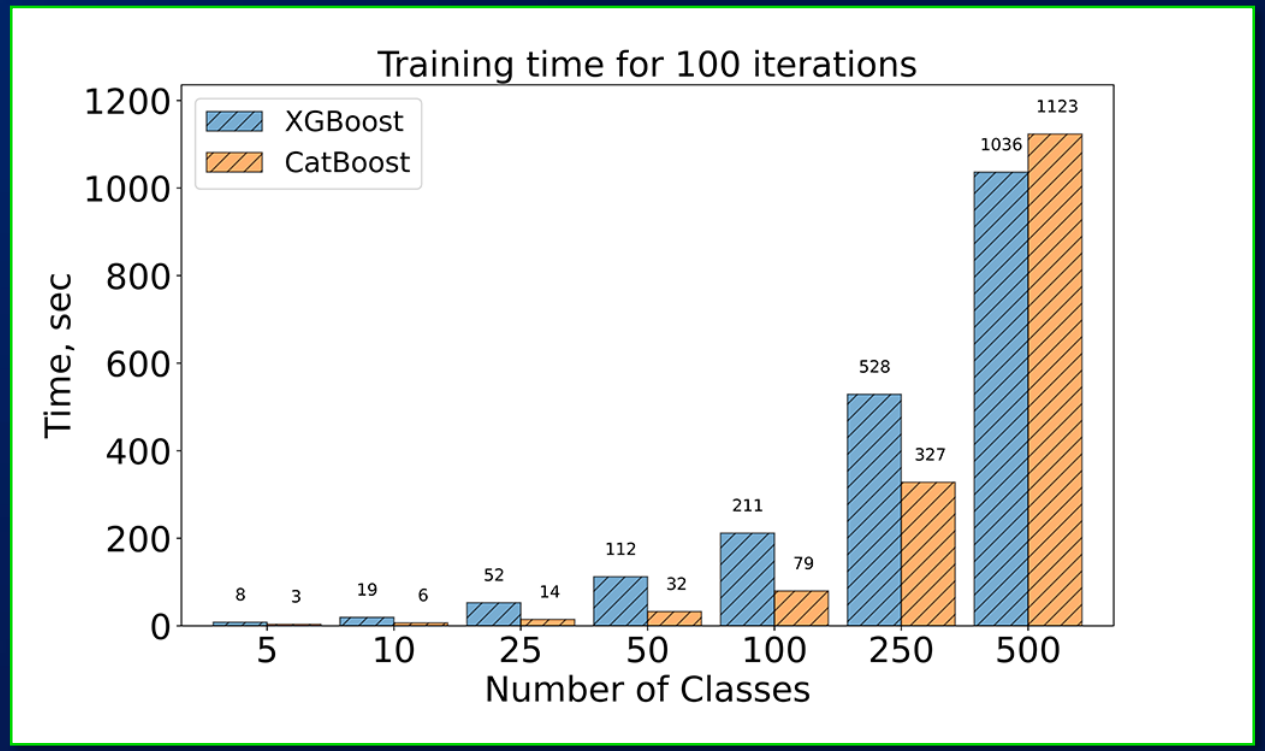
- **XGBoost**
- **LightGBM**
- **CatBoost**

And even for them, learning a GBDT model for moderately large datasets with high-dimensional output can require much time.

Motivation

XGBoost and **CatBoost** training time for 100 iterations on a synthetic dataset

(2000k instances, 100 features) for multiclass classification



Background on GBDT

- Dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^n$, where $x_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}^d$
- (x_k, y_k) are i.i.d. according to unknown $P(\cdot, \cdot)$
- $L(y, \hat{y})$ is a given loss function

The goal is to construct a model $F(x)$ to minimize the aggregation of loss L,

$$\mathcal{L}_n(F) = \sum_{k=1}^{n} L\big(y_k, F(x_k)\big).$$

Background on GBDT

Gradient Boosting is an algorithm which combines weak learners into a single strong learner in an iterative and greedy fashion.

- Choose a set of weak learners $\mathcal{F} \subset \{f : \mathbb{R}^m \to \mathbb{R}^d\}$
- At each step $t \in \mathbb{N}$
  1. Compute derivatives:

$$g_i^t = \nabla_a L(y_i, a)\big|_{a=F^{t-1}(x_i)} \quad \text{and} \quad H_i^t = \nabla_{aa}^2 L(y_i, x)\big|_{a=F^{t-1}(x_i)}.$$

  2. Find an approximate minimizer $f^t \in \mathcal{F}$ using the Newton method:

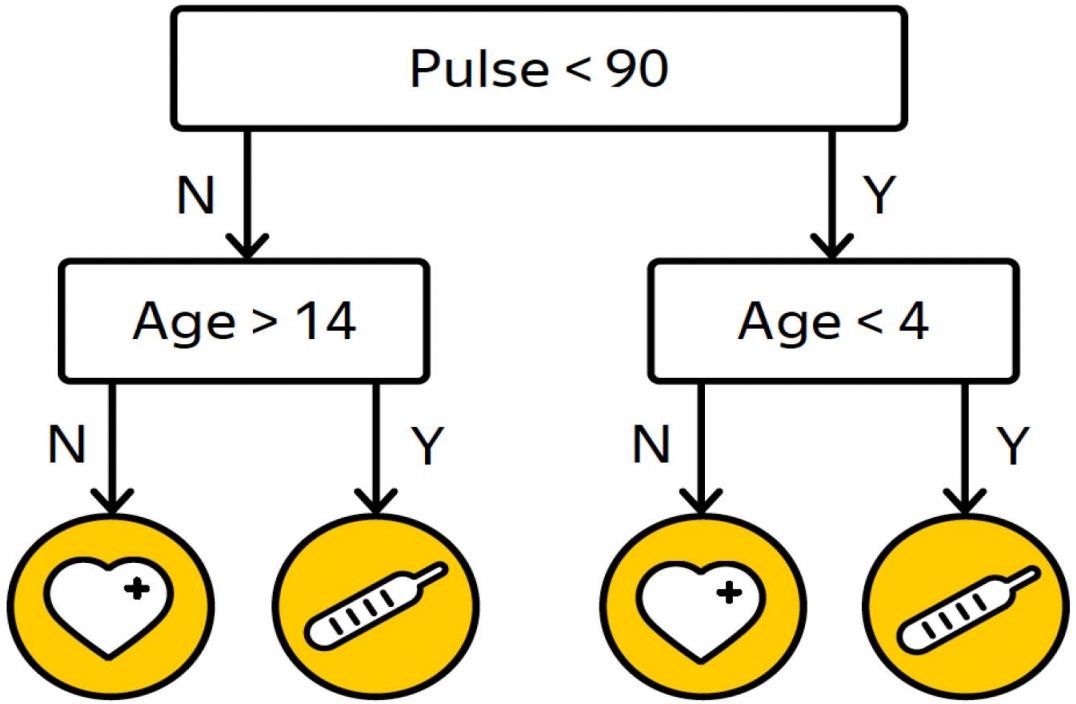
$$f^t \approx \operatorname*{argmin}_{f \in \mathcal{F}} \left\{ \sum_{i=1}^{n} \left((g_i^t)^\top f(x_i) + \frac{1}{2} \big(f(x_i)\big)^\top H_i^t f(x_i) \right) + \Omega(f) \right\},$$

     where $\Omega(f)$ is a regularization term.

  3. Update the model: $F^{t+1} = F^t + \varepsilon f^t$ ($\varepsilon > 0$ is a learning rate).

Background on GBDT

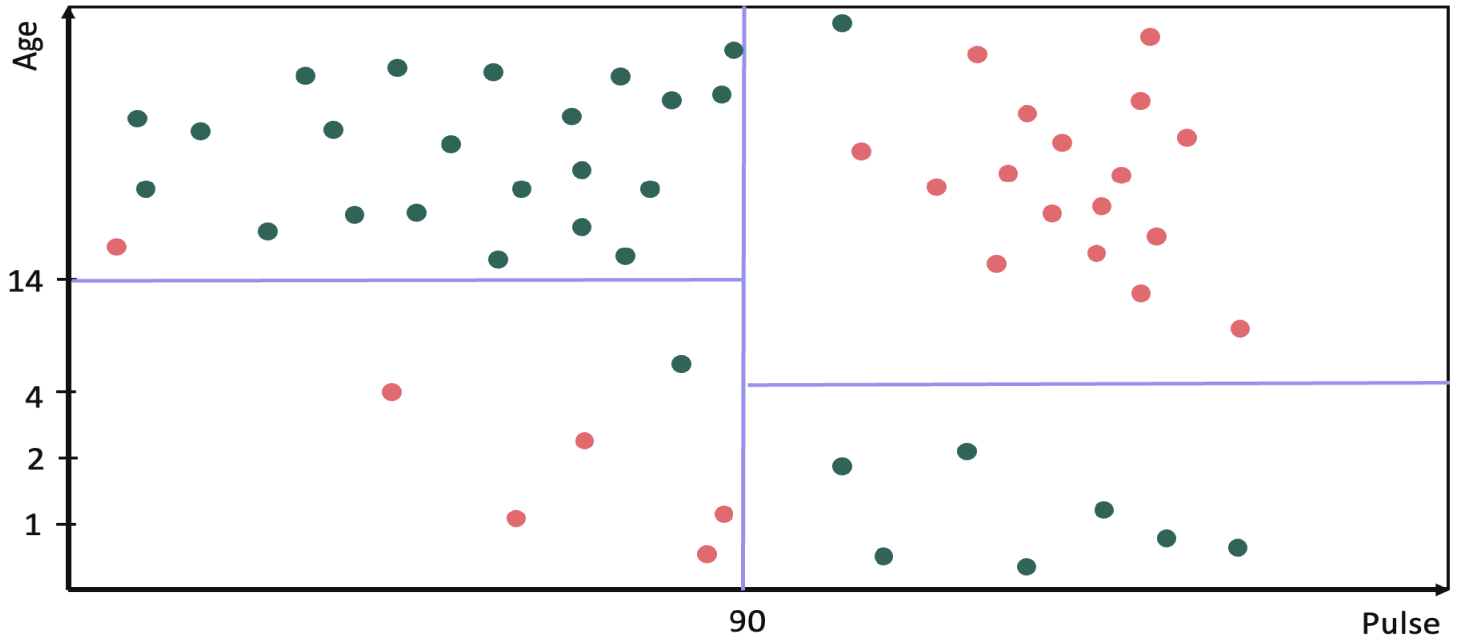
Gradient Boosted Decision Tree (GBDT) uses decision trees as weak learners.



Based on the construction mechanism, any decision tree can be expressed as

$$f(x) = \sum_{j=1}^{J} v_j \cdot [x \in R_j],$$

where J is the number of leaves, R_j is a j-th leaf, v_j is the value of j-th leaf. Here [predicate] denotes the indicator function.



Background on GBDT

The problem of learning f^t can be divided into two separate problems:

1. Finding the tree structure — division of a feature space into J leaves.
2. Finding the leaf values — finding leaf values which minimize the loss function for a tree with a given structure.

Background on GBDT

Fitting a decision tree

Since decision trees take constant values at each leaf, we can optimize the objective function from for each leaf R_j separately,

$$v_j = \underset{v \in \mathbb{R}^d}{\operatorname{argmin}} \left\{ \sum_{x_i \in R_j} \left(g_i^\top v + \frac{1}{2} v^\top H_i v \right) + \frac{\lambda}{2} \|v\|^2 \right\},$$

where we employ l_2 regularization on leaf values with $\lambda > 0$.

(we don't indicate the dependence of J, v_j, R_j, g_i, and H_i on the step t)

Background on GBDT

If the loss function L is separable w.r.t. different outputs, all Hessians $H_1, \ldots, H_n$ are diagonal. If it is not the case, it is a common practice to purposely simplify them to this extent in order to avoid matrix inversion.

For diagonal Hessians, the optimal leaf values are given by

$$v_j = -\frac{\sum_{i \in R} g_i^j}{\sum_{i \in R} h_i^j + \lambda}, \quad \text{where } g_i = \begin{pmatrix} g_i^1 \\ \vdots \\ g_i^d \end{pmatrix} \text{ and } H_i = \begin{pmatrix} h_i^1 & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & h_i^d \end{pmatrix}.$$

Background on GBDT

Substituting these leaf values back into the objective function, and omitting insignificant terms, we obtain

$$\text{Loss}(f_t) = -\frac{1}{2}\sum_{j=1}^{J} S(R_j), \quad \text{where} \quad S(R) = \sum_{j=1}^{d} \frac{\left(\sum_{x_i \in R} g_i^j\right)^2}{\sum_{x_i \in R} h_i^j + \lambda}.$$

The function $S(\cdot)$ will be referred to as the scoring function.

Background on GBDT

Finding the tree structure

- Commonly, a greedy algorithm that starts from a single leaf and iteratively adds branches to the tree is used.
- At a general step, to split one of existing leaves, we iterate through all leaves, features, and thresholds for each feature.
- The split of leaf R is based on a feature and threshold for this feature $R_{\text{left}} = \{x_i \in R \mid x_i^j \leq \text{threshold}\}$ and $R_{\text{right}} = \{x_i \in R \mid x_i^j > \text{threshold}\}$. To evaluate split candidates, we maximize the impurity score given by

$$S(R_{\text{left}}) + S(R_{\text{right}}).$$

This is equivalent to maximization of the information gain

$$\text{Gain} = -\frac{1}{2}\Big(S(R) - \big(S(R_{\text{left}}) + S(R_{\text{right}})\big)\Big).$$

AIJ Artificial Intelligence Journey

Key ideas of SketchBoost

Fast Split Scoring

Key idea: To compress the data associated with output dimensions during the split search (the most time-consuming step in GBDT) while keeping other boosting steps without change.

This is achieved by approximate computation of a scoring function used to find the best split of decision trees.

For the details please see our NeurIPS paper at
https://openreview.net/forum?id=WSxarC8t-T

Fast Split Scoring

Key idea: To exclude some of the output dimensions during the split search (the most time-consuming step) for single-tree GBDT.

Fast Split Scoring

For split search, we simplify the scoring function to

$$S_G(R) = \frac{\left\|G^\top v_R\right\|^2}{|R| + \lambda},$$

where

$$G = \begin{pmatrix} g_1^1 & \dots & g_1^d \\ \vdots & \ddots & \vdots \\ g_n^1 & \dots & g_n^d \end{pmatrix} \quad \text{and} \quad v_R = \begin{pmatrix} [x_1 \in R] \\ \vdots \\ [x_n \in R] \end{pmatrix}.$$

Fast Split Scoring

Sketching

To reduce the complexity of computing $S_G(R)$ in d, we will approximate $S_G(R)$ with $S_{G_k}(R)$ for some other matrix $G_k \in \mathbb{R}^{n \times k}$ with $k \ll d$.

We will refer to G_k as the sketch matrix.

Since there might be several good splits with almost equal impurity scores, replacing S_G with S_{G_k} might result in completely different tree structures.

Fast Split Scoring

Truncated SVD

Key Idea: To replace the gradient matrix G with its Truncated SVD version.

We start with Truncated SVD since, by the matrix approximation lemma, it provides the optimal deterministic solution to AMM.

Fast Split Scoring

Top Outputs

Key Idea: To choose k columns of G with the largest Euclidian norm.

Specifically, let us denote the columns of G by $g_1, \ldots, g_d$ and let $i_1, \ldots, i_d$ be the indexes which sort the columns of G in descending order by their norm, $\|g_{i_1}\| \geq \|g_{i_2}\| \geq \ldots \geq \|g_{i_d}\|$. We consider the following sketch

$$G = \begin{pmatrix} | & | & & | \\ g_1 & g_2 & \ldots & g_d \\ | & | & & | \end{pmatrix}, \quad G_k = \begin{pmatrix} | & | & & | \\ g_{i_1} & g_{i_2} & \ldots & g_{i_k} \\ | & | & & | \end{pmatrix}.$$

Fast Split Scoring

Random Sampling

Key Idea: To sample k columns of G with optimal (non-uniform) probabilities.

Namely, we define the non-uniform sampling probabilities by

$$p_i = \frac{\|g_i\|^2}{\sum_{j=1}^{d} \|g_j\|^2}, \qquad i = 1, \ldots, d.$$

Fast Split Scoring

Now we consider the following sketch

$$G_k = \begin{pmatrix} | & | & & | \\ \widehat{g}_1 & \widehat{g}_2 & \dots & \widehat{g}_k \\ | & | & & | \end{pmatrix},$$

where the columns $\widehat{g}_1, \dots, \widehat{g}_k$ are independent copies of the random vector $\widehat{g}$ such that

$$\widehat{g} = \frac{1}{\sqrt{kp_i}} g_i \quad \text{with probability } p_i.$$

Fast Split Scoring

Random Projections

Key Idea: To sample k random linear combinations of columns of G.

Previously, the sketch G_k was formed by sampling columns from G according to some probability distribution. This process can be viewed as multiplication of G by a matrix Π,

$$G_k = G\Pi,$$

where $\Pi \in \mathbb{R}^{d \times k}$ has independent columns, and each column is all zero except for a 1 in a random location.

Fast Split Scoring

Now we consider sampling matrices Π, every entry of which is an independently sampled random variable. Namely, we consider the sketch

$$G_k = G\Pi,$$

where $\Pi \in \mathbb{R}^{d\times k}$ is a random matrix filled with independent $\mathcal{N}(0, k^{-1})$ entries.

This approach is based on the Johnson-Lindenstrauss lemma. In fact, this lemma is true for many other distributions, but there was no significant difference between them in our numerical experiments.

AIJ Artificial Intelligence Journey

Implementation and Py-Boost library

Problems with existing GBDTs

Modern gradient boosting toolkits are very complex and are written in low-level programming languages. As a result,

- It is hard to modify or customize them to suit one's needs
- Ideas from recent papers are not easy to implement
- It is difficult to understand how boosting toolkits actually work
- For researchers, it is an involved process to test ideas and hypotheses in boosting

These are the reasons why we develop **Py-Boost** — a Python-based GBDT library available at **https://github.com/sb-ai-lab/Py-Boost**

Py-Boost key features

Fast with GPU

Despite the fact that Py-Boost is written in Python, it works only on GPU and uses Python GPU libraries such as CuPy and Numba.

Simple

Py-boost is a simplified gradient boosting library but it supports all main features and hyperparameters available in other implementations.

Easy to customize

Py-boost can be easily customized even if one is not familiar with GPU programming (just replace np with cp). What can be customized?

- Row/Col sampling strategy
- Training control
- Losses/metrics
- Multioutput handling strategy
- Anything via custom callbacks

Up-to-date

Some of recent results in tree boosting are already implemented in Py-boost (to be discussed later).

AIJ Artificial Intelligence Journey

Experimental results

Experimental design

SketchBoost algoritm is a part of **Py-Boost** library which implements the proposed sketching methods.

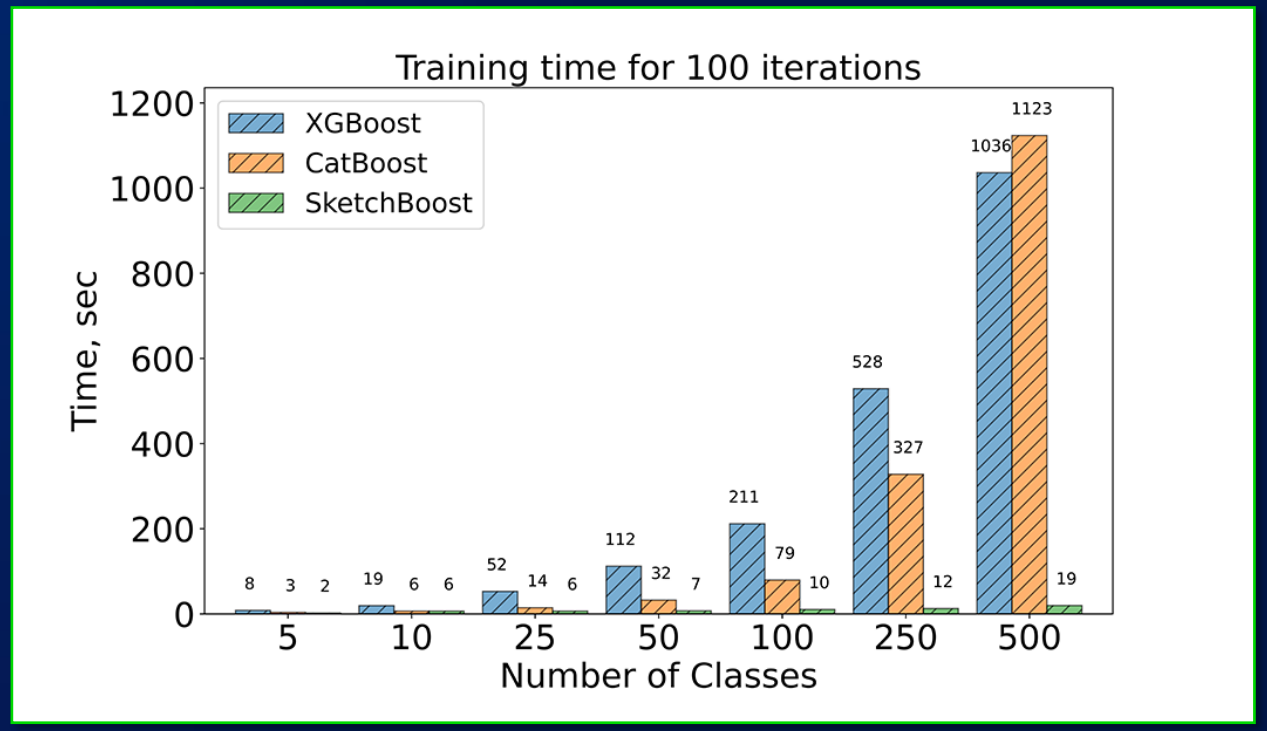
In our numerical study, we compare **SketchBoost** to

- existing state-of-art boosting toolkits (**XGBoost** and **CatBoost**)
- pure **Py-Boost** with no sketching strategies
- deep learning baseline (**TabNet**)

Numerical results

SketchBoost, **XGBoost** and **CatBoost** training time for 100 iterations on a synthetic dataset

(2000k instances, 100 features) for multiclass classification



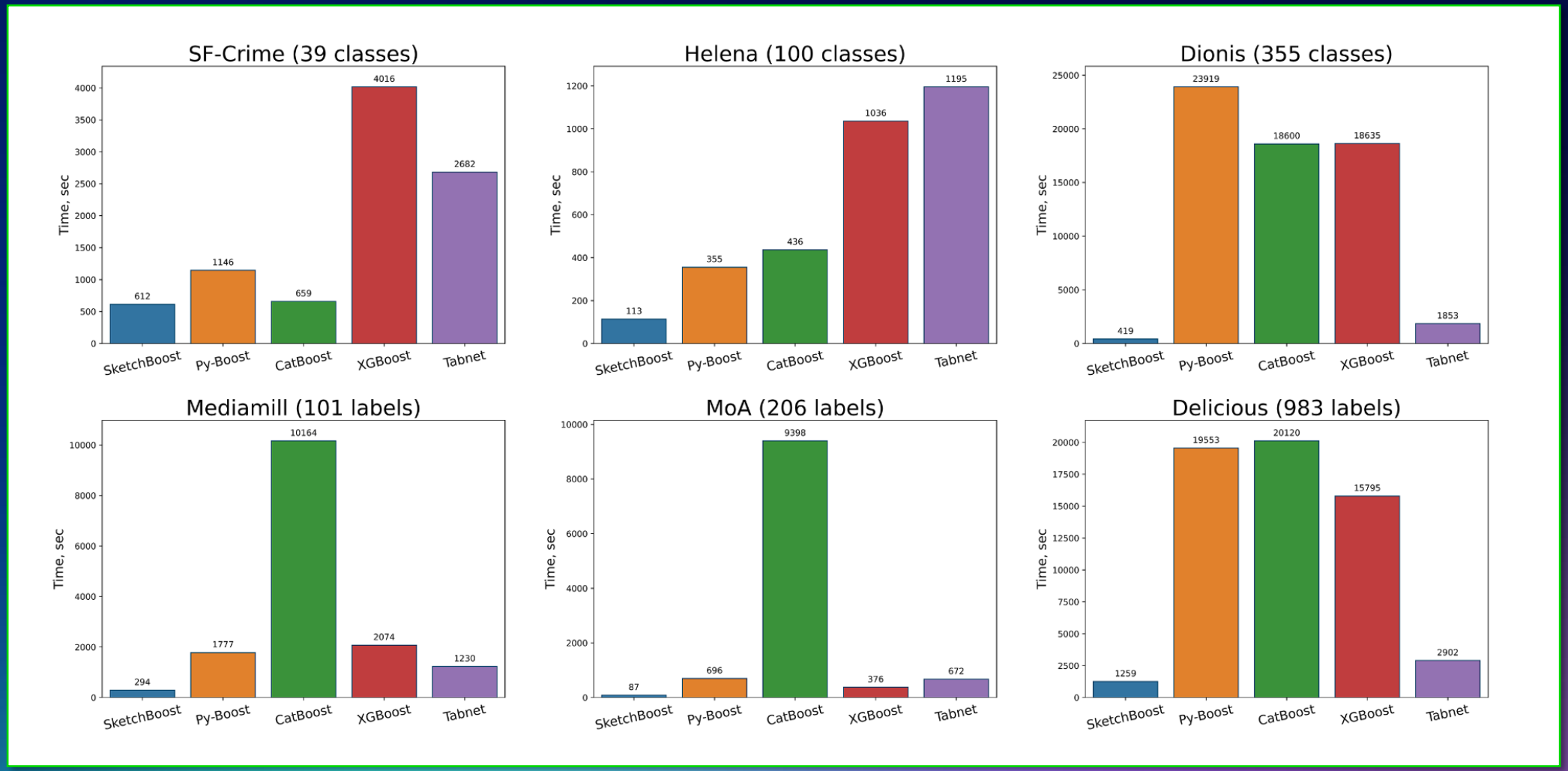
Datasets to compare

The experiments are conducted on datasets from Kaggle, OpenML, and Mulan website for multiclass and multilabel classification.

Dataset	Task	Rows	Features	Classes/Labels
SF-Crime	multiclass	878 049	10	39
Helena	multiclass	65 196	27	100
Dionis	multiclass	416 188	60	355
Mediamill	multilabel	43 910	120	101
MoA	multilabel	23 814	876	206
Delicious	multilabel	16 110	500	983

* a more extensive study is presented in the paper

Training time per fold



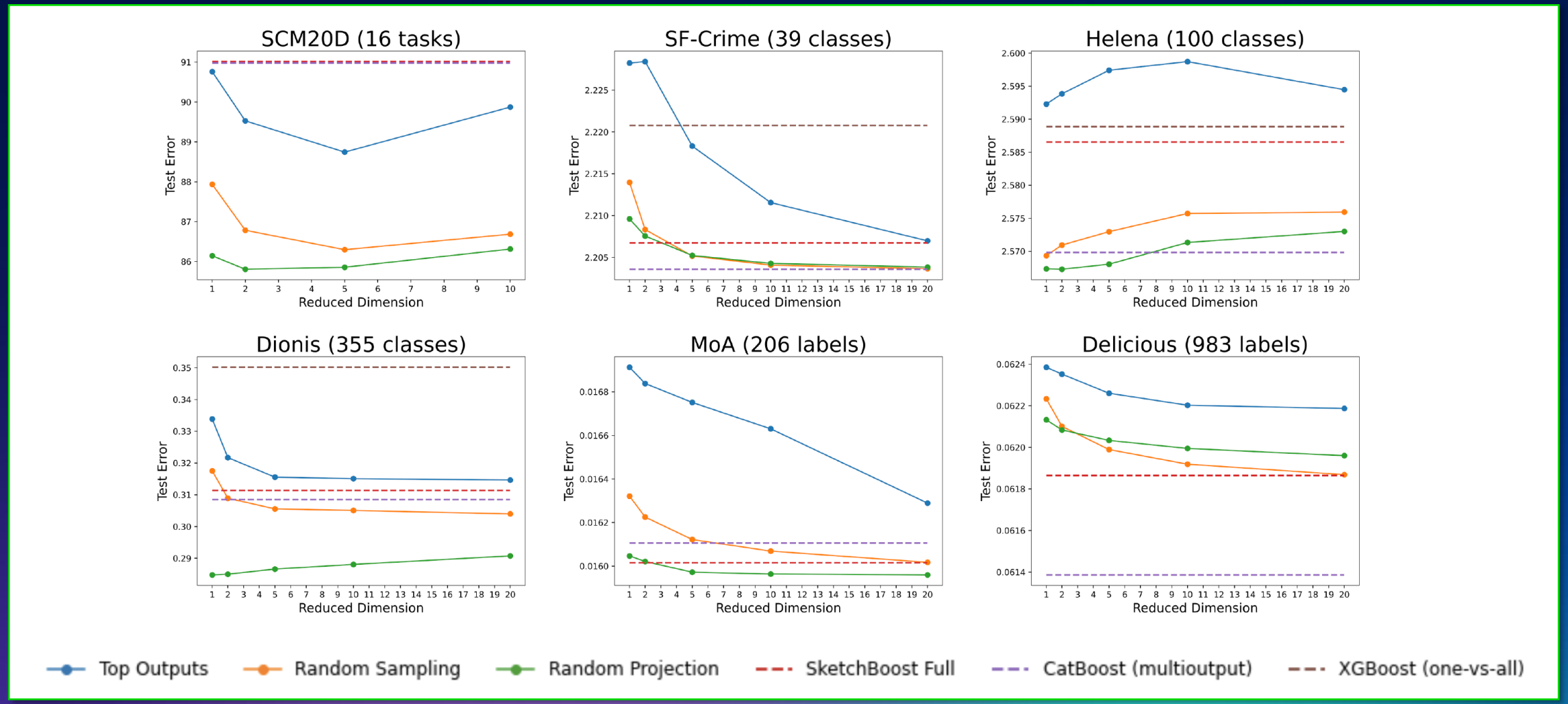
* CatBoost can be trained on GPU only for multiclass classification tasks.

Model performance

	Our Algorithms		Baselines		
Dataset	**SketchBoost**	**Py-Boost**	**CatBoost**	**XGBoost**	**TabNet**
Multiclass classification					
SF-Crime (39 classes)	2.2038	2.2067	**2.2036**	2.2208	2.4819
Helena (100 classes)	**2.5673**	2.5865	2.5698	2.5889	2.7197
Dionis (355 classes)	**0.2848**	0.3114	0.3085	0.3502	0.4753
Multilabel classification					
Mediamill (101 labels)	**0.0743**	0.0747	0.0754	0.0758	0.0859
MoA (206 labels)	**0.0160**	0.0160	0.0161	0.0166	0.0193
Delicious (983 labels)	0.0620	0.0619	**0.0614**	0.0620	0.0664

* Cross-entropy on test set

Dependence on K



Bonus: Kaggle Gold Medal with Py-Boost

Open Problem Multimodal competition on Kaggle consists of 2 multiregression tasks:

1) **200k features and 23k targets**
2) **22k features and 140 targets**

See more at competition page

https://www.kaggle.com/competitions/open-problems-multimodal/

#	△	Team	Members	Score	Entries	Last	Code
1	▲ 39	Shuji Suzuki		0.775690	49	2d	
2	▼ 1	senkin & tmp		0.774669	152	2d	
3	▲ 8	Makotu		0.773518	180	2d	
4	▲ 6	Oliver Wang		0.773145	194	2d	
5	▲ 12	Lucky Shake		0.772827	446	2d	
6	▼ 3	[Risk-ZaloPay] Aggressive		0.772712	110	2d	
7	▲ 9	chromosom		0.772635	155	1d	
8	▲ 11	[Deleted] d9a20f9e-aef5-4975-9de4-8b814b69413d		0.772355	45	6d	
9	▲ 111	vialactea		0.772142	54	1d	

Models (ensemble)

Citeseq task : 3x multilayer perceptrons, 1x Conv1D, 1x pyBoost (best single model).
Multiome task : 1x multilayer perceptron, 1x TabNet, 1x pyBoost (best single model).

pyBoost seems to be the new SOTA on multioutput tasks (at least among GBDT models). It's extremely fast to train as it uses GPU only and super easy to customize.

Py-Boost based solution was able to get 7th place utilizing the power on SketchBoost algorithm

***Solution is described at https://www.kaggle.com/competitions/open-problems-multimodal/discussion/366471**

AIJ Artificial Intelligence Journey

Conclusion

Conclusion

Our empirical study shows that SketchBoost achieve comparable and sometimes even better results than the existing state-of-the-art GBDT implementations but in remarkably less time.

These methods are implemented in SketchBoost which itself is a part of our Python-based implementation of GBDT called Py-Boost.

AIJ Artificial Intelligence Journey

Thank you for listening

Py-Boost
https://github.com/sb-ai-lab/Py-Boost

SketchBoost Paper
https://openreview.net/forum?id=WSxarC8t-T

Other projects of Sber AI Lab
https://github.com/sb-ai-lab