XGBoost vs LightGBM vs CatBoost

Knowledge sharing session

Rostislav Stoyanov

May 28, 2025

Part I

RECAP

Part I: Recap

Additive modelling

- Additive models over a base hypothesis space $\mathcal{H}$ take the form:

$$\mathcal{F} = \left\{ f(x) = \sum_{m=1}^{M} \nu_m h_m(x) \mid h_m \in \mathcal{H}, \nu_m \in \mathbb{R} \right\}.$$

  - h_m functions take values in $\mathbb{R}$ or some other vector space.
- For a dataset $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ and a loss function ℓ, try to find:

$$\arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \ell\left(y_i, f\left(x_i\right)\right).$$

- Difficult to solve in the general case (e.g. for trees), but if there is an alternative if we can find the solution for a single hypothesis fast.

Forward Stagewise Additive Modeling (FSAM)

- Use an iterative algorithm to fit the additive models:

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M :
   2.1 Compute:
$$(v_m, h_m) = \underset{\nu \in \mathbf{R}, h \in \mathcal{H}}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_{m-1}(x_i) + \underbrace{vh(x_i)}_{\text{new piece}}).$$
   2.2 Set $f_m = f_{m-1} + v_m h$.
3. Return: f_M.

- The size of the step ν is chosen by fixing a constant step or performing a line search after finding h_m at each step m.

Gradient boosting

- The difficult part is to find the step direction we want to take h_m. Instead of trying to find the optimal (ν, h) pair we use the idea of gradient descent.
- This means finding at each step m the negative gradient g_m w.r.t to the current function values f_{m-1} at the n training points. Each element $g_{im}, i = 1, \ldots, n$ is defined as:

$$g_{im} = \left[\frac{\partial L\left(y_i, f\left(x_i\right)\right)}{\partial f\left(x_i\right)} \right]_{f(x_i)=f_{m-1}(x_i)}$$

- g_m gives direction of steepest descent, but doesn't generalize and might not correspond to a base hypothesis. As a solution find the hypothesis that is closes to the negative gradient:

$$h_m = \min_{h \in \mathcal{H}} \sum_{i=1}^{n} (-g_{im} - h(x_i))^2$$

XGBoost

- XGBoost is a gradient boosting algorithm with several key modifications.
- Usage of second-order derivatives in order to make a better approximation of the step. In other words, Newton's method is used instead of pure gradient descent.
- Addition of a regularization term Ω:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w^2 \,,$$

  where T is the number of leaves, γ is penalization on the number of leaves, λ is the regularization term L2.
- Newton's method and regularization lead to the following objective function for each new tree (f_m) added at step m:

$$\mathcal{L}^{(m)} = \sum_{i=1}^{n} [g_m(x_i)\, f_m(x_i) + \frac{1}{2} h_m(x_i)\, f_m^2(x_i)] + \Omega(f_t)$$

- The objective for each tree allows for the optimal values for both the optimal values for each leaf:

$$w_j^* = -\frac{G_{jm}}{H_{jm} + \lambda}$$

and gain for splitting to be calculated based on derivatives:

$$-\frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right] - \gamma$$

  - $\sum_{i \in I_j} g_m(x_i) = G_{jm}$
  - $\sum_{i \in I_j} h_m(x_i) = H_{jm}$
  - G_L and G_R are gradients for left(and respectively right) node after splitting. Similarly, H_L and H_R are second-order statistics.

XGBoost

- XGBoost also provides weighted quantile sketch as an alternative to greedy tree building. This approach relies on building histograms and binning feature values in order to speed up computation. Instead of the counting data samples, hessian values are used for grouping the data points.
- There are also other technical improvements such as:
  - Custom loss functions.
  - Parallel processing.
  - Later GPU support and out-of-core computation have also been added.
  - Some other additions taken from algorithms we will see later in the session.

Outline

For this session, we are going to explore alternatives to XGBoost and attempt to compare them:

- LightGBM.
- CatBoost.

Part II

LightGBM

Part II: LightGBM

LightGBM

- Trees are grown in leaf-wise fashion (Fig.1a), instead of the level-wise method used in XGBoost(Fig.1b). This approach might lead to highly imbalanced trees and overfitting on small datasets, so maximum depth must be controlled.

(a) Leaf wise tree growth.

(b) Level-wise tree growth

Figure. Tree growing options.[1]

[1] Source: https://lightgbm.readthedocs.io/en/v3.3.5/Features.html

LightGBM

- Improvements for split-finding. Binning is done only once at the beginning of tree building, instead of at each level (weighted quantile sketch). Significant reduction of training time.
- Both of these improvements are currently available in XGBoost (with some differences in implementation). LightGBM, however, also provides some features can't be used in XGBoost and are the main focus of the original paper [Ke+]:
  - Gradient-based One-Side Sampling (GOSS) – used for data instance subsampling.
  - Exclusive Feature Bundling (EFB) – used for reduction of the number of features.
- Note: Improvements and/or differences for distributed learning are not explored in this presentation.

GOSS

- Sumsampling is used to prevent overfitting and to speed up the learning process for each tree.
- Variance of each component of the ensamble goes up, the pairwise correlation between trees decreases, which can lead to reduction of total variance of the model.
- Previous approaches for downsampling the training set include:
  - For AdaBoost, there are approaches that rely on the sample weights. These, however, are not applicable to GBDT as there is no sample weight in GBDT.
  - Stochastic gradient boosting (SGB), which uses random sampling (uniform) to select a subset to train each new learner. Can lead to worse performance.

GOSS

- The issue with SGB is that, while it samples uniformly, not all of the samples play the same role when calculating the gain. To have similar performance with sampled dataset as with the full one, we need to choose the most informative examples.
- Main idea in GOSS is that data samples x_i with larger gradients (in terms of absolute value $|g_i|$) are more important the ones with small gradients.
- Large gradient value indicates that the model can be improved significantly with respect to the data sample.
- Can't focus entirely on the samples with large gradient as this will shift the data distribution.

GOSS

- The GOSS subsampling algorithm works in the following way:
  1. Sort the data samples according to absolute gradients in a descending order.
  2. Select the top $a\times$ 100% instances, where a is a parameter, $0 \leq a < 1$.
  3. From the remaining $(1-a)$ percents of the data sample $b\times$ 100% instances. These instances are scaled by $\frac{1-a}{b}$. This is done in order to avoid changing the original data distribution too much.
- Some theoretical analysis is provided of the optimality of the sampling method [Ke+]. Empirical evidence also shows that GOSS is better than SGB and can achieve similar model performance with only 10-20% of the data [Ou20; Ke+].

Sampling ratio	0.1	0.15	0.2	0.25	0.3	0.35	0.4
SGB	0.5182	0.5216	0.5239	0.5249	0.5252	0.5263	0.5267
GOSS	0.5224	0.5256	0.5275	0.5284	0.5289	0.5293	0.5296

Table. Accuracy comparison on LETOR dataset for GOSS and SGB under different sampling ratios. [Ke+]

EFB

- High-dimensional data are usually very sparse. This allows for the creation an nearly lossless approach to feature reduction.
- Exclusive feature bundling (EFB) bundles mutually exclusive features (can't take non-zero value simultaneously). Such bundles are used to further reduce the time required for training.
- There are two questions that need to be answered when building bundles:
  1. Which features should we bundle together?
  2. How should bundles be constructed?

EFB

Identifying feature bundles

- It can be proven that this problem is NP-hard by reducing graph colouring to the problem of identifying feature bundles.
- A greedy algorithm for graph colouring is then used in order to produce the bundles.
- Additionally, EFB allows for features that are not 100% mutually exclusive to be added together in a bundle. This is controlled by a parameter γ, which specifies the maximal number of conflicts in each bundle.

EFB

Feature merging

- Because we are using a histogram-based algorithm, the idea is to let exclusive feature reside in different bins.
- Achieved by adding offsets to the original values of the features.
- Example from the paper [Ke+]:
  1. Feature A takes values in [0, 10), feature B takes values in [0, 20).
  2. Adding 10 to features in B makes them take values in [10, 30).
  3. Safe to merge features A and B in a bundle with range [0, 30).

Categorical feature support

- Instead of one-hot encoding, the optimal solution is to split on a categorical feature by partitioning its categories into 2 subsets. Want to group the categories that output similar leaf values.
- Splits are defined as value $\in$ categories.
- Issues is finding how to split the categories – there are $2^{(k-1)} - 1$ possible partitions for a feature with k categories.
- Turns out there is a more efficient solution that requires look at possible partitions only between points when split $(\mathcal{O}(k * log(k))$ [Fis58].
- Implemented in LightGBM and recently added as beta feature to XGBoost.

Part III

CatBoost

Part III: CatBoost

CatBoost

- Unique approach to handling categorical features.
- Instead of the classical gradient boosting algorithm, uses ordered boosting, which is permutation-driven.
- Both techniques are used to fight a prediction shift caused by target leakage.

Ordered target encoding

- A popular way of handling categorical features is to use target encoding:

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

  where:
  - i is the categorical feature.
  - x_k is the k-th example, x_i^k is the category of that example.
  - p is the average target value in the dataset and $a \geq 0$ is a parameter.
- The problem of this approach is target leakage coming from calculating $\hat{x}_k^i$ using y_k.

Ordered target encoding

- CatBoost uses an idea inspired by online learning algorithms which training examples sequentially in time.
- A artificial "time" is included by using a random permutation σ of the data.
- For each example target encoding is calculated using only previously available data.
- Using only one permutation leads to early examples in permutation having higher variance than later samples. As we will see later CatBoost uses more than one permutation in different stages of the algorithm.

Ordered target encoding

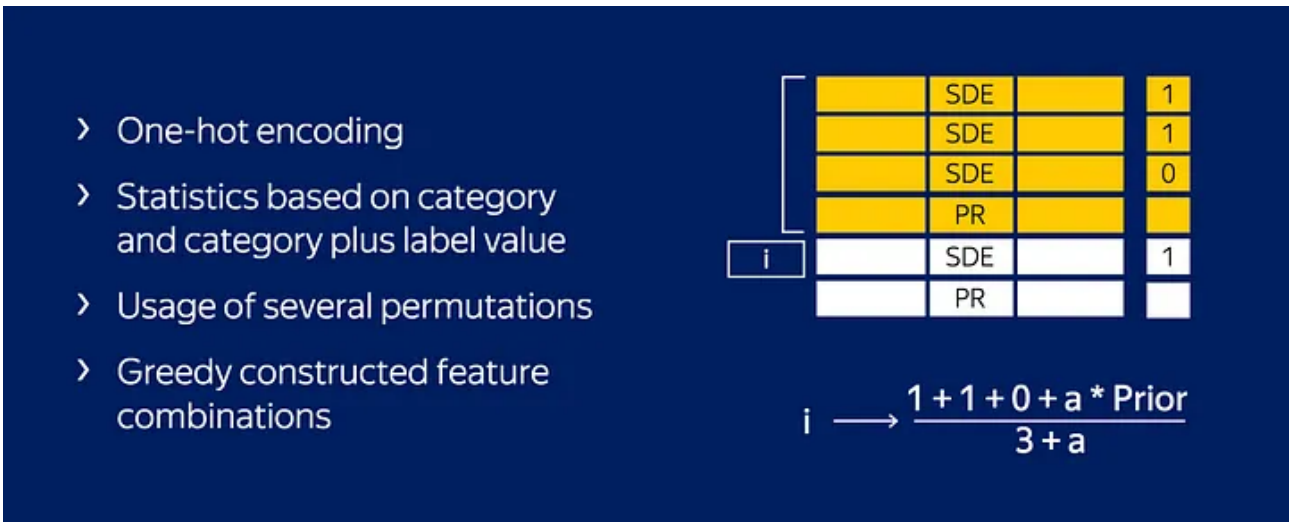

Figure. Ordered target encoding example. [2]

[2] Taken from `https://www.youtube.com/watch?v=oGRIGdsz7bM`

Prediction shift and ordered boosting

Prediction shift

- Similarly to the case of target encoding, the authors have also identified target leakage in the gradient boosting algorithm.
- The leakage comes from gradient estimation at step t being performed using the same target values on which the current model F^{t-1} is build.
- In the general the distribution $F^{t-1}(x_k)|x_k$ for a training example x_k is shifted from the distribution $F^{t-1}(x)|x$ for a test examples. This is called prediction shift.

- When calculating the gradient $g^t(x_k, y_k)$ we need to have the model F^{t-1} trained without example x_k.
- Need this for each sample, so no training data can be used?
- Instead take a permutation σ of the training data and maintain a set of n maintaining models M_i, each on trained using the first i samples in the dataset.
- The gradient for the j-th sample is calculated using model M_{j-1}.
- Not feasible in most tasks, however, modification for the tree boosting case is available.

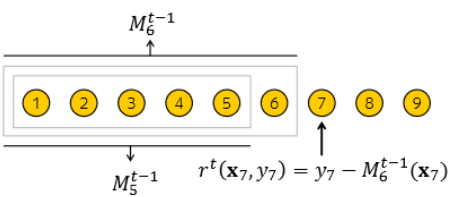

Figure. Ordered boosting principle, examples are ordered according to σ[Pro+19].

The CatBoost algorithm

- Before beginning tree construction CatBoost generates s+1 random permutations of the data, with $\sigma_1, \ldots, \sigma_s$ being used for split evaluation and σ_0 being used for leaf calculations.
- Each time a tree is build a permutation is sampled.
- The same permutation is used both for building a tree and calculating target encoding.
- During the learning process supporting models $M_{r,j}$ are maintained, where $M_{r,j}(i)$ is current prediction for the i-th example based on the first j examples in the permutation σ_r

- These models $M_{r,j}$ are used for generating the gradients at each step:

$$\text{grad}_{r,j}(i) = \left.\frac{\partial L\left(y_i, s\right)}{\partial s}\right|_{s=M_{r,j}(i)}$$

- The leaf value $\Delta(i)$ for example i is obtained individually by averaging the gradients $\text{grad}_{r,\sigma_r(i)-1}$ of the preceding examples p lying in the same leaf $leaf_r(i)$ the example i belongs to.

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 $i = 1..n$;

Figure. Leaf value calculation while building a tree [Pro+19].

- The goodness of fit against the gradients is evaluated using cosine similarity.
- After picking a tree structure, that tree structure is used to boost **all** of the models $M_{r',j}$.
- Each model $M_{r,j}$ gets different leaf values depending of the permutation:

if $Mode = Plain$ **then**
  $M_{r'}(i) \leftarrow M_{r'}(i) - \alpha\,\mathrm{avg}(grad_{r'}(p)$ for
  $p : leaf_{r'}(p) = leaf_{r'}(i))$ for $r' = 1..s, i = 1..n;$
if $Mode = Ordered$ **then**
  $M_{r',j}(i) \leftarrow M_{r',j}(i) - \alpha\,\mathrm{avg}(grad_{r',j}(p)$ for
  $p : leaf_{r'}(p) = leaf_{r'}(i), \sigma_{r'}(p) \leq j)$ for $r' = 1..s,$
  $i = 1..n, j \geq \sigma_{r'}(i) - 1;$

Figure. Boosting support models[Pro+19].

The CatBoost algorithm

- 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.

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 $i = 1..n$;

Figure. CatBoost [Pro+19].

Additional features

Feature combinations

- CatBoost also allows uses combinations of categorical features as features in attempt to catch any higher-order dependencies.
- The number of combinations grows exponentially, so can't process them all.
- Instead use a greedy approach.
- For each split of a tree, CatBoost combines all categorical features (and their combinations) already used for previous splits. Target encoding is calculated on the fly.

Additional features

Minimal variance subsampling (MVS)

- CatBoost also allows for subsampling. We are interested in the Minimal variance subsampling, which is first used in CatBoost and later implemented also in XGBoost.
- MVS is similar to GOSS, in that it provides non-uniform sampling based on the gradient values.
- Has a better theoretical background and in turn better performance.

Additional features

Minimal variance subsampling (MVS)

- Let f be a feature and v – value for that feature. The score of a split (f, v) is evaluated as:

$$S(f, v) := \sum_{l \in L} \frac{\left(\sum_{i \in l} g_i\right)^2}{\sum_{i \in l} h_i}$$

  where L is the set of obtained leaves, and leaf l consists of objects that belong to this leaf.
- Sampling from a set of size N – sequence of random variables $(\xi_1, \xi_2, \ldots, \xi_N)$, where $\xi_i \sim \text{Bernoulli}\,(p_i)$, and $\xi_i = 1$ indicates that i-th example was sampled.
- Sampling with sampling ratio s –$(\xi_1, \xi_2, \ldots, \xi_N)$, which samples $s \times 100\%$ of data on average:

$$\mathbb{E}\left(n_{\text{sampled}}\right) = \mathbb{E} \sum_{i=1}^{N} \xi_i = \sum_{i=1}^{N} p_i = N \cdot s.$$

Additional features

Minimal variance subsampling (MVS)

- To make statistics unbiased, inverse probability weighting estimation is performed by setting weights to each sample $w_i = \frac{1}{p_i}$.
- When sampling we are approximating $S(f, v)$ with:

$$\hat{S}(f, v) := \sum_{l \in L} \frac{\left(\sum_{i \in l} \frac{1}{p_i} \xi_i g_i \right)^2}{\sum_{i \in l} \frac{1}{p_i} \xi_i h_i},$$

- The aim is to minimize $\mathbb{E}\Delta^2$, where $\Delta^2 = (\hat{S}(f, v) - S(f, v))^2$
- Turns out there exists μ such that $p_i = \min\left(1, \frac{\sqrt{g_i^2 + \lambda h_i^2}}{\mu}\right)$ is a solution to the problem. μ is threshold for deterministic vs probabilistic sampling and is calculated depending on the sample rate.

Additional features

Minimal variance subsampling (MVS)

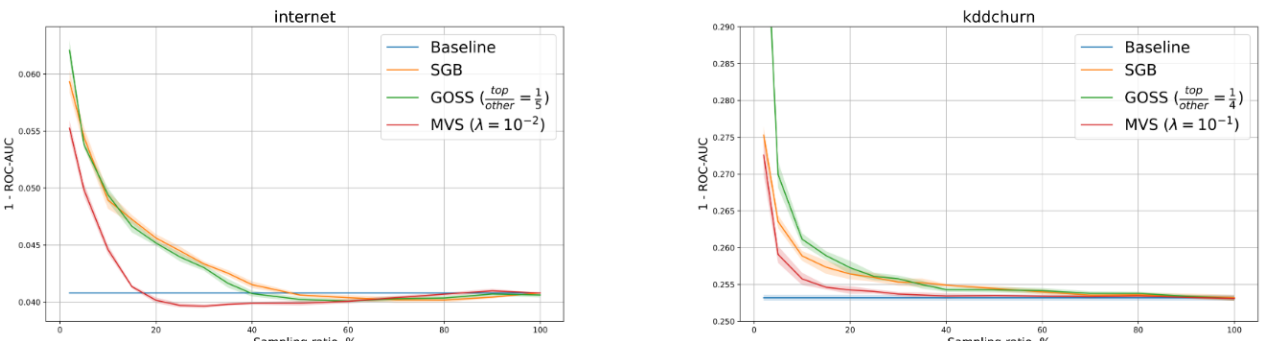

Figure. ROC-AUC error versus the fraction of sampled examples per one iteration [IG]

Useful resources

- https://towardsdatascience.com/what-makes-lightgbm-lightning-fast-a27cf0d9785e
- https://towardsdatascience.com/what-makes-lightgbm-light-8eb4dc258046
- https://www.youtube.com/watch?v=oGRIGdsz7bM
- https://www.youtube.com/watch?v=KXOTSkPL2X4
- https://lightgbm.readthedocs.io/en/v3.3.5/Features.html

References I

[Fis58] Walter D. Fisher. "On Grouping for Maximum Homogeneity". en. In: *Journal of the American Statistical Association* 53.284 (Dec. 1958), pp. 789–798. ISSN: 0162-1459, 1537-274X. DOI: 10.1080/01621459.1958.10501479. URL: http://www.tandfonline.com/doi/abs/10.1080/01621459.1958.10501479 (visited on 02/25/2023).

[IG] Bulat Ibragimov and Gleb Gusev. "Minimal Variance Sampling in Stochastic Gradient Boosting". en. In: (). URL: https://arxiv.org/abs/1910.13204.

[Ke+] Guolin Ke et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". en. In: (), p. 9. URL: https://papers.nips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html.

[Ou20] Rong Ou. *Out-of-Core GPU Gradient Boosting*. arXiv:2005.09148 [cs, stat]. May 2020. DOI: 10.48550/arXiv.2005.09148. URL: http://arxiv.org/abs/2005.09148.

[Pro+19] Liudmila Prokhorenkova et al. *CatBoost: unbiased boosting with categorical features*. arXiv:1706.09516 [cs]. Jan. 2019. DOI: 10.48550/arXiv.1706.09516. URL: http://arxiv.org/abs/1706.09516 (visited on 03/05/2023).