**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

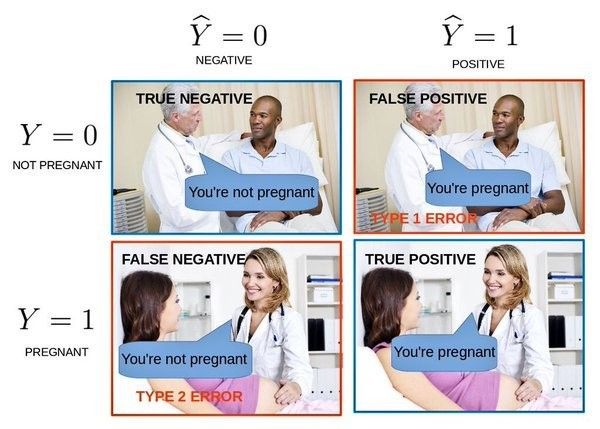
More than two classes. Example: classifying handwritten digits (0–9), types of fruits.

## 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:

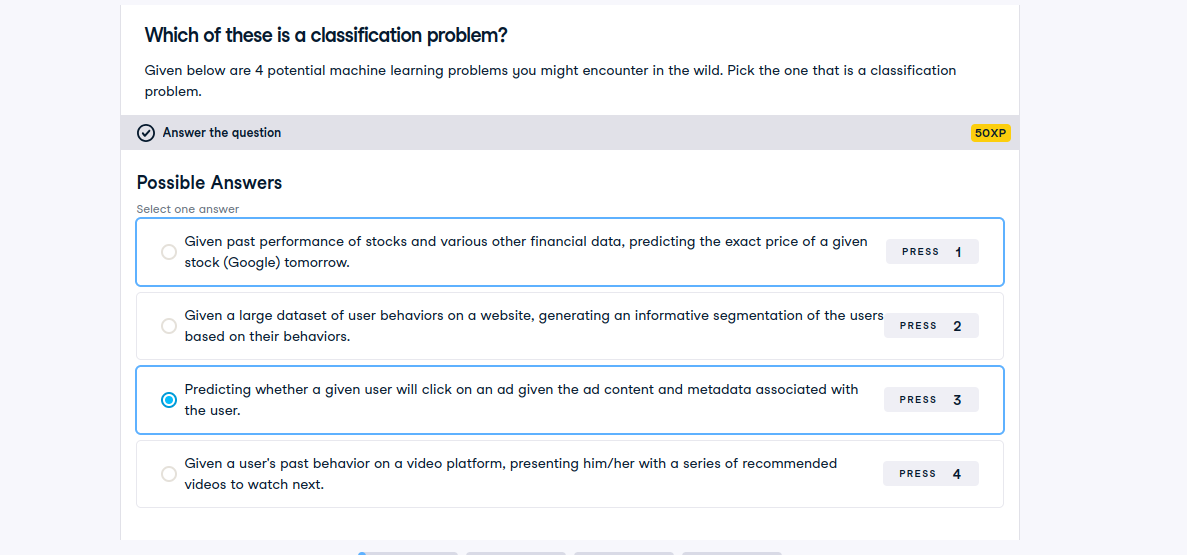
* **Label Encoding**: Assigns integer to each category (e.g., red=0, blue=1)
* **One-Hot Encoding**: Creates binary columns for each category (e.g., red → [1,0,0], blue → [0,1,0])
* **Target Encoding**: Replaces category with mean of target variable (used in boosting)

## 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

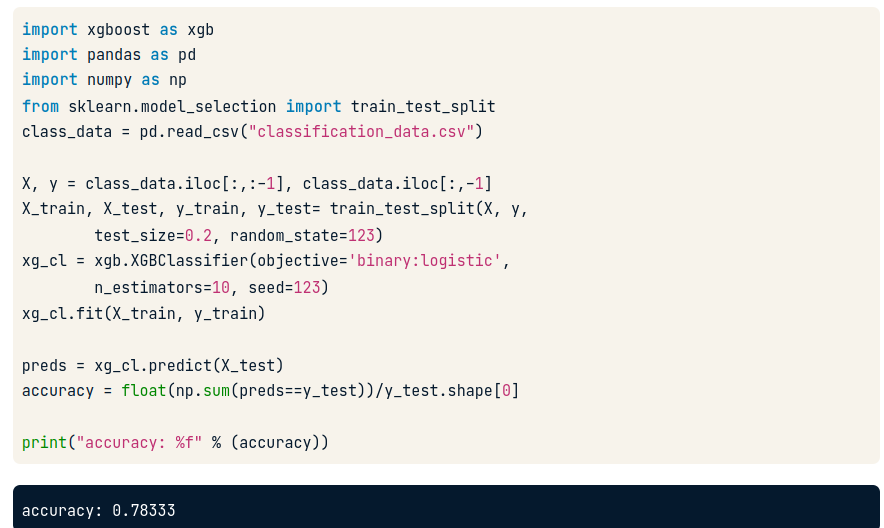
2. load our data in from file and split the entire dataset into matrix of samples by features ‘x’ and vector of target values ‘y’

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:

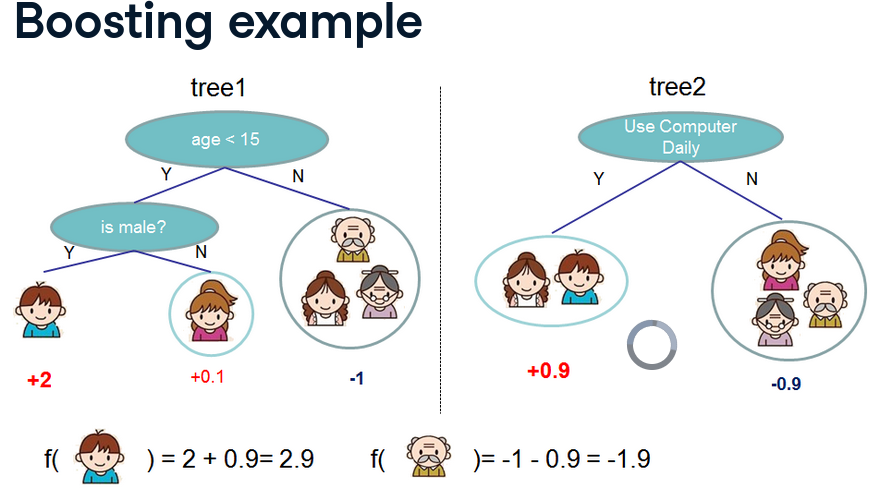


Where:

* 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**

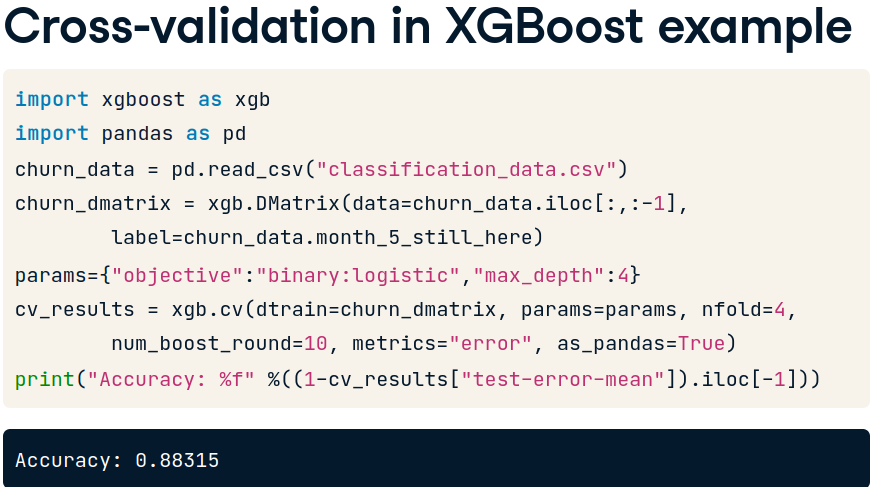
1. inport the shit

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****

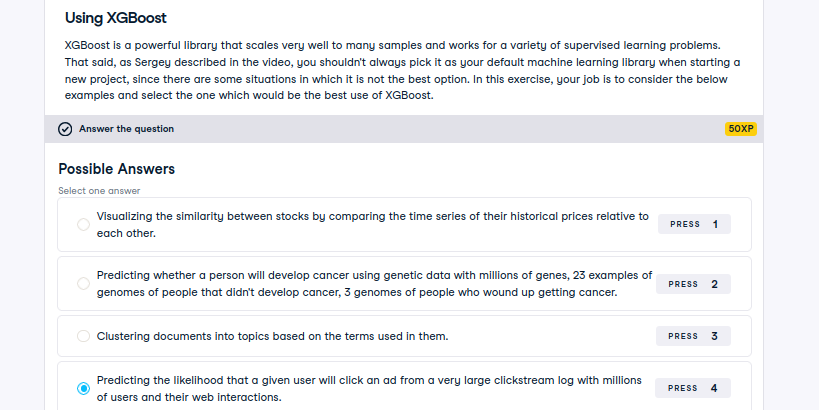
* Tasks like **sentiment analysis**, **translation**, or **question answering** require models that learn deep representations.
* XGBoost is a shallow learner—it doesn’t learn hierarchical features.

### 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:

\text{RMSE} = \sqrt{ \frac{1}{n} \sum\_{i=1}^{n} (y\_i - \hat{y}\_i)^2 }

#### Interpretation:

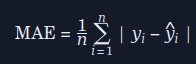
* Measures the **average magnitude of error**.
* **Squaring** the differences penalizes **larger errors** more than smaller ones.
* Treats **positive and negative errors equally**.
* Sensitive to **outliers**—a few large errors can dominate the score.

#### 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.