**Stuffs to know before XG boosting**

## **Supervised Classification**

Learn from labeled data to predict labels for new data. A function approximation method where input-output pairs are used to train a model to map inputs to discrete output classes

**Working Mechanism**

1. Collect labeled data
2. Split into training and test sets
3. Train model using algorithms (e.g., decision trees, SVM)
4. Validate on test set
5. Predict unseen data

**Applications**

* Email spam detection
* Medical diagnosis (e.g., cancer detection)
* Credit scoring
* Image recognition (e.g., handwritten digits)

**Features**

* Requires labeled data
* Output is categorical
* Evaluation via accuracy, precision, recall, F1-score

**Pros and Cons**

* High accuracy with good data
* Clear performance metrics − Needs labeled data − Can overfit if not regularized

**Analogy** Like a teacher grading papers based on a rubric learned from past examples

**Importance** Core of many real-world ML systems where decisions must be categorical and explainable

**Critical Nuances**

* Data imbalance skews performance
* Feature selection impacts accuracy
* Evaluation must go beyond accuracy (e.g., precision-recall tradeoff)

**Future Scope**

* Better handling of noisy labels
* Semi-supervised hybrids
* More interpretable models

**Comparison** vs Unsupervised: supervised uses labels vs Regression: classification predicts categories, regression predicts continuous values

**Example Scenario** Predict whether a loan applicant will default based on income, age, credit score

**Python Snippet**

from sklearn.tree import DecisionTreeClassifier

X = [[25, 50000], [40, 100000]]

y = [0, 1]

model = DecisionTreeClassifier()

model.fit(X, y)

print(model.predict([[30, 60000]]))

**Pseudocode**

Input: labeled data (X, y)

Train classifier on X, y

Predict label for new input

## **Decision Trees**

A flowchart-like structure that splits data based on feature values. A recursive partitioning algorithm that builds a tree where each node represents a decision rule and leaves represent class labels

**Working Mechanism**

1. Select best feature to split (using Gini or entropy)
2. Split data into subsets
3. Repeat recursively until stopping criteria met
4. Assign class labels to leaves

**Applications**

* Loan approval
* Fraud detection
* Customer churn prediction
* Medical triage systems

**Features**

* Hierarchical structure
* Handles both numerical and categorical data
* Easy to interpret
* Prone to overfitting

**Pros and Cons**

* Transparent decisions
* No need for feature scaling − Overfits easily − Unstable with small data changes

**Analogy** Like playing 20 questions to guess an object by narrowing down possibilities

**Importance** Used in many domains due to interpretability and simplicity

**Critical Nuances**

* Depth control is crucial
* Pruning improves generalization
* Feature importance can be extracted

**Future Scope**

* Integration with ensemble methods
* More robust splitting criteria
* Hybrid models with neural nets

**Comparison** vs Random Forest: forest uses multiple trees vs SVM: SVM is margin-based, tree is rule-based

**Example Scenario** Predict if a customer will buy a product based on age and income

**Python Snippet**

from sklearn import tree

clf = tree.DecisionTreeClassifier()

clf.fit([[20, 30000], [40, 90000]], [0, 1])

print(clf.predict([[30, 50000]]))

**Pseudocode**

Start with full dataset

While stopping criteria not met:

Choose best feature to split

Partition data

Assign labels to leaf nodes

## **Boosting**

Combine weak learners to form a strong learner An ensemble technique that sequentially trains models, each correcting errors of the previous one

**Working Mechanism**

1. Train weak learner (e.g., shallow tree)
2. Identify misclassified samples
3. Increase their weights
4. Train next learner on weighted data
5. Repeat and combine all learners

**Applications**

* Credit scoring
* Face detection
* Text classification
* Predictive maintenance

**Features**

* Sequential model training
* Focuses on hard examples
* Often uses decision trees
* High accuracy, low bias

**Pros and Cons**

* High performance
* Reduces bias − Sensitive to noise − Longer training time

**Analogy** Like a team of tutors, each focusing on what the student didn’t understand last time

**Importance** Boosting algorithms like XGBoost and AdaBoost dominate structured data competitions

**Critical Nuances**

* Learning rate controls contribution of each learner
* Overfitting possible with too many rounds
* Needs careful tuning

**Future Scope**

* Faster variants (e.g., LightGBM)
* Integration with deep learning
* Better interpretability tools

**Comparison** vs Bagging: boosting is sequential, bagging is parallel vs Stacking: stacking combines different models, boosting uses same base learner

**Example Scenario** Predict customer churn using multiple weak decision trees that learn from each other’s mistakes

**Python Snippet**

from sklearn.ensemble import AdaBoostClassifier

from sklearn.tree import DecisionTreeClassifier

model = AdaBoostClassifier(base\_estimator=DecisionTreeClassifier(max\_depth=1), n\_estimators=50)

model.fit([[25, 1], [40, 0]], [1, 0])

print(model.predict([[30, 1]]))

**Pseudocode**

Initialize weights

For each round:

Train weak learner

Update weights based on errors

Combine learners for final prediction

## **Classification and Its Types**

**Classification** is a supervised learning task where the goal is to assign labels to input data based on learned patterns.

### Binary Classification

Two possible classes. Example: cat vs not cat, disease vs no disease.

### Multiclass Classification

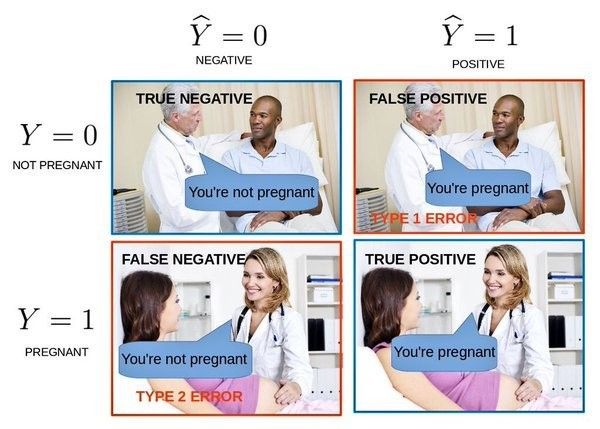
More than two classes. Example: classifying handwritten digits (0–9), types of fruits.

## Accuracy Evaluation Metrics

### Confusion Matrix

A 2x2 table for binary classification:

|  | Predicted Positive | Predicted Negative |
| --- | --- | --- |
| Actual Positive | True Positive (TP) | False Negative (FN) |
| Actual Negative | False Positive (FP) | True Negative (TN) |



**Example**: Predicting if someone is pregnant.

* Actual: Man (cannot be pregnant)
* Prediction: Pregnant → **False Positive**
* Actual: Woman, predicted not pregnant → **False Negative**

**Formulae**:

* Accuracy = (TP + TN) / (TP + TN + FP + FN)
* Precision = TP / (TP + FP)
* Recall = TP / (TP + FN)
* F1 Score = 2 \* (Precision \* Recall) / (Precision + Recall)

### AUC-ROC

* ROC = Receiver Operating Characteristic
* AUC = Area Under Curve
* Measures how well the model separates classes
* AUC of 1 = perfect, 0.5 = random guessing

For multiclass, confusion matrix becomes an N×N grid. AUC can be averaged across classes (macro/micro averaging).

## Data Format in Supervised Learning

Yes, it requires **feature vectors**. Each sample must be represented as a fixed-length vector of features. Why?

* Algorithms like decision trees, SVMs, neural nets operate on numerical matrices
* Enables mathematical operations like dot products, distance calculations

## Encoding Categorical Variables

Categorical data must be converted to numeric form.

### Common Methods:

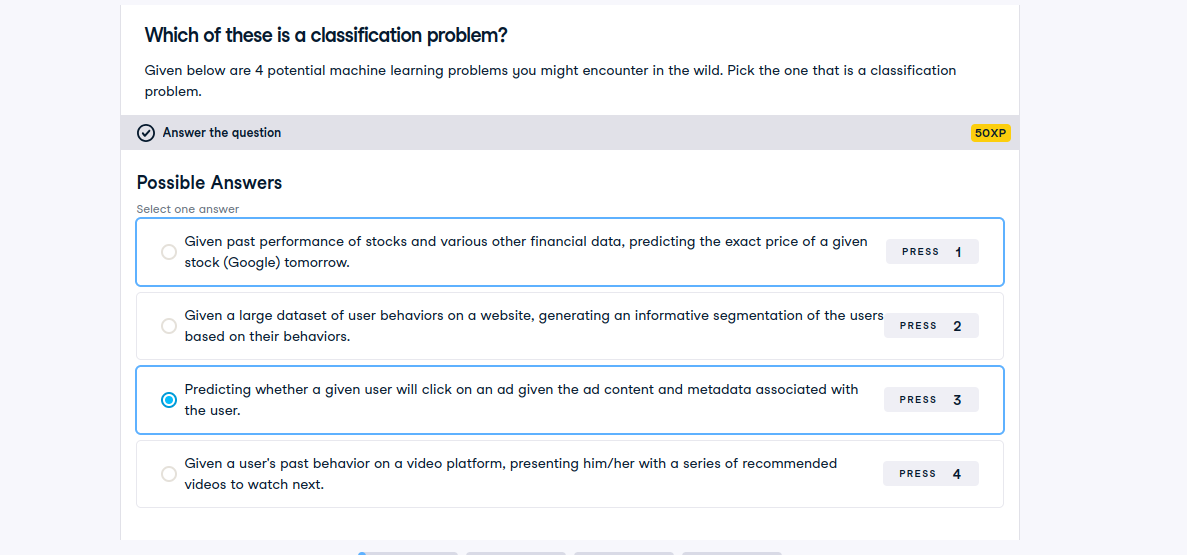
* **Label Encoding**: Assigns integer to each category (e.g., red=0, blue=1)
* **One-Hot Encoding**: Creates binary columns for each category (e.g., red → [1,0,0], blue → [0,1,0])
* **Target Encoding**: Replaces category with mean of target variable (used in boosting)

## Z-Score and Feature Scaling

**Z-Score** = (value − mean) / standard deviation It standardizes features to have mean 0 and std 1.

Why scale numeric features?

* Algorithms like logistic regression, SVM, k-NN are sensitive to feature magnitude
* Prevents dominant features from skewing results
* Helps in faster convergence during training



**XG boosting**

XGBoost (Extreme Gradient Boosting) is an optimized implementation of gradient boosting that uses decision trees as base learners. It minimizes a loss function using gradient descent and adds new trees sequentially to correct previous errors.

## Working Mechanism

1. Start with initial predictions (usually mean for regression, log odds for classification)
2. Compute residuals (errors between actual and predicted)
3. Fit a small decision tree to these residuals
4. Update predictions by adding the tree’s output scaled by a learning rate
5. Repeat steps 2–4 for a fixed number of rounds or until convergence
6. Final prediction is the sum of all trees’ outputs

Each tree focuses on the mistakes made by the previous ones.

## Real-World Applications

* Credit scoring and fraud detection in banking
* Predicting customer churn in telecom
* Diagnosing diseases from medical data
* Ranking search results in information retrieval
* Forecasting sales or demand in retail
* Classifying images or text in competitions

## Features and Characteristics

* Uses decision trees as base learners
* Supports regularization (L1 and L2) to prevent overfitting
* Handles missing values internally
* Parallelized tree construction for speed
* Supports early stopping
* Works with sparse data
* Compatible with classification, regression, ranking

## Pros and Cons

**Pros**

* High accuracy on structured/tabular data
* Fast training due to parallelization
* Built-in regularization
* Handles missing data
* Highly customizable

**Cons**

* Complex tuning (many hyperparameters)
* Less interpretable than simpler models
* Can overfit if not regularized
* Not ideal for unstructured data like images or audio

## Analogies

Think of XGBoost like a team of tutors helping a student. Each tutor focuses on what the student didn’t understand from the previous session. Over time, the student gets better because each tutor builds on the last.

## Importance and Relevance

XGBoost is one of the most widely used algorithms in data science competitions and industry applications. It consistently delivers top performance on structured data problems. Its speed, flexibility, and accuracy make it a go-to choice for practitioners.

## Critical Details or Nuances

* Learning rate controls how much each tree contributes. Lower is safer but slower.
* Tree depth affects model complexity. Shallow trees reduce overfitting.
* Column sampling (subsample) helps reduce correlation between trees.
* Objective functions can be customized (e.g., logistic, squared error)
* Feature importance can be extracted but is not always reliable

## Future Scope

* Integration with deep learning models for hybrid architectures
* Better interpretability tools (e.g., SHAP values)
* More efficient GPU-based training
* AutoML frameworks using XGBoost as core engine

## Comparison

**vs Random Forest**

* RF builds trees independently; XGBoost builds sequentially
* RF reduces variance; XGBoost reduces bias
* RF is easier to tune; XGBoost is more powerful but complex

**vs LightGBM**

* LightGBM uses histogram-based splits and leaf-wise growth
* XGBoost uses level-wise growth
* LightGBM is faster on large datasets but can overfit more easily

**vs Neural Networks**

* XGBoost excels on tabular data
* Neural nets dominate unstructured data (images, text)

## Example Scenario

A telecom company wants to predict customer churn. Features: age, contract type, monthly charges, tenure XGBoost trains on historical data to learn patterns of churn It builds trees that focus on misclassified customers Final model predicts churn probability for new customers Company uses this to target retention offers

**XGBoost implementatino**

1. inport the libraries and function to use all including pandas numpy train-test-split and xgboost we need to split our data into train test split to test on our data to ensure no overfitting has occurred and can generalize unseen data

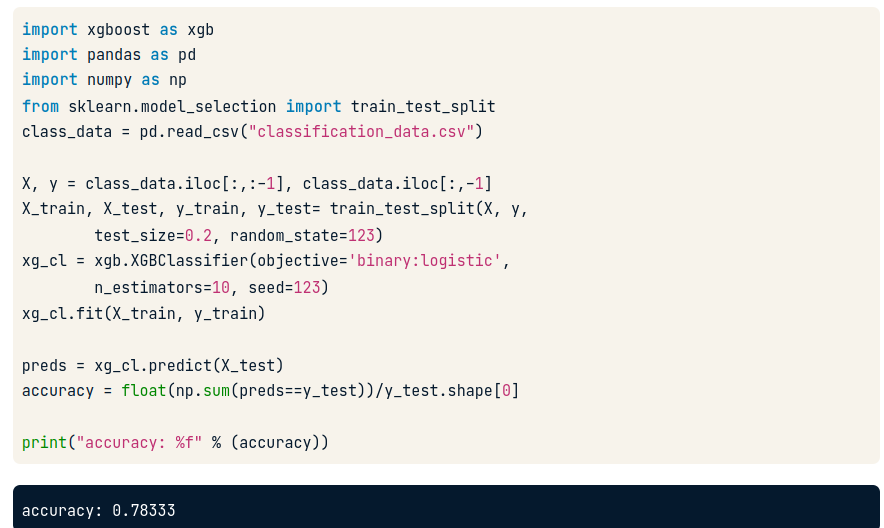
2. load our data in from file and split the entire dataset into matrix of samples by features ‘x’ and vector of target values ‘y’

3. splitting data for training and testing

4/ instantiate xgboost classifier instance with some parameters

5. fittign and generating model and training model for evaluating by generating test set comparing the output with target on test set

6. predicting accuracy



**Code in action**

# Import xgboost

import xgboost as xgb

# Create arrays for the features and the target: X, y

X, y = churn\_data.iloc[:,:-1], churn\_data.iloc[:,-1]

# Create the training and test sets

X\_train, X\_test, y\_train, y\_test= train\_test\_split(X,y, test\_size=0.2, random\_state=123)

# Instantiate the XGBClassifier: xg\_cl

xg\_cl = xgb.XGBClassifier(objective='binary:logistic', n\_estimators=10, seed=123)

# Fit the classifier to the training set

xg\_cl.fit(X\_train,y\_train)

# Predict the labels of the test set: preds

preds = xg\_cl.predict(X\_test)

# Compute the accuracy: accuracy

accuracy = float(np.sum(preds==y\_test))/y\_test.shape[0]

print("accuracy: %f" % (accuracy))

**Decision Tree**

It’s a recursive partitioning algorithm that builds a tree where each internal node represents a test on a feature, each branch represents an outcome of the test, and each leaf node represents a class label or prediction.

## Working Mechanism

1. Start with the full dataset
2. Choose the best feature to split on (using criteria like Gini impurity or entropy)
3. Split the data into subsets based on feature values
4. Repeat recursively for each subset
5. Stop when a stopping condition is met (e.g., max depth, minimum samples per leaf)
6. Assign class labels to leaf nodes

## Real-World Applications

* Loan approval systems
* Medical diagnosis (e.g., predicting disease based on symptoms)
* Fraud detection
* Customer churn prediction
* HR systems for candidate screening

## Features and Characteristics

* Hierarchical structure
* Handles both numerical and categorical data
* Easy to interpret
* Can be visualized
* Prone to overfitting without pruning

## Pros and Cons

**Pros**

* Transparent and interpretable
* No need for feature scaling
* Works with mixed data types

**Cons**

* Sensitive to small data changes
* Overfits easily
* Can be biased toward features with more levels

## Analogy

Think of it like playing “20 Questions.” Each question narrows down the possibilities until you reach a final answer.

## Importance and Relevance

Decision trees are foundational in machine learning. They’re used directly or as base learners in powerful ensemble methods like Random Forests and XGBoost. Their interpretability makes them valuable in regulated industries like finance and healthcare.

## Critical Details or Nuances

* **Gini vs Entropy**: Both measure impurity; Gini is faster, entropy is more informative
* **Pruning**: Reduces overfitting by trimming branches that add little value
* **Feature importance**: Trees can rank features by how often they’re used for splits
* **Handling missing values**: Some implementations can split based on surrogate splits

## Future Scope

* Integration with explainable AI tools
* Hybrid models combining trees with neural networks
* More robust splitting criteria for fairness and bias reduction

## Comparison

**vs Logistic Regression**

* Trees are non-linear, logistic regression is linear
* Trees are easier to interpret but less stable

**vs Random Forest**

* Forests use multiple trees to reduce variance
* Trees are faster but less accurate alone

## Example Scenario

A bank wants to predict loan default. Features: age, income, credit score Tree splits first on credit score, then income, then age Each leaf node gives a prediction: default or not Easy to explain to regulators and customers

**Overfitting**:  
The model has **low training error** but **high test/validation error**. It captures noise and irrelevant details, leading to poor generalization. Often caused by high model complexity (e.g., deep trees, too many features).

* **Underfitting**:  
  The model has **high training error** (and usually high test error too). It fails to capture the underlying trend due to insufficient capacity (e.g., linear model for nonlinear data).
* **Bias**:  
  The error due to **incorrect assumptions** in the learning algorithm. Mathematically:  
  Bias = E[ŷ] − y\_true  
  High bias → systematic error.
* **Variance**:  
  The error due to **sensitivity to fluctuations** in the training set. Mathematically:  
  Variance = E[(ŷ − E[ŷ])²]  
  High variance → model changes a lot with different training samples.

**Total Error ≈ Bias² + Variance + Irreducible Error**

### ****Step 1: Draw Multiple Bootstrap Samples from the Training Data****

### What is Bootstrap Sampling?

* **Bootstrap sampling** means: randomly pick samples **with replacement** from your original dataset.
* Each bootstrap sample is the **same size** as the original dataset, but some rows appear multiple times, and others are left out.

#### Layman Example:

Imagine you have a bag of 5 colored marbles:  
🔴 Red, 🔵 Blue, 🟢 Green, 🟡 Yellow, 🟣 Purple.

You close your eyes and draw **5 marbles one by one**, but **put each one back** before drawing the next (that’s “with replacement”).

One possible bootstrap sample:  
🔴, 🔴, 🟢, 🟡, 🔵  
→ Red appears twice; Purple is missing!

You do this **many times** (e.g., 100 times) to create 100 different “bags” of marbles.

#### In ML:

* Original dataset: 1,000 rows.
* Bootstrap sample 1: randomly pick 1,000 rows with replacement → some rows repeated, ~37% left out (called **out-of-bag** samples).
* Repeat to get, say, 100 such samples.

### 🌳 ****Step 2: For Each Sample, Build a Decision Tree****

* For each bootstrap sample, train a **full decision tree** (usually without pruning).
* But—here’s the twist:

### ****At Each Split, Randomly Select a Subset of Features****

* When deciding how to split a node (e.g., “Is age > 30?”), **don’t consider all features**.
* Instead, randomly pick **√p features** (for classification) or **p/3 features** (for regression), where p = total number of features.

#### Example:

Suppose you’re predicting if someone will like a movie, using:

* Age
* Gender
* Genre
* Runtime
* Director

That’s **5 features** (p = 5).  
At each split, Random Forest might randomly pick **only 2 features** (e.g., Genre and Director) to consider for the best split.

This **decorrelates the trees**—so they don’t all make the same mistakes!

### ****Step 3: Each Tree Makes a Prediction****

* Once all trees are built, you feed a new data point (e.g., a new customer) to **every tree**.
* Each tree independently predicts the outcome.

Example:

* Tree 1 → “Will like the movie” ✅
* Tree 2 → “Will not like” ❌
* Tree 3 → “Will like” ✅
* ... and so on for 100 trees.

### ****Step 4: Combine Predictions****

#### 🔹 ****For Classification****: ****Majority Vote****

* Count how many trees said “Yes” vs. “No”.
* Whichever gets **more votes** wins.

In our example: 65 trees say “Yes”, 35 say “No” → Final prediction = **“Will like”**

#### 🔸 ****For Regression****: ****Average the Predictions****

* Each tree predicts a number (e.g., house price: $300K, $320K, $290K…).
* Final prediction = **average** of all tree predictions.

Example: (300 + 320 + 290 + ...)/100 = **$305,000**

### ✅ ****Why This Works So Well****

1. **Bootstrap sampling** → reduces variance (trees see slightly different data).
2. **Random feature selection** → makes trees diverse (uncorrelated errors).
3. **Averaging/voting** → cancels out individual tree mistakes.
4. Result: **lower overfitting**, **better generalization**, and **robust performance**.

🌲 Random Forest = Wisdom of the Crowd + Controlled Randomness

## **🔹 Decision Trees as Base Learners in Ensemble Methods**

In ensemble methods like **Gradient Boosting** or **Random Forests**, **decision trees** are commonly used as **base learners**. These are typically **weak learners**, meaning they perform just slightly better than random guessing. The idea is to combine many such weak learners to build a strong predictive model.

### How Decision Trees Work in Ensembles

* **Feature Selection**: At each node, the algorithm evaluates all features and selects the one that best splits the data. The criterion could be:
  + **Gini impurity** (used in classification)
  + **Entropy** (information gain)
  + **Mean squared error** (for regression)
* **Split Point Selection**: For numerical features, the algorithm finds a threshold that best separates the data. For categorical features, it may group categories or use one-hot encoding depending on implementation.
* **Leaf Nodes**: A leaf is created when:
  + The data is pure (i.e., all samples belong to one class).
  + A stopping criterion is met (e.g., max depth, min samples per leaf).
  + In boosting, trees are often shallow (depth 3–5), so leaves may not be pure but still represent useful corrections.

## 🔹 Variance and Overfitting in Decision Trees

* **High Variance Learners**: Decision trees are **high variance** models. They can learn complex relationships but are prone to **overfitting**—memorizing training data and failing to generalize to unseen data.
* **Low Bias**: They can fit the training data very well (low bias), but this comes at the cost of poor generalization.

### Why Ensembles Help

* **Bagging (e.g., Random Forests)** reduces variance by averaging predictions from many trees trained on different subsets of data.
* **Boosting (e.g., XGBoost)** reduces bias by sequentially training trees to correct the errors of previous ones.

## 🔹 XGBoost and CART Trees

* **CART (Classification and Regression Trees)** is the tree algorithm used in XGBoost.
* **Real-Valued Splits**: CART handles continuous features by finding optimal thresholds (e.g., split on feature X at value 3.7).
* **Categorical Features**: These can be:
  + Converted to numerical via encoding (e.g., one-hot, label encoding).
  + Handled natively in some implementations (e.g., CatBoost, LightGBM).

### XGBoost Specifics

* Builds trees **sequentially**, each correcting the residuals of the previous.
* Uses **regularization** to penalize complex trees and reduce overfitting.

Make a simple decision tree using scikit-learn's DecisionTreeClassifier on the breast cancer dataset that comes pre-loaded with scikit-learn.

This dataset contains numeric measurements of various dimensions of individual tumors (such as perimeter and texture) from breast biopsies and a single outcome value (the tumor is either malignant, or benign).

We've preloaded the dataset of samples (measurements) into X and the target values per tumor into y. Now, you have to split the complete dataset into training and testing sets, and then train a DecisionTreeClassifier. You'll specify a parameter called max\_depth. Many other parameters can be modified within this model,

**example of the code of simple decision tree with scikit learn for demostrartion**

# Import necessary modules

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier

# Assume you have a dataset with features X and target y

# Split the data: 80% training, 20% testing

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=123)

# Instantiate the DecisionTreeClassifier with max\_depth=4

dt\_clf\_4 = DecisionTreeClassifier(max\_depth=4, random\_state=123)

# Fit the classifier to the training data

dt\_clf\_4.fit(X\_train, y\_train)

# Predict labels for the test set

y\_pred\_4 = dt\_clf\_4.predict(X\_test)

accuracy = float(np.sum(y\_pred\_4 == y\_test)) / y\_test.shape[0]

**Boosting**

**Boosting** is an **ensemble learning technique** that combines multiple **weak learners** (models that perform slightly better than random guessing) to form a **strong learner**. The goal is to reduce **bias and variance** by sequentially training models that correct the errors of their predecessors.

### Key Principles:

* **Sequential Learning**: Models are trained one after another, each trying to fix the mistakes of the previous.
* **Weighted Data**: Misclassified examples are given more weight so that future models focus on them.
* **Final Prediction**: The ensemble combines all weak learners, often by weighted voting or summing their outputs.

## 🔹 Components of Boosting

| Component | Description |
| --- | --- |
| Weak Learner | Typically shallow decision trees (e.g., stumps with depth=1 or 2) |
| Loss Function | Measures how far predictions are from actual values (e.g., log loss, MSE) |
| Weight Update | Adjusts sample weights or gradients to emphasize hard-to-predict examples |
| Model Aggregation | Combines predictions from all learners (e.g., weighted sum or majority vote) |

## 🔹 Types of Boosting Algorithms

| Algorithm | Description |
| --- | --- |
| AdaBoost | Adjusts sample weights based on errors; uses exponential loss |
| Gradient Boosting | Uses gradients of a loss function to guide model updates |
| XGBoost | Optimized version of gradient boosting with regularization and speed |
| LightGBM | Uses histogram-based splits and leaf-wise growth for speed and accuracy |
| CatBoost | Handles categorical features natively and combats prediction shift |

## 🔹 XGBoost: eXtreme Gradient Boosting

XGBoost is a **high-performance implementation** of gradient boosting. It builds an ensemble of **CART decision trees** (Classification and Regression Trees), each trained to correct the residuals (errors) of the previous tree.

### How It Works:

1. **Start with a base prediction** (e.g., mean of target values).
2. **Compute residuals**: Difference between actual and predicted values.
3. **Train a new tree** to predict these residuals.
4. **Update the prediction**: Add the new tree’s output to the previous prediction.
5. **Repeat** until a stopping criterion is met (e.g., number of trees, minimal gain).

### Mathematical Form:

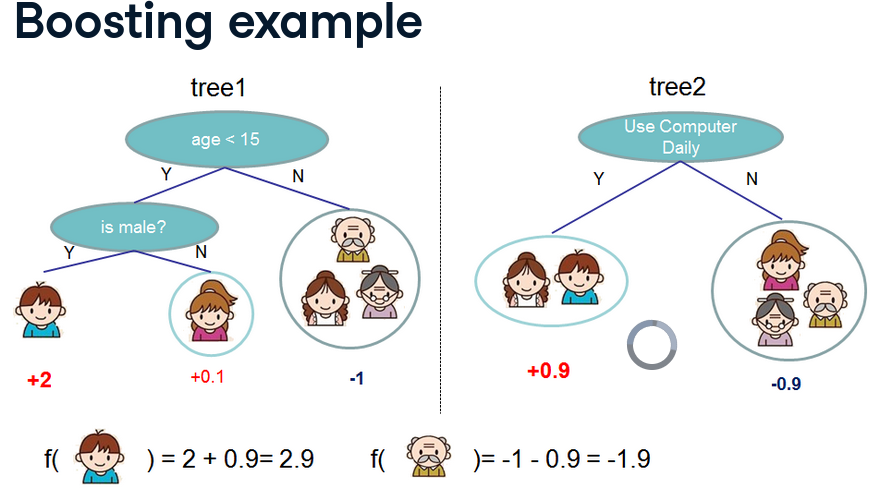


Where:

* y is the final prediction for instance i
* fk(xi) is the prediction from the kth tree

### Features of XGBoost:

* **Regularization** (L1 and L2) to prevent overfitting
* **Parallel processing** for speed
* **Handling missing values**
* **Custom loss functions**

The figure shows two decision trees used in a boosting framework:

### Tree 1:

* Splits on age < 15
  + If **yes**, checks is male?
    - If **yes**, predicts **+2**
    - If **no**, predicts **+0.1**
  + If **no**, predicts **−1**

### Tree 2:

* Splits on Use Computer Daily
  + If **yes**, predicts **+0.9**
  + If **no**, predicts **−0.9**

### Combined Prediction:

Boosting **adds** the outputs of both trees:

* For a person who is male and uses a computer daily: → Tree1: +2, Tree2: +0.9 → **Final: 2.9**
* For a person who is older than 15 and doesn’t use a computer: → Tree1: −1, Tree2: −0.9 → **Final: −1.9**

This illustrates how boosting **aggregates weak learners** to make stronger predictions.

## **🔹 What Is Cross-Validation?**

Cross-validation is a technique to evaluate a model’s generalization ability. cross validation is robust method to estimate expected profermance on unseen data generating nono overlapping train test split on your training data and reporing avg test set performance across all data splites. WInstead of training on one fixed train-test split, it:

* **Splits the training data into K folds** (non-overlapping subsets)
* **Trains the model on K−1 folds** and **tests on the remaining fold**
* **Repeats this process K times**, each time using a different fold as the test set
* **Averages the performance** across all K test sets to get a reliable estimate

This reduces the risk of overfitting to a particular train-test split and gives a more **stable estimate** of how the model will perform on unseen data.

## 🔹 XGBoost Native Learning API (xgb.cv)

The **native API** provides a built-in method for cross-validation: xgb.cv. It works directly with **DMatrix**, XGBoost’s optimized data structure.

## 🔹 Two Ways to Use XGBoost

### 1. ****Scikit-learn-Compatible API****

* Uses XGBClassifier or XGBRegressor
* Integrates with cross\_val\_score, GridSearchCV, etc.
* Familiar syntax for sklearn users

### 2. ****Native XGBoost Learning API****

* Uses xgb.train and xgb.cv
* Requires DMatrix format
* Offers **built-in cross-validation**, early stopping, and fine-grained control

| Feature | Native API (xgb.cv) | Sklearn API (XGBClassifier) |
| --- | --- | --- |
| Data Format | Requires DMatrix | Uses NumPy arrays or Pandas DataFrames |
| Cross-Validation | Built-in via xgb.cv | Uses cross\_val\_score, GridSearchCV |
| Early Stopping | Directly supported | Requires manual setup with eval\_set |
| Performance Reporting | Detailed per-round metrics | Only final fold scores |
| Flexibility | More control over boosting internals | Easier integration with sklearn tools |

## 🔹 Why Use Native API for Cross Validation?

* **Precise control** over boosting rounds and evaluation metrics
* **Efficient training** with internal optimizations
* **Early stopping** built-in, saving time and preventing overfitting
* **Detailed diagnostics** for each round of boosting

**Steps for Cross validation**

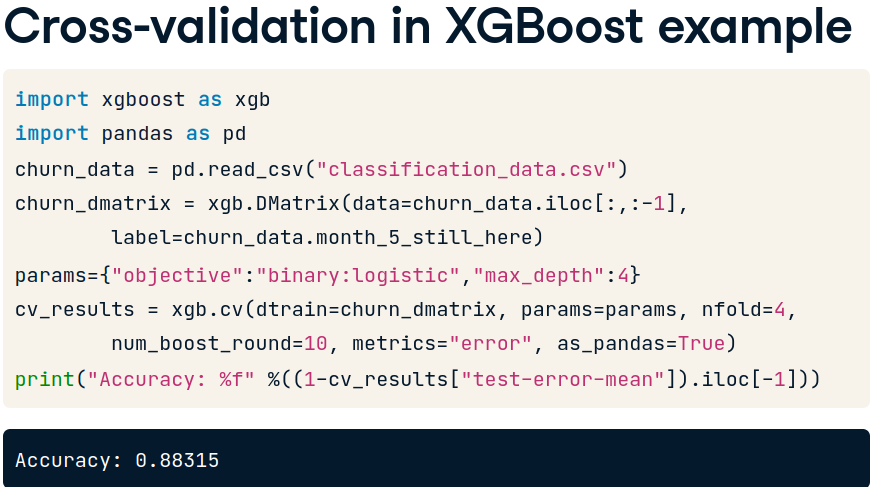
1. inport the shit

2. load the example datasets

3. convert dataset to the optimized datastructure that the creators of xgboost made called Dmatrix conversion explicit to d matrix is necessary

4. parameter dictionary to pass into our CV

5. call CV method and pass all the object and things created



Create a DMatrix called churn\_dmatrix from churn\_data using xgb.DMatrix(). The features are available in X and the labels in y.

* Perform 3-fold cross-validation by calling xgb.cv(). dtrain is your churn\_dmatrix, params is your parameter dictionary, nfold is the number of cross-validation folds (3), num\_boost\_round is the number of trees we want to build (5), metrics is the metric you want to compute (this will be "error", which we will convert to an accuracy).

# Create arrays for the features and the target: X, y

X, y = churn\_data.iloc[:,:-1], churn\_data.iloc[:,-1]

# Create the DMatrix from X and y: churn\_dmatrix

churn\_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params

params = {"objective":"reg:logistic", "max\_depth":3}

# Perform cross-validation: cv\_results

cv\_results = xgb.cv(dtrain=churn\_dmatrix, params=params,

nfold=3, num\_boost\_round=5,

metrics="error", as\_pandas=True, seed=123)

# Print cv\_results

print(cv\_results)

# Print the accuracy

print(((1-cv\_results["test-error-mean"]).iloc[-1]))

cv\_results stores the training and test mean and standard deviation of the error per boosting round (tree built) as a DataFrame. From cv\_results, the final round 'test-error-mean' is extracted and converted into an accuracy, where accuracy is 1-error. The final accuracy of around 75% is an improvement from earlier!

**AUC calculation**

compute any other metric you might be interested in. All you have to do is pass it (or a list of metrics) in as an argument to the metrics parameter of xgb.cv().

Your job in this exercise is to compute another common metric used in binary classification - the area under the curve ("auc"). As before, churn\_data is available in your workspace, along with the DMatrix churn\_dmatrix and parameter dictionary params.

Perform 3-fold cross-validation with 5 boosting rounds and "auc" as your metric. Print the "test-auc-mean" column of cv\_results

# Perform cross\_validation: cv\_results

cv\_results = xgb.cv(dtrain=churn\_dmatrix, params=params,

nfold=3, num\_boost\_round=5,

metrics="auc", as\_pandas=True, seed=123)

# Print cv\_results

print(cv\_results)

# Print the AUC

print((cv\_results["test-auc-mean"]).iloc[-1])

## **When to Use XGBoost**

XGBoost excels in **structured, tabular data** problems—especially in supervised learning tasks like classification and regression.

### 1. ****Large Number of Examples, Few Features****

* If your dataset has **thousands to millions of rows** and **dozens or hundreds of features**, XGBoost is ideal.
* It handles high-volume data efficiently and scales well with parallel processing.

### 2. ****Numerical and Categorical Features****

* Performs well with:
  + **Purely numerical data**
  + **Categorical data** (after encoding)
  + **Mixed types**
* While XGBoost doesn’t natively handle categorical variables, you can use label encoding or one-hot encoding. For native support, tools like **CatBoost** are better.

### 3. ****Missing Values****

* XGBoost can **automatically handle missing values** by learning optimal splits even when data is incomplete.

### 4. ****Feature Importance & Interpretability****

* It provides **feature importance scores**, making it useful for understanding which variables drive predictions.

### 5. ****Competitions & Benchmarks****

* Frequently used in **Kaggle competitions** and real-world business problems like:
  + Credit scoring
  + Fraud detection
  + Customer churn
  + Sales forecasting

## When Not to Use XGBoost

XGBoost is not a one-size-fits-all solution. It struggles in domains where **deep learning** dominates.

### 1. ****Image Recognition / Computer Vision****

* Images are **high-dimensional**, and relationships are spatial.
* XGBoost can't capture pixel-level patterns or convolutional hierarchies.
* Use **CNNs (Convolutional Neural Networks)** instead.

### 2. ****Natural Language Processing (NLP)****

* Text data is **sequential and contextual**.
* XGBoost can work with basic text features (e.g., TF-IDF), but it lacks semantic understanding.
* Use **transformers (e.g., BERT, GPT)** or **RNNs/LSTMs** for richer language modeling.

### 3. ****Understanding Tasks / Representation Learning****

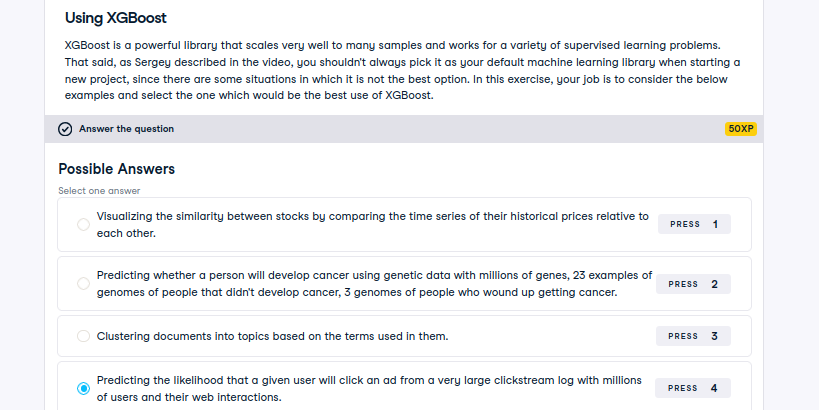
* Tasks like **sentiment analysis**, **translation**, or **question answering** require models that learn deep representations.
* XGBoost is a shallow learner—it doesn’t learn hierarchical features.

### 4. ****Very Small Training Sets (< 100 samples)****

* XGBoost is **data-hungry**. With very few samples:
  + It may **overfit** quickly.
  + It can’t generalize well.
* Use simpler models like **logistic regression**, **SVM**, or even **Bayesian methods** for small datasets.

### 5. ****High-Dimensional, Low-Sample Problems****

* If you have **more features than samples**, XGBoost may struggle.
* It can overfit and produce unstable trees.
* Dimensionality reduction or regularized models are better suited here.



## **🔹 Regression with XGBoost**

XGBoost supports regression tasks using objectives like:

* 'reg:squarederror' (default)
* 'reg:logistic'
* 'reg:pseudohubererror'

### Example Use Case:

Predicting **height in centimeters** based on features like:

* Age
* Birth year
* Nutrition score
* Parental height

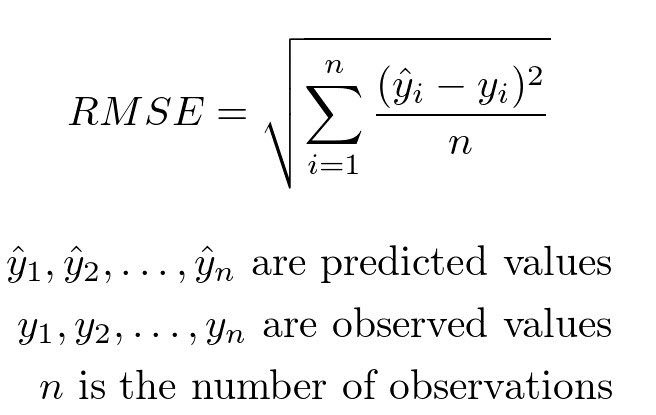
Here, the target variable (height) is **continuous**, not categorical.

## 🔹 Evaluating Regression Models

Unlike classification (which uses accuracy, precision, etc.), regression models are evaluated using **error metrics** that measure how close predictions are to actual values.

### 1. ****RMSE – Root Mean Squared Error****

#### Formula:



#### Interpretation:

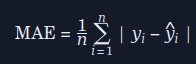
* Measures the **average magnitude of error**.
* **Squaring** the differences penalizes **larger errors** more than smaller ones.
* Treats **positive and negative errors equally**.
* Sensitive to **outliers**—a few large errors can dominate the score.

#### Use Case:

* When large errors are especially undesirable (e.g., predicting medical dosages, financial forecasts).

### 2. ****MAE – Mean Absolute Error****

#### Formula:



#### Interpretation:

* Measures the **average absolute difference** between predicted and actual values.
* **Linear penalty**: all errors contribute equally.
* **Not sensitive to outliers**—doesn’t exaggerate the impact of large errors.
* Easier to interpret: “on average, predictions are off by X units.”

#### Use Case:

* When you want a **robust metric** that’s not skewed by extreme values.

## 🔹 Objective Function in XGBoost

**Objective functions**

The **objective function** in XGBoost defines what the model is trying to optimize. It consists of two parts:

1. **Training Loss Function**: Measures how far off the predictions are from the actual values.
2. **Regularization Term**: Penalizes model complexity to prevent overfitting.

### General Form:



**Where:**

* \ell(yi, ^{y}i) is the **loss function** for each data point
* \Omega(f\_k) is the **regularization** for each tree f\_k

## **🔹 Loss Functions in XGBoost**

The **loss function** is a component of the objective function. It quantifies the error between predicted and actual values.

### 1. ****Regression Loss: reg:squarederror****

Used when predicting continuous values (e.g., height, price).

Squared error punishes large mistakes more than small ones.

* Symmetric: treats over- and under-predictions equally.

### 2. ****Binary Classification Loss: reg:logistic****

Used when predicting binary outcomes (e.g., spam vs. not spam).

* Predicts **probability** of class membership.
* Uses **logistic loss**:



* Output is a probability between 0 and 1, which can be thresholded to get class labels.

## 🔹 Base Learners in XGBoost

XGBoost builds its model using **base learners**, which are typically **decision trees**.

### Characteristics of Base Learners:

* Each tree is a **weak learner**: shallow, limited depth (often 3–6).
* Trees are trained **sequentially**, each correcting the errors of the previous.
* Final prediction is the **sum of outputs** from all trees:

### Why Decision Trees?

* The­y can model **non-linear relationships**.
* Handle both **numerical and categorical features**.
* Are interpretable and fast to train.

**Linear Base Learners**

# Load the dataset

boston\_data = pd.read\_csv("boston\_housing.csv")

# Separate features (X) and target (y)

X = boston\_data.iloc[:, :-1] # All columns except the last

y = boston\_data.iloc[:, -1] # Last column as target

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=123)

# Initialize the XGBoost regressor

xg\_reg = xgb.XGBRegressor(objective='reg:squarederror', n\_estimators=10, seed=123)

# Train the model

xg\_reg.fit(X\_train, y\_train)

# Make predictions on the test set

preds = xg\_reg.predict(X\_test)

# Evaluate model performance

mae = mean\_absolute\_error(y\_test, preds)

mse = mean\_squared\_error(y\_test, preds)

r2 = r2\_score(y\_test, preds)

# Print accuracy metrics

print("Model Performance Metrics:")

print(f"Mean Absolute Error (MAE): {mae:.2f}")

print(f"Mean Squared Error (MSE): {mse:.2f}")

print(f"R² Score: {r2:.2f}")

**{using API) linear base example**

# Load the dataset

boston\_data = pd.read\_csv("boston\_housing.csv")

# Separate features (X) and target (y)

X = boston\_data.iloc[:, :-1] # All columns except the last

y = boston\_data.iloc[:, -1] # Last column as target

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=123)

DM\_train = xgb.Dmatrix(data=X\_train, label=y\_train)

DM\_test = xgb.Dmatrix(data=X\_test, label=y\_train)

params= {“booster”:”gblinear”, “objective”:”reg:squarederror”}

xg\_reg = xgb.train(params=params, dtrain=DM\_train, num\_boost\_round=10)

preds =xg\_reg.predict(DM\_test)

**Base model linear lerner approach**

Ames, Iowa! This dataset of housing prices has been pre-loaded into a DataFrame called df. If you explore it in the Shell, you'll see that there are a variety of features about the house and its location in the city.

In this exercise, your goal is to use trees as base learners. By default, XGBoost uses trees as base learners, so you don't have to specify that you want to use trees here with booster="gbtree".

xgboost has been imported as xgb and the arrays for the features and the target are available in X and y, respectively.

Split df into training and testing sets, holding out 20% for testing. Use a random\_state of 123.

* Instantiate the XGBRegressor as xg\_reg, using a seed of 123. Specify an objective of "reg:squarederror" and use 10 trees. Note: You don't have to specify booster="gbtree" as this is the default.
* Fit xg\_reg to the training data and predict the labels of the test set. Save the predictions in a variable called preds.
* Compute the rmse using np.sqrt() and the mean\_squared\_error() function from sklearn.metrics, which has been pre-imported.

**Codes**

# Create the training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=123)

# Instantiate the XGBRegressor: xg\_reg

xg\_reg = xgb.XGBRegressor(objective='reg:squarederror', n\_estimators=10, seed=123)

# Fit the regressor to the training set

xg\_reg.fit(X\_train, y\_train)

# Predict the labels of the test set: preds

preds = xg\_reg.predict(X\_test)

# Compute the rmse: rmse

rmse = np.sqrt(mean\_squared\_error(y\_test, preds))

print("RMSE: %f" % (rmse))

**XGB API approach\**

This model, although not as commonly used in XGBoost, allows you to create a regularized linear regression using XGBoost's powerful learning API.

However, because it's uncommon, you have to use XGBoost's own non-scikit-learn compatible functions to build the model, such as xgb.train().

In order to do this you must create the parameter dictionary that describes the kind of booster you want to use (similarly to how [you created the dictionary in Chapter 1](https://campus.datacamp.com/courses/extreme-gradient-boosting-with-xgboost/10555?ex=9) when you used xgb.cv()). The key-value pair that defines the booster type (base model) you need is "booster":"gblinear".

Once you've created the model, you can use the .train() and .predict() methods of the model just like you've done in the past.

Here, the data has already been split into training and testing sets, so you can dive right into creating the DMatrix objects required by the XGBoost learning API.

* Create two DMatrix objects - DM\_train for the training set (X\_train and y\_train), and DM\_test (X\_test and y\_test) for the test set.
* Create a parameter dictionary that defines the "booster" type you will use ("gblinear") as well as the "objective" you will minimize ("reg:squarederror").
* Train the model using xgb.train(). You have to specify arguments for the following parameters: params, dtrain, and num\_boost\_round. Use 5 boosting rounds.
* Predict the labels on the test set using xg\_reg.predict(), passing it DM\_test. Assign to preds.

# Convert the training and testing sets into DMatrixes

DM\_train = xgb.DMatrix(data=X\_train, label=y\_train)

DM\_test = xgb.DMatrix(data=X\_test, label=y\_test)

# Create the parameter dictionary

params = {"booster": "gblinear", "objective": "reg:squarederror"}

# Train the model

xg\_reg = xgb.train(params=params, dtrain=DM\_train, num\_boost\_round=5)

# Predict the labels of the test set

preds = xg\_reg.predict(DM\_test) #

# Compute and print the RMSE

rmse = np.sqrt(mean\_squared\_error(y\_test, preds))

print("RMSE: %f" % (rmse))

**Prediction with cross validation**

# Create the DMatrix: housing\_dmatrix

housing\_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params

params = {"objective":"reg:squarederror", "max\_depth":4}

# Perform cross-validation: cv\_results

cv\_results = xgb.cv(dtrain=housing\_dmatrix, params=params, nfold=4, num\_boost\_round=5, metrics="rmse", as\_pandas=True, seed=123)

# Print cv\_results

print(cv\_results)

# Extract and print final boosting round metric

print((cv\_results["test-rmse-mean"]).tail(1))

**using MAE metric**

# Create the DMatrix: housing\_dmatrix

housing\_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params

params = {"objective":"reg:squarederror", "max\_depth":4}

# Perform cross-validation: cv\_results

cv\_results = xgb.cv(dtrain=housing\_dmatrix, params=params, nfold=4, num\_boost\_round=5, metrics="mae", as\_pandas=True, seed=123)

# Print cv\_results

print(cv\_results)

# Extract and print final boosting round metric

print((cv\_results["test-mae-mean"]).tail(1))

### Base Models in XGBoost

XGBoost supports two primary base learners:

* **Tree-based models (**booster='gbtree'**)**: These are decision trees built sequentially. They are powerful and widely used for structured/tabular data.
* **Linear models (**booster='gblinear'**)**: These are generalized linear models. They are faster and more interpretable but less flexible than trees.

Each booster type has its own regularization behavior and parameters.

### Regularization Parameters in XGBoost

These parameters modify the loss function to include penalties for complexity. They help balance the trade-off between accuracy and simplicity.

#### 1. ****Gamma (****gamma****)****

* **Applies to**: Tree-based models.
* **Purpose**: Controls whether a node should split.
* **Mechanism**: A split is only allowed if it results in a reduction in loss greater than gamma.
* **Effect**: Higher gamma means fewer splits, leading to simpler trees.
* **Use case**: Helps prune unnecessary branches and reduce overfitting.

#### 2. ****Alpha (****alpha****) – L1 Regularization****

* **Applies to**: Linear models and tree leaf weights.
* **Purpose**: Adds a penalty proportional to the absolute value of leaf weights.
* **Mechanism**: Encourages sparsity—many leaf weights become zero.
* **Effect**: Strong L1 regularization (high alpha) leads to simpler models with fewer active features or leaves.
* **Use case**: Useful when you suspect many features are irrelevant or when interpretability is important.

#### 3. ****Lambda (****lambda****) – L2 Regularization****

* **Applies to**: Linear models and tree leaf weights.
* **Purpose**: Adds a penalty proportional to the square of leaf weights.
* **Mechanism**: Smoothly shrinks leaf weights toward zero without forcing them to be zero.
* **Effect**: Helps prevent large leaf weights that could cause overfitting.
* **Use case**: Often used in combination with L1 to balance sparsity and smoothness.

### How Regularization Alters the Loss Function

In XGBoost, the objective function is:

object =loss+**Ω(f)**

Where:

* **Loss** is the training loss (e.g., squared error).
* **Ω(f)** is the regularization term that penalizes complexity.

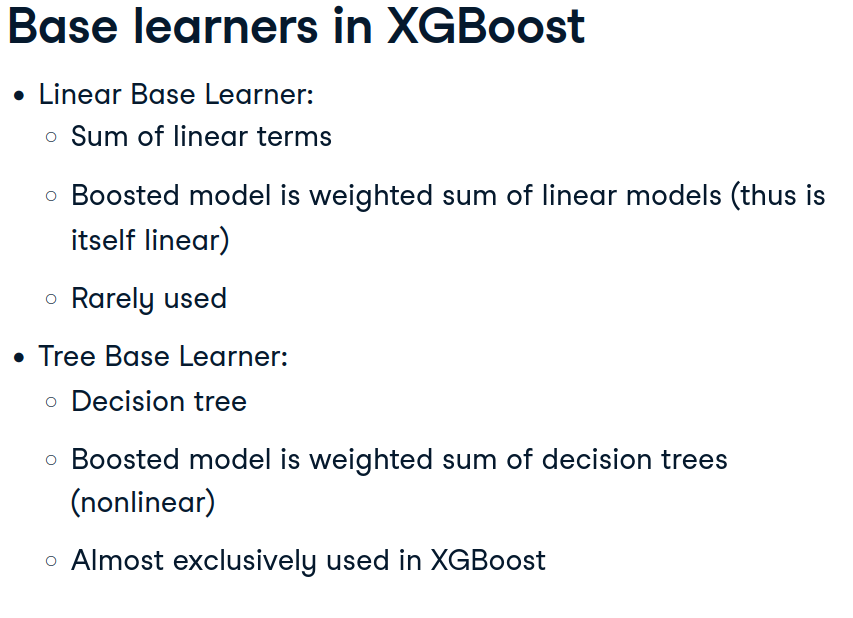
For tree models, Ω(f) includes:

* Number of leaves
* Leaf weights
* Regularization parameters (alpha, lambda, gamma)

### Practical Considerations

* **Tuning regularization parameters** is essential during hyperparameter optimization.
* **Cross-validation** helps assess the impact of regularization on generalization.
* **Over-regularization** can lead to underfitting—too simple a model that misses patterns.
* **Under-regularization** can lead to overfitting—too complex a model that memorizes noise.

| Technique | Keyword | Layman Meaning | Technical Effect |
| --- | --- | --- | --- |
| lambda | Smoothness | Shrinks model influence | Penalizes large squared weights |
| alpha | Sparsity | Ignores weak signals | Drives weights to zero via L1 penalty |
| gamma | Pruning | Limits unnecessary splits | Requires minimum loss gain to split |
| Early stopping | Caution | Stops training early | Monitors validation loss to halt training |
| Cross-validation | Testing | Tests on multiple slices | Evaluates generalization across folds |



* Use zip(list1, list2) to pair up the values.
* Wrap it with list() to get a full list of pairs.
* Pass that list into pd.DataFrame() and specify column names.

**Reinforce understaning with suppliment theory**

#### Boosting

* **Definition**: An ensemble technique that combines weak learners (e.g., shallow decision trees) sequentially to form a strong learner, focusing on correcting errors.
* **Working Mechanism**:
  + Train a weak learner (e.g., a shallow tree).
  + Identify misclassified samples and increase their weights.
  + Train subsequent learners on weighted data.
  + Combine all learners’ predictions (e.g., via weighted voting or summing).
* **Applications**:
  + Credit scoring, face detection, text classification, predictive maintenance.
* **Features**:
  + Sequential training.
  + Focuses on difficult examples.
  + Often uses decision trees.
  + High accuracy, low bias.
* **Pros and Cons**:
  + Pros: High performance, reduces bias.
  + Cons: Sensitive to noise, longer training times.
* **Critical Nuances**:
  + Learning rate controls each learner’s contribution.
  + Overfitting can occur with too many rounds.
  + Requires careful hyperparameter tuning.
* **Comparison**:
  + Unlike bagging (parallel trees, e.g., Random Forests), boosting is sequential.
  + Unlike stacking (combines different models), boosting uses the same base learner.

#### 4. XGBoost (Extreme Gradient Boosting)

* **Definition**: An optimized gradient boosting algorithm that uses decision trees as base learners, minimizing a loss function via gradient descent.
* **Working Mechanism**:
  + Start with an initial prediction (e.g., mean for regression, log odds for classification).
  + Compute residuals (errors between actual and predicted values).
  + Fit a shallow decision tree to predict residuals.
  + Update predictions by adding the tree’s output, scaled by a learning rate.
  + Repeat for a fixed number of rounds or until convergence.
  + Final prediction is the sum of all trees’ outputs.
* **Applications**:
  + Credit scoring, fraud detection, customer churn, disease diagnosis, sales forecasting, search ranking.
* **Features**:
  + Uses decision trees (default booster: gbtree).
  + Supports regularization (L1, L2) to prevent overfitting.
  + Handles missing values internally.
  + Parallelized tree construction for speed.
  + Supports early stopping and custom loss functions.
  + Works for classification, regression, and ranking tasks.
* **Pros and Cons**:
  + Pros: High accuracy, fast, robust to missing data, customizable.
  + Cons: Complex tuning, less interpretable, not ideal for unstructured data (e.g., images, text).
* **Critical Nuances**:
  + Learning rate balances speed and stability.
  + Tree depth controls model complexity.
  + Column sampling reduces tree correlation.
  + Feature importance scores are available but may not always be reliable.
* **Comparison**:
  + Unlike Random Forests (independent trees), XGBoost builds trees sequentially.
  + Unlike LightGBM (histogram-based, leaf-wise growth), XGBoost uses level-wise growth.
  + Unlike neural networks, XGBoost excels on tabular data but struggles with unstructured data.

#### 5. Classification and Evaluation Metrics

* **Types of Classification**:
  + **Binary**: Two classes (e.g., malignant vs. benign).
  + **Multiclass**: More than two classes (e.g., digit recognition 0–9).
* **Confusion Matrix** (for binary classification):
  + Tracks True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).
  + Example: Predicting pregnancy (man predicted pregnant = FP, woman predicted not pregnant = FN).
* **Evaluation Metrics**:
  + **Accuracy**: (TP + TN) / (TP + TN + FP + FN).
  + **Precision**: TP / (TP + FP) — proportion of positive predictions that are correct.
  + **Recall**: TP / (TP + FN) — proportion of actual positives correctly identified.
  + **F1-Score**: 2 \* (Precision \* Recall) / (Precision + Recall) — balances precision and recall.
  + **AUC-ROC**: Area Under the Receiver Operating Characteristic curve, measuring class separation (1 = perfect, 0.5 = random guessing).
* **Multiclass Extension**:
  + Confusion matrix becomes N×N.
  + AUC can use macro/micro averaging across classes.

#### 6. Data Preprocessing for Supervised Learning

* **Feature Vectors**:
  + Data must be represented as fixed-length numerical vectors for algorithms like decision trees or XGBoost.
  + Enables mathematical operations (e.g., distance calculations, dot products).
* **Encoding Categorical Variables**:
  + **Label Encoding**: Assigns integers to categories (e.g., red=0, blue=1).
  + **One-Hot Encoding**: Creates binary columns per category (e.g., red=[1,0,0], blue=[0,1,0]).
  + **Target Encoding**: Replaces categories with the mean of the target variable (common in boosting).
* **Feature Scaling**:
  + **Z-Score**: (value − mean) / standard deviation, standardizes features to mean 0, std 1.
  + **Why Scale?**: Algorithms like logistic regression or SVMs are sensitive to feature magnitudes. Scaling prevents dominant features and aids convergence.

#### 7. Overfitting and Underfitting

* **Overfitting**:
  + Model performs well on training data but poorly on test data.
  + Caused by high complexity (e.g., deep trees, too many features).
  + Captures noise instead of patterns, leading to poor generalization.
* **Underfitting**:
  + Model performs poorly on both training and test data.
  + Caused by insufficient model capacity (e.g., overly simple model for complex data).
* **Bias and Variance**:
  + **Bias**: Error due to overly simplistic assumptions (e.g., linear model for nonlinear data).
  + **Variance**: Error due to sensitivity to training data fluctuations.
  + Total error ≈ Bias² + Variance + Irreducible Error.
* **Mitigation**:
  + Regularization (e.g., L1, L2) to control complexity.
  + Cross-validation to assess generalization.
  + Ensemble methods like boosting to balance bias and variance.

#### 8. Cross-Validation

* **Definition**: A technique to evaluate model generalization by splitting data into K folds, training on K−1 folds, and testing on the remaining fold, repeating K times.
* **Purpose**:
  + Provides a robust estimate of performance on unseen data.
  + Reduces risk of overfitting to a single train-test split.
* **XGBoost’s Native Cross-Validation (xgb.cv)**:
  + Uses DMatrix (XGBoost’s optimized data structure).
  + Performs K-fold cross-validation with specified boosting rounds and metrics.
  + Outputs metrics (e.g., RMSE, error, AUC) per round as a DataFrame.
  + Supports early stopping to halt training if performance plateaus.
* **Why Use xgb.cv?**:
  + Built-in efficiency and control over boosting.
  + Detailed diagnostics for each boosting round.
  + Avoids manual splitting and looping.

#### 9. Regularization in XGBoost

* **Purpose**: Prevents overfitting by penalizing model complexity, balancing accuracy and simplicity.
* **Objective Function**:
  + Combines training loss (e.g., squared error) and a regularization term.
  + Form: Obj = Loss + Ω(f), where Ω(f) penalizes complex trees.
* **Key Regularization Parameters**:
  + **Gamma**: Minimum loss reduction required for a tree split (prunes unnecessary splits).
  + **Alpha (L1)**: Penalizes absolute leaf weights, encouraging sparsity (many weights become zero).
  + **Lambda (L2)**: Penalizes squared leaf weights, shrinking weights smoothly without forcing them to zero.
* **Effects**:
  + **Gamma**: Reduces tree complexity by limiting splits.
  + **Alpha**: Promotes simpler models with fewer active features.
  + **Lambda**: Prevents large weights, reducing overfitting.
* **Practical Considerations**:
  + Tuning regularization parameters is critical.
  + Over-regularization leads to underfitting; under-regularization leads to overfitting.
  + Use cross-validation to assess regularization impact.

#### 10. Regression with XGBoost

* **Definition**: Predicting continuous outcomes (e.g., house prices, height) using objectives like reg:squarederror.
* **Evaluation Metrics**:
  + **RMSE (Root Mean Squared Error)**:
    - Measures average magnitude of prediction errors.
    - Penalizes large errors more (due to squaring).
    - Sensitive to outliers.
  + **MAE (Mean Absolute Error)**:
    - Measures average absolute difference between predictions and actual values.
    - Linear penalty, robust to outliers.
    - Easier to interpret as “average error in units.”
* **Applications**:
  + Predicting house prices, sales forecasting, medical measurements.
* **Base Learners**:
  + Default: Decision trees (gbtree).
  + Alternative: Linear models (gblinear), faster but less flexible.

#### 11. When to Use XGBoost

* **Ideal Scenarios**:
  + Large datasets with thousands to millions of rows and moderate features.
  + Numerical or categorical data (after encoding).
  + Tasks with missing values (handled internally).
  + Structured/tabular data problems (e.g., credit scoring, churn prediction).
* **When Not to Use**:
  + Image recognition or NLP (use CNNs or transformers).
  + Very small datasets (<100 samples, risks overfitting).
  + High-dimensional, low-sample problems (use dimensionality reduction or simpler models).

### Potential Gaps or Additional Clarifications

The document covers the core concepts well, but here are a few additional theoretical points or nuances that might not have been explicitly emphasized but are relevant up to Chapter 3:

1. **Why Decision Trees as Base Learners?**:
   * Decision trees are chosen in XGBoost because they can model non-linear relationships, handle mixed data types, and are computationally efficient. Their hierarchical structure allows them to capture complex patterns, but they need regularization to avoid overfitting.
2. **Role of the Learning Rate**:
   * The learning rate (eta) in XGBoost scales the contribution of each tree. A lower learning rate requires more trees but improves stability and reduces overfitting. This trade-off is critical for tuning but may not have been deeply explored in the document.
3. **Early Stopping**:
   * XGBoost’s native API supports early stopping, which halts training when validation performance stops improving. This prevents overfitting and saves computation time, especially in cross-validation.
4. **Handling Missing Values**:
   * XGBoost automatically learns how to handle missing values by assigning them to the branch that minimizes loss. This is a key advantage over other algorithms that require imputation.
5. **Feature Importance**:
   * XGBoost can compute feature importance based on how often features are used in splits or their contribution to loss reduction. However, importance scores can be misleading if features are correlated or if the model is overfitted.
6. **Loss Functions in Detail**:
   * For regression, reg:squarederror uses squared loss, penalizing large errors heavily.
   * For classification, reg:logistic uses logistic loss, outputting probabilities that can be thresholded for binary predictions.
   * Custom loss functions can be defined, but this is advanced and likely beyond Chapter 3.
7. **CART Trees in XGBoost**:
   * XGBoost uses Classification and Regression Trees (CART), which handle both continuous and categorical features via optimal threshold splits. Understanding CART’s mechanics (e.g., how splits are chosen) is key to grasping XGBoost’s tree-building process.
8. **Regularization’s Impact on Interpretability**:
   * L1 (alpha) regularization promotes sparsity, which can make the model more interpretable by focusing on fewer features or leaves. L2 (lambda) ensures smoother weights, which can stabilize predictions but doesn’t inherently reduce the number of features used.
9. **Cross-Validation Metrics**:
   * In xgb.cv, metrics like RMSE, MAE, or error are computed per boosting round. The final round’s test metric (e.g., test-rmse-mean) reflects the model’s performance after all boosting rounds, providing a reliable estimate of generalization.
10. **Why DMatrix?**:
    * The DMatrix format optimizes memory usage and computation speed in XGBoost. It’s required for the native API (xgb.train, xgb.cv) and enables efficient handling of large datasets and sparse data.

| Feature | L1 Regularization (Lasso) | L2 Regularization (Ridge) |
| --- | --- | --- |
| Penalty term | Sum of **absolute** weights | Sum of **squared** weights |
| Formula | `alpha \*w | lambda \* w² |
| Effect on weights | Drives some weights to **zero** | Shrinks weights but keeps them |
| Resulting model | **Sparse** (feature selection) | **Smooth** (all features used) |
| Use case | When you want to **ignore** irrelevant features | When you want to **keep** all features but reduce impact |

**Implementing L2 regularization**

* Create your DMatrix from X and y as before.
* Create an initial parameter dictionary specifying an "objective" of "reg:squarederror" and "max\_depth" of 3.
* Use xgb.cv() inside of a for loop and systematically vary the "lambda" value by passing in the current l2 value (reg).
* Append the "test-rmse-mean" from the last boosting round for each cross-validated xgboost model.
* Hit 'Submit Answer' to view the results. What do you notice?

# Create the DMatrix: housing\_dmatrix

housing\_dmatrix = xgb.DMatrix(data=X, label=y)

reg\_params = [1, 10, 100]

# Create the initial parameter dictionary for varying l2 strength: params

params = {"objective":"reg:squarederror","max\_depth":3}

# Create an empty list for storing rmses as a function of l2 complexity

rmses\_l2 = []

# Iterate over reg\_params

for reg in reg\_params:

# Update l2 strength

params["lambda"] = reg

# Pass this updated param dictionary into cv

cv\_results\_rmse = xgb.cv(dtrain=housing\_dmatrix, params=params, nfold=2, num\_boost\_round=5, metrics="rmse", as\_pandas=True, seed=123)

# Append best rmse (final round) to rmses\_l2

rmses\_l2.append(cv\_results\_rmse["test-rmse-mean"].tail(1).values[0])

# Look at best rmse per l2 param

print("Best rmse as a function of l2:")

print(pd.DataFrame(list(zip(reg\_params, rmses\_l2)), columns=["l2","rmse"]))

**Visualizing indie xgboost in tree form**

Create a parameter dictionary with an "objective" of "reg:squarederror" and a "max\_depth" of 2.

* Train the model using 10 boosting rounds and the parameter dictionary you created. Save the result in xg\_reg.
* Plot the first tree using xgb.plot\_tree(). It takes in two arguments - the model (in this case, xg\_reg), and num\_trees, which is 0-indexed. So to plot the first tree, specify num\_trees=0.
* Plot the fifth tree.
* Plot the last (tenth) tree sideways. To do this, specify the additional keyword argument rankdir="LR"

# Create the DMatrix: housing\_dmatrix

housing\_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params

params = {"objective":"reg:squarederror", "max\_depth":2}

# Train the model: xg\_reg

xg\_reg = xgb.train(params=params, dtrain=housing\_dmatrix, num\_boost\_round=10)

# Plot the first tree

xgb.plot\_tree(xg\_reg, num\_trees=0)

plt.show()

# Plot the fifth tree

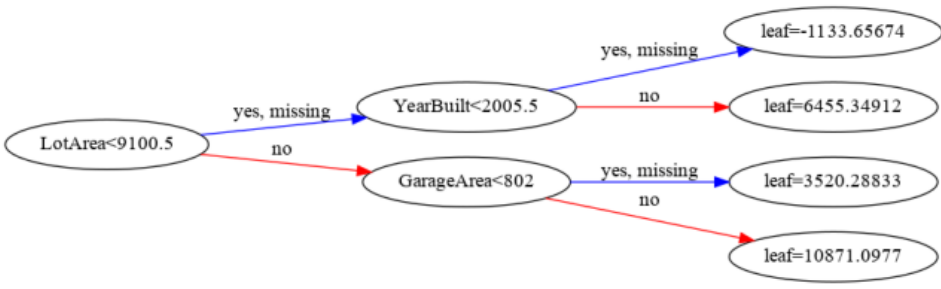
xgb.plot\_tree(xg\_reg, num\_trees=4)

plt.show()

# Plot the last tree sideways

xgb.plot\_tree(xg\_reg, num\_trees=9, rankdir='LR')

plt.show()



# **Visualizing feature importances: What features are most important in my dataset**

Another way to visualize your XGBoost models is to examine the importance of each feature column in the original dataset within the model.

One simple way of doing this involves counting the number of times each feature is split on across all boosting rounds (trees) in the model, and then visualizing the result as a bar graph, with the features ordered according to how many times they appear. XGBoost has a plot\_importance() function that allows you to do exactly this, and you'll get a chance to use it in this exercise!

## **Instructions**

* Create your DMatrix from X and y as before.
* Create a parameter dictionary with appropriate "objective" ("reg:squarederror") and a "max\_depth" of 4.
* Train the model with 10 boosting rounds, exactly as you did in the previous exercise.
* Use xgb.plot\_importance() and pass in the trained model to generate the graph of feature importances.

# Create the DMatrix: housing\_dmatrix

housing\_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params

params = {"objective":"reg:squarederror", "max\_depth":4}

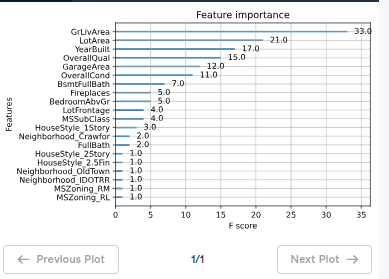
# Train the model: xg\_reg

xg\_reg = xgb.train(params=params, dtrain=housing\_dmatrix, num\_boost\_round=10)

# Plot the feature importances

xgb.plot\_importance(xg\_reg)

plt.show()



**Day 7 (Learning Utsab Day 1)**

### **Introduction to Model Tuning in Machine Learning**

Model tuning, also known as hyperparameter optimization, is the process of systematically adjusting the hyperparameters of a machine learning model to improve its performance on a given task. Hyperparameters are configuration settings that are not learned from the data but are set prior to training, such as learning rate, tree depth, or number of iterations in algorithms like XGBoost. Unlike model parameters (e.g., weights in a neural network), which are optimized during training via methods like gradient descent, hyperparameters control the overall behavior of the learning process and must be tuned externally.

The goal of tuning is to find the optimal combination of hyperparameters that minimizes a chosen loss function or maximizes a performance metric (e.g., accuracy, AUC-ROC, RMSE) on validation data, while avoiding overfitting to the training data. This is crucial because default hyperparameters provided by libraries may not be ideal for every dataset or problem, leading to suboptimal results. Tuning bridges the gap between a generic model and one tailored to specific data characteristics, such as feature distributions, noise levels, or sample sizes.

In the context of ensemble methods like gradient boosting, tuning is particularly important due to the iterative nature of building weak learners (e.g., decision trees) into a strong predictor. XGBoost, an implementation of gradient boosted trees, exemplifies this, as it combines boosting with regularization techniques to enhance speed, scalability, and accuracy.

## V1. Why Tune Your XGBoost Model?

### Motivation for Tuning

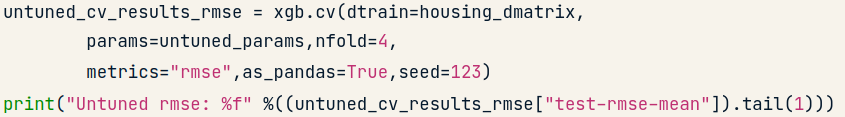
* You've learned how to use XGBoost for classification and regression.
* Now, it's time to **supercharge** those models through **parameter tuning**.
* To understand the impact, we compare:
  + A **basic untuned model**
  + A **tuned model**
* Goal: See how tuning affects **RMSE (Root Mean Squared Error)**.

## Untuned Model Example

### Basic Setup

* **Lines 1–6**: Load libraries and Ames housing dataset → convert to DMatrix.
* **Line 7**: Create a **minimal parameter configuration**:

params = {"objective": "reg:squarederror"}



* **Line 8**: Run **4-fold cross-validation** using:
* Simple parameter grid
* RMSE as evaluation metric
* **Line 9**: Print final RMSE → **~$34,600 (untuned baseline performance)**

### Improved Setup (Tuning the performance)

* **Lines 1–6**: Same data loading and conversion.
* **Line 7**: Create a **tuned parameter configuration**:

params = {

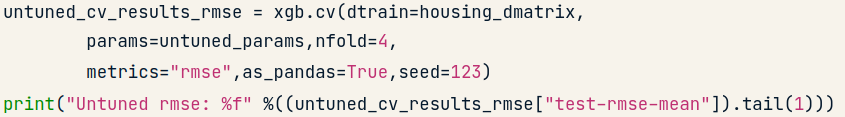
"objective": "reg:squarederror",

"colsample\_bytree": 0.3,

"learning\_rate": 0.1,

"max\_depth": 5

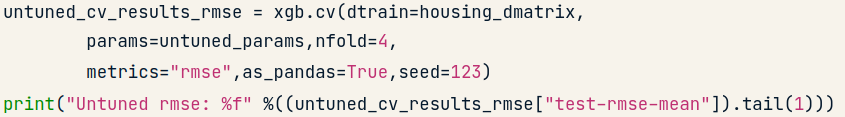
}



**Line 8**: Run **4-fold cross-validation** with:

* + Tuned parameters
  + 200 trees
  + RMSE as evaluation metric
* **Line 9**: Print final RMSE → **~$29,800**

That’s a **14% reduction in RMSE** — a major performance boost!



# Tuning the number of boosting rounds

Let's start with parameter tuning by seeing how the number of boosting rounds (number of trees you build) impacts the out-of-sample performance of your XGBoost model. You'll use xgb.cv() inside a for loop and build one model per num\_boost\_round parameter.

Here, you'll continue working with the Ames housing dataset. The features are available in the array X, and the target vector is contained in y.

# Tuning the number of boosting rounds

Let's start with parameter tuning by seeing how the number of boosting rounds (number of trees you build) impacts the out-of-sample performance of your XGBoost model. You'll use xgb.cv() inside a for loop and build one model per num\_boost\_round parameter.

Here, you'll continue working with the Ames housing dataset. The features are available in the array X, and the target vector is contained in y.

# Create the DMatrix: housing\_dmatrix

housing\_dmatrix = xgb.DMatrix(data=X,label=y)

# Create the parameter dictionary for each tree: params

params = {"objective":"reg:squarederror", "max\_depth":3}

# Create list of number of boosting rounds

num\_rounds = [5, 10, 15]

# Empty list to store final round rmse per XGBoost model

final\_rmse\_per\_round = []

# Iterate over num\_rounds and build one model per num\_boost\_round parameter

for curr\_num\_rounds in num\_rounds:

# Perform cross-validation: cv\_results

cv\_results = xgb.cv(dtrain=housing\_dmatrix, params=params, nfold=3, num\_boost\_round=curr\_num\_rounds, metrics="rmse", as\_pandas=True, seed=123)

# Append final round RMSE

final\_rmse\_per\_round.append(cv\_results["test-rmse-mean"].tail().values[-1])

# Print the resultant DataFrame

num\_rounds\_rmses = list(zip(num\_rounds, final\_rmse\_per\_round))

print(pd.DataFrame(num\_rounds\_rmses,columns=["num\_boosting\_rounds","rmse"]))

# Automated boosting round selection using early\_stopping

Now, instead of attempting to cherry pick the best possible number of boosting rounds, you can very easily have XGBoost automatically select the number of boosting rounds for you within xgb.cv(). This is done using a technique called **early stopping**.

**Early stopping** works by testing the XGBoost model after every boosting round against a hold-out dataset and stopping the creation of additional boosting rounds (thereby finishing training of the model early) if the hold-out metric ("rmse" in our case) does not improve for a given number of rounds. Here you will use the early\_stopping\_rounds parameter in xgb.cv() with a large possible number of boosting rounds (50). Bear in mind that if the holdout metric continuously improves up through when num\_boost\_rounds is reached, then early stopping does not occur.

Here, the DMatrix and parameter dictionary have been created for you. Your task is to use cross-validation with early stopping. Go for it!

***Mnemonic parameter for sgb.cv(child, fees param, fold validation test, round iteration, early halt, metric of test, as pands dataframe. seed)***

# Create your housing DMatrix: housing\_dmatrix

housing\_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary for each tree: params

params = {"objective": "reg:squarederror", "max\_depth": 4}

# Perform cross-validation with early stopping: cv\_results

cv\_results = xgb.cv(

dtrain=housing\_dmatrix,

params=params,

nfold=3,

num\_boost\_round=50,

early\_stopping\_rounds=10,

metrics="rmse",

as\_pandas=True,

seed=123

)

# Print cv\_results

print(cv\_results)

#### Why Tune Your Model?

Tuning your model is essential for several reasons:

1. **Improving Predictive Performance**: Default settings often yield acceptable but not optimal results. Tuning can significantly boost metrics like precision, recall, or F1-score by aligning the model more closely with the data's underlying patterns. For instance, in imbalanced datasets, tuning can emphasize minority class handling.
2. **Preventing Overfitting and Underfitting**: Hyperparameters control model complexity. Too simple (underfitting) and the model misses patterns; too complex (overfitting) and it memorizes noise. Tuning finds a balance, using techniques like cross-validation to generalize better to unseen data.
3. **Resource Efficiency**: Optimized models train faster and require less computational resources. For example, tuning the learning rate can reduce the number of iterations needed for convergence.
4. **Adaptation to Problem-Specific Needs**: Different tasks (e.g., regression vs. classification, time-series vs. tabular data) benefit from customized hyperparameters. In high-stakes applications like medical diagnosis or fraud detection, tuning ensures reliability and ethical outcomes.
5. **Competitive Edge**: In competitions (e.g., Kaggle) or production environments, tuned models outperform untuned ones, leading to better business decisions or higher rankings.

Pros of Tuning:

* Leads to state-of-the-art performance.
* Enhances model interpretability by revealing key hyperparameters.
* Allows integration with domain knowledge (e.g., setting priors based on expert insights).

Cons of Tuning:

* Computationally expensive, especially for large search spaces.
* Risk of overfitting to validation data if not done carefully (e.g., without nested cross-validation).
* Requires expertise to select appropriate hyperparameters and methods.

Alternatives to Manual Tuning:

* Automated tools like AutoML frameworks (e.g., Auto-Sklearn, TPOT) that combine model selection with hyperparameter optimization.
* Bayesian optimization, which uses probabilistic models to guide the search more efficiently than grid or random methods.
* Meta-learning approaches that learn from past tuning experiences across datasets.

Latest Trends:

* Integration with neural architecture search (NAS) for end-to-end optimization.
* Use of hardware accelerators (e.g., GPUs, TPUs) to speed up tuning.
* Ethical tuning, focusing on fairness metrics alongside accuracy.
* Federated tuning for privacy-preserving distributed data.

#### When Is Tuning Your Model a Bad Idea?

While tuning is powerful, it is not always advisable. Situations where it might be a bad idea include:

1. **Small Datasets**: With limited data, tuning can lead to overfitting the validation set, as there's insufficient signal to reliably estimate hyperparameter effects. Instead, rely on defaults or simple models.
2. **Time or Resource Constraints**: If computational budget is low (e.g., edge devices or real-time applications), the cost of tuning may outweigh benefits. Defaults in libraries like XGBoost are often robust.
3. **Exploratory Phases**: In early prototyping, focus on feature engineering or data quality rather than tuning, as poor data will undermine even optimized models.
4. **When Defaults Suffice**: For standard problems, pre-tuned defaults (e.g., XGBoost's out-of-the-box settings) perform well. Tuning might introduce unnecessary complexity without gains.
5. **High-Dimensional Search Spaces**: If the hyperparameter space is vast and poorly understood, random exploration might waste resources; better to use domain knowledge first.
6. **Risk of Data Leakage**: Improper tuning (e.g., using test data for validation) can inflate performance estimates, leading to deployment failures.

Pros of Skipping Tuning:

* Saves time and resources.
* Reduces risk of over-optimization.
* Encourages focus on foundational aspects like data preprocessing.

Cons of Skipping Tuning:

* Potential for suboptimal performance.
* Missed opportunities for insights into model behavior.

Alternatives:

* Use transfer learning or pre-trained models where hyperparameters are already optimized.
* Ensemble untuned models for robustness.
* Rule-based systems for simple problems instead of ML.

Latest Trends:

* "No-Tune" models emphasizing architectural innovations that minimize hyperparameter sensitivity (e.g., transformer variants).
* Self-tuning algorithms that adapt hyperparameters during training.

### Tuning in XGBoost: Focus on Boosting Rounds

#### Tuning the Number of Boosting Rounds

In XGBoost, boosting rounds (also called n\_estimators or num\_boost\_round) refer to the number of sequential decision trees added to the ensemble. Each round builds a new tree to correct errors from previous ones, using gradient boosting principles where residuals are minimized via Newton's method approximations.

Theory: Gradient boosting constructs an additive model F(x) = Σ f\_t(x), where each f\_t is a weak learner (tree) fitted to the negative gradient of the loss function. More rounds allow finer corrections but increase risk of overfitting.

Methods for Tuning:

1. Manual Iteration: Start with a high number and monitor validation performance.
2. Cross-Validation: Use k-fold CV to average performance across folds for different round values.
3. Learning Curves: Plot training vs. validation error to spot where divergence (overfitting) begins.

Steps (Theoretical):

* Split data into train/validation sets.
* Train with increasing rounds, evaluating at intervals.
* Select the round where validation error minimizes without rising.

Pros:

* Directly controls model capacity.
* Simple to implement conceptually.

Cons:

* Time-consuming if done naively.
* Sensitive to other hyperparameters (e.g., learning rate).

Alternatives:

* Adaptive boosting methods like AdaBoost, which weights based on errors rather than gradients.
* Early stopping (detailed below).

Latest Trends:

* Dynamic round adjustment in online learning scenarios.
* Integration with reinforcement learning for adaptive tuning.

#### Automated Boosting Round Selection Using Early Stopping

Early stopping is a regularization technique that halts training when validation performance stops improving, automating the selection of boosting rounds.

Theory: During training, monitor a metric (e.g., log loss) on a validation set after each round. If no improvement for a patience threshold (e.g., 10 rounds), stop to prevent overfitting. This leverages the bias-variance tradeoff: early rounds reduce bias, later ones increase variance.

Types:

* Simple Early Stopping: Based on a single metric.
* Advanced: With rolling windows or multiple metrics (e.g., combining accuracy and loss).

Methods:

* Built-in in XGBoost via eval\_set and early\_stopping\_rounds parameters.
* Custom implementations tracking metrics externally.

Steps (Theoretical):

* Prepare train and validation sets.
* Train iteratively, evaluating after each round.
* Track the best iteration based on metric improvement.
* Stop if no gain for specified rounds; revert to best.

Pros:

* Prevents overfitting automatically.
* Reduces training time.
* Robust to noisy validation data with proper patience.

Cons:

* Requires a held-out validation set, risking data scarcity.
* May stop prematurely on plateaus.
* Dependent on metric choice (e.g., AUC vs. accuracy).

Alternatives:

* Learning rate scheduling to mimic early stopping effects.
* Model checkpointing with manual review.

Latest Trends:

* Multi-metric early stopping in multi-objective optimization.
* Integration with active learning for dynamic validation sets.

### Overview of XGBoost's Hyperparameters

XGBoost hyperparameters fall into categories:

## Common Tree-Based Parameters in XGBoost

These are the most frequently tuned parameters when using the tree-based booster (booster='gbtree'):

### 1. ****learning\_rate (eta)****

* **Type**: float (typically between 0.01 and 0.3)
* **Purpose**: Controls how much each tree contributes to the final prediction.
* **Effect**: Lower values slow down learning and require more boosting rounds; higher values speed up learning but risk overfitting.

### 2. ****max\_depth****

* **Type**: int (positive integer)
* **Purpose**: Limits how deep each tree can grow.
* **Effect**: Deeper trees can model more complex patterns but may overfit. Shallower trees are simpler and may underfit.

### 3. ****subsample****

* **Type**: float (between 0 and 1)
* **Purpose**: Fraction of the training data used for each boosting round.
* **Effect**:
  + Low values → risk of underfitting due to limited data.
  + High values → risk of overfitting due to less randomness.
  + Helps introduce stochasticity for better generalization.

### 4. ****colsample\_bytree****

* **Type**: float (between 0 and 1)
* **Purpose**: Fraction of features used to build each tree.
* **Effect**:
  + Low values → stronger regularization, less chance of overfitting.
  + High values → more features used, potentially better fit but higher risk of overfitting.

### 5. ****gamma****

* **Type**: float (≥ 0)
* **Purpose**: Minimum loss reduction required to make a split.
* **Effect**: Higher values make the algorithm more conservative (fewer splits), acting as a regularizer.

### 6. ****alpha (reg\_alpha)****

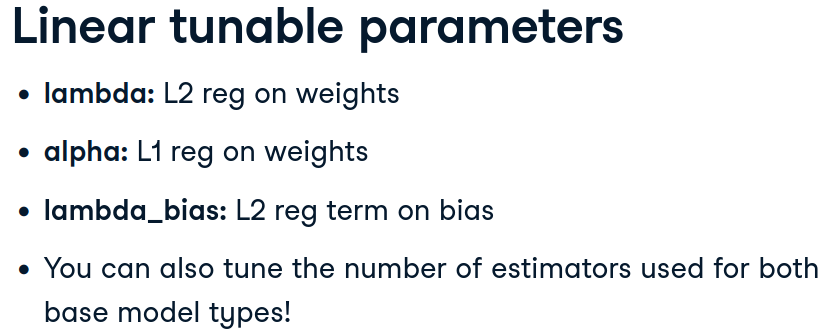
* **Type**: float (≥ 0)
* **Purpose**: L1 regularization term on weights.
* **Effect**: Encourages sparsity in feature weights, useful for high-dimensional data.

### 7. ****lambda (reg\_lambda)****

* **Type**: float (≥ 0)
* **Purpose**: L2 regularization term on weights.
* **Effect**: Helps prevent overfitting by penalizing large weights.

## Summary Tips

* Use **low learning\_rate** with **more boosting rounds** for smoother learning.
* Tune **max\_depth**, **subsample**, and **colsample\_bytree** to balance bias and variance.
* Apply **gamma**, **alpha**, and **lambda** for regularization to prevent overfitting.



Theory: These interact; e.g., high eta with low rounds approximates aggressive learning. Tuning considers tradeoffs: complexity vs. generalization, speed vs. accuracy.

Pros of XGBoost Hyperparameters:

* Fine-grained control.
* Built-in regularization outperforms vanilla boosting.

Cons:

* Interdependencies make tuning challenging.
* Over-tuning can lead to brittle models.

Latest Trends:

* Auto-tuning extensions in libraries like Optuna.
* Hyperparameter importance analysis via SHAP-like methods.

#### Tuning Eta (Learning Rate)

Eta (shrinkage) scales the contribution of each tree, typically between 0.01-0.3.

Theory: Lower eta makes learning conservative, requiring more rounds but improving generalization by smoothing the optimization path in the functional space.

Methods: Grid search over logarithmic scales (e.g., 0.001, 0.01, 0.1).

Steps:

* Fix other params.
* Train with varying eta, more rounds for lower values.
* Select based on validation plateau.

Pros: Reduces overfitting risk.  
Cons: Low eta increases compute time.  
Alternatives: Adaptive rates like in Adam optimizer.  
Trends: Cyclical learning rates for faster convergence.

# Tuning eta

It's time to practice tuning other XGBoost hyperparameters in earnest and observing their effect on model performance! You'll begin by tuning the "eta", also known as the learning rate.

The learning rate in XGBoost is a parameter that can range between 0 and 1, with higher values of "eta" penalizing feature weights more strongly, causing much stronger regularization.

# Create your housing DMatrix: housing\_dmatrix

housing\_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary for each tree (boosting round)

params = {"objective":"reg:squarederror", "max\_depth":3}

# Create list of eta values and empty list to store final round rmse per xgboost model

eta\_vals = [0.001, .01, .1]

best\_rmse = []

# Systematically vary the eta

for curr\_val in eta\_vals:

params["eta"] = curr\_val

# Perform cross-validation: cv\_results

cv\_results = xgb.cv(

dtrain=housing\_dmatrix,

params=params,

nfold=3,

num\_boost\_round=10,

early\_stopping\_rounds=5,

metrics="rmse",

as\_pandas=True,

seed=123

)

# Append the final round rmse to best\_rmse

best\_rmse.append(cv\_results["test-rmse-mean"].tail().values[-1])

# Print the resultant DataFrame

print(pd.DataFrame(list(zip(eta\_vals, best\_rmse)), columns=["eta","best\_rmse"]))

#### Tuning Max\_Depth

Max\_depth limits tree depth, controlling complexity (default 6).

Theory: Deeper trees capture interactions but overfit; shallower underfit. Relates to VC dimension for generalization bounds.

Methods: Search from 3-10, considering data size.

Steps:

* Start shallow, increase until overfitting.
* Use CV for robustness.

Pros: Direct complexity control.  
Cons: High values slow training.  
Alternatives: Min\_child\_weight for similar effects.  
Trends: Depth-pruning in neural decision trees.

# Create your housing DMatrix

housing\_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary

params = {"objective": "reg:squarederror"}

# Create list of max\_depth values

max\_depths = [2, 5, 10, 20]

best\_rmse = []

# Systematically vary the max\_depth

for curr\_val in max\_depths:

params["max\_depth"] = curr\_val

# Perform cross-validation

cv\_results = xgb.cv(

dtrain=housing\_dmatrix,

params=params,

nfold=2,

num\_boost\_round=10,

early\_stopping\_rounds=5,

metrics="rmse",

seed=123,

as\_pandas=True

)

# Append the final round RMSE to best\_rmse

best\_rmse.append(cv\_results["test-rmse-mean"].tail().values[-1])

# Print the resultant DataFrame

print(pd.DataFrame(list(zip(max\_depths, best\_rmse)), columns=["max\_depth", "best\_rmse"]))

#### Tuning Colsample\_Bytree

Colsample\_bytree is the fraction of features sampled per tree (0-1).

Theory: Introduces randomness like in random forests, reducing correlation among trees and overfitting.

Methods: Search 0.3-1.0.

Steps:

* Fix others, vary fraction.
* Evaluate diversity vs. performance.

Pros: Enhances ensemble diversity.  
Cons: Too low may miss features.  
Alternatives: Colsample\_bylevel for per-level sampling.  
Trends: Feature grouping in high-dimensional data.

# Create your housing DMatrix

housing\_dmatrix = xgb.DMatrix(data=X,label=y)

# Create the parameter dictionary

params={"objective":"reg:squarederror","max\_depth":3}

# Create list of hyperparameter values: colsample\_bytree\_vals

colsample\_bytree\_vals = [0.1, 0.5, 0.8, 1]

best\_rmse = []

# Systematically vary the hyperparameter value

for curr\_val in colsample\_bytree\_vals:

params["colsample\_bytree"] = curr\_val

# Perform cross-validation

cv\_results = xgb.cv(dtrain=housing\_dmatrix, params=params, nfold=2,

num\_boost\_round=10, early\_stopping\_rounds=5,

metrics="rmse", as\_pandas=True, seed=123)

# Append the final round rmse to best\_rmse

best\_rmse.append(cv\_results["test-rmse-mean"].tail().values[-1])

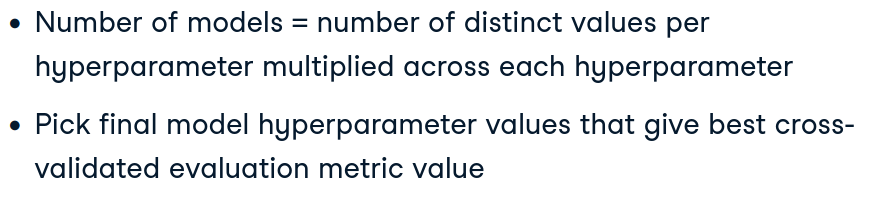
# Print the resultant DataFrame

print(pd.DataFrame(list(zip(colsample\_bytree\_vals, best\_rmse)), columns=["colsample\_bytree","best\_rmse"]))

### Hyperparameter Search Methods

#### Review of Grid Search and Random Search

Grid Search: Exhaustively evaluates all combinations in a predefined grid. Iiiif two hyper params to tune 4 values then space will try 16 of them

import pandas as pd # Load pandas for data manipulation

import xgboost as xgb # Load XGBoost library

import numpy as np # Load NumPy for numerical operations

from sklearn.model\_selection import GridSearchCV # Import GridSearchCV for hyperparameter tuning

housing\_data = pd.read\_csv("ames\_housing\_trimmed\_processed.csv")

# Load the preprocessed Ames housing dataset from CSV into a DataFrame

X, y = housing\_data[housing\_data.columns.tolist()[:-1]], housing\_data[housing\_data.columns.tolist()[-1]]

# Split the dataset into features (X) and target (y). Assumes last column is the target variable.

housing\_dmatrix = xgb.DMatrix(data=X, label=y)

# Convert the data into XGBoost’s optimized DMatrix format (optional for scikit-learn interface, but useful for native XGBoost)

gbm\_param\_grid = {

'learning\_rate': [0.01, 0.1, 0.5, 0.9], # Try 4 different learning rates (eta)

'n\_estimators': [200], # Fix number of boosting rounds to 200

'subsample': [0.3, 0.5, 0.9] # Try 3 different subsample ratios

}

# Define the grid of hyperparameters to search over. Total combinations = 4 × 1 × 3 = 12

gbm = xgb.XGBRegressor()

# Create an XGBoost regressor object using scikit-learn API

grid\_mse = GridSearchCV(

estimator=gbm, # The model to tune

param\_grid=gbm\_param\_grid, # The hyperparameter grid

scoring='neg\_mean\_squared\_error',# Use negative MSE as scoring (scikit-learn convention)

cv=4, # Use 4-fold cross-validation

verbose=1 # Print progress during training

)

# Set up the grid search with cross-validation and scoring metric

grid\_mse.fit(X, y)

# Fit the grid search object to the data. Trains 12 models (one for each parameter combo) using 4-fold CV.

print("Best parameters found: ", grid\_mse.best\_params\_)

# Print the best combination of hyperparameters found during grid search

print("Lowest RMSE found: ", np.sqrt(np.abs(grid\_mse.best\_score\_)))

# Convert the best negative MSE score to RMSE and print it

Stuffs happening here:

* You load the data and split it into features and target.
* You define a grid of hyperparameters to search over: 4 learning rates × 3 subsample values × 1 fixed number of trees = **12 combinations**.
* You use GridSearchCV to train and evaluate each combination using **4-fold cross-validation**.
* After fitting, you extract the best-performing hyperparameter set and its corresponding **lowest RMSE**, which in this case is around **$28,530**.

**Day 3**

**Random Search:** Samples randomly from distributions.

Theory: Grid assumes uniform importance; random is efficient for non-uniform spaces.

Pros (Grid): Systematic, reproducible.  
Cons (Grid): Curse of dimensionality.  
Pros (Random): Faster for large spaces.  
Cons (Random): May miss optima.

**Example:**

import pandas as pd

import xgboost as xgb

import numpy as np

from sklearn.model\_selection import RandomizedSearchCV housing\_data = pd.read\_csv("ames\_housing\_trimmed\_processed.csv") X, y = housing\_data[housing\_data.columns.tolist()[:-1]], housing\_data[ housing\_data.columns.tolist() [-1]]

housing\_dmatrix = xgb.DMatrix (data=X, label=y)

gbm\_param\_grid = {'learning\_rate': np.arange(0.05,1.05,.05),

'n\_estimators': [200],

'subsample': np.arange(0.05,1.05,.05)}

gbm xgb.XGBRegressor()

randomized\_mse = RandomizedSearchCV (estimator=gbm, param\_distributions=gbm\_param\_grid, n\_iter=25, scoring='neg\_mean\_squared\_error', cv=4, verbose=1)

randomized\_mse.fit(x, y)

print("Best parameters found: ", randomized\_mse.best\_params\_)

print("Lowest RMSE found: ", np.sqrt(np.abs (randomized\_mse.best\_score\_)))

* **Why** n\_iter=25 **matters**: You’re sampling 25 out of 400 possible combinations (20×20), which is far more efficient than grid search.
* **Use** random\_state: Ensures reproducibility of results.
* **Scoring Metric**: Use 'neg\_root\_mean\_squared\_error' for regression tasks. The negative sign is a quirk of sklearn scoring conventions.
* **Cross-validation (**cv=4**)**: Helps generalize performance across folds.
* **Avoid Overfitting**: Consider adding early stopping manually if needed, as RandomizedSearchCV doesn’t support it natively

## Random Search: Concept Review

* **Core Idea**: Unlike grid search, random search doesn't exhaustively try every combination. Instead, it samples a fixed number of random combinations from the hyperparameter space.
* **Efficiency Advantage**: The number of models trained is independent of the size of the hyperparameter grid. You control the number of iterations (n\_iter), making it scalable.
* **Workflow**:
  1. Define the hyperparameter ranges.
  2. Randomly sample combinations for a set number of iterations.
  3. Train and evaluate each model.
  4. Select the best-performing configuration.

#### Grid Search with XGBoost

Steps:

* Define grid (e.g., eta: [0.1,0.2], depth: [3,5]).
* Use CV to score each combo.
* Pick best.

Pros: Thorough.  
Cons: Exponential time. O(2^n)

#### Random Search with XGBoost

Steps:

* Define distributions (e.g., uniform for eta).
* Sample N times, evaluate via CV.
* Select top.

Pros: Scalable.  
Cons: Stochastic.

#### Limits of Grid Search and Random Search

Limits: Grid inefficient; random lacks structure. Both ignore interactions, sensitive to bounds.

Alternatives: Bayesian (e.g., GP-based), evolutionary algorithms.

#### When Should You Use Grid Search and Random Search?

Use Grid: Small spaces, when exhaustiveness matters (e.g., few params).  
Use Random: Large spaces, exploratory tuning.  
Combine with CV always.

Latest Trends: Hybrid methods, multi-fidelity optimization (cheap evals first).

**Gridsearch with Xgboost exercise**

using scikit-learn's GridSearch and RandomizedSearch capabilities with internal cross-validation using the GridSearchCV and RandomizedSearchCV functions. You will use these to find the best model exhaustively from a collection of possible parameter values across multiple parameters simultaneously. Let's get to work, starting with GridSearchCV!

gbm\_param\_grid = {

'colsample\_bytree': [0.3, 0.7],

'n\_estimators': [50],

'max\_depth': [2, 5]

}

# Instantiate the regressor: gbm

gbm = xgb.XGBRegressor()

# Perform grid search: grid\_mse

grid\_mse = GridSearchCV(estimator= gbm,

param\_grid=gbm\_param\_grid,

scoring= 'neg\_mean\_squared\_error',

cv =4,

verbose = 1)

# Fit grid\_mse to the data

grid\_mse.fit(X,y)

# Print the best parameters and lowest RMSE

print("Best parameters found: ", grid\_mse.best\_params\_)

print("Lowest RMSE found: ", np.sqrt(np.abs(grid\_mse.best\_score\_)))

Create a parameter grid called gbm\_param\_grid that contains a list of "colsample\_bytree" values (0.3, 0.7), a list with a single value for "n\_estimators" (50), and a list of 2 "max\_depth" (2, 5) values.

* Instantiate an XGBRegressor object called gbm.
* Create a GridSearchCV object called grid\_mse, passing in: the parameter grid to param\_grid, the XGBRegressor to estimator, "neg\_mean\_squared\_error" to scoring, and 4 to cv. Also specify verbose=1 so you can better understand the output.
* Fit the GridSearchCV object to X and y.
* Print the best parameter values and lowest RMSE, using the .best\_params\_ and .best\_score\_ attributes, respectively, of grid\_mse.

# **Random search with XGBoost**

Often, GridSearchCV can be really time consuming, so in practice, you may want to use RandomizedSearchCV instead, as you will do in this exercise. The good news is you only have to make a few modifications to your GridSearchCV code to do RandomizedSearchCV. The key difference is you have to specify a param\_distributions parameter instead of a param\_grid parameter.

gbm\_param\_grid = {

'n\_estimators': [25],

'max\_depth': range(2,12)

}

# Instantiate the regressor: gbm

gbm = xgb.XGBRegressor(n\_estimators=10)

# Perform random search: grid\_mse

randomized\_mse = RandomizedSearchCV(estimator =gbm,

param\_distributions = gbm\_param\_grid,

scoring='neg\_mean\_squared\_error',

cv =4,

n\_iter = 5,

verbose =1 )

# Fit randomized\_mse to the data

randomized\_mse.fit(X,y)

# Print the best parameters and lowest RMSE

print("Best parameters found: ", randomized\_mse.best\_params\_)

print("Lowest RMSE found: ", np.sqrt(np.abs(randomized\_mse.best\_score\_)))

Create a parameter grid called gbm\_param\_grid that contains a list with a single value for 'n\_estimators' (25), and a list of 'max\_depth' values between 2 and 11 for 'max\_depth' - use range(2, 12) for this.

* Create a RandomizedSearchCV object called randomized\_mse, passing in: the parameter grid to param\_distributions, the XGBRegressor to estimator, "neg\_mean\_squared\_error" to scoring, 5 to n\_iter, and 4 to cv. Also specify verbose=1 so you can better understand the output.
* Fit the RandomizedSearchCV object to X and y.

**What are pipelines in scikit-learn**

Pipelines are structured objects that allow chaining multiple steps in a machine learning workflow, typically preprocessing followed by modeling. They ensure transformations are applied consistently and in the correct order during training, cross-validation, and prediction.

**Structure and functionality**

A pipeline is built from a list of named steps, each defined as a tuple. The first element is a string name such as scaler or model. The second is a transformer or estimator object like StandardScaler or RandomForestRegressor. These steps are executed sequentially using fit and predict methods. Pipelines follow the fit transform predict paradigm where each step receives the output of the previous one.

**Why pipelines matter**

Automation helps streamline the workflow. Consistency prevents data leakage by applying preprocessing identically during training and testing. Integration allows pipelines to be used directly with cross\_val\_score for cross-validation and with GridSearchCV or RandomizedSearchCV for hyperparameter tuning. Modularity lets you swap out steps without rewriting the workflow.

**Example workflow**

Data loading involves splitting the dataset into features and target. Pipeline creation includes applying a transformer such as scaling followed by fitting a model. Evaluation uses cross-validation and scoring metrics like RMSE or accuracy. Optimization involves tuning hyperparameters using grid or random search where the pipeline itself becomes the estimator.

**Preprocessing strategies**

LabelEncoder and OneHotEncoder are not pipeline compatible. LabelEncoder converts string categories into integers. OneHotEncoder converts integer categories into binary dummy variables. This two-step process cannot be embedded directly in a pipeline because LabelEncoder is not designed for column-wise transformations in DataFrames.

DictVectorizer is pipeline compatible. It converts a list of dictionaries into a numerical matrix. It combines label encoding and one-hot encoding in one step. It is useful for categorical or text features. The DataFrame must be converted into a list of dictionaries before applying.

**Insights**

Pipelines are essential for building robust and reproducible machine learning workflows. They encapsulate all steps from raw data to prediction. For complex datasets like Ames Housing, pipelines are especially useful due to multiple preprocessing needs. For categorical preprocessing, prefer pipeline-compatible tools like DictVectorizer or use ColumnTransformer to apply different transformations to different columns.

### **Preprocessing I: LabelEncoder and OneHotEncoder**

This approach involves two sequential steps using scikit-learn’s preprocessing tools:

* **LabelEncoder**: Converts categorical string values into integer labels. Each unique string is mapped to a distinct integer.
* **OneHotEncoder**: Takes those integer labels and transforms them into binary dummy variables, creating one column per category.

**Limitation**: This two-step method cannot be directly embedded within a scikit-learn pipeline. LabelEncoder is not designed to work column-wise on DataFrames, making it incompatible with pipeline-based workflows that require consistent handling of multiple columns.

### Preprocessing II: DictVectorizer

DictVectorizer offers a more streamlined and pipeline-friendly alternative:

* It belongs to scikit-learn’s feature extraction module and is commonly used in text processing.
* It transforms a list of dictionaries (each representing a row of feature mappings) into a numeric feature matrix.
* This single step effectively combines the functionality of both LabelEncoder and OneHotEncoder.

**Requirement**: Since pandas DataFrames are not in dictionary format by default, you must convert them using to\_dict(orient='records') to apply DictVectorizer.

**Advantage**: DictVectorizer is compatible with pipelines and simplifies categorical preprocessing, especially when working with mixed-type data.

**Shell based exercise solving**

In [1]:

df.info()

<class 'pandas.core.frame.DataFrame'> RangeIndex: 1460 entries, 0 to 1459 Data columns (total 21 columns): # Column Non-Null Count Dtype --- ------ -------------- ----- 0 MSSubClass 1460 non-null int64 1 LotFrontage 1201 non-null float64 2 LotArea 1460 non-null int64 3 OverallQual 1460 non-null int64 4 OverallCond 1460 non-null int64 5 YearBuilt 1460 non-null int64 6 Remodeled 1460 non-null int64 7 GrLivArea 1460 non-null int64 8 BsmtFullBath 1460 non-null int64 9 BsmtHalfBath 1460 non-null int64 10 FullBath 1460 non-null int64 11 HalfBath 1460 non-null int64 12 BedroomAbvGr 1460 non-null int64 13 Fireplaces 1460 non-null int64 14 GarageArea 1460 non-null int64 15 MSZoning 1460 non-null object 16 PavedDrive 1460 non-null object 17 Neighborhood 1460 non-null object 18 BldgType 1460 non-null object 19 HouseStyle 1460 non-null object 20 SalePrice 1460 non-null int64 dtypes: float64(1), int64(15), object(5) memory usage: 239.7+ KB

In [2]:

df.head

Out[2]:

<bound method NDFrame.head of MSSubClass LotFrontage LotArea OverallQual OverallCond ... PavedDrive Neighborhood BldgType HouseStyle SalePrice 0 60 65.0 8450 7 5 ... Y CollgCr 1Fam 2Story 208500 1 20 80.0 9600 6 8 ... Y Veenker 1Fam 1Story 181500 2 60 68.0 11250 7 5 ... Y CollgCr 1Fam 2Story 223500 3 70 60.0 9550 7 5 ... Y Crawfor 1Fam 2Story 140000 4 60 84.0 14260 8 5 ... Y NoRidge 1Fam 2Story 250000 ... ... ... ... ... ... ... ... ... ... ... ... 1455 60 62.0 7917 6 5 ... Y Gilbert 1Fam 2Story 175000 1456 20 85.0 13175 6 6 ... Y NWAmes 1Fam 1Story 210000 1457 70 66.0 9042 7 9 ... Y Crawfor 1Fam 2Story 266500 1458 20 68.0 9717 5 6 ... Y NAmes 1Fam 1Story 142125 1459 20 75.0 9937 5 6 ... Y Edwards 1Fam 1Story 147500 [1460 rows x 21 columns]>

In [3]:

In [3]:

X = df.drop(columns=df.columns[-1])  
y = df[df.columns[-1]]

In [4]:

X.isnull().sum().sort\_values(ascending=False)

Out[4]:

LotFrontage 259 MSSubClass 0 BldgType 0 Neighborhood 0 PavedDrive 0 MSZoning 0 GarageArea 0 Fireplaces 0 BedroomAbvGr 0 HalfBath 0 FullBath 0 BsmtHalfBath 0 BsmtFullBath 0 GrLivArea 0 Remodeled 0 YearBuilt 0 OverallCond 0 OverallQual 0 LotArea 0 HouseStyle 0 dtype: int64

In [5]:

X = X.fillna("Missing")  # For categorical  
X = X.fillna(X.mean(numeric\_only=True))  # For numeric

In [6]:

X\_encoded = pd.get\_dummies(X)

In [7]:

    X\_encoded.dtypes.value\_counts()

Out[7]:

uint8 157 int64 14 dtype: int64

In [1]:

df["LotArea"].mean()

Out[1]:

10516.828082191782

In [2]:

df["SalePrice"].std()

Out[2]:

79442.50288288662

In [3]:

df.isnull().sum().sum()

Out[3]:

259

In [4]:

df["LotFrontage"].isnull().sum()

Out[4]:

259

In [3]:

***hence lot frontage has missig values so option c***

# ***Encoding categorical columns I: LabelEncoder***

Now that you've seen what will need to be done to get the housing data ready for XGBoost, let's go through the process step-by-step.

First, you will need to fill in missing values - as you saw previously, the column LotFrontage has many missing values. Then, you will need to encode any categorical columns in the dataset using one-hot encoding so that they are encoded numerically. You can watch [this video](https://campus.datacamp.com/courses/supervised-learning-with-scikit-learn/preprocessing-and-pipelines?ex=1) from [Supervised Learning with scikit-learn](https://www.datacamp.com/courses/supervised-learning-with-scikit-learn) for a refresher on the idea.

The data has five categorical columns: MSZoning, PavedDrive, Neighborhood, BldgType, and HouseStyle. Scikit-learn has a [LabelEncoder](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html) function that converts the values in each categorical column into integers. You'll practice using this here.

*# Import LabelEncoder*

from sklearn.preprocessing import LabelEncoder

# Fill missing values with 0

df.LotFrontage = df.LotFrontage.fillna(0)

# Create a boolean mask for categorical columns

categorical\_mask = (df.dtypes == object)

# Get list of categorical column names

categorical\_columns = df.columns[categorical\_mask].tolist()

# Print the head of the categorical columns

print(df[categorical\_columns].head())

# Create LabelEncoder object: le

le = LabelEncoder()

# Apply LabelEncoder to categorical columns

df[categorical\_columns] = df[categorical\_columns].apply(lambda x: le.fit\_transform(x))

# Print the head of the LabelEncoded categorical columns

print(df[categorical\_columns].head())

# ***Encoding categorical columns II: OneHotEncoder***

Okay - so you have your categorical columns encoded numerically. Can you now move onto using pipelines and XGBoost? Not yet! In the categorical columns of this dataset, there is no natural ordering between the entries. As an example: Using LabelEncoder, the CollgCr Neighborhood was encoded as 5, while the Veenker Neighborhood was encoded as 24, and Crawfor as 6. Is Veenker "greater" than Crawfor and CollgCr? No - and allowing the model to assume this natural ordering may result in poor performance.

As a result, there is another step needed: You have to apply a one-hot encoding to create binary, or "dummy" variables. You can do this using scikit-learn's [OneHotEncoder](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html).

# Import OneHotEncoder

from sklearn.preprocessing import OneHotEncoder

# Create OneHotEncoder: ohe

ohe = OneHotEncoder(sparse=False)

# Apply OneHotEncoder to the full DataFrame

df\_encoded = ohe.fit\_transform(df)

# Print first 5 rows of the resulting dataset

print(df\_encoded[:5, :])

# Print the shape of the original DataFrame

print(df.shape)

# Print the shape of the transformed array

print(df\_encoded.shape)

# **Encoding categorical columns III: DictVectorizer**

Alright, one final trick before you dive into pipelines. The two step process you just went through - LabelEncoder followed by OneHotEncoder - can be simplified by using a [DictVectorizer](http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.DictVectorizer.html).

Using a DictVectorizer on a DataFrame that has been converted to a dictionary allows you to get label encoding as well as one-hot encoding in one go.

­# Import DictVectorizer

from sklearn.feature\_extraction import DictVectorizer

# Convert df into a dictionary: df\_dict

­df\_dict = df.to\_dict(orient="records")

# Create the DictVectorizer object: dv

dv = DictVectorizer(sparse=False)

# Apply dv on df: df\_encoded

df\_encoded = dv.fit\_transform(df\_dict)

# Print the resulting first five rows

print(df\_encoded[:5, :])

­

# Print the vocabulary

print(dv.vocabulary\_)

Day 4

# **Preprocessing within a pipeline**

Now that you've seen what steps need to be taken individually to properly process the Ames housing data, let's use the much cleaner and more succinct DictVectorizer approach and put it alongside an XGBoostRegressor inside of a scikit-learn pipeline.

# Import necessary modules

from sklearn.feature\_extraction import DictVectorizer

from sklearn.pipeline import Pipeline

# Fill LotFrontage missing values with 0

X['LotFrontage'] = X['LotFrontage'].fillna(0)

X\_dict = X.to\_dict("records")

# Setup the pipeline steps: steps

steps = [("ohe\_onestep", DictVectorizer(sparse=False)),

("xgb\_model", xgb.XGBRegressor())]

# Create the pipeline: xgb\_pipeline

xgb\_pipeline = Pipeline(steps)

# Fit the pipeline

xgb\_pipeline.fit(X\_dict, y)

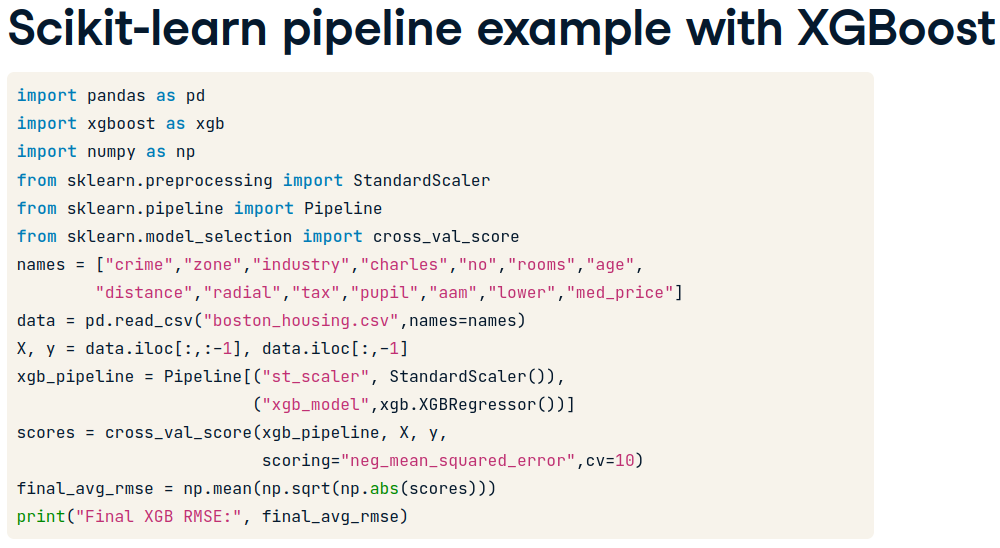
**Import Required Modules**

* + Import all necessary libraries including Pipeline from sklearn.pipeline and XGBRegressor from xgboost.

1. **Load and Prepare Data**
   * Load the dataset.
   * Split it into the feature matrix X and target vector y.
2. **Pipeline Construction**
   * The only difference from using a native scikit-learn model is here: instead of a RandomForestRegressor or LinearRegression, you pass an instance of xgb.XGBRegressor() into the pipeline.
3. **Model Evaluation**
   * Use 10-fold cross-validation to compute the **negative mean squared error (MSE)**.
   * Convert the negative MSE scores into **root mean squared error (RMSE)** values.
   * Average the RMSE across all folds to assess model performance.

### Performance Comparison xgboost with only pipeline ooutperformed randomforest here

* **XGBoost RMSE**: ~4.03 units
* **Random Forest RMSE**: ~4.50 units



# **Cross-validating your XGBoost model**

In this exercise, you'll go one step further by using the pipeline you've created to preprocess **and** cross-validate your model.

# Import necessary modules

from sklearn.feature\_extraction import DictVectorizer

from sklearn.pipeline import Pipeline

from sklearn.model\_selection import cross\_val\_score

import xgboost as xgb

import numpy as np

# Fill LotFrontage missing values with 0

X['LotFrontage'] = X['LotFrontage'].fillna(0)

# Convert X to dictionary format

X\_dict = X.to\_dict("records")

# Setup the pipeline steps

steps = [

("ohe\_onestep", DictVectorizer(sparse=False)),

("xgb\_model", xgb.XGBRegressor(max\_depth=2, objective="reg:squarederror"))

]

# Create the pipeline

xgb\_pipeline = Pipeline(steps)

# Perform 10-fold cross-validation

cross\_val\_scores = cross\_val\_score(

xgb\_pipeline,

X\_dict,

y,

scoring="neg\_mean\_squared\_error",

cv=10

)

# Print the 10-fold RMSE

rmse\_scores = np.sqrt(np.abs(cross\_val\_scores))

print("10-fold RMSE: ", np.mean(rmse\_scores))

# Kidney disease case study I: Categorical Imputer

You'll now continue your exploration of using pipelines with a dataset that requires significantly more wrangling. The [chronic kidney disease dataset](https://archive.ics.uci.edu/ml/datasets/chronic_kidney_disease) contains both categorical and numeric features, but contains lots of missing values. The goal here is to predict who has chronic kidney disease given various blood indicators as features.

As Sergey mentioned in the video, you'll be introduced to a new library, [sklearn\_pandas](https://github.com/pandas-dev/sklearn-pandas), that allows you to chain many more processing steps inside of a pipeline than are currently supported in scikit-learn. Specifically, you'll be able to use the DataFrameMapper() class to apply any arbitrary sklearn-compatible transformer on DataFrame columns, where the resulting output can be either a NumPy array or DataFrame.

We've also created a transformer called a Dictifier that encapsulates converting a DataFrame using .to\_dict("records") without you having to do it explicitly (and so that it works in a pipeline). Finally, we've also provided the list of feature names in kidney\_feature\_names, the target name in kidney\_target\_name, the features in X, and the target in y.

In this exercise, your task is to apply sklearn's SimpleImputer to impute all of the categorical columns in the dataset. You can refer to how the numeric imputation mapper was created as a template. Notice the keyword arguments input\_df=True and df\_out=True? This is so that you can work with DataFrames instead of arrays. By default, the transformers are passed a numpy array of the selected columns as input, and as a result, the output of the DataFrame mapper is also an array. Scikit-learn transformers have historically been designed to work with numpy arrays, not pandas DataFrames, even though their basic indexing interfaces are similar.

# Import necessary modules

from sklearn\_pandas import DataFrameMapper

from sklearn.impute import SimpleImputer

# Check number of nulls in each feature column

nulls\_per\_column = X.isnull().sum()

print(nulls\_per\_column)

# Create a boolean mask for categorical columns

categorical\_feature\_mask = X.dtypes == object

# Get list of categorical column names

categorical\_columns = X.columns[categorical\_feature\_mask].tolist()

# Get list of non-categorical column names

non\_categorical\_columns = X.columns[~categorical\_feature\_mask].tolist()

# Apply numeric imputer

numeric\_imputation\_mapper = DataFrameMapper(

[([numeric\_feature], SimpleImputer(strategy="median")) for numeric\_feature in non\_categorical\_columns],

input\_df=True,

df\_out=True

)

# Apply categorical imputer

categorical\_imputation\_mapper = DataFrameMapper(

[(category\_feature, SimpleImputer(strategy="most\_frequent")) for category\_feature in categorical\_columns],

input\_df=True,

df\_out=True

)

# **Kidney disease case study II: Feature Union**

Having separately imputed numeric as well as categorical columns, your task is now to use scikit-learn's [FeatureUnion](http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.FeatureUnion.html) to concatenate their results, which are contained in two separate transformer objects - numeric\_imputation\_mapper, and categorical\_imputation\_mapper, respectively.

You may have already encountered FeatureUnion in [Machine Learning with the Experts: School Budgets](https://campus.datacamp.com/courses/machine-learning-with-the-experts-school-budgets/improving-your-model?ex=7). Just like with pipelines, you have to pass it a list of (string, transformer) tuples, where the first half of each tuple is the name of the transformer.

# Import FeatureUnion

from sklearn.pipeline import FeatureUnion

# Combine the numeric and categorical transformations

numeric\_categorical\_union = FeatureUnion([

("num\_mapper", numeric\_imputation\_mapper),

("cat\_mapper", categorical\_imputation\_mapper)

])

# **Kidney disease case study III: Full pipeline**

It's time to piece together all of the transforms along with an XGBClassifier to build the full pipeline!

Besides the numeric\_categorical\_union that you created in the previous exercise, there are two other transforms needed: the Dictifier() transform which we created for you, and the DictVectorizer().

After creating the pipeline, your task is to cross-validate it to see how well it performs.

# Create full pipeline

pipeline = Pipeline([

("featureunion", numeric\_categorical\_union),

("dictifier", Dictifier()),

("vectorizer", DictVectorizer(sort=False)),

("clf", xgb.XGBClassifier(max\_depth=3))

])

# Perform cross-validation

cross\_val\_scores = cross\_val\_score(

pipeline,

kidney\_data,

y,

scoring="roc\_auc",

cv=3

)

# Print avg. AUC

print("3-fold AUC: ", np.mean(cross\_val\_scores))

**imports needed**

pip install pandas numpy scikit-learn xgboost scipy sklearn-pandas liac-arff

## **Why Use Hyperparameter Tuning in a Pipeline?**

Pipelines allow you to chain preprocessing and modeling steps together. When combined with automated hyperparameter tuning (like RandomizedSearchCV), you get:

* Cleaner, reproducible workflows
* Seamless integration of preprocessing and modeling
* Efficient search over model configurations

## Step-by-Step Explanation

### 1. ****Import Required Libraries****

You’ll need:

* pandas and numpy for data handling
* StandardScaler for feature scaling
* Pipeline to chain steps
* XGBRegressor from xgboost
* RandomizedSearchCV for tuning
* mean\_squared\_error for RMSE conversion

### 2. ****Load the Boston Housing Dataset****

Assuming you have boston\_housing.csv, load it and split into features X and target y.

### 3. ****Create the Pipeline****

Include:

* "scaler": StandardScaler() for numeric feature scaling
* "xgb\_model": XGBRegressor() as the estimator

### 4. ****Define Hyperparameter Grid****

Use double underscores to target parameters inside the pipeline:

{

"xgb\_model\_\_subsample": [...],

"xgb\_model\_\_max\_depth": [...],

"xgb\_model\_\_colsample\_bytree": [...]

}

### 5. ****Run RandomizedSearchCV****

Pass the pipeline and parameter grid to RandomizedSearchCV, set scoring to "neg\_mean\_squared\_error" and cv=5.

### 6. ****Evaluate Best Model****

Convert the best score to RMSE and print the best estimator.

**An example**

# Import necessary libraries

import pandas as pd

import numpy as np

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.metrics import mean\_squared\_error

import xgboost as xgb

# Load the Boston Housing dataset

df = pd.read\_csv("boston\_housing.csv")

# Split into features and target

X = df.drop("MEDV", axis=1)

y = df["MEDV"]

# Create pipeline

pipeline = Pipeline([

("scaler", StandardScaler()),

("xgb\_model", xgb.XGBRegressor(objective="reg:squarederror"))

])

# Define hyperparameter grid

param\_grid = {

"xgb\_model\_\_subsample": [0.5, 0.7, 1.0],

"xgb\_model\_\_max\_depth": [3, 5, 7],

"xgb\_model\_\_colsample\_bytree": [0.5, 0.7, 1.0]

}

# Setup RandomizedSearchCV

random\_search = RandomizedSearchCV(

estimator=pipeline,

param\_distributions=param\_grid,

scoring="neg\_mean\_squared\_error",

cv=5,

n\_iter=10,

random\_state=42

)

# Fit the model

random\_search.fit(X, y)

# Evaluate best model

best\_score = random\_search.best\_score\_

rmse = np.sqrt(-best\_score)

print("Best RMSE:", rmse)

# Print best model

print("Best Estimator:\n", random\_search.best\_estimator\_)