XGBoost For Classification

```
import xgboost as xgb
```

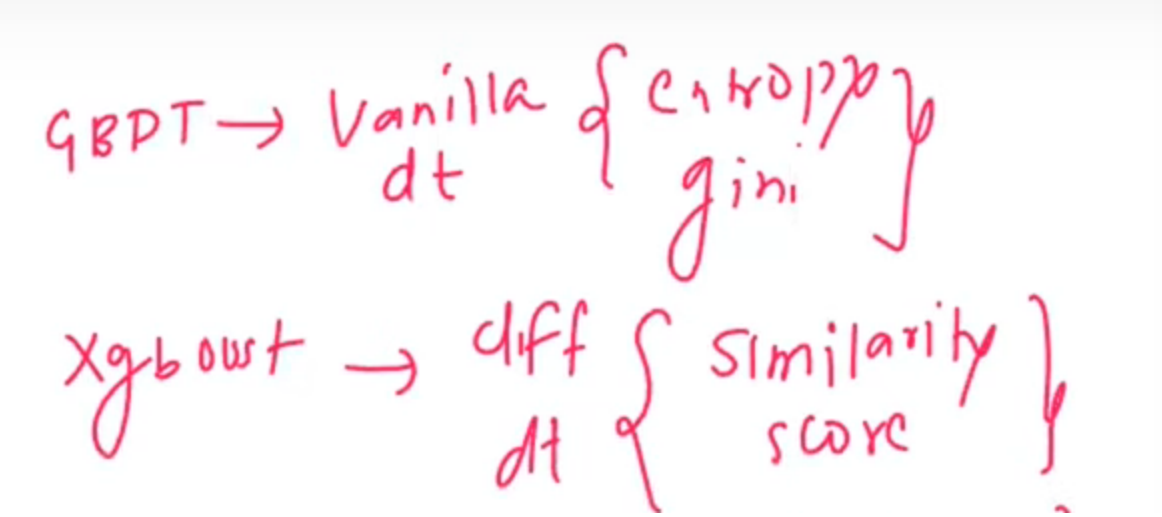
```
xgb.XGBClassifier()
```

How XGBoost Works for Classification

1. **Objective:**
    - For classification, XGBoost predicts the probability of an instance belonging to a particular class (binary or multi-class).
    - It typically uses **logistic regression** as the **base loss function** for **binary classification** (log loss/cross-entropy) or **softmax** for **multi-class problems.**
        - Loss Function:
            - **logistic regression : binary classification**
            - **softmax** : **multi-class problems.**
2. **Tree Building:**
    - It constructs decision trees iteratively.
    - Each tree focuses on the residuals (errors) of the previous trees, adjusting predictions to reduce the overall error.
3. **Gradient Boosting:**
    - The algorithm calculates the gradient of the loss function and fits each new tree to the negative gradient (pseudo-residuals).
4. **Regularization:**
    - XGBoost adds penalties for model complexity (e.g., number of leaves, depth of trees) via L1 (Lasso) and L2 (Ridge) regularization, making it less prone to overfitting than traditional gradient boosting.
5. P**rediction:**

- For binary classification, it outputs a probability (0 to 1), which is thresholded (e.g., 0.5) to assign a class.
- For multi-class, it uses softmax to assign the class with the highest probability.

Decision Trees: Gradient Boost vs XGBoost

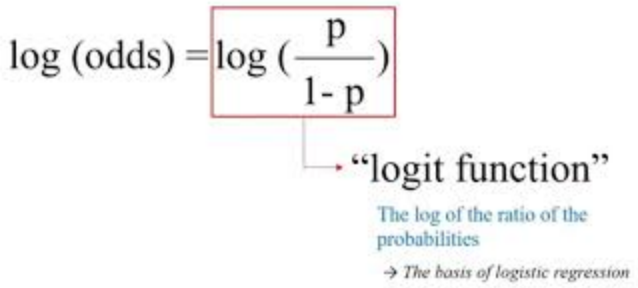


> **In Gradient boost, the DTs calculate entropy & gini impurity, while in XGBoost, it calculates similarity score.**

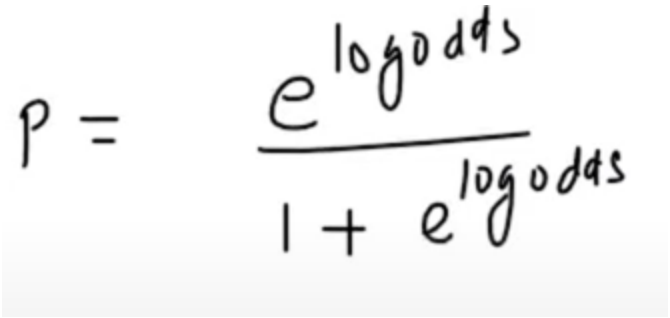
Step-by-Step

1. Initialize the Model

- Start with an initial prediction for all samples. For classification, this is typically the **log-odds** of the target class probabilities.
  - Example: If 60% of the samples belong to class 1, the initial prediction might be the log-odds of 0.6.



- Convert this👆 into **Probability**



---

2. Calculate Residuals (Errors)

- Compute the difference between the **actual class labels** and the **predicted probabilities** (these are called **residuals**).
  - Example: If the predicted probability for a sample is 0.7 but the actual label is 1, the residual is:

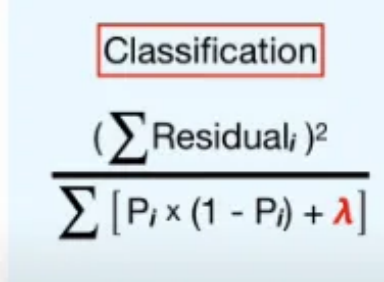
    $$1 - 0.7 = 0.3$$

---

3. Build a Tree to Predict Residuals

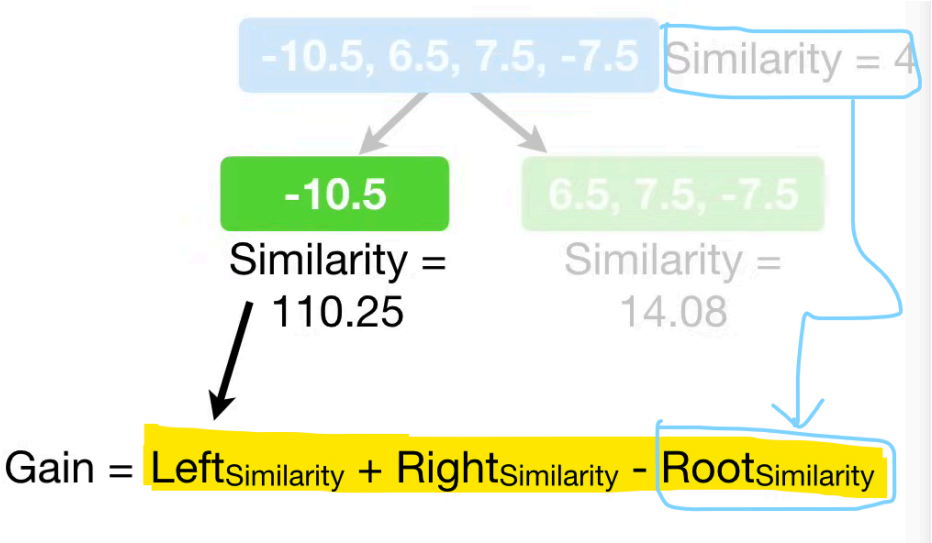
- Train a decision tree to predict the residuals.
- **Similarity Score**:
  - For each node, XGBoost calculates a **similarity score** to measure how "similar" (homogeneous) the residuals are in that node.
  - Formula (simplified):

$$\text{Similarity} = \frac{(\text{Sum of residuals})^2}{\text{Number of residuals} + \lambda}$$



  - (where λ is a regularization term to prevent overfitting).

- **Gain**:
  - The "gain" of a split is the difference between the similarity score of the parent node and the combined similarity scores of its child nodes.
  - Splits with higher gain are prioritized (they create more homogeneous groups).



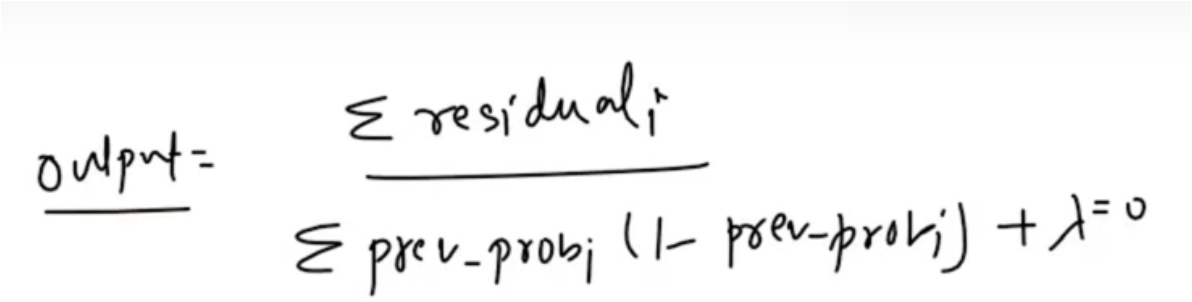
- **Stopping Condition**:
  - A split is only made if the gain exceeds a threshold (gamma), the `gamma` hyperparameter).

- This prevents overly complex trees.

4. Calculate Output Values for Leaves

- Each leaf (terminal node) in the tree is assigned an **output value** that adjusts the predictions.
- Output value formula (simplified):

$$\text{Output} = \frac{\text{Sum of residuals}}{\text{Number of residuals} + \lambda}$$



- This value represents the "correction" needed for the residuals in that leaf.

5. Update Predictions

- Add the tree's output values (scaled by the `learning_rate`) to the previous predictions.
- Example: If a leaf's output is 0.2 and the learning rate is 0.1, the prediction increases by 0.02.

6. Repeat for Multiple Trees

- Repeat steps 2–5 for `n_estimators` trees.
- Each new tree focuses on the residuals (errors) left by the previous trees.

7. Convert Probabilities to Class Labels

- After training, the model outputs **probabilities** for each class.
- For binary classification, a threshold (e.g., 0.5) is used to convert probabilities to class labels (0 or 1).
- For multi-class classification, the class with the highest probability is selected.

Key Features of XGBClassifier

1. **Regularization**:
   - Uses L1 (`reg_alpha`) and L2 (`reg_lambda`) regularization to prevent overfitting.
2. **Handling Missing Values**:
   - Automatically learns the best imputation strategy during training.
3. **Early Stopping**:
   - Stops training if validation performance doesn't improve for a specified number of rounds.
4. **Feature Importance**:
   - Provides insights into which features contribute most to the predictions.

Example: Binary Classification

Suppose you're predicting whether a customer will churn (1) or not (0):

1. **Initial Prediction**: Start with the log-odds of the churn rate (e.g., 0.2).
2. **First Tree**: Corrects predictions for customers with high usage but low churn probability.
3. **Second Tree**: Adjusts for customers with low satisfaction scores.
4. **Final Prediction**: Combines all corrections from all trees to output probabilities (e.g., 0.85 for churn).

Why XGBClassifier is Powerful

- **Speed**: Optimized for fast training and prediction.

- **Accuracy**: Often achieves state-of-the-art performance on classification tasks.
- **Flexibility**: Supports custom objectives, evaluation metrics, and handling of missing values.

Python Example: Binary Classification

```
import xgboost as xgb
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_
matrix

# Load dataset
data = load_breast_cancer()
X, y = data.data, data.target

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
e=42)

# Initialize XGBClassifier
model = xgb.XGBClassifier(
    objective='binary:logistic',  # Binary classification
    n_estimators=100,           # Number of trees
    learning_rate=0.1,          # Learning rate
    max_depth=5,                # Maximum depth of trees
    subsample=0.8,              # Fraction of samples used per tree
    colsample_bytree=0.8,       # Fraction of features used per tree
    gamma=1,                    # Minimum loss reduction for a split
    reg_lambda=1,               # L2 regularization
    reg_alpha=0,                # L1 regularization
    eval_metric='logloss',      # Evaluation metric
    early_stopping_rounds=10,   # Early stopping
```

```
    random_state=42            # Random seed
)

# Train the model
model.fit(X_train, y_train, eval_set=[(X_test, y_test)], verbose=False)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=data.target_names))

# Confusion matrix
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

```
Accuracy: 0.96

Classification Report:
              precision    recall  f1-score   support

   malignant       0.98      0.93      0.95        43
      benign       0.96      0.99      0.97        71

    accuracy                           0.96       114
   macro avg       0.97      0.96      0.96       114
weighted avg       0.97      0.96      0.96       114


Confusion Matrix:
[[40  3]
 [ 1 70]]
```

Results with vanilla `XGBClassifier()` 👇

```
Accuracy: 0.96

Classification Report:
              precision    recall  f1-score   support

   malignant       0.95      0.93      0.94        43
      benign       0.96      0.97      0.97        71

    accuracy                           0.96       114
   macro avg       0.96      0.95      0.95       114
weighted avg       0.96      0.96      0.96       114


Confusion Matrix:
[[40  3]
 [ 2 69]]
```

Multi-Class Classification Example

For multi-class classification, change the `objective` to `multi:softmax` and specify the number of classes (`num_class`):

```
# Initialize XGBClassifier for multi-class classification
model = xgb.XGBClassifier(
    objective='multi:softmax',  # Multi-class classification
    num_class=3,                # Number of classes
    n_estimators=100,
    learning_rate=0.1,
    max_depth=5,
    subsample=0.8,
    colsample_bytree=0.8,
    gamma=1,
    reg_lambda=1,
    eval_metric='mlogloss',     # Multi-class log loss
    early_stopping_rounds=10,
```

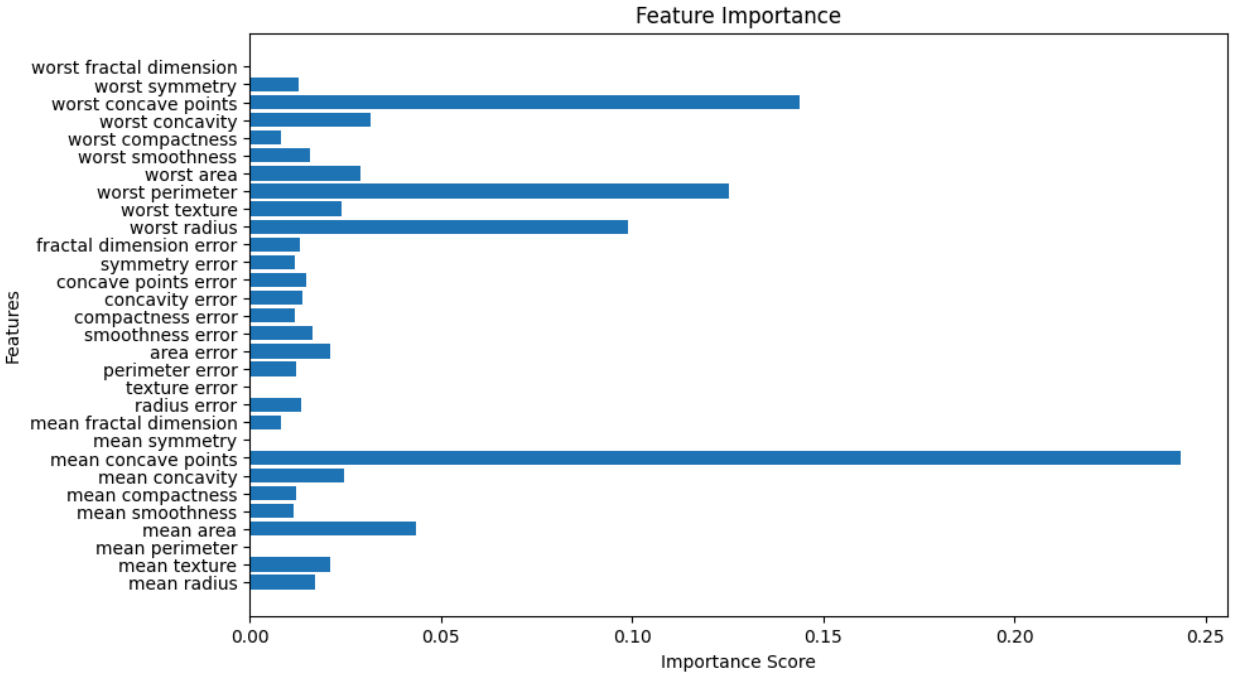
```
    random_state=42
)
```

Feature Importance

```
import matplotlib.pyplot as plt

# Get feature importance
importance = model.feature_importances_

# Plot feature importance
plt.figure(figsize=(10, 6))
plt.barh(data.feature_names, importance)
plt.title("Feature Importance")
plt.xlabel("Importance Score")
plt.ylabel("Features")
plt.show()
```



💡 Important Notes

- **For Imbalanced datasets** (use `scale_pos_weight` for class imbalance)

How to use `scale_pos_weight` ?

- **Answer:** Set it as the ratio of negative to positive samples (e.g., neg_samples / pos_samples) for binary classification to balance imbalanced classes.
- **Details:**
  - Used in binary classification to give more weight to the minority class.
  - Example: If you have **900 negatives (0) and 100 positives (1)**,
    - ***scale_pos_weight = 900 / 100 = 9.***
  - `xgb.XGBClassifier(scale_pos_weight=9)`
  - Helps when precision/recall for the minority class is critical (e.g., fraud detection).

How to Manage Imbalanced Data?

There are **four** key techniques:

Method	How It Works	Pros	Cons
Oversampling (SMOTE, ADASYN)	Generates synthetic minority samples	More data, better recall	Can introduce noise
Undersampling	Removes some majority class samples	Faster training	Risk of losing useful data
Class Weighting (`scale_pos_weight`)	Adjusts class importance	No data loss	Can lead to overfitting
Balanced Ensemble Models	Uses multiple models trained on balanced subsets	Handles extreme imbalance	Computationally expensive

- **Evaluation Metric:** Use F1-score, AUC-PR, or balanced accuracy instead of plain accuracy.

What's `stratify` ?

✅ **Ensures train-test split maintains class ratio**

- If you have **80% class 0, 20% class 1**, a normal split might **randomly select too many from one class**.
- **Stratify keeps the same ratio** in both train & test sets.
- Without it, random splitting might skew the distribution (e.g., test set with no minority class).

Usage: `train_test_split(X, y, test_size=0.2, stratify=y).`

Does XGBoost Auto-Select `multi:softmax` or `binary:logistic` ?

✅ **Yes, based on** `num_class` **.**

Scenario	Objective Auto-Selected
Binary Classification (0/1)	`binary:logistic`
Multi-Class Classification (3+ classes)	`multi:softmax`
Custom Probabilities	`multi:softprob`

To **force** a choice, explicitly se

```
xgb.XGBClassifier(objective='multi:softmax', num_class=3)
```

How to Deal with Missing Data?

✅ XGBoost **automatically handles missing values** (no need for imputation).

- It **learns optimal split directions** for missing values.
- If a feature is missing in a row, XGBoost **chooses the best path** instead of discarding data.
- If you want to manually set missing values, use `missing=np.nan` .

```
xgb.XGBClassifier(missing=np.nan)
```

Does it need one-hot encoding?

Answer: ✅ No, it works with categorical data directly!

XGBoost doesn't require one-hot encoding; it handles categorical data natively (with some prep).

- **If you use `XGBClassifier()` (>= v1.3), it supports categorical features natively.**
- **One-Hot Encoding (OHE) expands features** → Can increase feature count **to thousands**, slowing down training.
- Instead of OHE, **use Label Encoding** or let XGBoost handle categories.

Example: Label Encoding Instead of OHE

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df['category'] = le.fit_transform(df['category'])
```

- **XGBoost can handle categorical features directly using `LabelEncoder` or by specifying `enable_categorical=True` (in newer versions).**

```
model = xgb.XGBClassifier(enable_categorical=True)
```

💡 **If the data contains string categories (e.g., city names), XGBoost cannot process them directly. It expects numerical input and provides two ways to handle it:**

Automatic Categorical Handling (XGBoost 1.3+)

- XGBoost **natively** supports categorical data.
- Convert the column **to a Pandas categorical type** and pass it to `XGBClassifier()`.

```
df['city'] = df['city'].astype('category')
model = xgb.XGBClassifier()
model.fit(df[['city']], y)
```

If you're using a newer version of XGBoost, you can enable categorical feature support with `enable_categorical=True`:

```
model = xgb.XGBClassifier(enable_categorical=True)
```

✨When you set `enable_categorical=True` in XGBoost, the library **automatically detects and handles categorical columns**. You **do not need to explicitly specify the column names** of the categorical features.

Label Encoding (For Older Versions or Compatibility)

- Convert categorical data **into numeric labels** using `LabelEncoder`.
- **Best for tree-based models** (keeps feature space small).

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['city'] = le.fit_transform(df['city'])
```

one-hot encoding vs. LabelEncoding

- One-hot encoding increases dimensionality, while LabelEncoding is more efficient for tree-based models.
    - **One-Hot Encoding**: Creates a binary column for each category. Increases dimensionality and can lead to sparse data.
    - **LabelEncoding**: Converts categories into integers. More efficient for tree-based models like XGBoost.

Method	How It Works	Best Used When	Impact on Performance
One-Hot Encoding (OHE)	Converts categorical columns into multiple binary columns	Small datasets, linear models	Increases feature space (1000s of new cols)
Label Encoding	Assigns numeric values (A → 0, B → 1)	Tree-based models (XGBoost, RF)	Works well, keeps feature space small

✅**Use Label Encoding for XGBoost.**

Cons: Implies order (e.g., 0 < 1 < 2), which might mislead tree splits if no real order exists.

✅OHE only when needed for compatibility with other models.

Do we need to scale the data?

- Answer: **No**, XGBoost doesn't require scaling since it's tree-based.

Model	Needs Scaling?	Why?
Linear Models (Logistic Regression, SVM, kNN)	✅ Yes	These models are sensitive to feature magnitudes
Tree-Based Models (XGBoost, Decision Tree, Random Forest)	❌ No	Splits are based on feature ordering, not distance

How does XGBoost handle White Spaces?

✅ **XGBoost does NOT automatically remove white spaces.**

- Leading/trailing spaces in categorical features **can cause issues**.
- Remove them manually before training:

```
df = df.apply(lambda x: x.str.strip() if x.dtype == "object" else x)
```

- ***Rows***: If a feature value is " " (string), XGBoost expects numeric input and will fail unless converted (e.g., to NaN or 0).
- ***Columns:*** Whitespace in column names (e.g., "fixed acidity") is fine in DMatrix as long as Pandas handles it upfront.

- Fix: Use `df.replace(' ', np.nan)` for **values**, and
- `df.columns = df.columns.str.replace(' ', '_')` for **names**

DMatrix

✅ **DMatrix is an optimized data structure for XGBoost that improves speed & memory efficiency.**

- **Converts data into an efficient internal format.**
- **Reduces memory usage** by storing sparse data effectively.
- **Speeds up training** by avoiding repeated conversions.

Components:

- `data`: Feature matrix (e.g., X_train).
- `label`: Target vector (e.g., y_train).
- Optional: `weight`, `missing` (for custom handling).

Why use DMatrix?

- Faster training and evaluation compared to standard NumPy arrays or Pandas DataFrames.
- Supports advanced features like cross-validation and early stopping.

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

# Define parameters
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'eta': 0.1
}

# Train model using DMatrix
```

```
bst = xgb.train(params, dtrain, num_boost_round=100)

# Make predictions
y_pred = bst.predict(dtest)
```

How It Works:

1. **Conversion**: Takes `X_train` (e.g., a 1000×11 matrix) and `y_train` (e.g., a 1000-element vector) and builds a compressed object.
2. **Missing Values:** Identifies `np.nan` (or custom missing) and prepares them for special handling.
3. **Optimization**: Stores data in a column-major format, enabling fast access for feature splits and parallel processing.

Why Not Raw Data?:

- Raw Pandas/NumPy data requires repeated preprocessing (e.g., missing value checks) during training, slowing it down.
- `DMatrix` does this once upfront, caching results for speed.