XGBoost for Regression

```
import xgboost as xgb
```

```
model = xgb.XGBRegressor()
```

https://xgboost.readthedocs.io/en/stable/parameter.html

Key Features of XGBRegressor:

1. **Boosting Algorithm**: XGBoost uses a boosting approach to combine multiple weak learners (decision trees) to create a strong model.
2. **Regularization**: It includes both L1 (Lasso) and L2 (Ridge) regularization to prevent overfitting.
3. **Parallelization**: XGBoost is designed for parallel computing, which significantly speeds up training, especially on large datasets.
4. **Handling Missing Data**: It can handle missing data internally by automatically learning what the missing values represent.
5. **Importance of Features**: XGBoost provides functionality to calculate feature importance, which is useful for interpreting the model.

How Does XGBRegressor Work?

1. Initial Prediction

- Start with a simple initial prediction for all samples (e.g., the average of the target values).
- ***Example: If house prices average = 300k, the first prediction for all houses is 300k***

---

2. Calculate Residuals

- Compute the difference between the actual target values and the current predictions (these are called **residuals**).
- Example: ***If a house actually costs 350k but the model predicted 350k, the residual is $50k.***

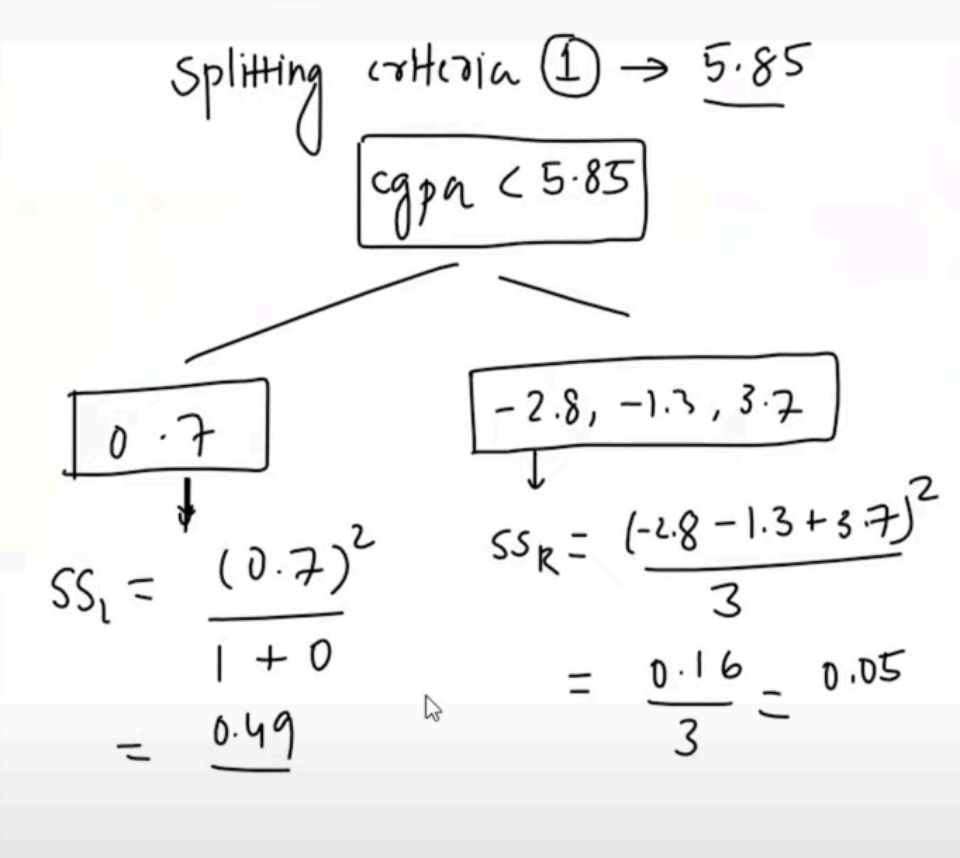
---

3. Build a Tree to Predict Residuals

- **Similarity Score:**
  - For every possible split in the tree, XGBoost calculates a **similarity score** for the node.
  - This score measures how "similar" (homogeneous) the residuals are in that node.
  - **Higher similarity = better grouping.**
  - Formula (simplified):

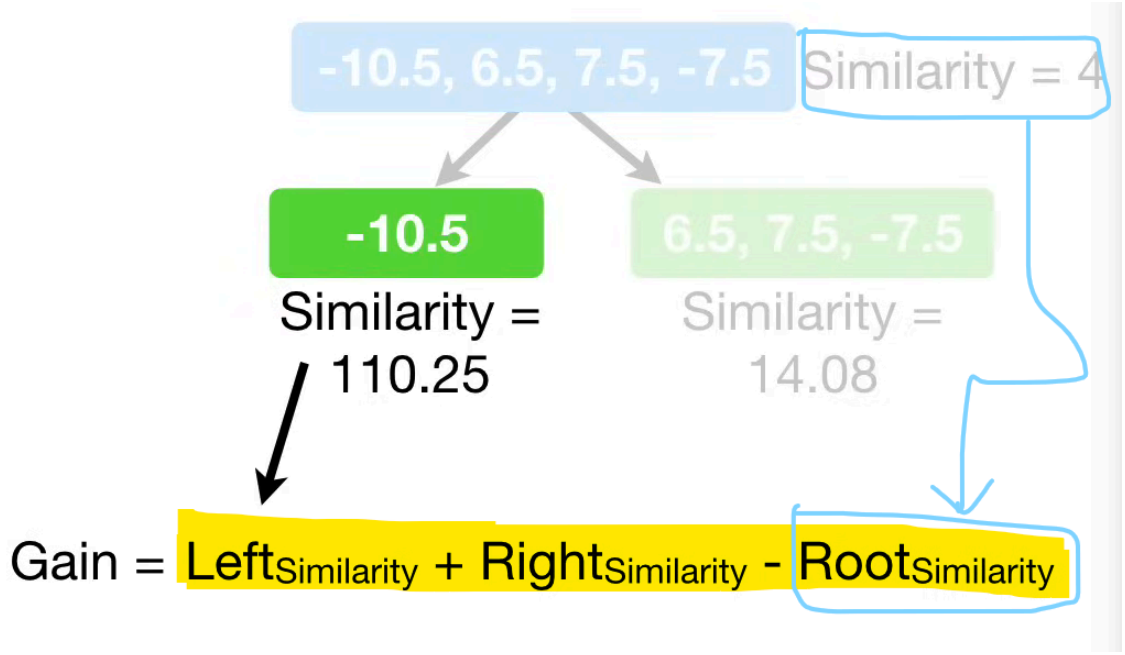
$$\text{Similarity} = \frac{(\text{Sum of residuals})^2}{\text{Number of residuals} + \lambda}$$

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

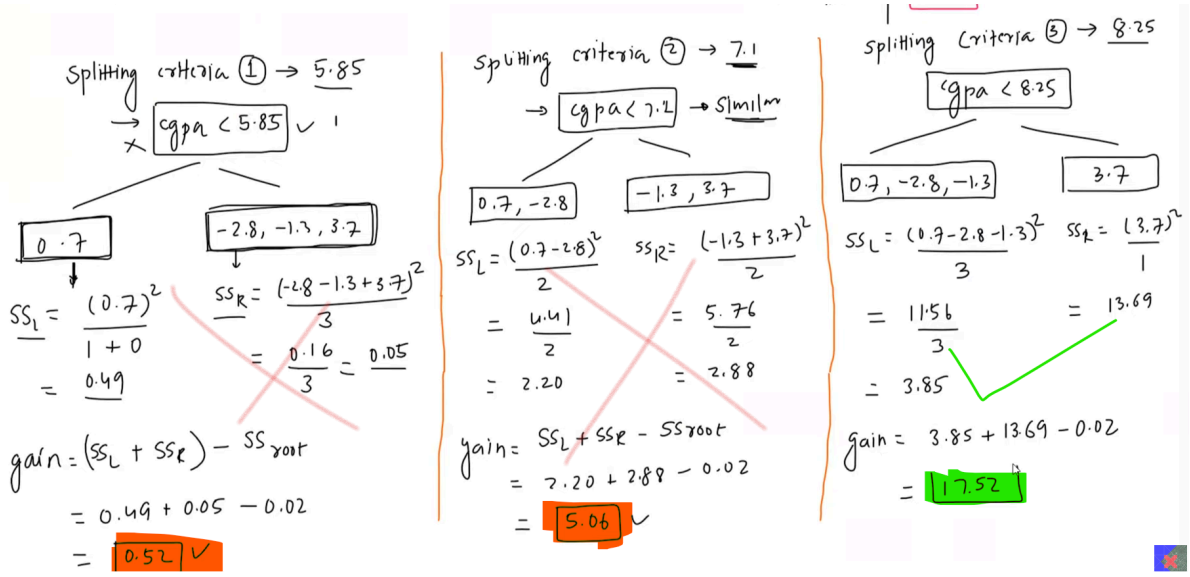


💡 **If there are -ve & +ve, they will cancel each other & SS will go down.**

- **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).



- **Calculate SS & Gain for other splits and compare them.**

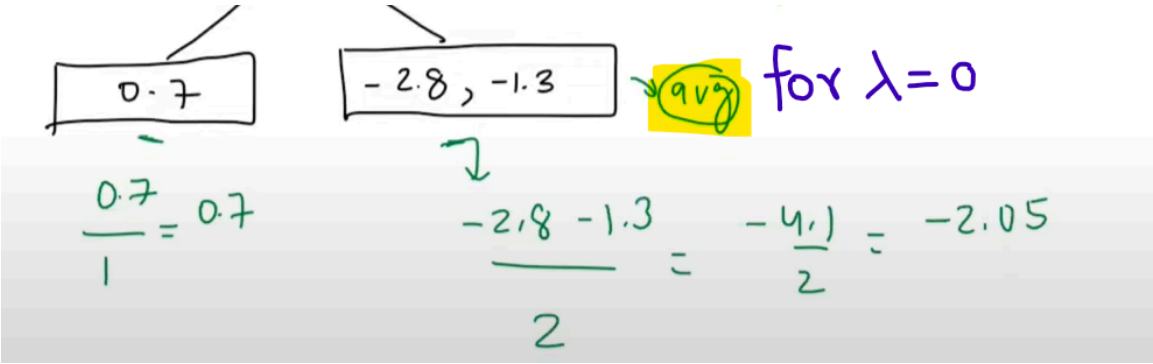


- **Stopping Condition:**
  - A split is only made if the gain exceeds a threshold (γ, the `gamma` hyperparameter).
  - This prevents overly complex trees.

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):

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



- 👆 If there are multiple values in a leaf & lambda=0 (no regularization), the ***predicted value = average***
- 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 10k and the learning rate is 0.1, the prediction increases by 1*k*
- Default `learning_rate=0.3`

6. Repeat for Multiple Trees

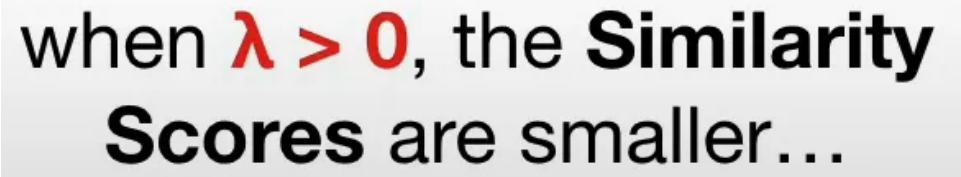
- Repeat steps 2–5 for `n_estimators` trees.

- Each new tree focuses on the residuals (errors) left by the previous trees.

---

7. Regularization

- XGBRegressor uses:
  - **Gamma (γ)**: Penalizes splits that don't improve the model enough.
    - **If Gain-Gamma= Positive → Keep the branch**
    - **If Gain-Gamma= Negative → Remove the branch**
    - **Setting Gamma=0 does not turn of pruning**
  - **Lambda (λ) (`reg_lambda`)**: Reduces the impact of individual trees (L2 regularization).



  - **λ** helps reduce overfitting
  - **Subsampling**: Uses a fraction of the data/features per tree (`subsample`, `colsample_bytree`).

Default Parameters for XGBRegressor

Parameter	Default Value	Role
`n_estimators`	`100`	Number of boosting trees. More trees improve performance but increase training time.
`learning_rate` (eta)	`0.3`	Shrinks tree contributions. Higher values (e.g., 0.3) speed up learning but risk overfitting.
`max_depth`	`6`	Maximum depth of trees. Deeper trees capture complex patterns but may overfit.
`subsample`	`1.0`	Fraction of training data used per tree. `1.0` = use all data.

`colsample_bytree`	`1.0`	Fraction of features used per tree. `1.0` = use all features.
`gamma`	`0`	Minimum loss reduction for a split. Higher values prevent splits that don't improve gains.
`reg_lambda` (lambda)	`1.0`	L2 regularization on leaf weights. Penalizes large weights to reduce overfitting.
`reg_alpha` (alpha)	`0.0`	L1 regularization on leaf weights. Encourages sparsity (rarely used by default).
`objective`	`reg:squarederror`	Loss function for regression (mean squared error).
`eval_metric`	Auto-detected	Metric for validation (e.g., `rmse` for regression).
`early_stopping_rounds`	`None`	Disabled by default. If set (e.g., `10`), stops training if validation error doesn't improve.
`random_state`	`None`	No fixed seed. Set to an integer (e.g., `42`) for reproducible results.

```
XGBRegressor(
    n_estimators=100,
    learning_rate=0.3,
    max_depth=6,
    subsample=1.0,
    colsample_bytree=1.0,
    gamma=0,
    reg_lambda=1,
    reg_alpha=0,
    objective='reg:squarederror',
    random_state=None
)
```

Important Hyperparameters of XGBRegressor:

- **`n_estimators`**: Number of boosting rounds (trees).
    - More trees can improve performance but may lead to overfitting.

    - ***100–1000***
- `learning_rate` / `eta`: Step size to shrink the contribution of each tree.
    - Lower values (e.g., 0.01, 0.1) usually give better performance but require more trees.
    - ***0.01–0.3***
- `max_depth`: Maximum depth of a tree. Controls the complexity of the model.
    - A larger depth allows the model to capture more complex patterns but may cause overfitting.
    - ***3–10***

`subsample`

- **What it does**: Controls the fraction of training data used for each tree.
- **Example**: If `subsample=0.8`, each tree is trained on 80% of the data, randomly selected.
- **Why it matters**:
    - Adds randomness to the model, making it less likely to overfit.
    - Helps the model generalize better to unseen data.
- **Default**: `1.0` (use all data).
- ***0.5–1.0***

colsample_bytree

- **What it does**: Controls the fraction of features (columns) used for each tree.
- **Example**: If `colsample_bytree=0.8`, each tree uses 80% of the features, randomly selected.
- **Why it matters**:
    - Adds randomness and reduces overfitting.
    - Helps the model focus on different subsets of features for each tree.
- **Default**: `1.0` (use all features).
- ***0.5–1.0***

- `objective`: Defines the loss function.
  - For regression, use `'reg:squarederror'`.

Regularization

Parameter	Description	Typical Values
`gamma`	- Minimum loss reduction required to split a node. - A higher value can lead to a more conservative model.	0–5
`reg_lambda` (lambda)	L2 regularization on leaf weights.	0–10
`reg_alpha` (alpha)	L1 regularization on leaf weights.	0–10

Learning Task

Parameter	Description	Example Values
`objective`	Loss function to optimize (default: `reg:squarederror`).	`reg:squaredlogerror`
`eval_metric`	Evaluation metric (e.g., `rmse`, `mae`).	`rmse`, `mae`

Parameter	Focus	Effect
`reg_lambda`	Leaf weights (L2 penalty)	Shrinks all weights, making them smaller.
`reg_alpha`	Leaf weights (L1 penalty)	Shrinks some weights to zero, creating sparsity.
`gamma`	Tree structure	Prevents splits that don't improve the model enough.

Python Code for XGBoost Regressor

```
import xgboost as xgb
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import mean_squared_error, r2_score
import pandas as pd

# Load dataset
data = fetch_california_housing()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
e=42)

# Initialize XGBRegressor
model = xgb.XGBRegressor(
    n_estimators=200,
    learning_rate=0.1,
    max_depth=5,
    subsample=0.8,
    colsample_bytree=0.8,
    gamma=1,
    early_stopping_rounds=10,
    random_state=42
)

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

# Predict
y_pred = model.predict(X_test)

# Evaluate
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"RMSE: {mse**0.5:.2f}")
print(f"R²: {r2:.2f}")
```

```
# Feature Importance
importance = pd.DataFrame({
    'Feature': data.feature_names,
    'Importance': model.feature_importances_
}).sort_values('Importance', ascending=False)

print("\nFeature Importance:")
print(importance)
```

```
RMSE: 0.47
R²: 0.83

Feature Importance:
      Feature  Importance
0      MedInc    0.502641
5    AveOccup    0.130767
2    AveRooms    0.091424
7   Longitude    0.081292
6    Latitude    0.072321
1    HouseAge    0.064989
3   AveBedrms    0.039533
4  Population    0.017033
```

For `model = xgb.XGBRegressor()` (vanilla XGBR)→ The results are almost same

```
RMSE: 0.47
R²: 0.83

Feature Importance:
      Feature  Importance
0      MedInc    0.489623
5    AveOccup    0.148580
7   Longitude    0.107952
6    Latitude    0.090290
1    HouseAge    0.070058
2    AveRooms    0.043091
3   AveBedrms    0.025678
4  Population    0.024728
```

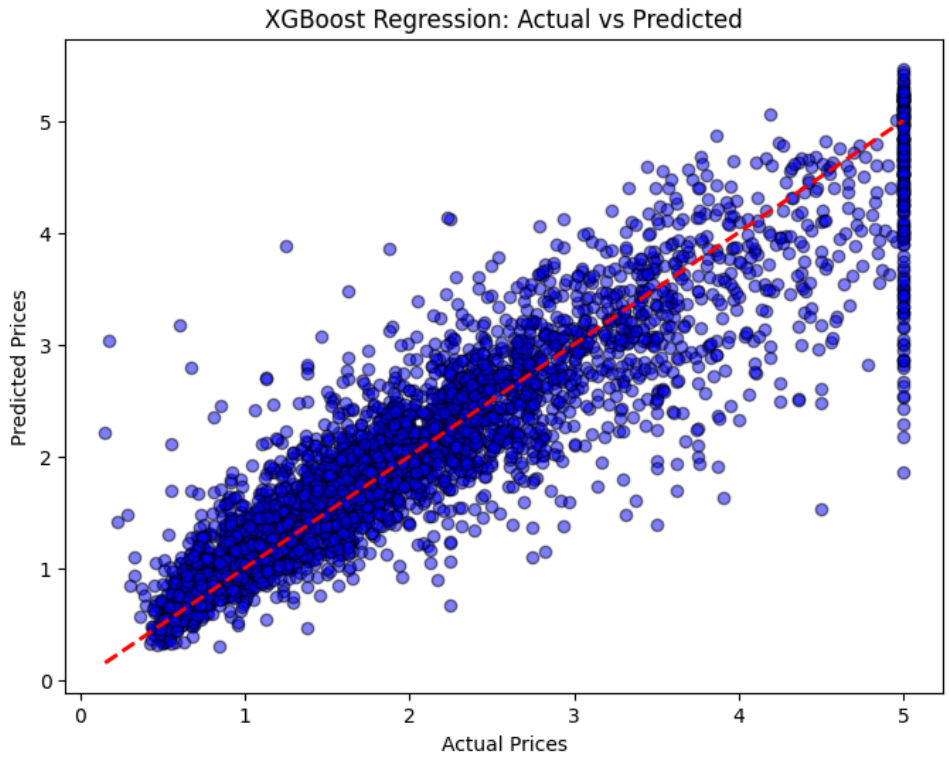
Advanced Features

a. Early Stopping

- Halts training if validation performance doesn't improve for `early_stopping_rounds`.
- Example: `early_stopping_rounds=10`.

Actual vs. Predicted values

```
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, alpha=0.5, color='blue', edgecolor='k')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("XGBoost Regression: Actual vs Predicted")
plt.show()
```



Hyperparameter Tuning using GridSearchCV

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'subsample': [0.7, 0.8, 1.0],
    'colsample_bytree': [0.7, 0.8, 1.0],
    'reg_alpha': [0, 0.1, 1],
    'reg_lambda': [1, 1.5, 2]
}

grid_search = GridSearchCV(estimator=xgb.XGBRegressor(objective='reg:squarederror', random_state=42),
                           param_grid=param_grid, cv=3, scoring='r2', n_jobs=-1, verbose=1)
grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)
best_model = grid_search.best_estimator_
y_pred_best = best_model.predict(X_test)
print("Optimized R² Score:", r2_score(y_test, y_pred_best))
```

Fitting 3 folds for each of 1458 candidates, totalling 4374 fits

Best Parameters:
{'colsample_bytree': 0.7, 'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200, 'reg_alpha': 1, 'reg_lambda': 1, 'subsample': 1.0}

Optimized R^2 Score:
0.8491272055978294

> 💡 **NOTE: 👆Took 4 min, 40 sec.**
>
> **Used 100% CPU**

When to Use XGBRegressor?

- Large datasets with mixed data types.
- High-dimensional feature spaces.
- Competitions (e.g., Kaggle) and production systems requiring speed/accuracy.

Advantages of XGBRegressor:

- **Accuracy**: It often performs very well on regression tasks.
- **Handling Non-linearity**: Since it's based on decision trees, it can handle non-linear relationships between features and the target.
- **Regularization**: Helps prevent overfitting through L1 and L2 regularization.
- **Parallelization**: Speeds up training by using multiple CPU cores.
- **Customizable**: Many hyperparameters to tune for optimal performance.

XGBoost vs Gradient Boost

Feature	Gradient Boosting	XGBoost
Speed	Slower, as it's not optimized for performance.	Faster, due to parallel processing and hardware optimization.
Regularization	No built-in regularization (prone to overfitting).	Built-in L1 and L2 regularization to prevent overfitting.

Feature	Gradient Boosting	XGBoost
Handling Missing Values	Requires manual handling of missing data.	Automatically handles missing values during training.
Tree Pruning	Grows trees fully, then prunes (if implemented manually).	Prunes trees during growth (depth-first) for efficiency.
Cross-Validation	Requires external tools (e.g., `GridSearchCV`).	Built-in cross-validation during training.
Early Stopping	Not natively supported.	Supports early stopping to halt training if no improvement.
Flexibility	Limited to basic gradient boosting.	Supports custom objectives, evaluation metrics, and more.
Scalability	Less scalable for large datasets.	Highly scalable, designed for big data.

1. `reg_lambda` **and** `reg_alpha`:
    - Both control the **size of leaf weights** (how much each leaf contributes to the prediction).
    - `reg_lambda` shrinks all weights, while `reg_alpha` shrinks some weights to zero.
    - Example: If a leaf weight is too large, `reg_lambda` reduces it, and `reg_alpha` might set it to zero if it's not important.
2. `gamma`:
    - Controls **whether a split is made** in the first place.
    - If a split doesn't improve the model enough (based on the similarity score), it's skipped.
    - Example: If `gamma=1`, a split must improve the model by at least 1 unit to be allowed

Parameter	Focus	Effect
`reg_lambda`	Leaf weights (L2 penalty)	Shrinks all weights, making them smaller.

`reg_alpha`	Leaf weights (L1 penalty)	Shrinks some weights to zero, creating sparsity.
`gamma`	Tree structure	Prevents splits that don’t improve the model enough.