XGBoost

1 What is XGBoost?

XGBoost (Extreme Gradient Boosting) is an advanced implementation of the gradient boosting algorithm. It is designed to be highly efficient, scalable, and accurate. It builds an ensemble of decision trees sequentially, where each new tree tries to correct the errors made by the previous ones using gradient descent.
It supports:

- Regression
- Classification
- Ranking
- Custom objective functions

1.1 Scenario: Custom Objective Function in XGBoost

Let's say we're doing regression, but you want to use Mean Absolute Error (MAE) instead of the default Mean Squared Error (MSE). XGBoost does not support MAE directly because it's not differentiable everywhere — but you can still plug it in as a custom objective using smoothed approximation.

```
import xgboost as xgb
import numpy as np
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error

# 1. Create synthetic regression data
X, y = make_regression(n_samples=1000, n_features=10, noise=10)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# 2. DMatrix
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# 3. Define custom MAE (smoothed with epsilon)
def custom_mae_objective(preds, dtrain):
    labels = dtrain.get_label()
    grad = np.sign(preds - labels)  # first derivative
    hess = np.ones_like(preds)      # second derivative (approx.)
```

```
    return grad, hess

# 4. Optional: custom eval metric
def mae_eval_metric(preds, dtrain):
    labels = dtrain.get_label()
    return 'mae', mean_absolute_error(labels, preds)

# 5. Train with custom objective
params = {
    'max_depth': 3,
    'eta': 0.1,
    'silent': 1,
}
bst = xgb.train(
    params,
    dtrain,
    num_boost_round=100,
    obj=custom_mae_objective,
    feval=mae_eval_metric,
    evals=[(dtest, 'test')],
    early_stopping_rounds=10
)

# 6. Predict
preds = bst.predict(dtest)
print("Final MAE:", mean_absolute_error(y_test, preds))
```

Notes:

- `obj=custom_mae_objective`: tells XGBoost to use our custom gradient & hessian.
- `feval=mae_eval_metric`: custom evaluation printed during training.
- The MAE gradient is sign(pred - label) and hessian is approximated as constant for simplicity.

2 How does XGBoost differ from traditional gradient boosting?

XGBoost improves traditional gradient boosting in several key ways:

Feature	**Traditional GBM**	**XGBoost**
Regularization	No or minimal	L1 `alpha` and L2 `lambda`
Parallelization	No	Yes (during tree construction)
Tree Pruning	Pre-pruning	Post-pruning (using `gamma`)
Handling Missing Values	Manual or preprocessed	Built-in automatic handling
Efficiency	Slower	Fast (due to optimized C++ backend)
Second-order optimization	Optional	Always uses gradient + hessian (2nd order)

In short, XGBoost is faster, more regularized, and more scalable than traditional GBM.

3 What kind of problems can XGBoost solve?

XGBoost can handle a wide variety of machine learning tasks:

- Regression (e.g., predicting house prices)
- Binary classification (e.g., spam detection)
- Multiclass classification (e.g., digit recognition)
- Ranking (e.g., search engine ranking)
- Time series prediction (with feature engineering)
- Anomaly detection

It also supports custom loss functions, making it highly flexible.

4 What are the main advantages of using XGBoost?

Here are the key advantages:

- Regularization (L1 & L2) $\rightarrow$ prevents overfitting
- High performance $\rightarrow$ fast training with parallel processing
- Automatic handling of missing values
- Built-in cross-validation and early stopping
- Handles large datasets and sparse data efficiently
- Feature importance extraction for interpretability
- Custom objective functions support

5 What are the key parameters in XGBoost?

Here are some core hyperparameters grouped by function:

5.1 Tree Structure:

- `max_depth`: Maximum depth of a tree
- `min_child_weight`: Minimum sum of instance weight in a child
- `gamma`: Minimum loss reduction to make a split
- `subsample`: Fraction of training data for each tree
- `colsample_bytree`: Fraction of features used per tree

5.2 Learning:

- `eta` (or `learning_rate`): Step size shrinkage
- `n_estimators`: Number of trees (boosting rounds)
- `objective`: Loss function (e.g., `reg:squarederror`, `binary:logistic`)

5.3 Regularization:

- `lambda`: L2 regularization term
- `alpha`: L1 regularization term

5.4 Others:

- `booster`: Booster type (gbtree, gblinear, dart)
- `scale_pos_weight`: Balances classes in imbalanced datasets
- `early_stopping_rounds`: For automatic stopping on validation performance

6 Loss Function in XGBoost

XGBoost supports various loss functions depending on the task:

- **Regression:** Mean Squared Error (MSE), Mean Absolute Error (MAE), Huber, etc.
- **Classification:** Logistic loss (binary), Softmax loss (multiclass).

However, the key idea in XGBoost is that it uses a second-order Taylor approximation of the loss function to optimize it efficiently. That is:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t)$$

Where:

- g_i: first derivative (gradient) of the loss w.r.t. prediction
- h_i: second derivative (Hessian)
- $f_t(x_i)$: prediction from the current tree
- $\Omega(f_t)$: egularization term on the tree (controls complexity)

This second-order info allows faster convergence and more robust learning.

7 What is the objective function in XGBoost, and how is it optimized?

The objective function combines:

- **Training loss** $l(y_i, \hat{y}_i^{(t)})$: measures how well the model fits the data.
- **Regularization** $\Omega(f_k)$: penalizes complexity to avoid overfitting.

$$\text{Obj} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^{t} \Omega(f_k)$$

Regularization term:

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

Where:

- T: number of leaves
- w_j: score on leaf j
- γ, λ: regularization parameters

Optimization:

- Uses a greedy algorithm to build trees one by one.
- For each split, it calculates the gain using gradients and Hessians.
- The split with highest gain is chosen.

8 How does XGBoost perform tree pruning?

XGBoost uses **post-pruning**, not pre-pruning like traditional decision trees.

- During tree construction, it first grows the tree fully.
- Then it calculates the gain from each split.
- If the gain < 0, the split is pruned (i.e., not used).

This helps reduce overfitting and avoids unnecessary splits.

9 What is the role of the gamma parameter?

- γ is the minimum loss reduction required to make a split.
- Acts as a regularization parameter.
- Higher $\gamma \rightarrow$ fewer splits (more conservative).
- It prunes splits that do not result in significant improvement.

Gain calculation:

$$\text{Gain} = \text{Score}_{\text{left}} + \text{Score}_{\text{right}} - \text{Score}_{\text{parent}} - \gamma$$

If Gain < 0: no split is performed.

10 Handling Overfitting in XGBoost

XGBoost prevents overfitting through:

- **Regularization:** via `lambda` (L2), `alpha` (L1) and `gamma`.
- **Tree-specific limits:**
  - `max_depth` - limits depth of trees.
  - `min_child_weight` - minimum sum of instance weight per leaf.
  - `subsample` - andomly samples rows before growing trees.
  - `colsample_bytree` -andomly samples columns (features).
- **Learning rate (`eta`)**: Shrinks weight of new trees.
- **Early stopping**: stops training when validation score stops improving.

10.1 Regularization in XGBoost

Regularization helps prevent overfitting by discouraging overly complex models (e.g., very deep trees or large leaf weights). XGBoost uses both L1 and L2 regularization, as well as a split-level penalty.

10.1.1 Lambda (L2 Regularization)

- Adds a penalty on the squared values of leaf weights.
- Helps keep the leaf scores small and smooth.
- Encourages the model to avoid placing too much weight on any one leaf.

Mathematically:

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

Where:

- T: number of leaves
- w_j: weight of leaf j
- Higher `lambda` leads to stronger shrinkage of leaf scores, resulting in a more conservative model.

10.1.2 Alpha (L1 Regularization)

- Adds a penalty on the absolute values of the leaf weights.
- Encourages sparsity, i.e., it can push some weights to zero, effectively pruning parts of the tree.

Mathematically:

$$\Omega(f) = \gamma T + \alpha \sum_{j=1}^{T} |w_j| + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Useful when only a few features are expected to be important. Similar to Lasso regularization in linear models.

10.1.3 Gamma (Minimum Split Loss Reduction)

- Applies a penalty for adding new splits (nodes) to the tree.
- Controls tree complexity at the structure level (not leaf weights).

Split Gain Formula:

$$\text{Gain} = \text{Score}_{\text{left}} + \text{Score}_{\text{right}} - \text{Score}_{\text{parent}} - \gamma$$

If Gain < 0, the split is not made.
Higher `gamma` reduces the number of splits, resulting in simpler trees.

11 What is the difference between histogram-based and exact greedy algorithm in XGBoost?

Exact Greedy:

- Calculates the best split by evaluating every possible split.
- Accurate but slow and memory-intensive for large datasets.

Histogram-Based:

- Bins continuous features into discrete buckets (histograms).
- Finds the best split using histogram summaries.
- Faster and scalable, with minor accuracy trade-off.

Modern XGBoost uses histogram-based splitting by default.

12 How does XGBoost differ from LightGBM and CatBoost?

Feature	**XGBoost**	**LightGBM**	**CatBoost**
Splitting Strategy	Level-wise (default)	Leaf-wise (with depth limit)	Symmetric Tree (Oblivious Tree)
Speed	Fast	Faster than XGBoost on large data	Slightly slower, but efficient for cat vars
Categorical Handling	Needs preprocessing (e.g., one-hot)	Basic support	Native support
Accuracy	High	Higher in many cases (faster convergence)	Very high (especially for categorical)
GPU Support	Yes	Yes	Yes
Interpretability	Good	Moderate	Moderate

13 Training Process

13.1 How Do Trees Learn from Previous Errors?

This is the gradient boosting part of XGBoost. Each tree tries to correct the mistakes of the previous ensemble.

Step 1: Start with initial prediction
Usually:

$$\hat{y}^{(0)} = 0 \Rightarrow P(\text{class } 1) = 0.5$$

Step 2: Iteratively add trees
For each round t:

1. Compute the gradient (g_i) and hessian (h_i) of the loss function for each sample.
    - These represent how wrong the current prediction is, and how confident the model is.
2. Train a tree to predict these gradients.

- This tree outputs values (leaf scores) that help reduce the current error.

3. Add the new tree's output to the running total:

$$\hat{y}^{(t)} = \hat{y}^{(t-1)} + \eta \cdot f_t(x)$$

   - η is the learning rate.

Repeat this for all boosting rounds.

13.2 What the Next Tree Learns?

The next tree is trained to predict $f_t(x)$ such that it minimizes this expression:

$$\sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right]$$

So the tree learns how much to correct each sample using the gradients and Hessians.
This becomes a weighted regression problem, where:

- Gradient g_i is the target.
- Hessian h_i is the weight.

13.3 Final Step: How a Leaf Value is Calculated?

Once a tree is built, each leaf node gets a score based on the gradients and Hessians of samples that land in it:

$$w_j = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

Where:

- w_j: value of leaf j
- I_j: set of samples in leaf j
- λ: L2 regularization

14 How xgboost calculate class probabilities?

14.1 Binary Classification

Objective:
Use `binary:logistic` as the objective function.

Prediction Flow:

- XGBoost outputs raw scores (logits) from the trees:

$$f(x) = \sum_{k=1}^{K} f_k(x)$$

  K is the total number of trees.

- Apply the sigmoid function to convert the raw score into a probability:

$$P(y = 1 \mid x) = \frac{1}{1 + e^{-f(x)}}$$

- We get a probability between 0 and 1 for the positive class (class 1).

In code:

```
model = xgb.XGBClassifier(objective='binary:logistic')
probs = model.predict_proba(X_test)  # returns [P(class 0), P(class 1)]
```

14.2 Multiclass Classification

Objective:
Use `multi:softprob` (or `multi:softmax` for class labels directly).

Prediction Flow:

- XGBoost builds `num_class` trees per boosting round (1 per class).
- Outputs a raw score (logit) for each class.
- Applies softmax function across classes:

$$P(y = i \mid x) = \frac{e^{f_i(x)}}{\sum_j e^{f_j(x)}}$$

  Where $f_i(x)$ is the raw score for class i.

In code:

```
model = xgb.XGBClassifier(objective='multi:softprob', num_class=3)
probs = model.predict_proba(X_test)  # shape: (n_samples, n_classes)
```

Step-by-step

✓ **Each tree outputs a value at each leaf (called the leaf weight):**

- This is not a class label, but a raw score or logit.
- A sample reaches a leaf based on its feature values.
- The output of the leaf becomes the contribution from that tree.

For a sample x, if it lands in a leaf with value 0.8, then:

$$f_1(x) = 0.8$$

These values from all K trees are added:

$$f(x) = \sum_{k=1}^{K} f_k(x)$$

This sum is a logit, not a probability yet.

✓ **Then, we apply sigmoid to turn it into a probability:**

$$P(y = 1 \mid x) = \frac{1}{1 + e^{-f(x)}}$$

So the final class probability depends on the sum of all leaf outputs across trees.

15 Gain in XGBoost

Analogy
Imagine you're deciding whether to build a new road:

- **Gain** = how useful the road will be.
- **Gamma** = cost to build it.

If the usefulness (gain) is greater than the cost (gamma), build it. Otherwise, skip it.

Gain measures how much the loss function improves when a node is split into two child nodes. It is a key metric used to decide where to split the tree during training.
In simple terms: Gain = how much better the model becomes after a split.

15.1 Loss-based Tree Building

Instead of impurity, XGBoost builds trees by minimizing a loss function, like log loss or squared error.To choose a split:
It uses the second-order Taylor approximation of the loss:

$$L \approx \sum_i \left[g_i w + \frac{1}{2} h_i w^2 \right]$$

Where:

- g_i: first derivative (gradient)
- h_i: second derivative (hessian)
- w: prediction (leaf weight)

Gain Formula in XGBoost

Let's say we split a node into left (L) and right (R).

Let:

$$G_L = \sum_{i \in L} g_i, \quad H_L = \sum_{i \in L} h_i$$
$$G_R = \sum_{i \in R} g_i, \quad H_R = \sum_{i \in R} h_i$$
$$G = G_L + G_R, \quad H = H_L + H_R$$

Then, the gain from the split is:

$$\text{Gain} = \frac{1}{2} \left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda} \right) - \gamma$$

Where:

- λ: L2 regularization term (to avoid overfitting)
- γ: minimum gain required to make a split (pruning threshold)

Intuition Behind the Formula

- Numerators G^2 represent "signal strength" — how consistent the gradients are.
- Denominators $H + \lambda$ account for confidence and regularization.
- Subtracting the parent score from child scores tells us how much better the split is.
- γ penalizes overly complex trees.

In Practice XGBoost loops over all features and all possible thresholds for splitting, and picks the one with the maximum gain.

- XGBoost remembers the split with the highest gain.
- If the gain is greater than 0 (minimum split loss), then it commits the split.
- If gain < 0 , it doesn't split.

16 Techniques to improve an underperforming XGBoost model

- Tune hyperparameters
- Try different loss functions (e.g., logloss, error, auc)
- Feature engineering (interactions, encoding, binning, scaling not needed)
- Add more trees with a smaller learning rate
- Reduce overfitting:
  - Lower `max_depth`
  - Increase `gamma`
  - Use `subsample` and `colsample_bytree` ¡ 1.0
- Handle class imbalance

17 Handling highly imbalanced datasets in XGBoost

Techniques:

- Use `scale_pos_weight = (# negative / # positive)` — gives higher penalty to minority class, here # negative $>$ # positive.
- Use eval metric like `auc`, `f1`, or `logloss` (accuracy can be misleading).
- Try `SMOTE`, undersampling, or oversampling.
- Tune `max_delta_step` (useful when classes are extremely imbalanced).
- Apply threshold tuning (e.g., change decision threshold from 0.5 to 0.3).

18 Interpreting XGBoost Feature Importance

Types of importance scores:

Type	Meaning
Gain	Average gain (loss reduction) from splits using the feature (best)
Cover	Avg. number of samples affected by those splits
Frequency	How often a feature is used in trees

Which one to trust?

- Gain is usually the most informative and stable.
- Use `plot_importance(model, importance_type='gain')`

19 How to use cross-validation in XGBoost

- Using xgb.cv() from XGBoost's native API

```python
import xgboost as xgb

# Prepare DMatrix
dtrain = xgb.DMatrix(X, label=y)

# Set parameters
params = {
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'eta': 0.1,
    'max_depth': 4
}

# Perform cross-validation
cv_results = xgb.cv(
    params=params,
    dtrain=dtrain,
    num_boost_round=200,
    nfold=5,
    early_stopping_rounds=10,
    metrics='auc',
    as_pandas=True,
    seed=42
)

print(cv_results.tail())
```

- Using `GridSearchCV` with `XGBClassifier` (sklearn API)

```python
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV

model = XGBClassifier(use_label_encoder=False, eval_metric='logloss')
```

```
param_grid = {
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1],
    'n_estimators': [100, 200],
    'subsample': [0.8, 1.0]
}

grid = GridSearchCV(
    model,
    param_grid,
    cv=5,
    scoring='roc_auc',
    verbose=1
)
grid.fit(X, y)

print("Best params:", grid.best_params_)
print("Best AUC:", grid.best_score_)
```