Boosting is an ensemble learning technique that focuses on **reducing bias** and improving model accuracy. It builds models **sequentially**, where each new model focuses on correcting the errors made by the previous models. The final predictions are combined in a weighted manner, giving more importance to the stronger models.

Steps of Boosting

1. Train Initial Model: Train the first weak learner (e.g., a decision tree) on the dataset.
2. Identify Errors: Evaluate the errors (misclassified samples) of the current model.
3. Update Weights: Increase the importance (weights) of the misclassified samples so that the next model focuses more on them.
4. Combine Models: Sequentially add new models, and their predictions are combined based on their performance.

Why Boosting Works?

1. Bias Reduction: Sequentially correcting errors minimizes bias.
2. Weighted Contributions: Stronger models have more influence, improving final predictions.
3. Flexibility: Boosting adapts well to various data distributions.

Common Boosting Algorithms

1. AdaBoost: Adjusts weights of misclassified samples.
2. Gradient Boosting: Minimizes the loss function by training models on the residual errors.
3. XGBoost/LightGBM: Optimized versions of gradient boosting.

A weak learner is a simple model that performs slightly better than random guessing. Here, we use a Decision Tree with a maximum depth of 1. This means:

- The tree can only make one split, resulting in a very simple decision boundary.
- This is why it’s called a "shallow" decision tree.

Example of Boosting in Action

Imagine the dataset has 3 misclassified samples (A, B, and C):

1. First Learner: Correctly classifies most samples but misclassifies A and B.
2. Second Learner: Focuses on A and B, getting them right but misclassifies C.
3. Third Learner: Focuses on C and corrects it. Finally, the ensemble combines all predictions, ensuring that errors from earlier models are corrected by later models.

Training and Prediction

Boosting happens sequentially, so the model works in the following way:

1. Train the First Weak Learner:
    - Train the decision tree on the dataset.
    - Evaluate the errors (misclassified points).
2. Focus on Errors:
    - Increase the weight of misclassified samples so they are given more importance.
    - This ensures the next weak learner focuses more on these difficult cases.
3. Repeat for All Learners:
    - Each new weak learner is trained on the updated dataset (with new weights).
    - This process continues for all 50 weak learners.
4. Combine Results:

- Each learner's prediction is weighted based on its accuracy.
- The final prediction is a weighted combination of all weak learners.

We are useing the Iris dataset for a classification example with AdaBoostClassifier.

```
# Import required libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
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, random_state=42)
```

```
# Initialize a base model (Decision Tree with max depth)
base_model = DecisionTreeClassifier(max_depth=1, random_state=42)
```

```
# Create an AdaBoostClassifier

adaboost_model = AdaBoostClassifier(
    estimator=base_model,       # Base learner
    n_estimators=50,            # Number of weak learners
    learning_rate=1.0,          # Controls the contribution of each learner
    random_state=42
)
```

AdaBoostClassifier Parameters

1. n_estimators=50
   - We train 50 weak learners (decision trees in this case) sequentially.
   - Each weak learner focuses on fixing the errors made by the previous ones.
   - Impact: Increasing n_estimators can lead to better performance but may risk overfitting if set too high.
2. learning_rate=1.0
   - This controls how much influence each weak learner has on the final prediction.
   - A lower learning rate means the model learns more gradually (but may require more estimators).
   - Impact: It adjusts the weight updates during boosting. A higher learning rate increases the contribution of each learner but risks instability.

```
# Train the AdaBoost model
adaboost_model.fit(X_train, y_train)
```

```
▸ AdaBoostClassifier
▸ estimator: DecisionTreeClassifier
  ▸ DecisionTreeClassifier
```

```
# Make predictions
y_pred = adaboost_model.predict(X_test)
```

```
# Evaluate accuracy

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

```
Accuracy of AdaBoost Classifier: 1.00
```

Code Walkthrough

1. Base Model: A shallow decision tree (max depth=1) is used as a weak learner.
2. AdaBoostClassifier Parameters:
    - n_estimators=50: Use 50 weak learners sequentially.
    - learning_rate=1.0: Controls how much each model contributes to the final prediction.
3. Training and Prediction: Sequentially trains models, focusing on correcting previous errors.

Here's an example of using Gradient Boosting for classification with the Iris dataset. We are useing GradientBoostingClassifier from Scikit-learn.

Gradient Boosting

Gradient Boosting is a boosting technique where each weak learner is trained to predict the **residual errors (differences between the true values and predictions)** of the previous learners. The model minimizes a loss function (e.g., log-loss for classification or mean squared error for regression) using gradient descent.

```
# Import required libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
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, random_state=42)
```

```
# Create a GradientBoostingClassifier
gb_model = GradientBoostingClassifier(
    n_estimators=100,      # Number of weak learners
    learning_rate=0.1,     # Step size for weight updates
    max_depth=3,           # Maximum depth of each decision tree
    random_state=42
)
```

```
# Train the Gradient Boosting model
gb_model.fit(X_train, y_train)
```

```
▾ GradientBoostingClassifier
GradientBoostingClassifier(random_state=42)
```

```
# Make predictions
y_pred = gb_model.predict(X_test)
```

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

```
Accuracy of Gradient Boosting Classifier: 1.00
```

Code Walkthrough

1. Model Initialization:
    - n_estimators=100: Use 100 decision trees in sequence.
    - learning_rate=0.1: Each tree contributes less, allowing for gradual improvement.
    - max_depth=3: Restrict trees to a maximum depth of 3 for simplicity and to avoid overfitting.
2. Training:
    - The model trains sequentially, where each tree focuses on correcting the residual errors of the previous trees.
3. Prediction:
    - Final predictions are made by combining the results of all trees.
4. Evaluation:
    - The accuracy of the model is computed on the test set.

Unlike AdaBoost, where you specify a weak learner explicitly, Gradient Boosting uses shallow decision trees (typically one with a small maximum depth) internally to fit the residuals. You control these weak learners indirectly using parameters like max_depth, n_estimators, and learning_rate.

How Gradient Boosting Works Internally

1. Weak Learners:
    - By default, the weak learners are shallow decision trees (max depth controlled by max_depth).
    - These trees are trained sequentially.
2. Residual Learning:
    - The first tree is trained on the original target values.
    - Each subsequent tree is trained on the residual errors (differences between true and predicted values from all previous trees).
3. Weighted Predictions:
    - The predictions from all trees are combined (weighted by the learning rate) to make the final prediction.

Why Gradient Boosting Works

1. Error Correction: Each tree is trained on the residuals, incrementally reducing the error.
2. Flexibility: Can optimize for various loss functions (e.g., classification, regression).
3. Hyperparameters: Fine-tuning parameters like n_estimators, learning_rate, and max_depth helps control performance and avoid overfitting.