XGBoost (eXtreme Gradient Boosting) is one of the most popular and powerful boosting techniques in machine learning, particularly for **structured/tabular data**. It is an advanced implementation of Gradient Boosting, optimized for speed and performance.

XGBoost is a scalable and efficient version of Gradient Boosting. It incorporates several enhancements over traditional Gradient Boosting, including:

1. Regularization to prevent overfitting.
2. Parallelization to speed up training.
3. Handling missing values natively.
4. Tree pruning for better optimization.
5. Custom loss functions for flexibility.

## Key Features of XGBoost

1. Regularization:
    - Adds L1 (Lasso) and L2 (Ridge) regularization terms to the loss function to control model complexity.
2. Tree Pruning:
    - Uses "max_depth" instead of the "depth-first" splitting to stop trees early, avoiding unnecessary splits.
3. Weighted Quantile Sketch:
    - Handles weighted datasets efficiently for split-point selection.
4. Parallel Processing:
    - Boosting itself is sequential, but operations like finding the best split for trees are parallelized.
5. Sparsity Awareness:
    - Handles missing values gracefully by learning the best way to deal with them.

Here’s an example of using XGBoost for classification on the Iris dataset.

```
!pip install xgboost
```

```
Requirement already satisfied: xgboost in /usr/local/lib/python3.10/dist-packages (2.1.3
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from xg
Requirement already satisfied: nvidia-nccl-cu12 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from xg
```

```
# Import required libraries
import xgboost as xgb
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
# Load the Iris dataset
iris = load_iris()
X, y = iris.data, iris.target
```

```
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3
```

```
# Create a DMatrix (optimized data structure for XGBoost)

train_data = xgb.DMatrix(data=X_train, label=y_train)
test_data = xgb.DMatrix(data=X_test, label=y_test)
```

```
# Define parameters for the XGBoost model
params = {
    "objective": "multi:softmax",  # For classification with multiple c
    "num_class": 3,                # Number of classes in the target
    "max_depth": 3,                # Maximum depth of trees
    "eta": 0.1,                    # Learning rate
    "subsample": 0.8,              # Fraction of samples to use for tra
    "colsample_bytree": 0.8,       # Fraction of features to use for ea
    "seed": 42                     # Random seed for reproducibility
}
```

```
# Train the XGBoost model
xgb_model = xgb.train(params, train_data, num_boost_round=100)
```

```
# Make predictions
y_pred = xgb_model.predict(test_data)
```

```
# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of XGBoost Classifier: {accuracy:.2f}")
```

```
Accuracy of XGBoost Classifier: 1.00
```

## Code Walkthrough

1. Data Preparation:
   - DMatrix: A data structure optimized for XGBoost, holding both features and labels.
2. Model Parameters:
   - objective="multi:softmax": Specifies a classification task with multiple classes.
   - num_class=3: Number of unique target classes in the dataset.
   - max_depth=3: Restricts the depth of each tree to control overfitting.
   - eta=0.1: Learning rate (controls how much each tree contributes to the model).
   - subsample=0.8: Uses 80% of the training data for each tree.
   - colsample_bytree=0.8: Uses 80% of the features for each tree.
3. Training:
   - xgb.train(): Trains the XGBoost model sequentially on residual errors.
4. Prediction:
   - Predictions are made using the trained model on the test data.
5. Evaluation:
   - Accuracy is calculated to evaluate the model's performance.

### Advantages of XGBoost

1. Extremely fast and efficient.
2. Highly flexible (supports various objectives and custom loss functions).
3. Handles missing data automatically.
4. Outperforms traditional Gradient Boosting in many scenarios.

## Disadvantages of XGBoost

- May require careful hyperparameter tuning for best performance.
- Computationally intensive for very large datasets.

Key Points:

1. Binary Classification: If the task were binary classification (e.g., 0 and 1), we would use "objective": "binary:logistic", and num_class wouldn’t be needed.
2. Multi-Class Classification: For multi-class tasks like the Iris dataset, "multi:softmax" ensures the model handles all three classes correctly.

## When to Use Each

1. multi:softmax:
   - Use when you only need the predicted class (e.g., 0, 1, or 2).
2. multi:softprob:
   - Use when you need the probabilities for all classes (e.g., [0.2, 0.5, 0.3]).