CSE 435:Data Mining

Chapter 6: Classification: basic concepts and methods

Md Atikuzzaman
Lecturer
Department of Computer Science & Engineering
atik@cse.green.edu.bd

GREEN UNIVERSITY OF BANGLADESH

Outline

What is a Decision Tree?

A Decision Tree is a **non-parametric**, supervised learning algorithm that builds a model in the form of a tree structure. It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed.

Key Components

- **Root Node:** The top-most node, representing the entire dataset.
- **Internal Node:** Represents a test on an attribute.
- **Branch:** Represents the outcome of the test.
- **Leaf Node:** A terminal node that holds a class label.

The Goal

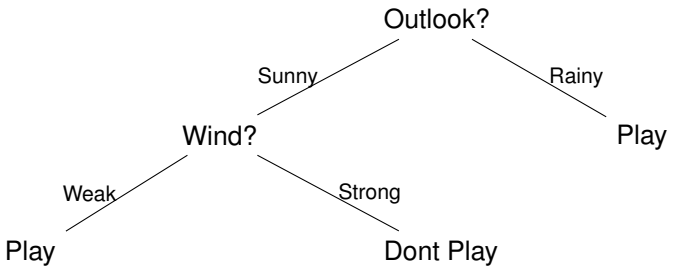
To partition the data space recursively until the leaf nodes are as "pure" as possible (i.e., contain samples from only one class).

Visualizing a Decision Tree

Sample Data Set

Outlook	Wind	Decision
Sunny	Weak	Play
Sunny	Strong	Don't Play
Rainy	Weak	Play
Rainy	Strong	Play
Sunny	Weak	Play
Sunny	Strong	Don't Play
Rainy	Weak	Play

Resulting Tree



How it Works

At each node, the algorithm selects the feature that maximizes the **purity** of resulting child nodes.

Splitting Criteria: How to Choose the Best Feature?

The "best" split is the one that maximizes the **purity** of the resulting child nodes. We measure this using an impurity function.

Gini Impurity (used by CART)

Measures the probability of incorrectly classifying a randomly chosen element.

$$Gini(D) = 1 - \sum_{i=1}^{k} (p_i)^2$$

where p_i is the probability of class i at node D.

- $Gini = 0$: Node is pure (all one class).
- $Gini = 0.5$: Node is maximally impure (for $k = 2$).

Entropy (used by ID3, C4.5)

Measures the level of disorder or uncertainty in the node.

$$H(D) = -\sum_{i=1}^{k} p_i \log_2(p_i)$$

where p_i is the probability of class i at node D.

- $H = 0$: Node is pure.
- $H = 1$: Node is maximally impure (for $k = 2$).

Splitting Criteria: Information Gain

Algorithms like ID3 and C4.5 use **Information Gain (IG)** to select the best feature to split on.

Information Gain Formula

- Expected information (entropy) needed to classify a tuple in D:

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i)$$

- Information needed (after using A to split D into v partitions) to classify D:

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j)$$

- Information gained by branching on attribute A:

$$Gain(A) = Info(D) - Info_A(D)$$

- Gain ratio: Overcomes the problem (as a normalization to information gain)

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2(\frac{|D_j|}{|D|})$$

Step-by-Step: Finding the Root Node

Goal: Find which attribute ("Weather" or "Temp") is the best to split the data on first. We use **Information Gain** to measure this.

Mini-Dataset (D):

Weather	Temp	Play?
Sunny	Hot	No
Sunny	Mild	Yes
Rainy	Hot	No
Rainy	Mild	Yes
Sunny	Hot	No

Overall Data (D):

- 5 samples total
- 2 Play = Yes
- 3 Play = No

This is our starting point, or "Root Node". It's **impure** because it contains a mix of "Yes" and "No" outcomes.

Step 1: Calculate Parent Entropy Info(D)

We first measure the total uncertainty of the entire dataset.

Root Node Visualization:

Root Node (D)

5 Samples Total
2 Yes / 3 No

Calculation:

- $p_{Yes} = 2/5 = 0.4$
- $p_{No} = 3/5 = 0.6$

$$Info(D) = -\sum p_i \log_2(p_i)$$

$$Info(D) = -(0.4 \log_2(0.4) + 0.6 \log_2(0.6))$$

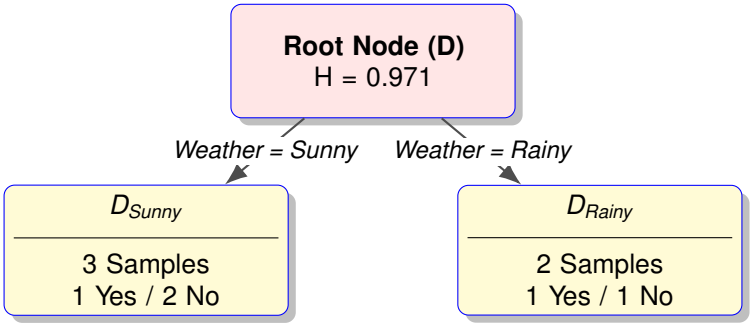
$$Info(D) = \mathbf{0.971}$$

Starting Entropy

Our starting uncertainty is **0.971**. The goal of a split is to reduce this number as much as possible.

Step 2: Test Split for "Weather" (1/2)

Let's see what happens if we split the data by "Weather".



This split creates two new child nodes, neither of which is pure. Now, we must calculate their entropy.

Step 2: Calculate Gain for "Weather" (2/2)

Parent Entropy: $Info(D) = 0.971$

Child Entropies:

- D_{Sunny}: (1 Yes, 2 No)

$$Info(D_S) = -\left(\frac{1}{3}\log_2\frac{1}{3} + \frac{2}{3}\log_2\frac{2}{3}\right) = \mathbf{0.918}$$

- D_{Rainy}: (1 Yes, 1 No)

$$Info(D_R) = -\left(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{2}\log_2\frac{1}{2}\right) = \mathbf{1.0}$$

Weighted Average Entropy:

$$Info_{\text{Weather}}(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j)$$

$$= \left[\frac{3}{5}Info(D_S) + \frac{2}{5}Info(D_R)\right]$$

$$= [0.6(0.918) + 0.4(1.0)] = \mathbf{0.951}$$

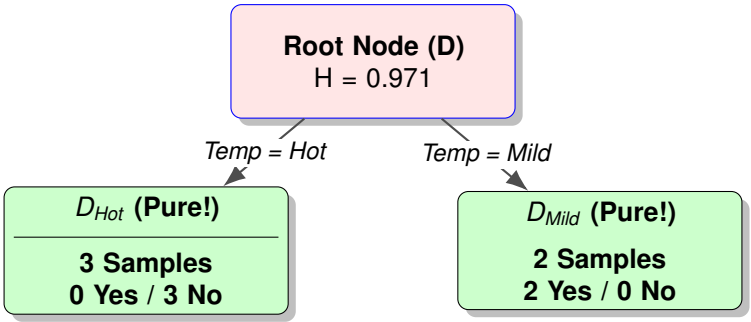
Information Gain (Weather):

$$Gain(\text{Weather}) = Info(D) - Info_{\text{Weather}}(D)$$

$$Gain(\text{Weather}) = 0.971 - 0.951 = \mathbf{0.02}$$

Step 3: Test Split for "Temp" (1/2)

Now let's try splitting by "Temp" instead.



This split looks much better! Both child nodes are **perfectly pure**.

Step 3: Calculate Gain for "Temp" (2/2)

Parent Entropy: $H(D) = 0.971$

Child Entropies:

- D_{Hot}: (0 Yes, 3 No)

$$H(D_H) = \mathbf{0} \quad \text{(Pure!)}$$

- D_{Mild}: (2 Yes, 0 No)

$$Info(D_M) = \mathbf{0} \quad \text{(Pure!)}$$

Weighted Average Entropy:

$$Info_{\text{Temp}}(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j)$$

$$= \left[\frac{3}{5} \times Info(D_H) + \frac{2}{5} \times Info(D_M)\right]$$

$$= [0.6(0) + 0.4(0)] = \mathbf{0}$$

Information Gain (Temp):

$$Gain(\text{Temp}) = Info(D) - Info_{\text{Temp}}(D)$$

$$Gain(\text{Temp}) = 0.971 - 0 = \mathbf{0.971}$$

Step 4: Compare and Conclude

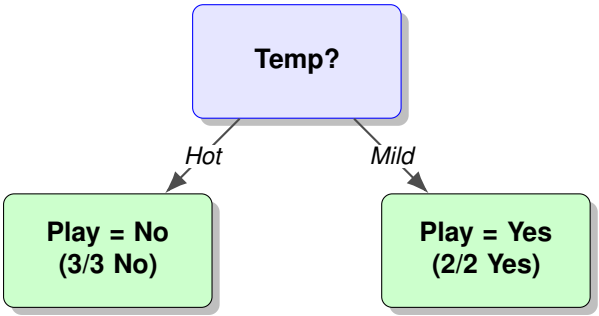
We compare the Information Gain from both potential splits.

Conclusion

- $Gain(D, \text{Temp}) = \mathbf{0.971}$
- $Gain(D, \text{Weather}) = \mathbf{0.02}$

Since $0.971 > 0.02$, the algorithm chooses **"Temp"** as the root node.

Final Tree Structure:



Step-by-Step: Finding the Root Node

Goal: Find which attribute ("Weather" or "Temp") is the best to split the data on first. We use **Gini Index** to measure this.

Mini-Dataset (D):

Weather	Temp	Play?
Sunny	Hot	No
Sunny	Mild	Yes
Rainy	Hot	No
Rainy	Mild	Yes
Sunny	Hot	No

Overall Data (D):

- 5 samples total
- 2 Play = Yes
- 3 Play = No

This is our starting point, or "Root Node". It's **impure** because it contains a mix of "Yes" and "No" outcomes.

Step 1: Calculate Parent Gini Index Gini(D)

We first measure the total impurity of the entire dataset.

Root Node Visualization:

Root Node (D)

5 Samples Total
2 Yes / 3 No

$$Gini(D) = 1 - \sum p_i^2$$

$$Gini(D) = 1 - \left((0.4)^2 + (0.6)^2\right)$$

$$Gini(D) = 1 - (0.16 + 0.36) = 1 - 0.52$$

$$Gini(D) = \mathbf{0.48}$$

Calculation:

- $p_{Yes} = 2/5 = 0.4$
- $p_{No} = 3/5 = 0.6$

Starting Impurity

Our starting impurity is **0.48**. The goal of a split is to reduce this number as much as possible. (0 = Pure).

Step 2: Calculate Gini Gain for "Weather"

Parent Gini: $Gini(D) = 0.48$

Child Gini Indices:

- D_{Sunny}: (1 Yes, 2 No)

$$Gini(D_S) = 1 - \left(\left(\frac{1}{3}\right)^2 + \left(\frac{2}{3}\right)^2\right)$$

$$= 1 - \left(\frac{1}{9} + \frac{4}{9}\right) = 1 - \frac{5}{9} = \mathbf{0.444}$$

- D_{Rainy}: (1 Yes, 1 No)

$$Gini(D_R) = 1 - \left(\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2\right)$$

Weighted Average Gini:

$$Gini_{\text{Weather}}(D) = \sum \frac{|D_j|}{|D|} \times Gini(D_j)$$

$$= \left[\frac{3}{5}Gini(D_S) + \frac{2}{5}Gini(D_R)\right]$$

$$= [0.6(0.444) + 0.4(0.5)]$$

$$= 0.266 + 0.2 = \mathbf{0.466}$$

Step 3: Calculate Gini Gain for "Temp"

Parent Gini: $Gini(D) = 0.48$

Child Gini Indices:

- D_{Hot}: (0 Yes, 3 No)

$$Gini(D_H) = 1 - \left(0^2 + 1^2\right) = \mathbf{0}$$

- D_{Mild}: (2 Yes, 0 No)

$$Gini(D_M) = 1 - \left(1^2 + 0^2\right) = \mathbf{0}$$

Weighted Average Gini:

$$Gini_{\text{Temp}}(D) = \sum \frac{|D_j|}{|D|} \times Gini(D_j)$$

$$= \left[\frac{3}{5}Gini(D_H) + \frac{2}{5}Gini(D_M)\right]$$

$$= [0.6(0) + 0.4(0)] = \mathbf{0}$$

Gini Gain (Temp):

$$\Delta Gini(\text{Temp}) = Gini(D) - Gini_{\text{Temp}}(D)$$

$$\Delta Gini(\text{Temp}) = 0.48 - 0 = \mathbf{0.48}$$

Step 4: Compare and Conclude

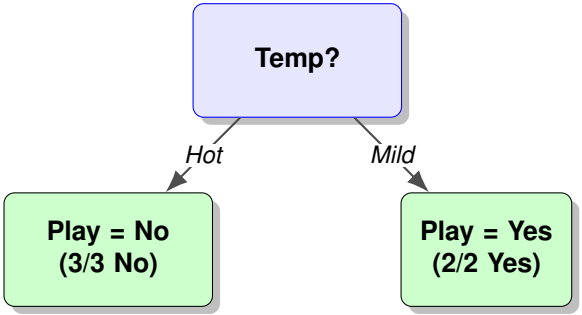
We compare the Gini Gain from both potential splits.

Conclusion

- $\Delta Gini(D, \text{Temp}) = \mathbf{0.48}$
- $\Delta Gini(D, \text{Weather}) = \mathbf{0.014}$

Since $0.48 > 0.014$, the algorithm chooses **"Temp"** as the root node.

Final Tree Structure:



How to Handle Continuous-Valued Attributes?

The Problem

Decision trees naturally split on **categorical** data (e.g., Weather = Sunny, Rainy, or Overcast).

But what about **continuous** data like Temperature (e.g., 65.2, 78.1, 85.5...)?

- We cannot create a unique branch for every single value.
- This would "memorize" the data (overfitting) and fail to generalize.

The Solution: Discretization

We must convert the continuous attribute into discrete intervals. The algorithm must find the **best split point** (e.g., 'Temp <= 72.5') to divide the data.

The "Best Split" Algorithm (Supervised)

This is the method used internally by algorithms like CART and C4.5.

1. **Sort:** Take all unique values of the attribute from the dataset and sort them in ascending order.
2. **Find Candidates:** Create a set of candidate split points. The most common method is to use the **midpoints** between all adjacent sorted values.
3. **Test Splits:** For *each* candidate split point S:
   - Divide the data into two sets: (Attribute $\leq S$) and (Attribute $> S$).
   - Calculate the impurity (Gini or Entropy) of this binary split.
4. **Select Best:** Choose the split point S that **maximizes the Gain** (or, equivalently, minimizes the weighted average impurity).

Result

The node becomes a binary question, e.g., "Is Temperature ≤ 72.5?".

Alternative: Discretize Continuous Values First

Before running the algorithm, you can "bin" continuous data to turn it into a categorical attribute.

Method 1: Equal Width Binning

Divide the total range (max - min) into k equal-sized bins. **Example (k=3):**

- Range: 65 to 90 (25 units)
- Bin Size: 25 / 3 $\approx$ 8.33
- **Bins:** [65, 73.3), [73.3, 81.6), [81.6, 90]
- **Labels:** "Low", "Medium", "High"

Method 2: Equal Frequency Binning

Divide the sorted data into k bins with (roughly) the same number of samples. **Example (k=3):**

- Data: {65, 70, 75, 80, 85, 90}
- 6 samples / 3 bins = 2 samples/bin
- **Bins:** {65, 70}, {75, 80}, {85, 90}
- **Labels:** "Bin 1", "Bin 2", "Bin 3"

Binning is (unsupervised) data pre-processing. The **"Best Split" method** is (supervised) and built directly into the tree algorithm. The "Best Split" method usually performs better because it uses the class labels (Yes/No) to find the most meaningful split.

Example (1/4): Using the "Best Split" Method

We will now use the **supervised "Best Split" method**.

Mini-Dataset (D):

Temp	Play?
65	No
70	No
75	Yes
80	Yes
85	Yes
90	No

Overall Data (D):

- 6 samples total
- 3 Play = Yes
- 3 Play = No

Parent Gini Calculation:

$$Gini(D) = 1 - \sum p_i^2$$

$$= 1 - \left(\left(\frac{3}{6}\right)^2 + \left(\frac{3}{6}\right)^2 \right)$$

$$= 1 - (0.5^2 + 0.5^2)$$

$$= 1 - (0.25 + 0.25) = \mathbf{0.5}$$

Example (2/4): Find Split Candidates

Step 1: Sort unique "Temp" values:

$$\{65, 70, 75, 80, 85, 90\}$$

Step 2: Find midpoints between adjacent values:

- $(65 + 70)/2 = \mathbf{67.5}$
- $(70 + 75)/2 = \mathbf{72.5}$
- $(75 + 80)/2 = \mathbf{77.5}$
- $(80 + 85)/2 = \mathbf{82.5}$
- $(85 + 90)/2 = \mathbf{87.5}$

Candidate Splits

We now have 5 binary splits to test:
(Temp $\leq$ 67.5), (Temp $\leq$ 72.5), (Temp $\leq$ 77.5), (Temp $\leq$ 82.5), (Temp $\leq$ 87.5)

Example (3/4): Test Each Split

We must calculate the Gini Gain for all 5 candidates. Let's try two of them:

Test 1: Split at $Temp \leq 72.5$
Left (≤ 72.5**):** {65(N), 70(N)}

- 0 Yes, 2 No $\rightarrow$ **Pure!**
- $Gini = \mathbf{0.0}$

Right (> 72.5**):** {75(Y), 80(Y), 85(Y), 90(N)}

- 3 Yes, 1 No
- $Gini = 1 - ((\frac{3}{4})^2 + (\frac{1}{4})^2) = \mathbf{0.375}$

Weighted Gini:

$$= \left(\frac{2}{6}\right)(0.0) + \left(\frac{4}{6}\right)(0.375) = \mathbf{0.25}$$

Gini Gain: $0.5 - 0.25 = \mathbf{0.25}$

Test 2: Split at $Temp \leq 87.5$
Left (≤ 87.5**):** {65(N), 70(N), 75(Y), 80(Y), 85(Y)}

- 3 Yes, 2 No
- $Gini = 1 - ((\frac{3}{5})^2 + (\frac{2}{5})^2) = \mathbf{0.48}$

Right (> 87.5**):** {90(N)}

- 0 Yes, 1 No $\rightarrow$ **Pure!**
- $Gini = \mathbf{0.0}$

Weighted Gini:

$$= \left(\frac{5}{6}\right)(0.48) + \left(\frac{1}{6}\right)(0.0) = \mathbf{0.4}$$

Gini Gain: $0.5 - 0.4 = \mathbf{0.1}$

Example (4/4): Select the Best Split

After testing all 5 candidate splits, we get the following results:

Split Point	Weighted Gini	Gini Gain ($0.5 - Gini$)
Temp $\leq$ 67.5	0.400	0.100
Temp $\leq$ 72.5	**0.250**	**0.250 (Max!)**
Temp $\leq$ 77.5	0.333	0.167
Temp $\leq$ 82.5	0.333	0.167
Temp $\leq$ 87.5	0.400	0.100

Conclusion

The split at **Temp** $\leq$ 72.5 gives the highest Gini Gain. This becomes the question for our root node.

An Important Note

The attribute "Temperature" is **not** "used up". The *same attribute* can be used again for a new split (with a different value) in a child node. (E.g., the branch for *Temp* $>$ 72.5 could be split again at *Temp* $\leq$ 87.5)

The Problem: Overfitting in Decision Trees

What is Overfitting?

A model overfits when it learns the training data too closely capturing both genuine patterns and **random noise**. As a result, it performs well on training data but poorly on unseen data.

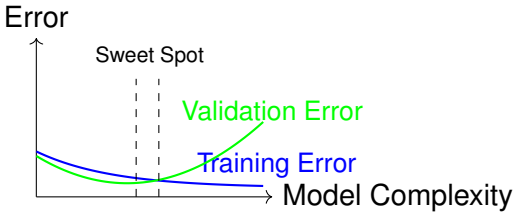
Typical Signs:

- Very high training accuracy
- Poor test accuracy
- Low bias but very high variance

In Decision Trees:

- Grows to full depth, creating overly specific branches
- Fits every training example, including outliers

Error vs. Model Complexity



Training error keeps decreasing, while validation error starts increasing after a point.

Visualizing Overfitting: Decision Boundaries

- Smooth, general decision boundary
- Correctly classifies most points
- Ignores minor noise better generalization

- Very complex, fragmented boundary
- Fits outliers and noise perfectly
- Fails badly on unseen data

The Solution: Pruning the Tree

Concept

Pruning simplifies a decision tree by removing branches that add little or no predictive power. It helps balance model complexity and accuracy.

Pre-Pruning (Early Stopping)

Stop early before overfitting occurs.

Post-Pruning (Cost-Complexity)

Grow fully then prune back.

Goal: Find the simplest tree with the best generalization performance.

Pre-Pruning (Early Stopping)

How It Works

Stop tree growth when any stopping rule (hyperparameter) is met:

`max_depth = k` Stop when tree depth reaches k.

`min_samples_leaf = n` Each leaf must contain at least n samples.

`min_impurity_decrease = x` Split only if impurity decreases by $\geq x$.

Pros

- Simple and efficient
- Faster training time

Cons

- Can stop too early
- Might miss beneficial deeper splits

Post-Pruning (Cost-Complexity)

How It Works

First, grow the full tree; then find the optimal subtree T that minimizes:

$$R_\alpha(T) = R(T) + \alpha|T|$$

- $R(T)$ Total misclassification error (training loss).
- $|T|$ Number of leaf nodes (tree size / complexity).
- α Complexity penalty controlling pruning aggressiveness.

Process: *Generate a sequence of subtrees for different α values and choose the best via cross-validation.*

Pros

- Achieves better biasvariance trade-off
- Typically yields higher test accuracy

Cons

- Computationally expensive
- Requires full tree + cross-validation

Why Do We Need Evaluation?

The Goal

The goal of a classifier is to **generalize** to new, unseen data. High accuracy on the data it was trained on is meaningless.

Warning: Never Test on Your Training Set!

A model will "memorize" the training data (overfitting). This leads to a 100% (or very high) accuracy on training data, but it will fail miserably on new data.

The Solution

Always assess a classifier's accuracy using a **separate test set** (or validation set) of class-labeled tuples that the model has **never seen before**.

The Foundation: The Confusion Matrix

For a binary classification problem (e.g., "Yes" or "No"), the confusion matrix tabulates a model's performance.

		Predicted Class	
		Class = Yes	Class = No
Actual Class	Class = Yes	**True Positive (TP)**	**False Negative (FN)**
	Class = No	**False Positive (FP)**	**True Negative (TN)**

- **TP:** Correctly predicted "Yes".
- **FN:** Incorrectly predicted "No" (it was "Yes"). **(Type II Error)**
- **FP:** Incorrectly predicted "Yes" (it was "No"). **(Type I Error)**
- **TN:** Correctly predicted "No".

Metric 1: Accuracy

How can we measure accuracy?

Accuracy is the percent of all predictions that were correct.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Example

A model classifies 100 tumors (60 malignant, 40 benign):

		Pred: Malignant	Pred: Benign
Actual	Malignant	TP = 50	FN = 10
Actual	Benign	FP = 5	TN = 35

$$\text{Accuracy} = \frac{50 + 35}{50 + 10 + 5 + 35} = \frac{85}{100} = \mathbf{85}\%$$

The Pitfall: Imbalanced Data

Accuracy can be very misleading.

Consider a dataset of 1000 people, where 990 are healthy and 10 have a rare disease.

Let's build a "dumb" classifier that **always predicts "Healthy"**.

"Dumb" Model Confusion Matrix

		Pred: Sick	Pred: Healthy
Actual	Sick	TP = 0	FN = 10
Actual	Healthy	FP = 0	TN = 990

$$\text{Accuracy} = \frac{0 + 990}{0 + 10 + 0 + 990} = \frac{990}{1000} = \mathbf{99}\%$$

This model has 99% accuracy but is **completely useless**, as it fails to identify a single sick patient.

Other Metrics to Consider (1/2)

Precision (Positive Predictive Value)

Of all the times the model predicted "Yes", what percentage was correct?

$$\text{Precision} = \frac{TP}{TP + FP}$$

Use when: The cost of a **False Positive** is high. (e.g., spam detection: you don't want to mark a real email as spam).

Recall (Sensitivity / True Positive Rate)

Of all the actual "Yes" cases, what percentage did the model find?

$$\text{Recall} = \frac{TP}{TP + FN}$$

Use when: The cost of a **False Negative** is high. (e.g., medical diagnosis: you don't want to miss a sick patient).

Other Metrics to Consider (2/2)

F1-Score (Harmonic Mean)

A single score that balances both Precision and Recall. It is the harmonic mean, which punishes extreme (very low) values.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

This is often the best "single number" metric for imbalanced classes.

Specificity (True Negative Rate)

Of all the actual "No" cases, what percentage did the model correctly identify?

$$\text{Specificity} = \frac{TN}{TN + FP}$$

This is the "Recall" for the negative class.

Methods for Estimating a Classifier's Accuracy

How do we split our data to get a reliable estimate of performance?

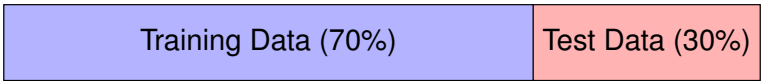
1. The Holdout Method
2. Cross-Validation
3. The Bootstrap

All these methods are ways to generate a test set, which is then used to build the confusion matrix and calculate the metrics.

Estimator 1: The Holdout Method

The simplest method.

1. **Split:** Divide the initial labeled data into two disjoint sets:
   - Training Set (e.g., 70-80% of data)
   - Test Set (e.g., 20-30% of data)
2. **Train:** Build the model using *only* the training set.
3. **Test:** Evaluate the model on the *test set* to get a performance estimate.



Problems

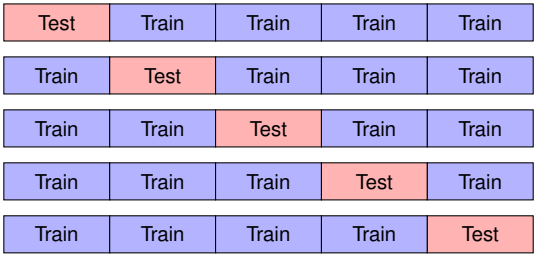
- **Wastes data:** We don't get to train on 30% of our data.
- **High Variance:** The estimate depends heavily on *which* 30% ended up in the test set. A "lucky" or "unlucky" split can give a misleadingly high or low score.

Estimator 2: k-Fold Cross-Validation

The standard solution to the Holdout method's problems.

Process (e.g., 5-Fold CV)

1. **Split:** Divide the data into k (e.g., 5) equal "folds".
2. **Iterate:** Run k experiments:
   - **Fold 1:** Test on F 1, Train on F 2, 3, 4, 5.
   - ... ...
   - **Fold 5:** Test on F 5, Train on F 1, 2, 3, 4.
3. **Average:** The final performance is the **average** of the 5 individual scores.



Advantages

Much more robust estimate. Mitigates the "unlucky split" problem. Uses all data for both training and testing.

Estimator 3: The Bootstrap

Process (Used when the dataset is small)

1. **Sample:** Given a dataset D of n tuples, create a new "bootstrap sample" D_i by sampling n tuples from D **with replacement**.
2. **Split:** D_i becomes the training set. The tuples from D that were *not* selected for D_i form the test set (the "out-of-bag" samples).
3. **Train & Test:** Train the model on D_i and test it on the out-of-bag samples.
4. **Repeat:** Repeat this process many times (e.g., $b = 200$ times).
5. **Average:** The final accuracy is the average of all b test accuracies.

Example of Bootstrap Sampling (n=10)

Original D: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Sample D_1: {2, 1, 7, 10, 2, 5, 9, 1, 6, 5} $\rightarrow$ *Train*
Test Set 1: {3, 4, 8} $\rightarrow$ *Test*

Comparing Classifiers (1/2): ROC Curves

Receiver Operating Characteristic (ROC) Curve

Many classifiers don't just output "Yes" or "No". They output a *probability* (e.g., 0.85). We then use a **threshold** (e.g., > 0.5) to make the decision.
An ROC curve visualizes a classifier's performance across **all possible thresholds**.

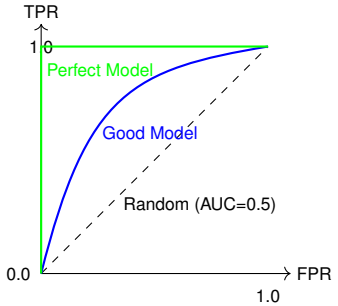
ROC Axes:

- **Y-Axis:** True Positive Rate (Recall)

$$\text{TPR} = \frac{TP}{TP + FN}$$

- **X-Axis:** False Positive Rate

$$\text{FPR} = \frac{FP}{FP + TN}$$



Comparing Classifiers (2/2): AUC

Area Under the Curve (AUC)

It's hard to compare two ROC curves just by looking. The **Area Under the Curve (AUC)** converts the entire curve into a single number.

Interpreting AUC:

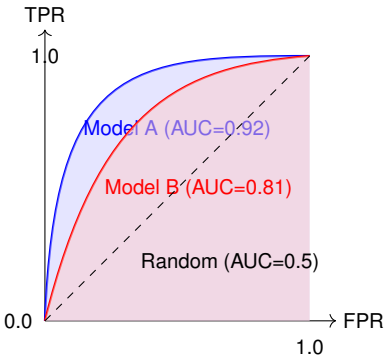
- **AUC = 1.0:** Perfect classifier.
- **AUC = 0.5:** Random guessing.
- **AUC < 0.5:** Worse than random (model is backwards).

Comparison

You have two models:

- Model A (AUC = 0.92)
- Model B (AUC = 0.81)

Conclusion: Model A is better at distinguishing between "Yes" and "No".



Summary of Evaluation

- **Never test on training data.** Always use a separate test set.
- **Accuracy is not enough.** It is misleading for imbalanced datasets.
- **Use the right metric.**
  - Use **Precision** if False Positives are "expensive".
  - Use **Recall** if False Negatives are "expensive".
  - Use **F1-Score** for a balance.
- **Use the right estimation method.**
  - **Holdout** is simple but has high variance.
  - **k-Fold Cross-Validation** is the industry standard.
  - **Bootstrap** is good for very small datasets.
- **Use ROC/AUC to compare models.** It provides a comprehensive view of a model's performance across all thresholds.

What are Ensemble Methods?

The Core Idea: "Wisdom of the Crowd"

Ensemble methods combine the predictions of multiple machine learning models (called "weak learners") to create a single, more robust, and accurate model (a "strong learner").

Why use them?

- **Reduce Variance:** Less sensitive to the specific training data. (e.g., Bagging)
- **Reduce Bias:** Correct for the errors of previous models. (e.g., Boosting)
- **Improve Performance:** By blending the strengths of different models. (e.g., Stacking)

Model 1 $\rightarrow$ Prediction 1
Model 2 $\rightarrow$ Prediction 2
Model 3 $\rightarrow$ Prediction 3

$\Downarrow$

Final Ensemble Prediction

Bagging (Bootstrap Aggregating)

How it Works: Parallel Training

The goal is to reduce variance by averaging out the noise.

1. **Bootstrap:** Create N random subsets of the training data (sampling *with replacement*).
2. **Train:** Train N models (e.g., Decision Trees) in parallel, one on each subset.
3. **Aggregate:** Combine the results.
   - **Classification:** Majority Voting
   - **Regression:** Averaging

Key Example: Random Forest

A Random Forest is a Bagging method using Decision Trees. It adds one more layer of randomness: at each split, the tree only considers a *random subset of features*.

Boosting

How it Works: Sequential Training

The goal is to reduce bias by learning from mistakes.

1. Train a simple model (Learner 1) on the data.
2. Identify misclassified samples. Increase their "weight" or importance.
3. Train a new model (Learner 2) that is forced to focus on the high-weight (difficult) samples.
4. Repeat, with each new learner correcting the errors of the previous ones.
5. **Aggregate:** Combine all learners using a weighted vote (better-performing learners get a higher say).

Key Examples

AdaBoost (Adaptive Boosting), **Gradient Boosting** (GBM), **XGBoost**

Stacking (Stacked Generalization)

How it Works: Learning to Combine

The goal is to blend different models to capture different patterns in the data.

Level 0: Base Learners

- Train several *different* models (e.g., SVM, Random Forest, k-NN) on the training data.
- Generate their predictions for the data (often using k-fold cross-validation to prevent data leakage).

Level 1: Meta-Learner

- The predictions from Level 0 are now used as input features for a new, final model.
- This "meta-learner" (e.g., Logistic Regression) learns the best way to combine the base learner predictions.

$$(\text{Data} \rightarrow [\text{SVM, RF, } k\text{-NN}]) \rightarrow (\text{Predictions}) \rightarrow [\text{Logistic Regression}] \rightarrow \text{Final Output}$$

Ensemble Comparison

Feature	Bagging	Boosting	Stacking
Model Building	Parallel	Sequential	Parallel, then Sequential
Main Goal	Reduce Variance	Reduce Bias	Improve Predictions
Model Type	Homogeneous	Homogeneous	Heterogeneous
Key Idea	Averaging	Weighted Voting	Learning to Combine
Example	Random Forest	XGBoost	Custom Blends

The Challenge: Imbalanced Data

What is it?

A classification dataset where one class is much more frequent than the other.

- **Majority Class:** The common one.
- **Minority Class:** The rare one.

Examples

- Fraud Detection (99.9% Not Fraud)
- Medical Diagnosis (98% Healthy)
- Ad Click-Through (99.5% No Click)

The "Accuracy Paradox"

- Imagine a 99% / 1% split.
- A "dumb" model that always predicts the **majority class** is 99% accurate.
- ...but it is completely useless, as it never finds the minority class (which is usually the one we care about!)

Solution: Stop using accuracy. Focus on metrics like **Precision**, **Recall**, **F1-Score**, and **AUC-ROC**.

Strategy 1: Data-Level Solutions (Resampling)

Undersampling

- **What:** Randomly remove samples from the majority class.
- **Pro:** Reduces dataset size, speeds up training.
- **Con:** Can lose important information and patterns from the majority class.

Oversampling

- **What:** Randomly duplicate samples from the minority class.
- **Pro:** No information is lost.
- **Con:** Can lead to overfitting, as the model sees the same exact samples multiple times.

A Better Way: SMOTE

SMOTE (Synthetic Minority Over-sampling Technique)

- Instead of duplicating, it creates new synthetic samples.
- It selects a minority sample, finds its neighbors, and creates a new point on the line segment between them.
- This provides new, plausible data and avoids simple overfitting.

Strategy 2 : Algorithm Solutions

Algorithm-Level (Cost-Sensitive Learning)

- **What:** Modify the model's loss function to penalize misclassifying the minority class more heavily.
- **Example:** Tell the model that a "False Negative" (missing a fraud case) is 100× worse than a "False Positive" (flagging a good transaction).
- Many models have a `class_weight='balanced'` parameter that does this automatically.

Strategy 3: Ensemble Solutions

Ensemble-Level Solutions

- **Balanced Bagging (e.g., Balanced Random Forest):**
- Each "bag" (bootstrap sample) is created by undersampling the majority class.
- The final ensemble is trained on many different, balanced datasets.
- **Boosting (e.g., AdaBoost):**
- Boosting algorithms naturally increase the weight of misclassified (hard) samples.
- The minority class is often "hard," so the model automatically learns to focus on it.

Summary of Key Points

- Decision Trees use impurity measures (Entropy, Gini).
- Evaluation uses metrics beyond accuracy.
- Ensemble methods improve robustness.
- Imbalanced data needs resampling or cost-sensitive methods.
- ROC/AUC provides a fair comparison between models.

References

1. J. Han, M. Kamber, J. Pei, *Data Mining: Concepts and Techniques*, 4th Ed., Morgan Kaufmann, 2012.
2. D. J. Hand, H. Mannila, P. Smyth, *Principles of Data Mining*, A Bradford Book, 2001.
3. R. O. Duda, P. E. Hart, D. G. Stork, *Pattern Classification*, 2nd Ed., Wiley, 2001.