Lecture 11: Gradient Boosting, XGBoost, LightGBM, CatBoost & Optuna

Dr. Ratnesh Srivastava, CSIT, Guru Ghasidas Vishwavidyalaya, Bilaspur, C.G.

July 26, 2025

Abstract

These lecture notes provide a comprehensive overview of advanced gradient boosting techniques: Gradient Boosting Machine (GBM), XGBoost, LightGBM, and CatBoost. Each algorithm is explained with a focus on mathematical intuition, core concepts, worked examples, and practical considerations. The notes also cover hyperparameter tuning using Optuna, detailing its benefits and providing a practical example. This material is designed to enhance student comprehension and practical application in machine learning.

Contents

1 Gradient Boosting (GBM) - The Foundational Concept

Gradient Boosting is an ensemble machine learning technique that builds models sequentially, with each new model correcting errors made by previous ones.

1.1 Core Intuition

Imagine you're learning math:

1. Take an an initial constant guess, $F_0(x)$, for your prediction (e.g., the mean value for a regression problem).
2. Calculate the errors, also known as residuals, which are the differences between the actual target values and your current predictions (residuals = actual − predicted).
3. Learn from these mistakes by building a simple model, often a decision tree, $h_m(x)$, to predict these errors.
4. Update your overall prediction: New prediction = Old prediction + (Learning Rate × Prediction from Error Model).
5. Repeat this process until the errors are minimized or a stopping criterion is met.

1.2 Mathematical Insight

The goal of Gradient Boosting is to find a function $F(x)$ that minimizes some chosen loss function, $L(y, F(x))$, where y represents the true target value and $F(x)$ is the model's prediction. For example, if we use the Mean Squared Error (MSE) as our loss function:

$$L(y_i, F(x_i)) = \text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - F(x_i))^2$$

Here, y_i is the actual target for the i-th data point, $F(x_i)$ is the model's current prediction for the i-th data point, and n is the total number of data points.

The direction to reduce this loss most rapidly is given by the negative gradient of the loss function with respect to the prediction $F(x)$. For MSE, the negative gradient is simply the residual itself:

$$\nabla_F L = -\frac{\partial L}{\partial F} = -\frac{\partial}{\partial F}\left(\frac{1}{2}(y - F(x))^2\right) = (y - F(x))$$

In this expression, $\nabla_F L$ denotes the gradient of the loss function L with respect to the model's output F. The term $\frac{\partial L}{\partial F}$ is the partial derivative of the loss with respect to F. The final result $(y - F(x))$ is exactly the residual. Thus, for squared error, residuals are equivalent to the negative gradient. Each new tree in Gradient Boosting is trained to predict these negative gradients (or "pseudo-residuals" for other loss functions).

1.3 Solved Example: Predicting Exam Scores

Data: We have a small dataset of students with their study hours and actual exam scores.

Hours Studied	Actual Score
1	40
2	60
3	80
4	100

Step-by-Step Solution:

1. **Initial Model (Tree #0), $F_0(x)$:** We start with a very simple model that predicts the mean of the actual scores. $F_0 = \frac{40+60+80+100}{4} = 70$. The initial predictions for all students are $[70, 70, 70, 70]$.

2. **Calculate Residuals:** The residuals are the differences between the actual scores (y_i) and the current predictions ($F_0(x_i)$): Residuals $= [40-70, 60-70, 80-70, 100-70] = [-30, -10, 10, 30]$.

3. **Build Tree #1 ($h_1(x)$), to predict residuals:** We train a simple regression tree (our weak learner) to predict these residuals. Let's assume our simple tree has one split: if Hours ≤ 2.5.

   - For students with Hours ≤ 2.5 (Hours=1,2), the average residual is $(-30-10)/2 = -20$. This will be the prediction for this leaf.
   - For students with Hours > 2.5 (Hours=3,4), the average residual is $(10+30)/2 = 20$. This will be the prediction for this leaf.

4. **Update Predictions, $F_1(x)$ ($\eta = 0.1$, learning rate):** We update our overall model $F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x)$, where η (eta) is the learning rate, controlling the step size of each tree's contribution.

   - Student 1 (1 Hour): Current prediction $70 + 0.1 \times (-20) = 68$
   - Student 2 (2 Hours): Current prediction $70 + 0.1 \times (-20) = 68$
   - Student 3 (3 Hours): Current prediction $70 + 0.1 \times 20 = 72$
   - Student 4 (4 Hours): Current prediction $70 + 0.1 \times 20 = 72$

5. **New Residuals:** Based on the updated predictions, we calculate the new residuals: $[40-68, 60-68, 80-72, 100-72] = [-28, -8, 8, 28]$.

6. **Repeat** this process by building Tree #2 on these new residuals, then Tree #3, and so on, until the residuals approach zero or a predefined number of trees are built.

Key Insight: Each new tree in Gradient Boosting corrects the **errors** (specifically, the negative gradients of the loss function) of the previous ensemble, iteratively pushing the overall model's predictions closer to the true target values.

1.4 Q&A

- **Q:** Why is it important to use a small learning rate (η) in Gradient Boosting?
- **A:** A small learning rate means that each individual tree contributes only a small portion to the overall prediction. This prevents the model from "overfitting" the training data by making too aggressive corrections in any single step. It's like taking measured, small strides towards a target instead of leaping past it, ensuring a more stable and generalizable learning process. This acts as a regularization technique.

2 XGBoost - Optimized Gradient Boosting

XGBoost (eXtreme Gradient Boosting) is an optimized, distributed, and highly flexible gradient boosting library designed for speed and performance. It builds upon the foundational concepts of GBM by incorporating several key enhancements.

2.1 Core Intuition

XGBoost can be thought of as a highly engineered and feature-rich version of GBM, offering several advancements:

1. **Regularization:** It explicitly adds penalties to the objective function to discourage complex models, thus preventing overfitting. This is akin to adding "simplicity" or "parsimony" as a grading criterion for a student's project, encouraging them not to overcomplicate things.
2. **Hessian (2nd Derivative):** Unlike basic GBM which uses only first-order gradients (residuals), XGBoost utilizes both first-order (g_i) and second-order (h_i, also known as Hessian) gradients of the loss function. This provides more detailed information about the curvature of the loss function, allowing for smarter, more precise steps towards the minimum. It's like adjusting your stride length based on the steepness of the terrain, not just the general direction.
3. **Handles Missing Values:** XGBoost has an internal mechanism to intelligently learn the best way to handle missing values during training, rather than requiring explicit pre-processing or imputation.
4. **Parallel Processing:** It is highly optimized to exploit multi-core CPUs, allowing for faster tree construction by parallelizing certain operations.
5. **Improved Tree Pruning:** It uses a more sophisticated tree pruning technique (pruning based on regularization) that is more robust than typical decision tree pruning.

2.2 Mathematical Insight

XGBoost optimizes a more sophisticated objective function that includes both the traditional loss term and a regularization term. For the t-th iteration, the objective function to minimize for the new tree $f_t(x)$ is:

$$\text{Obj}^{(t)} = \sum_{i=1}^{N} L(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

Here, $L(y_i, \hat{y}_i^{(t-1)} + f_t(x_i))$ is the loss for the i-th data point, where $\hat{y}_i^{(t-1)}$ is the prediction from the ensemble of $t-1$ trees, and $f_t(x_i)$ is the prediction of the current t-th tree. $\Omega(f_t)$ is the regularization term for the t-th tree, which penalizes complexity:

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^2$$

In this regularization term:

- T: Represents the total number of leaves in the t-th tree.
- ω_j: Denotes the output weight (predicted value) for the j-th leaf.
- γ (gamma): Is a regularization parameter that penalizes the number of leaves. A higher γ encourages more aggressive pruning, leading to simpler trees.
- λ (lambda): Is an L2 regularization parameter applied to the leaf weights. It discourages large leaf output values, preventing overfitting.

To efficiently optimize this objective, XGBoost uses a second-order Taylor expansion of the loss function. This expansion involves:

- g_i: The first-order gradient of the loss function with respect to the current prediction, $\frac{\partial L(y_i, \hat{y}_i)}{\partial \hat{y}_i}$. This is analogous to the residuals in GBM.
- h_i: The second-order gradient (Hessian) of the loss function with respect to the current prediction, $\frac{\partial^2 L(y_i, \hat{y}_i)}{\partial \hat{y}_i^2}$. This provides information about the curvature of the loss function.

For a given tree structure, the optimal weight for a leaf j, ω_j^*, is calculated as:

$$\omega_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

Here, I_j represents the set of data points that fall into leaf j. This formula directly incorporates both gradients and the L2 regularization parameter λ.

2.3 Solved Example: Same Exam Scores with XGBoost

Let's continue with the exam scores data to illustrate how XGBoost's features conceptually apply. **Data:** $(1, 40), (2, 60), (3, 80), (4, 100)$ Assume we are trying to predict the residuals, as in GBM, with a specific loss function that yields g_i and h_i. **Parameters for illustration:** $\gamma = 0.25, \lambda = 1$, Learning Rate ($\eta$)=0.1.

1. **Initial Prediction:** $F_0 = 70$.
2. **Gradients (g_i):** For Mean Squared Error (MSE), g_i is the residual. So, $g_i = [-30, -10, 10, 30]$.
3. **Hessians (h_i):** For MSE, the second derivative $\frac{\partial^2}{\partial F^2} \frac{1}{2}(y - F)^2 = 1$. So, $h_i = 1$ for all data points i.
4. **Calculate Split Quality for Hours ≤ 2.5:** XGBoost searches for the best split by evaluating a "gain" formula that incorporates sums of g_i and h_i for potential left and right nodes, along with the regularization parameters.
   - For the proposed Left leaf (Hours=1,2): Sum of gradients $\sum_{i \in I_{\text{left}}} g_i = -30 + (-10) = -40$. Sum of Hessians $\sum_{i \in I_{\text{left}}} h_i = 1 + 1 = 2$.
   - For the proposed Right leaf (Hours=3,4): Sum of gradients $\sum_{i \in I_{\text{right}}} g_i = 10 + 30 = 40$. Sum of Hessians $\sum_{i \in I_{\text{right}}} h_i = 1 + 1 = 2$.
5. **Weight Calculation for Leaves:** The optimal output weight (w_j^*) for each leaf is calculated using the formula derived from the objective function.
   - $w^*_{\text{left}} = -\frac{\sum_{i \in I_{\text{left}}} g_i}{\sum_{i \in I_{\text{left}}} h_i + \lambda} = -\frac{-40}{2+1} = \frac{40}{3} \approx 13.33$
   - $w^*_{\text{right}} = -\frac{\sum_{i \in I_{\text{right}}} g_i}{\sum_{i \in I_{\text{right}}} h_i + \lambda} = -\frac{40}{2+1} = -\frac{40}{3} \approx -13.33$

   Notice how these weights are calculated directly from gradients and Hessians, and how the regularization parameter λ influences them.
6. **Update Predictions ($F_1(x)$, using $\eta = 0.1$):** The overall model is updated by adding the new tree's predictions scaled by the learning rate.
   - Student 1 (1 Hour): $70 + 0.1 \times 13.33 \approx 71.33$
   - Student 2 (2 Hours): $70 + 0.1 \times 13.33 \approx 71.33$
   - Student 3 (3 Hours): $70 + 0.1 \times (-13.33) \approx 68.67$
   - Student 4 (4 Hours): $70 + 0.1 \times (-13.33) \approx 68.67$

Key Insight: XGBoost's built-in regularization (parameters like γ and λ) helps to control the complexity of the trees and the magnitude of their predictions (leaf weights). This pulls predictions towards zero and prevents extreme corrections, significantly reducing the risk of overfitting compared to basic GBM.

2.4 Q&A

- **Q:** Why is XGBoost often faster than a vanilla GBM implementation (e.g., scikit-learn's 'GradientBoostingRegressor')?
- **A:** XGBoost is highly optimized through several mechanisms: it uses parallel processing for tree construction, implements approximate split finding algorithms for very large datasets (avoiding exhaustive search), and employs more efficient data structures. Conceptually, it's like solving a complex puzzle with a smart plan and the help of friends (parallelization and optimized algorithms) instead of trying to do it all alone.

3 LightGBM - Speed-Optimized Boosting

LightGBM (Light Gradient Boosting Machine) is developed by Microsoft and is renowned for its high speed and efficiency, especially on large datasets, while typically maintaining accuracy comparable to XGBoost.

3.1 Core Intuition

LightGBM is specifically designed for handling big data by implementing smart optimizations that reduce computation and memory usage:

1. **GOSS (Gradient-based One-Side Sampling):** This technique focuses on data instances that contribute most significantly to the error (those with large gradients). It keeps all such "hard" instances and randomly samples only a small proportion of "easy" instances (those with small gradients). This is like prioritizing difficult exam questions because they reveal more about what needs to be learned, while still briefly reviewing the easy ones.
2. **EFB (Exclusive Feature Bundling):** This method bundles mutually exclusive (or nearly exclusive) features into a single "bundle" feature. Mutually exclusive means they rarely take non-zero values simultaneously. This is like combining "zip code" and "city" into a broader "location" feature if they are represented sparsely and one implies the other.
3. **Leaf-Wise Growth (Best-first):** Unlike traditional level-wise tree growth (where all nodes at a given depth are expanded before moving to the next level, like in XGBoost by default), LightGBM grows trees by repeatedly splitting the leaf with the maximum gain. This typically leads to deep, asymmetrical trees. It's similar to a depth-first search, where the algorithm quickly explores the most promising branches first.

3.2 Mathematical Insight

GOSS Sampling: GOSS works by sorting data instances based on the absolute values of their gradients (which reflect how much they contribute to the error). Let g_i be the

gradient for instance i. It then selects a fraction a of the instances with the largest absolute gradients. From the remaining instances (those with smaller gradients), it randomly samples a fraction b. To ensure these sampled small-gradient instances don't get overlooked, their corresponding weights are multiplied by a factor of $\frac{1-a}{b}$. This selective sampling significantly reduces the number of data points used for training each tree without losing much information.

Histogram-based Algorithm: Both LightGBM and (optionally) XGBoost use histogram-based algorithms to find optimal split points, which is a key to their speed. Instead of iterating through all unique values of a continuous feature to find the best split, they discretize continuous feature values into discrete bins (histograms). This significantly reduces the computational cost. For a split point d for feature j, the gain is related to:

$$\text{Gain} \propto \left(\frac{(\sum_{i \in I_L} g_i)^2}{n_L} + \frac{(\sum_{i \in I_R} g_i)^2}{n_R} \right)$$

Where I_L and I_R are the sets of data points in the left and right child nodes after the split, and n_L, n_R are their respective counts. This represents the variance gain from the split.

3.3 Solved Example: 10,000 Students Dataset

Problem: Predict scores from study hours, sleep, and 97 other (potentially sparse) features for a large dataset of 10,000 students.

LightGBM Workflow (Conceptual):

(a) **GOSS Sampling:**

- For building a specific tree, LightGBM might identify, say, $a = 10\%$ of students with the largest absolute errors (e.g., |residual| > 20). If there are 10,000 students, this means 1,000 students.
- From the remaining 9,000 students, it might randomly sample $b = 10\%$ (900 students).
- Result: For this tree, LightGBM trains on a total of $1,000 + 900 = 1,900$ students (only 19% of the original data). This dramatically speeds up computation for each tree.

(b) **EFB Bundling:**

- LightGBM analyzes features to find mutually exclusive ones. For instance, if you have one-hot encoded features like 'Payment$_M$ethod$_C$ash', '$Payment_Method_Card$', 'Pay
- This reduces the number of histograms to build and potential split points to consider, accelerating the training process.

(c) **Leaf-Wise Growth:**

- Instead of splitting all nodes at a given level (level-wise growth, common in older GBMs and default in XGBoost), LightGBM finds the single leaf node that, if split, would result in the maximum gain (greatest error reduction).

- Example:
  - After the root, LightGBM identifies that splitting on 'Hours ¡= 2.5' would give the highest error reduction, say 80%. It performs this split.
  - Now, among the newly created leaves and any existing shallow leaves, it finds that splitting on 'Sleep ¡= 6' within the leaf generated from 'Hours ¡= 2.5' would give the next highest gain, say 15%. It then makes this split, potentially growing one branch of the tree very deep while others remain shallow.

Result (Hypothetical): This often leads to:

- Training time: LightGBM 22 sec vs. XGBoost's 61 sec (significantly faster)
- Accuracy: LightGBM 92% vs. XGBoost 91.5% (comparable or sometimes slightly better)

Key Insight: LightGBM's innovations, especially GOSS and EFB, allow it to prioritize high-impact data points and features, while leaf-wise growth finds the most effective splits faster. This results in significantly faster training times and lower memory consumption without compromising much on accuracy, making it ideal for large datasets.

3.4 Q&A

- **Q:** When should I be cautious about using LightGBM?
- **A:** While fast, LightGBM's aggressive leaf-wise growth can make it more prone to overfitting on very small datasets (e.g., less than 1,000 rows), as it might create overly specific branches. In such cases, there might not be enough data for its sampling and bundling strategies to be robust, and algorithms like XGBoost (with careful tuning) or even CatBoost (if applicable) might offer more stability.

4 CatBoost - Master of Categorical Features

CatBoost (Categorical Boosting) is a powerful gradient boosting library developed by Yandex, uniquely designed to handle categorical features directly and effectively, solving common issues like target leakage and prediction shift that arise with other methods.

4.1 Core Intuition

CatBoost differentiates itself by fundamentally changing how it handles categorical data, often eliminating the need for manual encoding:

i. **Ordered Target Encoding (Permutation-driven Training):** This is CatBoost's most significant innovation. Instead of traditional target encoding (which can leak information from the target variable), CatBoost encodes

categorical features by calculating target statistics (e.g., average score for a city) using only historical data from a random permutation of the dataset. This ensures that the model never "sees" the target value of the current data point when encoding its categorical features. This is like calculating the average score per city *before* seeing the current student's actual score, preventing data leakage.

ii. **Symmetric Trees (Oblivious Trees):** CatBoost primarily uses decision trees where all nodes at the same level of the tree have the exact same split condition. This leads to balanced, symmetrical trees.

4.2 Mathematical Insight

Ordered Encoding: For a categorical feature, CatBoost transforms its values into numerical ones using an ordered statistic. For a given categorical feature value and an instance k within a random permutation of the training data, the encoded value for that category is computed as:

$$\text{encoded}_k = \frac{\sum_{j=1}^{k-1}[x_j = x_k]y_j + a \cdot p}{\sum_{j=1}^{k-1}[x_j = x_k] + a}$$

Where:

- $[x_j = x_k]$: Is an indicator function which is 1 if instance j and instance k share the same categorical value for the feature being encoded, and 0 otherwise. This sum counts how many times the category has appeared before instance k.
- y_j: Represents the target value of instance j.
- a: Is a smoothing parameter (sometimes called 'prior$_c ount$').$It's a small positive value that helps to$
$Is the global average of the target variable across the entire dataset. This 'prior' is used to give a reason$
This dynamic encoding, performed on-the-fly for each permutation, ensures that the model does not learn from its own target values, effectively preventing target leakage and "prediction shift" (a discrepancy between training and testing data distribution related to categorical feature statistics).

Symmetric Trees: In symmetric trees (also known as oblivious trees), a decision split (e.g., "feature A > value X") is applied to all nodes at a particular depth. This strict, level-wise identical split structure leads to a balanced tree, where all paths from the root to any leaf have the same length and use the same sequence of features for splits. This structure acts as a form of regularization and also makes prediction very efficient due to simplified evaluation paths.

4.3 Solved Example: Student Scores with Categorical Data

Consider an example with student scores, now including a categorical feature: 'City'. **Data:**

-

City	Hours	Score
Tokyo	1	40
Paris	2	60
Tokyo	3	80
New York	4	?

Let's assume a random permutation of the training data could be (Tokyo, Paris, Tokyo). And 'New York' is a test data point. Let smoothing parameter $a = 1$, and the global average score $p = 60$.

Ordered Encoding Process (Conceptual, based on one permutation):

i. **For the 1st 'Tokyo' (from permutation):** Since this is the first instance of 'Tokyo' in this specific permutation, there are no prior observations. The encoded value uses the global mean and smoothing: $\text{encoded}_{\text{Tokyo (1st)}} = \frac{0+1\cdot 60}{0+1} = 60$.

ii. **For 'Paris' (from permutation):** Similarly, no prior 'Paris' observations. $\text{encoded}_{\text{Paris}} = \frac{0+1\cdot 60}{0+1} = 60$.

iii. **For the 2nd 'Tokyo' (from permutation):** There is one prior 'Tokyo' observation in this permutation (the 1st one, with score 40). $\text{encoded}_{\text{Tokyo (2nd)}} = \frac{40+1\cdot 60}{1+1} = \frac{100}{2} = 50$.

This process is repeated for every permutation and for every categorical feature.

For 'New York' (test data): When encountering a category in the test set (like 'New York') that was not present in the training set or requires encoding, CatBoost handles it using pre-learned statistics from the training phase (e.g., replacing it with the global average, or a value based on frequency of occurrence).

Tree Structure (Symmetric):

```
       [Root] (e.g., split on Encoded_City > Threshold_1)
      /      \
   [Node A] [Node B]
   (e.g., split on Hours <= Threshold_2) (e.g., split on Hours <= Threshold_2)
   / \      / \
 ... ...  ... ...
```

In a symmetric tree, if "Hours $\leq$ Threshold$_2$" is chosen as a split condition at Level 2, it applies identically to *both* branches (Node A and Node B), forcing a balanced structure.

Key Insight: CatBoost's ordered encoding fundamentally prevents target leakage by ensuring that the numerical transformation for a categorical feature for any given data point is based only on *prior* observations within a shuffled dataset. This approach accurately mimics the real-world scenario of predicting on unseen data, leading to more robust and reliable models, especially when dealing with high-cardinality categorical features.

4.4 Q&A

- **Q:** Why are "symmetric trees" beneficial for CatBoost?
- **A:** Symmetric trees (also called oblivious trees) mean that all nodes at the same level of the tree use the identical split condition. This constrained structure offers several benefits:
  i. **Faster Prediction:** The identical split conditions across a level simplify the prediction path, making inference significantly faster, especially on hardware like GPUs.
  ii. **Reduced Overfitting:** The enforced symmetry acts as a strong regularization mechanism. It prevents the tree from making highly specific, complex splits on local data patterns that might not generalize well.
  iii. **Improved Quality:** Despite the constraint, this structure can sometimes lead to better generalization and accuracy by focusing on more robust, global splits.

5 Hyperparameter Tuning with Optuna

After understanding these powerful boosting algorithms, the next crucial step is to effectively tune their hyperparameters. This is where **Optuna** comes in.

5.1 Core Intuition

Hyperparameters are settings that are configured *before* the training process begins (e.g., learning rate, number of trees, maximum tree depth). Their correct selection is crucial for a model's performance. Optuna is an open-source hyperparameter optimization framework that automates this often tedious and complex trial-and-error process:

i. **Defines Search Space:** You specify the ranges or discrete choices for each hyperparameter you want to tune (e.g., 'learning$_r ate$'$could be between 0.01 and 0.3, or$'$max_d ept$
ii. **Prunes Unpromising Trials:** A critical feature is its ability to automatically stop trials that are not performing well early in their training process, based on a predefined metric. This saves significant computational time and resources.
iii. **Intelligent Sampling (Bayesian Optimization):** Optuna doesn't just randomly or exhaustively search. It uses intelligent algorithms, such as TPE (Tree-structured Parzen Estimator) as its default, to learn from the results of past trials. This knowledge guides the sampling process, suggesting more promising hyperparameter combinations for future trials, making the search much more efficient than blind grid or random search.

5.2 Mathematical Insight

Optuna's default sampler, TPE (Tree-structured Parzen Estimator), is a Bayesian optimization algorithm. It works by modeling the probability distributions of

hyperparameters given good objective values ($P(x|y < y^*)$) and hyperparameters given bad objective values ($P(x|y \geq y^*)$), where y^* is a threshold objective value (e.g., the current best observed accuracy).

TPE attempts to find hyperparameters x that maximize the Expected Improvement (EI):

$$\text{EI}(x) = E[\max(0, f(x^*) - f(x))]$$

Here:

- $f(x)$: Represents the objective function (e.g., validation error or accuracy) for a candidate set of hyperparameters x.
- $f(x^*)$: Denotes the current best observed objective value found so far during the optimization process.
- $E[\cdot]$: Represents the expected value.

Essentially, EI quantifies how much improvement we can expect by trying a particular set of hyperparameters x. TPE proposes x values that are likely to yield a significant improvement over the current best, thus focusing the search on more promising regions of the hyperparameter space.

5.3 Solved Example: Tuning CatBoost for Titanic Dataset

Goal: Maximize survival prediction accuracy for the Titanic dataset using CatBoost, tuned by Optuna.

```
import optuna
from catboost import CatBoostClassifier
from sklearn.model_selection import cross_val_score, train_test_split
import pandas as pd
import numpy as np

# Load and preprocess data (assuming titanic.csv is available)
# Dummy data for demonstration if titanic.csv is not present
try:
    data = pd.read_csv('titanic.csv')
    # Basic preprocessing as per your notes
    data['Age'].fillna(data['Age'].median(), inplace=True)
    # Handle Embarked missing values (fill with mode)
    data['Embarked'].fillna(data['Embarked'].mode()[0], inplace=True)
    X = data[['Age', 'Sex', 'Pclass', 'Fare', 'Embarked']]
    y = data['Survived']
    # Define categorical features for CatBoost
    cat_features_indices = ['Sex', 'Embarked', 'Pclass']
except FileNotFoundError:
    print("titanic.csv not found. Using synthetic data for demonstration.")
    from sklearn.datasets import make_classification
```

```
    X, y = make_classification(n_samples=1000, n_features=10, n_informative=5,
    X = pd.DataFrame(X, columns=[f'feature_{i}' for i in range(10)])
    X['Sex'] = np.random.choice(['male', 'female'], size=1000)
    X['Embarked'] = np.random.choice(['S', 'C', 'Q'], size=1000)
    X['Pclass'] = np.random.randint(1, 4, size=1000)
    cat_features_indices = ['Sex', 'Embarked', 'Pclass']

# Split data (using X, y from either real or synthetic data)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_s

# Define the objective function for Optuna
def objective(trial):
    # Hyperparameters suggested by Optuna for each trial
    params = {
        'depth': trial.suggest_int('depth', 3, 8), % Integer parameter for tree
        'learning_rate': trial.suggest_float('lr', 0.01, 0.3, log=True), % Floa
        'l2_leaf_reg': trial.suggest_float('l2', 1, 10), % Float parameter for
        'iterations': trial.suggest_int('iterations', 50, 500), # Integer param
        'random_seed': 42,
        'verbose': 0, # Suppress verbose output during trials
        'od_type': 'Iter', # Enable overfit detector
        'od_wait': 50 # Number of iterations to wait before stopping if no impr
    }

    # Create CatBoost Classifier model with the suggested hyperparameters
    model = CatBoostClassifier(
        **params,
        cat_features=cat_features_indices
    )

    # Evaluate the model using cross-validation on the training data
    # cross_val_score returns an array of scores from each fold. We take the me
    try:
        score = cross_val_score(model, X_train, y_train, cv=3, scoring='accurac
    except Exception as e:
        # Handle cases where some parameter combinations might lead to errors
        print(f"Trial failed with params {params}: {e}")
        # Return negative infinity if maximizing or positive infinity if minimi
        # to signal a failed trial
        return -float('inf')

    # Optuna aims to maximize this returned score
    return score

# Create a study object to manage the optimization process
```

```
# direction='maximize' means Optuna will try to find parameters that yield the
# pruner=optuna.pruners.MedianPruner(...) is used to stop unpromising trials ea
# n_startup_trials: Number of trials before pruning is activated.
# n_warmup_steps: Number of steps (e.g., epochs or boosting rounds) a trial mus
#                 before it can be considered for pruning.
study = optuna.create_study(direction='maximize',
                            pruner=optuna.pruners.MedianPruner(n_startup_trials=

# Run the optimization
print("Starting Optuna optimization...")
study.optimize(objective, n_trials=30, timeout=300) # Run for 30 trials or a ma

# Print the results of the optimization
print(f"\nBest accuracy: {study.best_value:.4f}")
print(f"Best hyperparameters found: {study.best_params}")

# Example of how Optuna might visualize trials (conceptual output)
# Actual visualization uses plot_optimization_history, plot_parallel_coordinate
```

Conceptual Output:

```
Starting Optuna optimization...
... (Optuna trial logs) ...
Best accuracy: 0.8432
Best hyperparameters found: {'depth': 6, 'lr': 0.127, 'l2': 4.82, 'iterations':
```

Trial Visualization (Conceptual):

- Trial 10: accuracy=0.801 (This trial might be **pruned early** by Optuna because its intermediate performance was significantly worse than the median performance of other trials at the same stage, saving computation.)
- Trial 15: accuracy=0.829
- Trial 22: accuracy=0.843 $\rightarrow$ **BEST** (This trial found the best accuracy among all trials).

Key Insight: Optuna intelligently discards unpromising trials after a few folds or iterations (through "pruning"), saving significant training time (often 50% or more) compared to traditional grid or random search. At the same time, its intelligent sampling (like TPE) helps it converge to better hyperparameters more efficiently.

6 Summary Cheat Sheet

Algorithm	Best For	Key Feature	
Gradient Boosting (GBM)	Small datasets	Sequential error correction	S
XGBoost	General-purpose	Regularization + Hessian	H
LightGBM	Big data (>10k rows)	GOSS + Leaf-wise growth	
CatBoost	Categorical data	Ordered encoding + Symmetric trees	

7 Student Exercise

Dataset: Titanic survival prediction (commonly available 'titanic.csv') **Features:** 'Age', 'Sex', 'Pclass', 'Fare', 'Embarked', etc. **Target:** 'Survived'

Tasks:

i. Implement CatBoost with default parameters (as a baseline model).

ii. Tune CatBoost hyperparameters using Optuna (focus on 'depth', 'learning$_rate$', '$l2_leaf_reg$', '$itera$
$hotencoding for LGBM/XGBoost, as they don't natively handle categorical features like CatBoos$

Solution Framework:

iii.
```
# Load data
import pandas as pd
from sklearn.model_selection import train_test_split
from catboost import CatBoostClassifier
import lightgbm as lgb
import xgboost as xgb
import optuna # Import optuna for Task 2

# Ensure titanic.csv is in your working directory or provide the full path
try:
    data = pd.read_csv('titanic.csv')
except FileNotFoundError:
    print("Error: 'titanic.csv' not found. Please ensure it's in the same direc
    exit()

# Select features and target
X = data[['Age', 'Sex', 'Pclass', 'Fare', 'Embarked']]
y = data['Survived']

# Preprocess for all models (common steps)
# Fill missing 'Age' values with the median
X['Age'].fillna(X['Age'].median(), inplace=True)
# Fill missing 'Embarked' values with the mode (most frequent value)
X['Embarked'].fillna(X['Embarked'].mode()[0], inplace=True)
# For 'Fare', if there are missing values (unlikely in Titanic), fill with medi
```

```
X['Fare'].fillna(X['Fare'].median(), inplace=True)


# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_s

# --- Task 1: CatBoost baseline ---
# Define categorical features for CatBoost's native handling
cat_features_for_cb = ['Sex','Embarked','Pclass']

print("\n--- Task 1: CatBoost Baseline ---")
cb_baseline = CatBoostClassifier(cat_features=cat_features_for_cb, verbose=0, r
cb_baseline.fit(X_train, y_train)
print(f"CatBoost baseline accuracy: {cb_baseline.score(X_val, y_val):.4f}")

# --- Task 2: Optuna tuning for CatBoost ---
print("\n--- Task 2: Optuna Tuning for CatBoost ---")

# Re-define the objective function here for clarity if you run this as a single
def objective_catboost_tuning(trial):
    params = {
        'depth': trial.suggest_int('depth', 3, 8),
        'learning_rate': trial.suggest_float('lr', 0.01, 0.3, log=True),
        'l2_leaf_reg': trial.suggest_float('l2', 1, 10),
        'iterations': trial.suggest_int('iterations', 50, 500),
        'random_seed': 42,
        'verbose': 0,
        'od_type': 'Iter',
        'od_wait': 50
    }

    model = CatBoostClassifier(
        **params,
        cat_features=cat_features_for_cb
    )

    try:
        score = cross_val_score(model, X_train, y_train, cv=3, scoring='accurac
    except Exception as e:
        print(f"Trial failed with params {params}: {e}")
        return -float('inf')

    return score

study = optuna.create_study(direction='maximize',
```

```
                                pruner=optuna.pruners.MedianPruner(n_startup_trials=
study.optimize(objective_catboost_tuning, n_trials=30, timeout=300)

print(f"Best tuned CatBoost accuracy: {study.best_value:.4f}")
print(f"Best tuned CatBoost params: {study.best_params}")


# --- Task 3: Compare with LightGBM and XGBoost ---
print("\n--- Task 3: Comparison with LightGBM and XGBoost ---")

# Preprocess for LGBM/XGBoost (one-hot encode categorical features)
# Create a copy to avoid modifying original X
X_encoded = X.copy()
# One-hot encode 'Sex', 'Embarked', 'Pclass' columns
X_encoded = pd.get_dummies(X_encoded, columns=['Sex', 'Embarked', 'Pclass'], dr

# Split the encoded data
X_train_enc, X_val_enc, y_train_enc, y_val_enc = train_test_split(X_encoded, y,

# Train LightGBM
lgb_model = lgb.LGBMClassifier(random_state=42)
lgb_model.fit(X_train_enc, y_train_enc)
print(f"LightGBM accuracy: {lgb_model.score(X_val_enc, y_val_enc):.4f}")

# Train XGBoost
# For XGBoost, set use_label_encoder=False to suppress a deprecation warning
xgb_model = xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss', r
xgb_model.fit(X_train_enc, y_train_enc)
print(f"XGBoost accuracy: {xgb_model.score(X_val_enc, y_val_enc):.4f}")
```

Discussion Question: Why does CatBoost often outperform other boosting algorithms on categorical data without manual encoding? **Answer:** Its ordered encoding specifically prevents target leakage, which is a common pitfall when converting categorical features to numerical values using target-based statistics. By performing this encoding based only on preceding data in a permutation, it mimics true unseen data conditions. Additionally, its use of symmetric trees optimizes for faster GPU usage and acts as a form of regularization, leading to robust performance on datasets rich in categorical features.