Northeastern University
Khoury College of
Computer Sciences

Decision Trees & Ensemble Learning

DS 4400 | Machine Learning and Data Mining I

Zohair Shafi

Spring 2026

Monday | March 9th, 2026

Today’s Outline

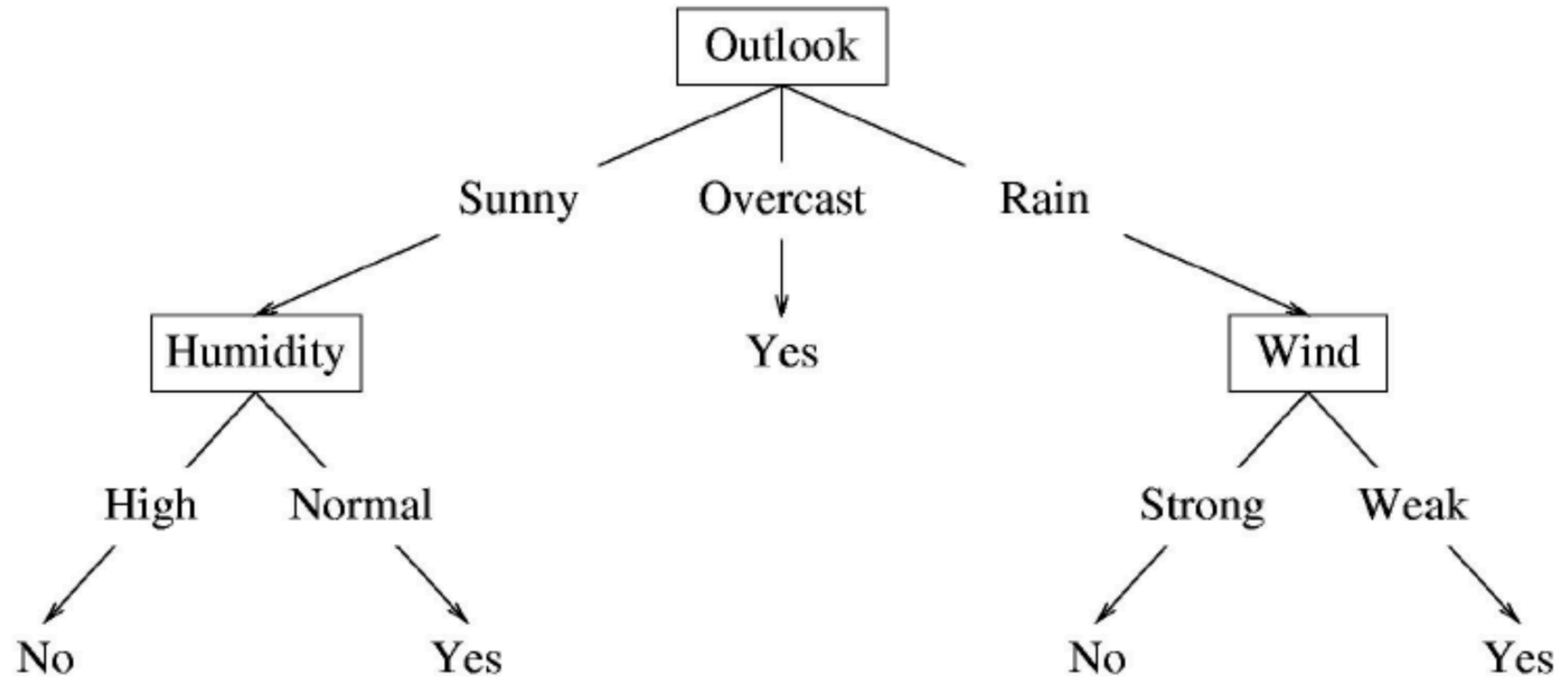
- Decision Trees
- Ensemble Learning
    - Bagging
    - Boosting

Decision Trees

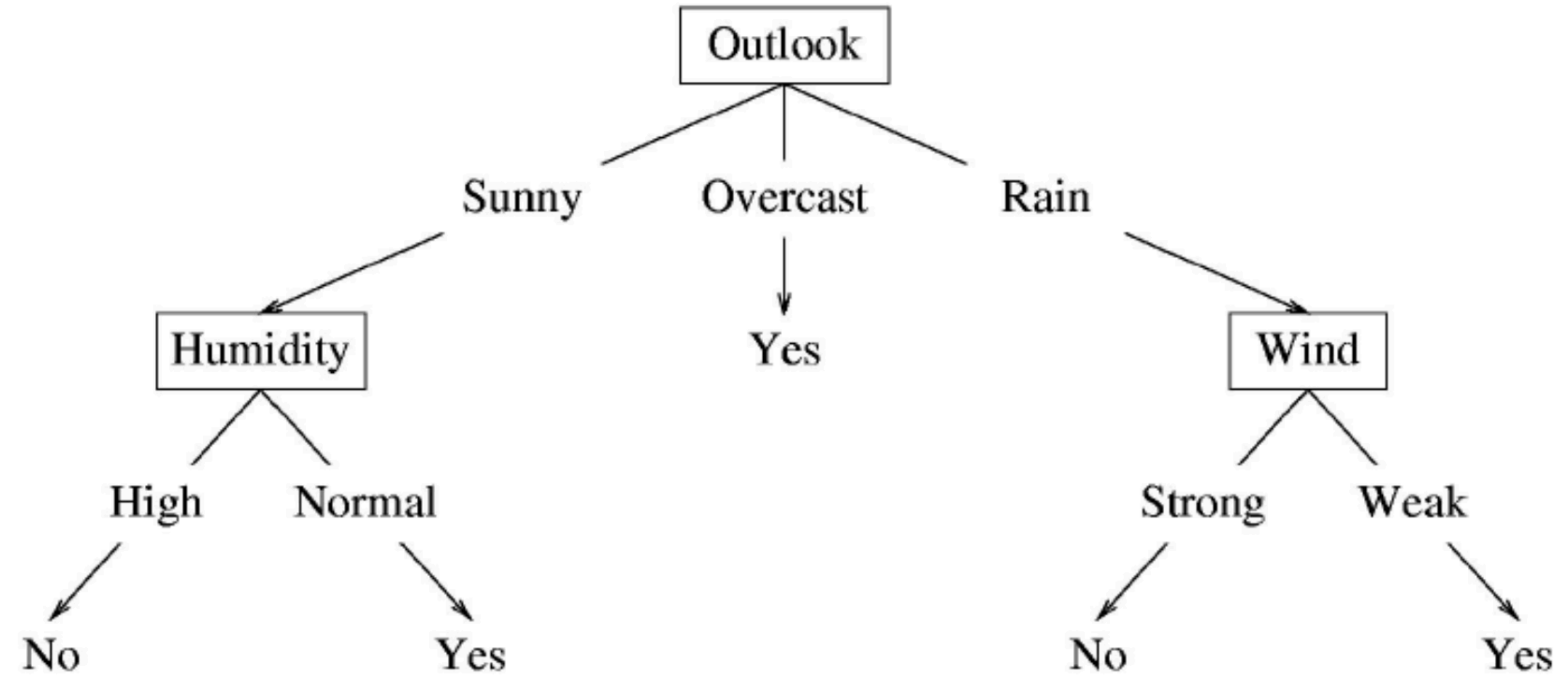
Play Outside? (y)	Humidity (X1)	Wind (X2)	Outlook (X3)
Yes	Normal	Strong	Sunny
Yes	Low	Weak	Overcast
No	High	Weak	Rain

Decision Trees

Play Outside? (y)	Humidity (X1)	Wind (X2)	Outlook (X3)
Yes	Normal	Strong	Sunny
Yes	Low	Weak	Overcast
No	High	Weak	Rain



Decision Trees



- Decision trees recursively partition the feature space using simple rules, creating a tree structure that mirrors human decision-making.
- Each internal node is a test on a feature x_i
- Each branch is an outcome of the test (or selects a value for x_i)
- Each leaf is a class prediction Y

Decision Trees

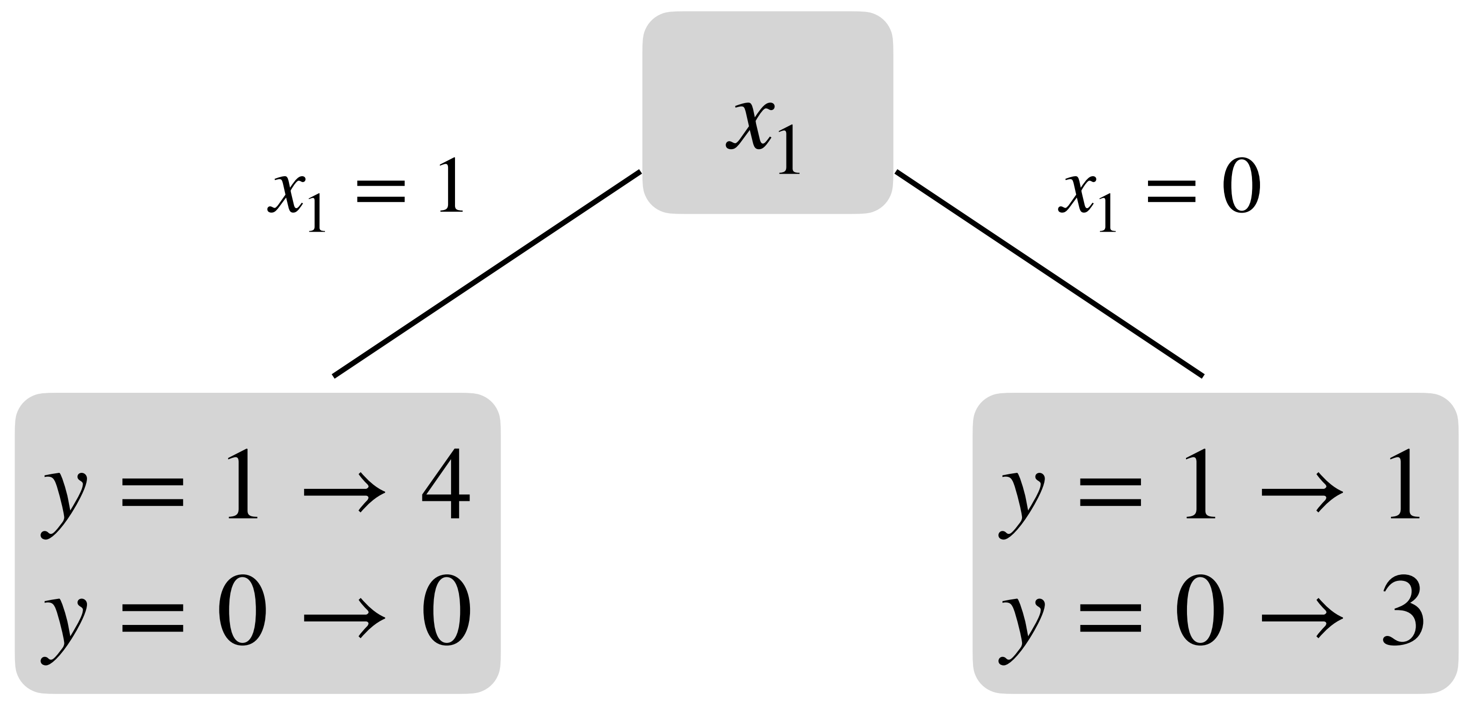
- Learning “optimal”, i.e., the simplest and smallest decision trees are an NP-complete problem.
- We resort to a greedy heuristic
  - Start from an empty tree
  - Of all available features x_i, split on the **best feature**
  - Recurse

Decision Trees

y	x_1	x_2
1	1	1
1	1	0
1	1	1
1	1	0
1	0	1
0	0	0
0	0	1
0	0	0

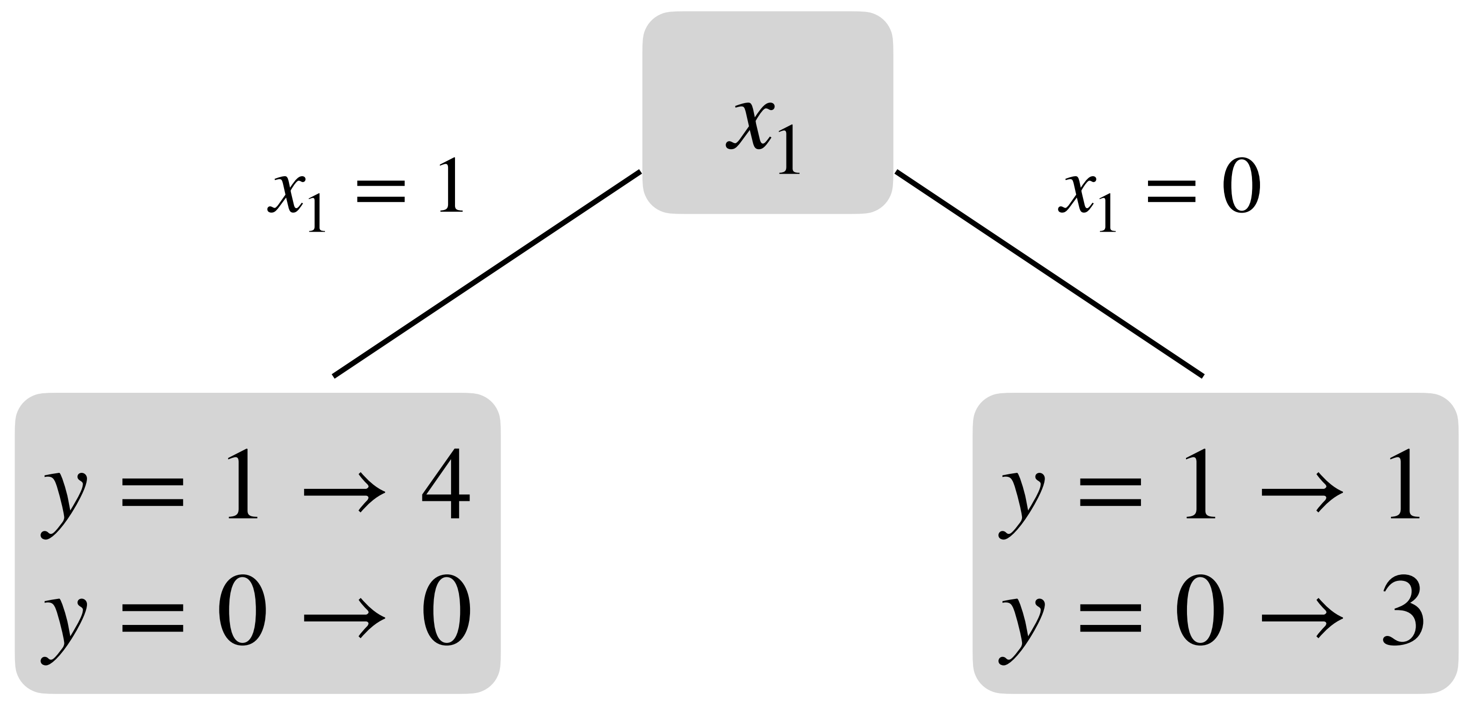
Decision Trees

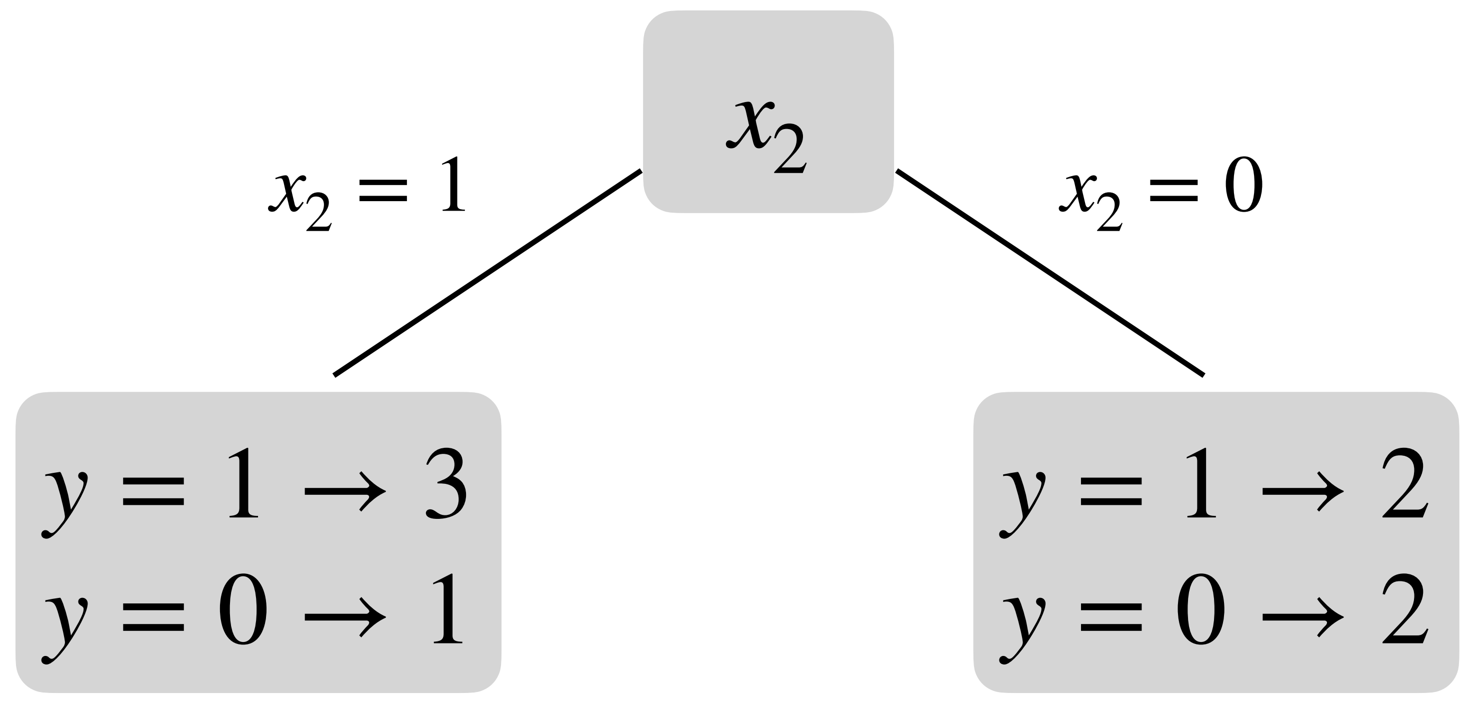
y	x_1	x_2
1	1	1
1	1	0
1	1	1
1	1	0
1	0	1
0	0	0
0	0	1
0	0	0



Decision Trees

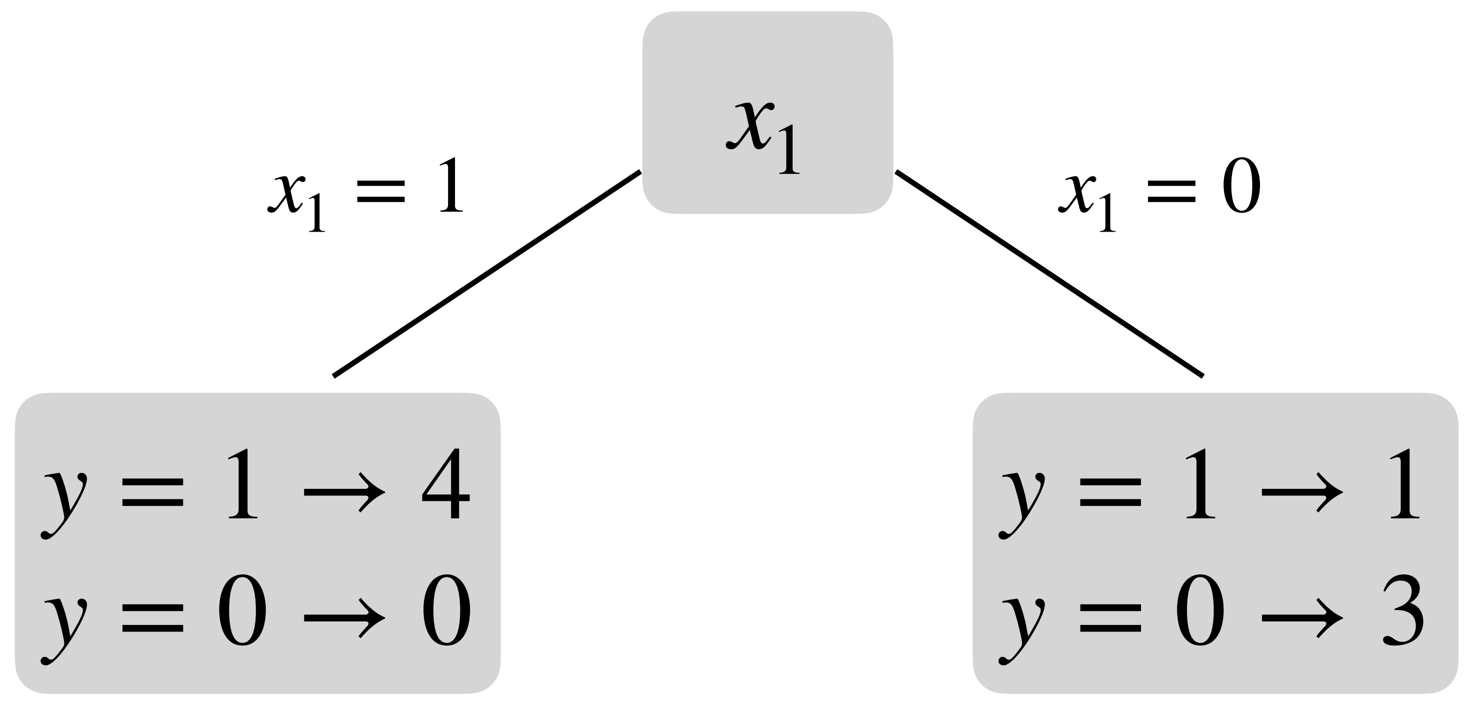
y	x_1	x_2
1	1	1
1	1	0
1	1	1
1	1	0
1	0	1
0	0	0
0	0	1
0	0	0

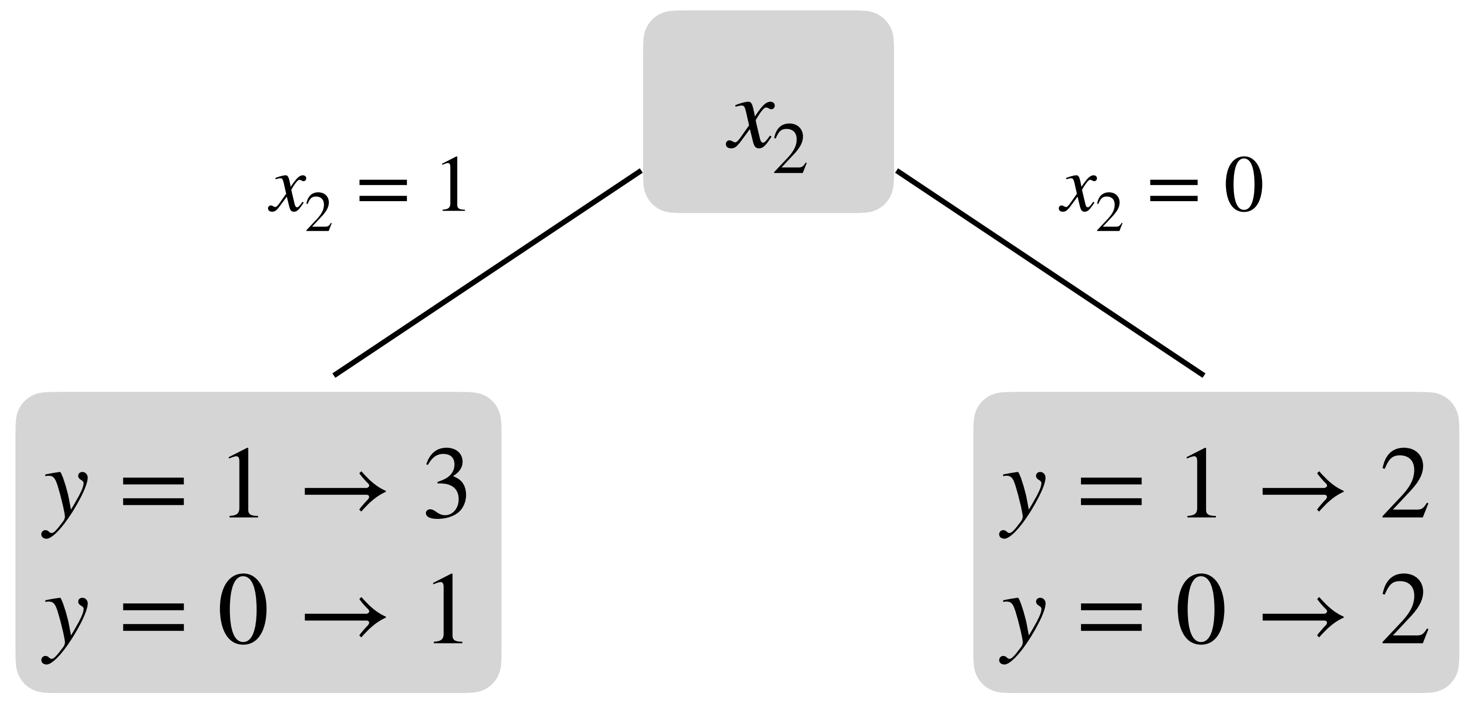




Decision Trees

y	x_1	x_2
1	1	1
1	1	0
1	1	1
1	1	0
1	0	1
0	0	0
0	0	1
0	0	0





How do we decide which is better?

Entropy Measures - Information Gain

Use counts at leaf nodes to define probabilities, so we can measure uncertainty

Entropy

- Entropy measures **uncertainty** or **surprise**.
- The **more uncertain** you are about an outcome, the **higher the entropy**.
- **Scenario 1**: A coin that always lands heads.

Entropy

- Entropy measures **uncertainty** or **surprise**.
- The **more uncertain** you are about an outcome, the **higher the entropy**.
- **Scenario 1**: A coin that always lands heads.
  - Before flipping, are you uncertain about the outcome?
  - No. You know it will be heads. There's no surprise, no uncertainty.
  - Entropy = 0 (minimum)

Entropy

- Entropy measures **uncertainty** or **surprise**.
- The **more uncertain** you are about an outcome, the **higher the entropy**.
- **Scenario 2**: A fair coin (50% heads, 50% tails).

Entropy

- Entropy measures **uncertainty** or **surprise**.
- The **more uncertain** you are about an outcome, the **higher the entropy**.
- **Scenario 2**: A fair coin (50% heads, 50% tails).
  - Before flipping, are you uncertain?
  - Yes. You genuinely don't know what will happen. Maximum surprise possible for two outcomes.
  - Entropy = 1 bit (maximum for binary outcome)

Entropy

- Entropy measures **uncertainty** or **surprise**.
- The **more uncertain** you are about an outcome, the **higher the entropy**.
- **Scenario 3**: A biased coin (90% heads, 10% tails).
  - Some uncertainty, but not much. You'd bet on heads and usually be right.
  - Entropy = 0.47

Entropy

- Entropy measures **uncertainty** or **surprise**.
- The **more uncertain** you are about an outcome, the **higher the entropy**.
- **<u>Entropy is maximized when all outcomes are equally likely.</u>**
- For outcomes with probabilities $p_1, p_2, \ldots, p_n$:

$$H = -\sum_i p_i log_2(p_i)$$

Entropy

- Why log formulation?
- For outcomes with probabilities $p_1, p_2, \ldots, p_n$:

$$H = -\sum_i p_i log_2(p_i)$$

<u>1. We want to measure how **surprising** some outcome is</u>

If $p_1 < p_2$, then surprise$(p_1) >$ surprise(p_2)

Entropy

- Why log formulation?
- For outcomes with probabilities $p_1, p_2, \ldots, p_n$:

$$H = -\sum_i p_i log_2(p_i)$$

1. We want to measure how **surprising** some outcome is

If $p_1 < p_2$, then surprise$(p_1) >$ surprise(p_2)

2. Certain events have 0 surprise, i.e., events guaranteed to happen

If $p_1 = 1$, surprise$(p_1) = 0$

Entropy

- Why log formulation?
- For outcomes with probabilities $p_1, p_2, ..., p_n$:

$$H = -\sum_i p_i log_2(p_i)$$

1. We want to measure how **surprising** some outcome is

If $p_1 < p_2$, then surprise$(p_1) >$ surprise(p_2)

2. Certain events have 0 surprise, i.e., events guaranteed to happen

If $p_1 = 1$, surprise$(p_1) = 0$

3. Surprise of independent events should add up

If p_1 and p_2 are independent, then surprise$(p_1, p_2) =$ surprise$(p_1) +$ surprise(p_1)

The log function hits all three conditions exactly

Entropy

- Why log formulation?
- For outcomes with probabilities $p_1, p_2, \ldots, p_n$:

$$H = -\sum_i p_i log_2(p_i)$$

<u>1. We want to measure how **surprising** some outcome is</u>

If $p_1 < p_2$, then surprise$(p_1) >$ surprise(p_2)

<u>2. Certain events have 0 surprise, i.e., events guaranteed to happen</u>

If $p_1 = 1$, surprise$(p_1) = 0$

<u>3. Surprise of independent events should add up</u>

If p_1 and p_2 are independent, then surprise$(p_1, p_2) =$ surprise$(p_1) +$ surprise(p_1)

Entropy

- For outcomes with probabilities $p_1, p_2, \ldots, p_n$:

$$H = -\sum_i p_i log_2(p_i)$$

- Why “Bits” and log_2?
    - Entropy answers: “How many **yes/no** questions do I need to identify the outcome?”
    - Fair coin: 1 question (“Is it heads?”) - 1 bit
    - Four equally likely outcomes: 2 questions - 2 bits
        - “Is it in the first half?”
        - “Is it the first of those two?”
- In general, n equally likely outcomes = $log_2(n)$ bits

Entropy

- Why Entropy Matters in Decision Trees
  - We want splits that **reduce uncertainty** about the class label.
  - A split that creates pure nodes (all one class) reduces entropy to zero.
    - **Before split:** Mixed classes, high entropy
    - **After good split**: Purer nodes, lower entropy
    - Information gain = entropy reduction

Entropy

- Node 1 has 75% class A, 25% class B:

$$H = -0.75log_2(0.75) - 0.25log_2(0.25) = 0.811 \text{ bits}$$

Entropy

- Node 1 has 75% class A, 25% class B:

$$H = -0.75log_2(0.75) - 0.25log_2(0.25) = 0.811 \text{ bits}$$

- Node 2 has 50% class A and 50% class B:

$$H = -0.5log_2(0.5) - 0.5log_2(0.5) = 1 \text{ bit}$$

- Node 1 has **lower entropy** (less uncertainty) than node 2.

Measuring Split Quality: Impurity Functions

- A good split separates classes.
- We measure node **impurity** - how mixed the classes are - and choose splits that maximize impurity reduction.

Measuring Split Quality: Impurity Functions

Entropy

- Let p_k be the proportion of class k samples in a node

$$Entropy(D) = -\sum_{k=1}^{K} p_k log_2(p_k)$$

- **Interpretation:** Expected number of bits needed to encode the class of a random sample.

Measuring Split Quality: Impurity Functions

Entropy

- Let p_k be the proportion of class k samples in a node

$$Entropy(D) = -\sum_{k=1}^{K} p_k log_2(p_k)$$

- **Interpretation:** Expected number of bits needed to encode the class of a random sample.
- **Properties**:
  - Minimum = 0 when node is pure
  - Maximum = $log_2(K)$ when uniform distribution
  - For binary: max = 1 bit at p = 0.5

Measuring Split Quality: Impurity Functions

Entropy

- Let p_k be the proportion of class k samples in a node

$$Entropy(D) = -\sum_{k=1}^{K} p_k log_2(p_k)$$

- **Interpretation:** Expected number of bits needed to encode the class of a random sample.
- **Properties**:
  - Minimum = 0 when node is pure
  - Maximum = $log_2(K)$ when uniform distribution
  - For binary: max = 1 bit at p = 0.5

- **Example (binary classification):**
  - Pure node: **Entropy = 0**
  - 50-50 split: Entropy = -0.5log_2(0.5) - 0.5log_2(0.5) = 1 bit
  - 90-10 split: Entropy = -0.9 log_2(0.9) - 0.1 log_2(0.1) ≈ 0.47 bits

Measuring Split Quality: Impurity Functions

Gini Impurity

- Let p_k be the proportion of class k samples in a node

$$Gini(D) = 1 - \sum_{k=1}^{K} p_k^2 = \sum_{k=1}^{K} p_k(1 - p_k)$$

Measuring Split Quality: Impurity Functions

Gini Impurity

- Let p_k be the proportion of class k samples in a node

$$Gini(D) = 1 - \sum_{k=1}^{K} p_k^2 = \sum_{k=1}^{K} p_k(1 - p_k)$$

- **Interpretation**: Probability of misclassifying a randomly chosen sample if labeled according to the class distribution.

Measuring Split Quality: Impurity Functions

Gini Impurity

- Let p_k be the proportion of class k samples in a node

$$Gini(D) = 1 - \sum_{k=1}^{K} p_k^2 = \sum_{k=1}^{K} p_k(1 - p_k)$$

- **Interpretation**: Probability of misclassifying a randomly chosen sample if labeled according to the class distribution.
- **Properties**:
  - Minimum = 0 when node is pure (all one class)
  - Maximum = $1 - \frac{1}{K}$ when classes are equally distributed
  - For binary: max = 0.5 at p = 0.5

Measuring Split Quality: Impurity Functions

Gini Impurity

- Let p_k be the proportion of class k samples in a node

$$Gini(D) = 1 - \sum_{k=1}^{K} p_k^2 = \sum_{k=1}^{K} p_k(1 - p_k)$$

- **Interpretation**: Probability of misclassifying a randomly chosen sample if labeled according to the class distribution.

- **Properties**:
  - Minimum = 0 when node is pure (all one class)
  - Maximum = $1 - \frac{1}{K}$ when classes are equally distributed
  - For binary: max = 0.5 at p = 0.5

- **Example (binary classification):**
  - Node with 100% class A: Gini = 1 - 1^2 = **0 (pure)**
  - Node with 50% each: Gini = 1 - 0.5^2 - 0.5^2 = 0.5 (maximum impurity)
  - Node with 90% class A: Gini = 1 - 0.9^2 - 0.1^2 = 0.18

Measuring Split Quality: Impurity Functions

Information Gain

- We want splits that reduce **impurity** (i.e., Gini, Entropy).
- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{classes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$

Measuring Split Quality: Impurity Functions

Information Gain

- We want splits that reduce **impurity** (i.e., Gini, Entropy).
- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \overset{\text{weighted average impurity of child nodes}}{\boxed{\sum_{k \in \text{classes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)}}$$

Measuring Split Quality: Impurity Functions
Information Gain

- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$

100 Days
60 Play, 40 Don’t Play

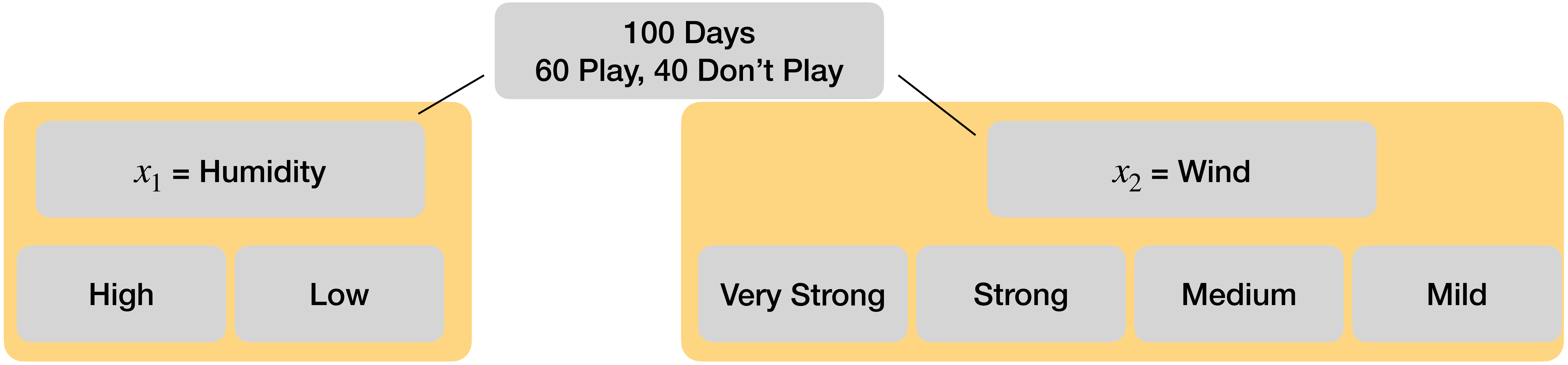
Measuring Split Quality: Impurity Functions

Information Gain

Step 1: Decide between Wind and Humidity

- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$

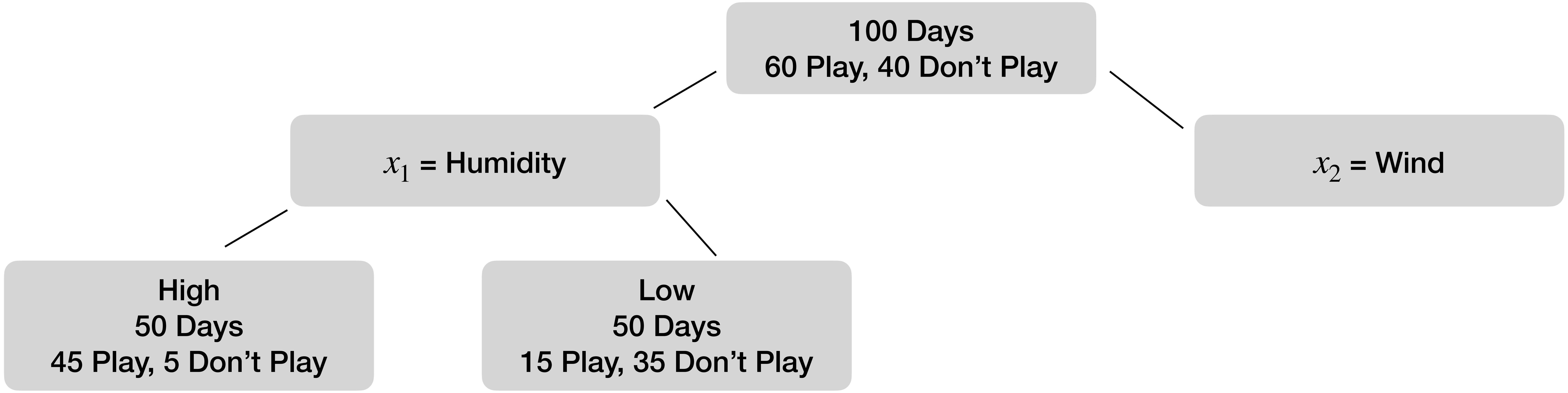


Measuring Split Quality: Impurity Functions

Information Gain

- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$



$H = -0.9 log_2(0.9) - 0.1 log_2(0.1) = 0.469 \ bits$

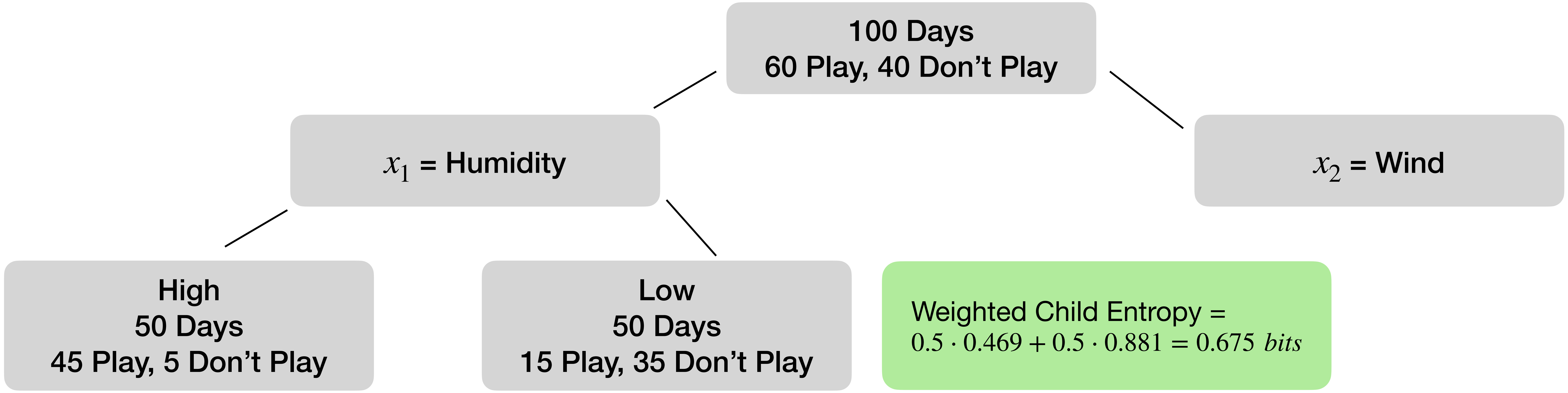
$H = -0.3 log_2(0.3) - 0.7 log_2(0.7) = 0.881 \ bits$

Measuring Split Quality: Impurity Functions

Information Gain

- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$



Weighted Child Entropy =
$0.5 \cdot 0.469 + 0.5 \cdot 0.881 = 0.675 \ bits$

$H = -0.9log_2(0.9) - 0.1log_2(0.1) = 0.469 \ bits$

$H = -0.3log_2(0.3) - 0.7log_2(0.7) = 0.881 \ bits$

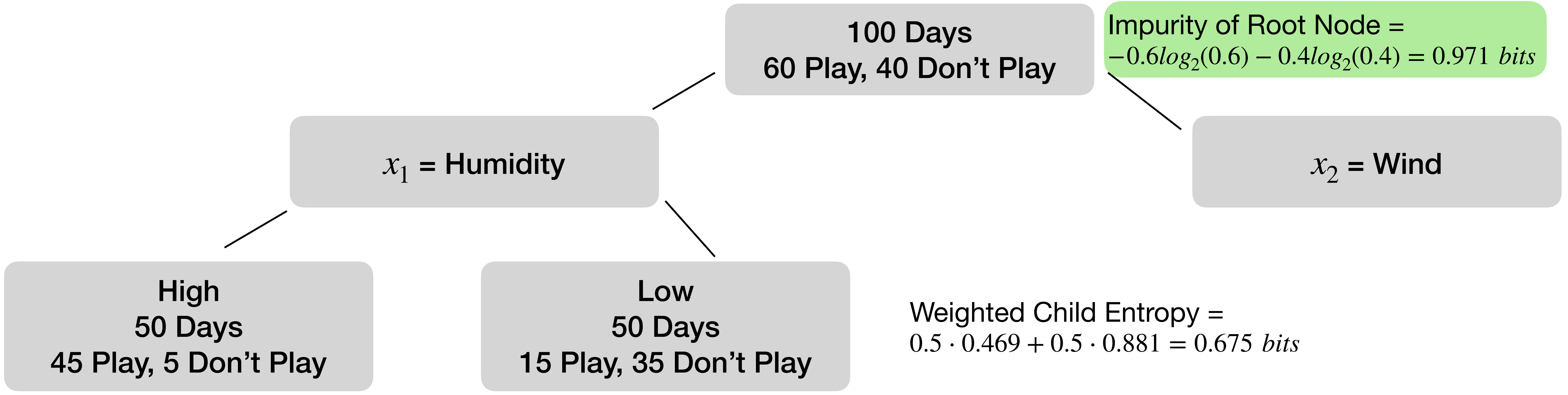
Measuring Split Quality: Impurity Functions

Information Gain

- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$

Impurity of Root Node =
$-0.6log_2(0.6) - 0.4log_2(0.4) = 0.971 \ bits$



Weighted Child Entropy =
$0.5 \cdot 0.469 + 0.5 \cdot 0.881 = 0.675 \ bits$

$H = -0.9log_2(0.9) - 0.1log_2(0.1) = 0.469 \ bits$

$H = -0.3log_2(0.3) - 0.7log_2(0.7) = 0.881 \ bits$

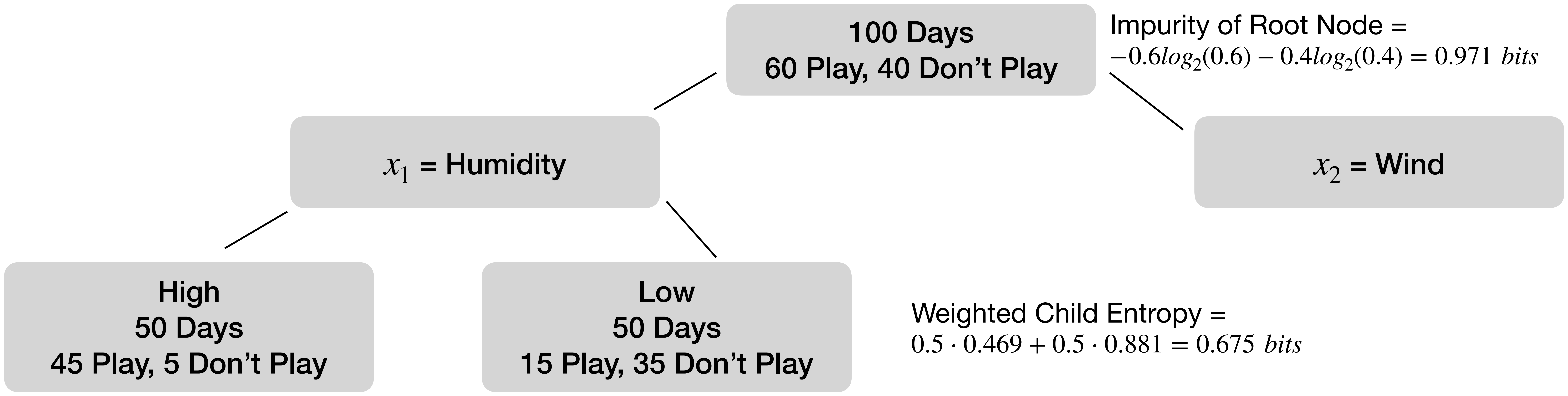
Measuring Split Quality: Impurity Functions

Information Gain

Information Gain = 0.971 - 0.675 = **0.296 bits**

- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$



Impurity of Root Node =
$-0.6 log_2(0.6) - 0.4 log_2(0.4) = 0.971 \ bits$

Weighted Child Entropy =
$0.5 \cdot 0.469 + 0.5 \cdot 0.881 = 0.675 \ bits$

$H = -0.9 log_2(0.9) - 0.1 log_2(0.1) = 0.469 \ bits$

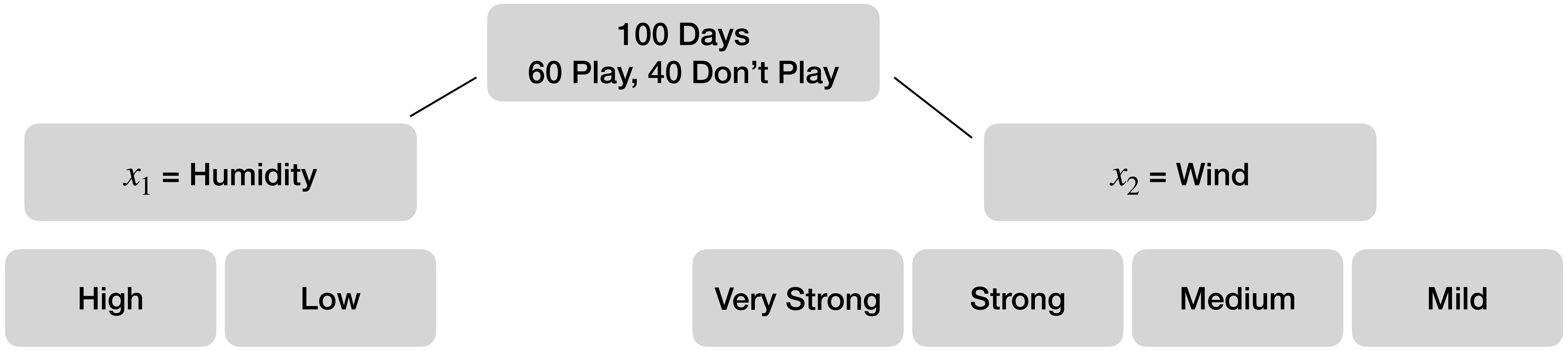
$H = -0.3 log_2(0.3) - 0.7 log_2(0.7) = 0.881 \ bits$

Measuring Split Quality: Impurity Functions

Information Gain

- Information gain measures this reduction:

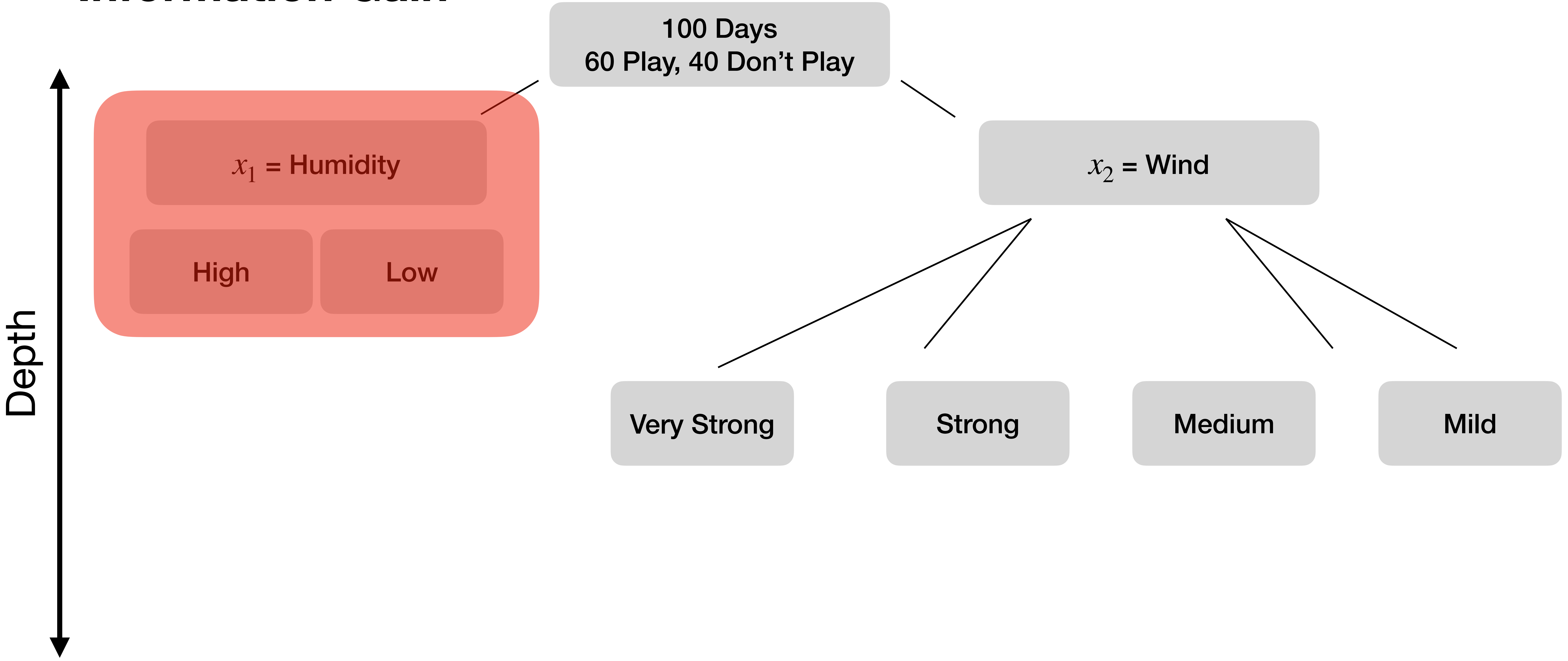
$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$



IG = 0.576

Measuring Split Quality: Impurity Functions

Information Gain



Measuring Split Quality: Impurity Functions

Information Gain

14 Days
6 Play, 4 Don't Play

Play	Humidity	Wind
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	High	Weak
1	High	Weak
1	High	Weak
1	High	Weak
0	High	Strong
0	High	Strong
0	High	Strong
0	High	Strong

Measuring Split Quality: Impurity Functions

Information Gain

14 Days
6 Play, 4 Don’t Play

Play	Humidity	Wind
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	High	Weak
1	High	Weak
1	High	Weak
1	High	Weak
0	High	Strong
0	High	Strong
0	High	Strong
0	High	Strong

Measuring Split Quality: Impurity Functions

Information Gain

14 Days
6 Play, 4 Don't Play

x_1 = Humidity

High

Low

Play	Humidity	Wind
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	High	Weak
1	High	Weak
1	High	Weak
1	High	Weak
0	High	Strong
0	High	Strong
0	High	Strong
0	High	Strong

Measuring Split Quality: Impurity Functions

Information Gain

14 Days
6 Play, 4 Don’t Play

x_1 = Humidity

High

Low

Play

Play	Humidity	Wind
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	High	Weak
1	High	Weak
1	High	Weak
1	High	Weak
0	High	Strong
0	High	Strong
0	High	Strong
0	High	Strong

Measuring Split Quality: Impurity Functions

Information Gain

14 Days
6 Play, 4 Don’t Play

x_1 = Humidity

High

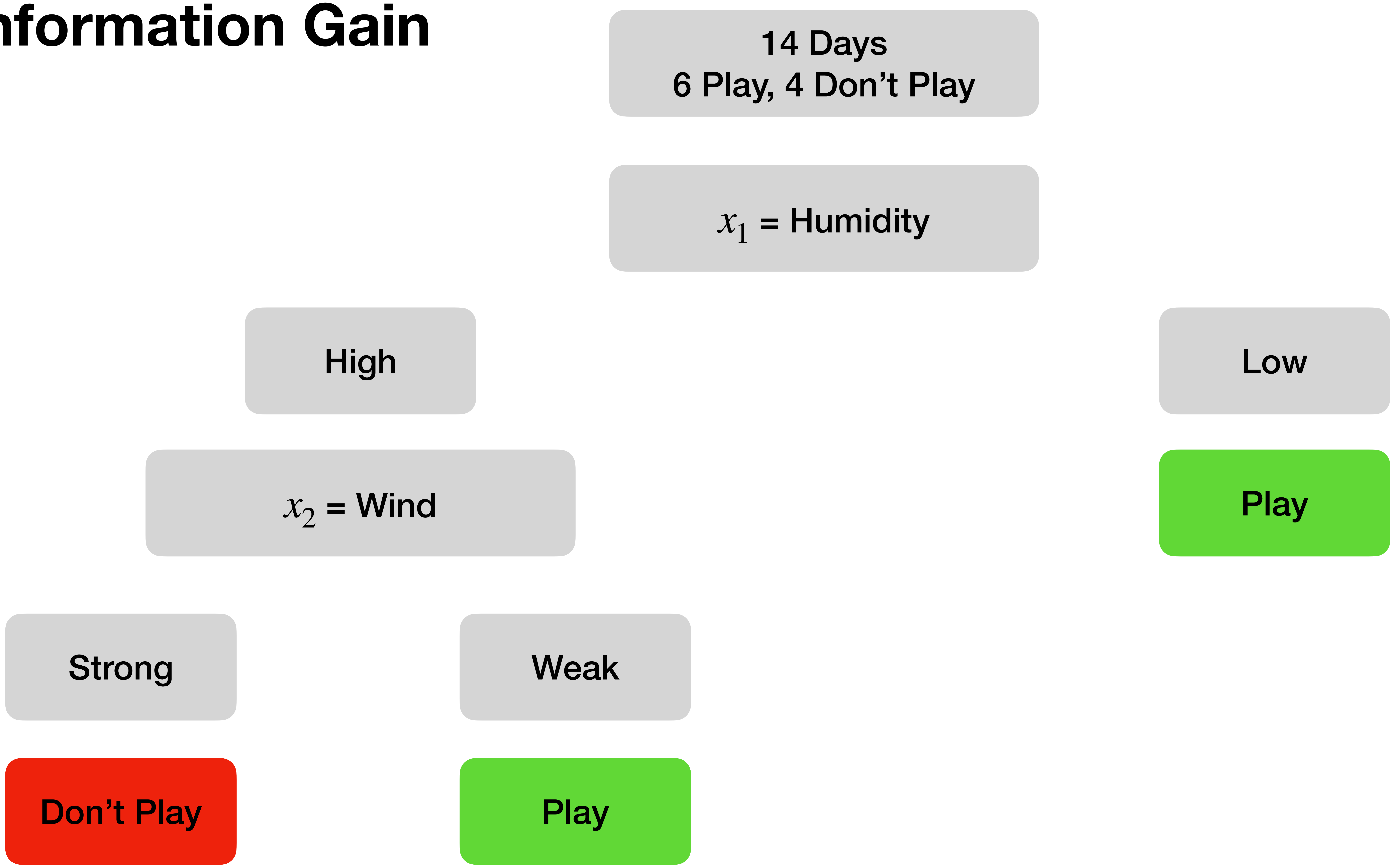
Low

Play

Play	Humidity	Wind
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	High	Weak
1	High	Weak
1	High	Weak
1	High	Weak
0	High	Strong
0	High	Strong
0	High	Strong
0	High	Strong

Measuring Split Quality: Impurity Functions

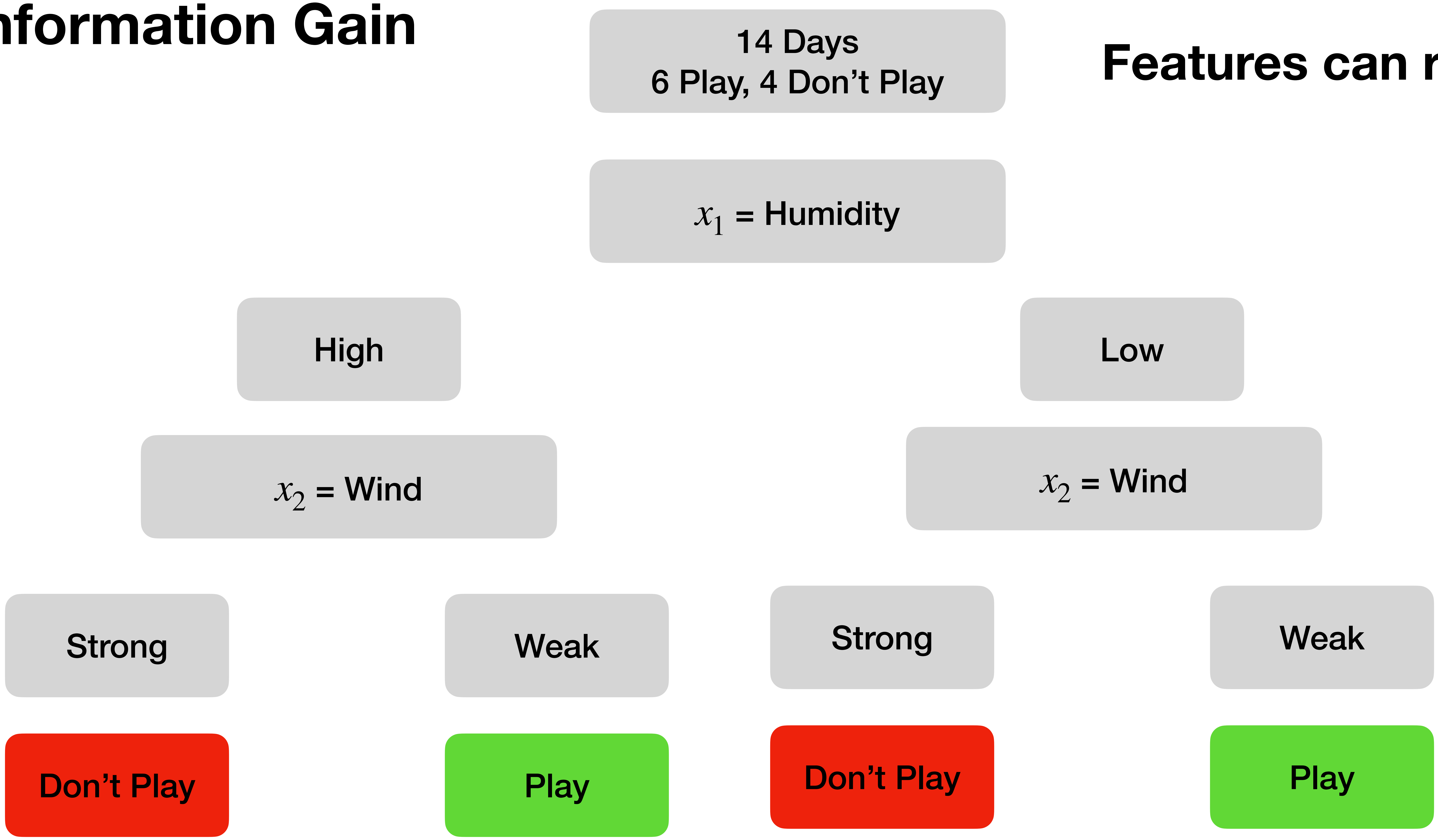
Information Gain



Play	Humidity	Wind
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	Low	Strong
1	Low	Weak
1	High	Weak
1	High	Weak
1	High	Weak
1	High	Weak
0	High	Strong
0	High	Strong
0	High	Strong
0	High	Strong

Measuring Split Quality: Impurity Functions

Information Gain



Features can repeat too!

Play	Humidity	Wind
0	Low	Strong
1	Low	Weak
0	Low	Strong
1	Low	Weak
0	Low	Strong
1	Low	Weak
1	High	Weak
1	High	Weak
1	High	Weak
1	High	Weak
0	High	Strong
0	High	Strong
0	High	Strong
0	High	Strong

Splitting

- Splitting on Continuous Features
- Splitting Categorical Features

Splitting

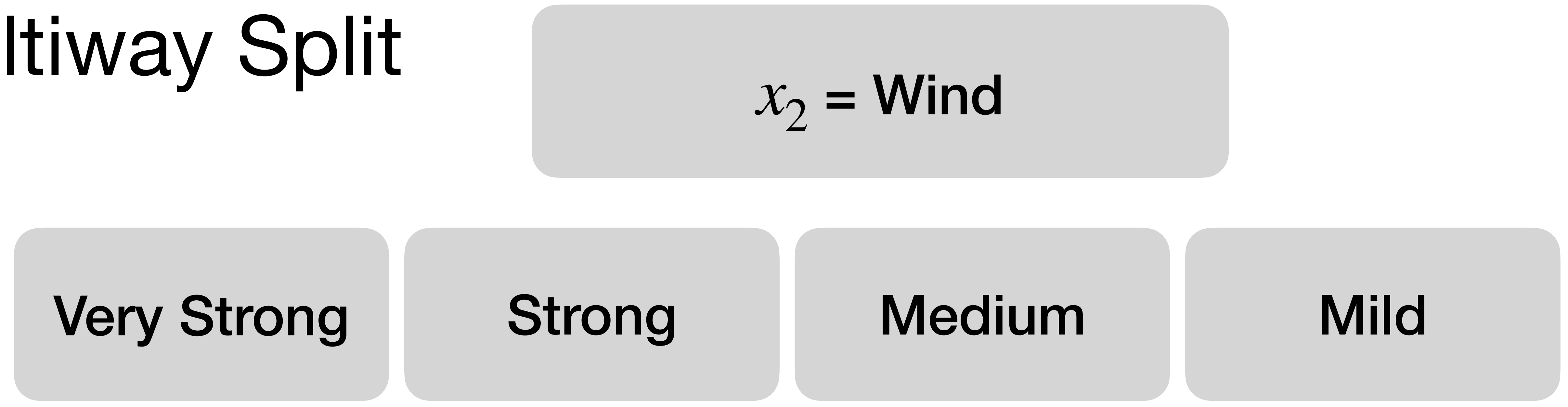
Continuous Features

- For continuous features, we must find the best threshold:
    - Sort unique values: $v_2 < v_2 < v_3 < \ldots < v_m$
    - Consider thresholds at midpoints: $\frac{(v_1 + v_2)}{2}$, $\frac{(v_2 + v_3)}{2}$, ...
    - For each threshold t: split as $x \leq t$ vs. $x \geq t$
    - Compute information gain for each
    - Choose threshold with highest gain

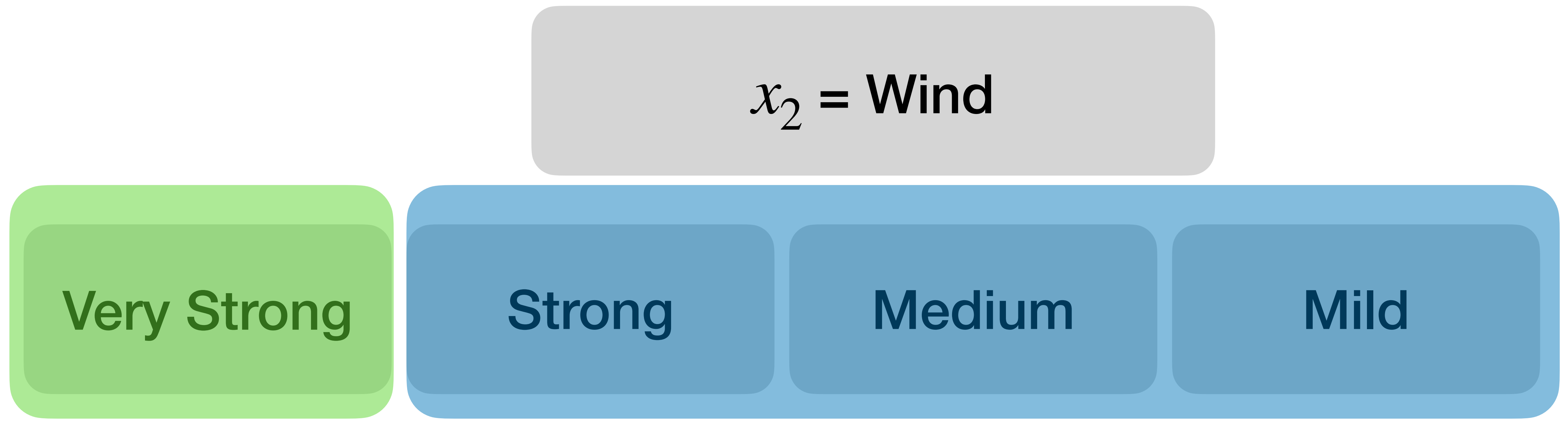
Splitting

Categorical Features

- Multiway Split



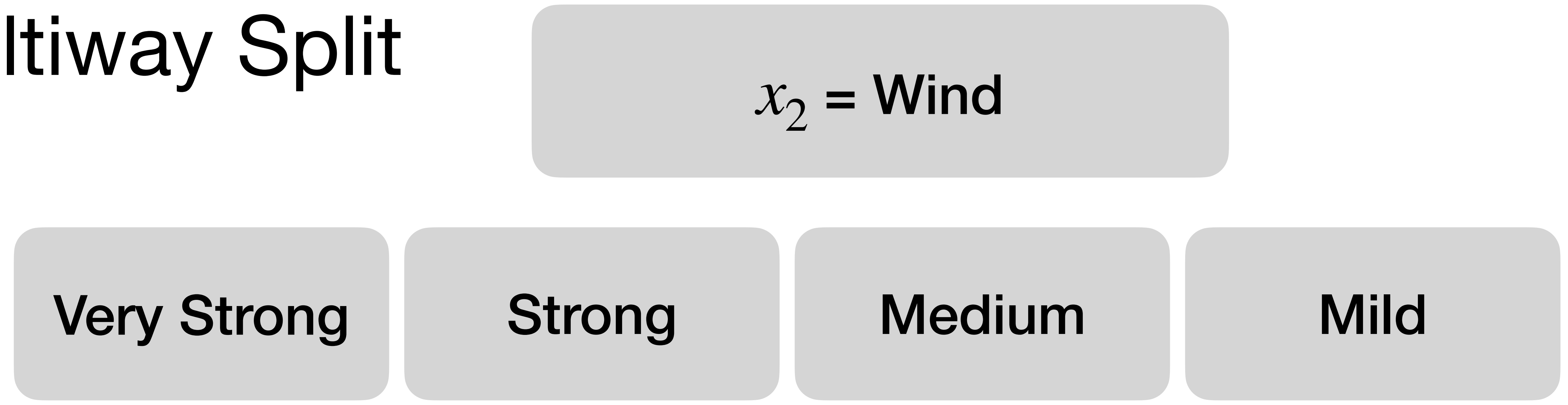
- Binary Split



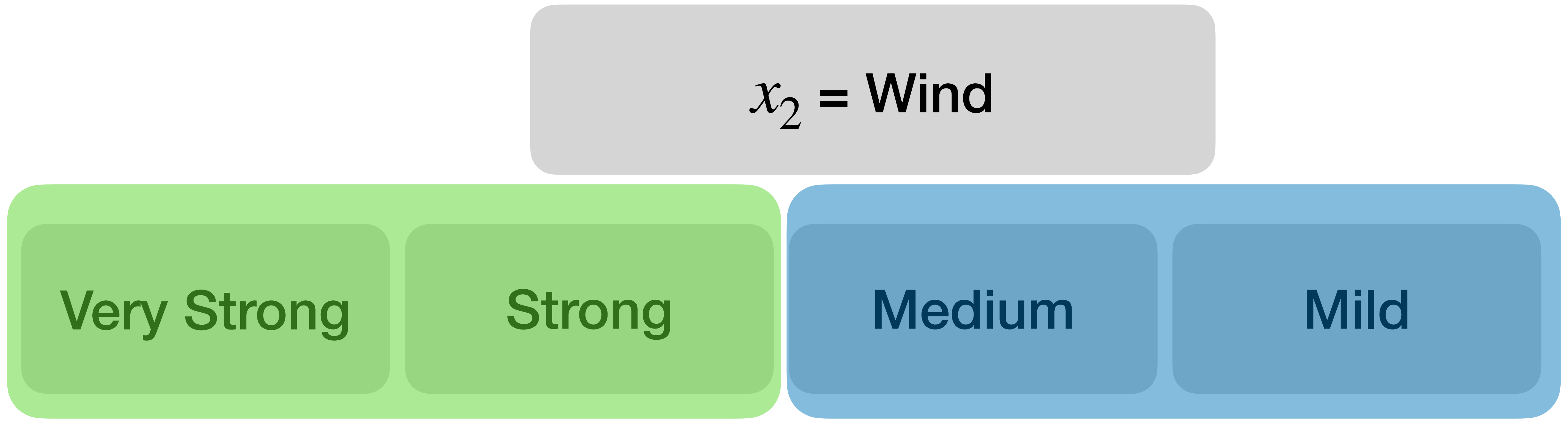
Splitting

Categorical Features

- Multiway Split



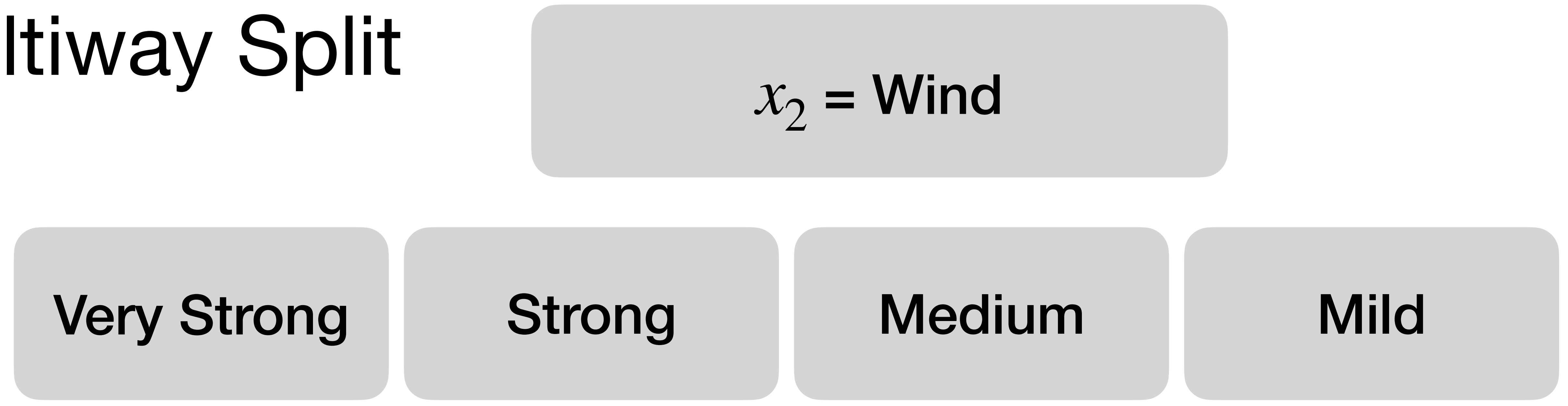
- Binary Split



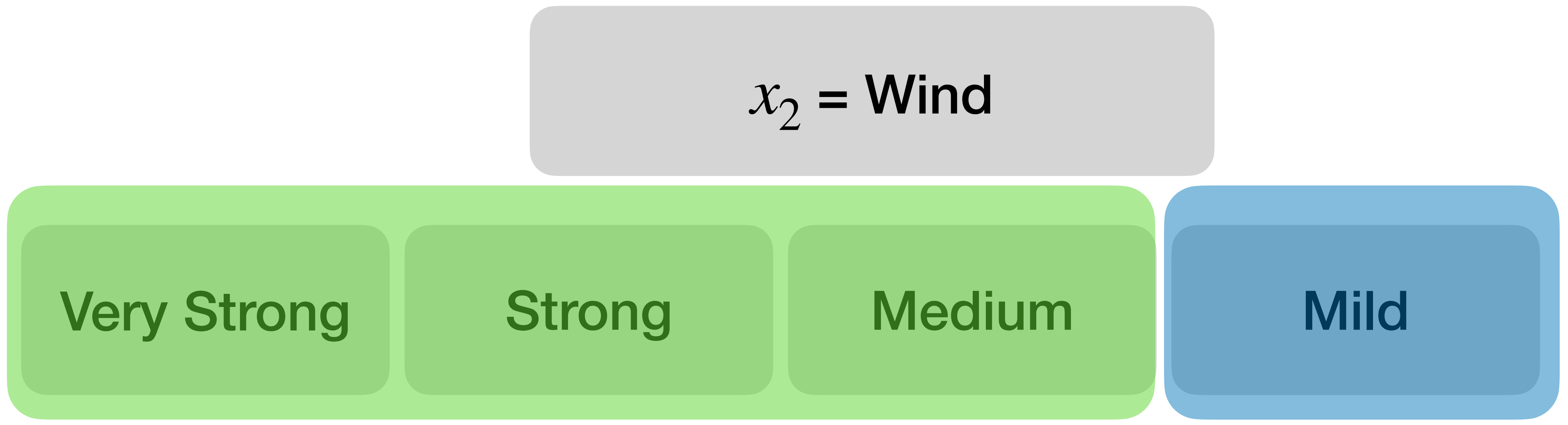
Splitting

Categorical Features

- Multiway Split



- Binary Split



For k categories, there are $2^{(k-1)} - 1$ possible binary partitions.

Stopping Criteria

When to stop splitting and create a leaf

- **Maximum depth reached**: depth $\geq$ **max_depth**

Stopping Criteria

When to stop splitting and create a leaf

- **Maximum depth reached**: depth $\geq$ **max_depth**
- **Minimum samples:** node has fewer than **min_samples_split** samples

Stopping Criteria

When to stop splitting and create a leaf

- **Maximum depth reached**: depth $\geq$ **max_depth**
- **Minimum samples:** node has fewer than **min_samples_split** samples
- **Minimum samples per leaf:** split would create child with < **min_samples_leaf**

Stopping Criteria

When to stop splitting and create a leaf

- **Maximum depth reached**: depth $\geq$ **max_depth**
- **Minimum samples:** node has fewer than **min_samples_split** samples
- **Minimum samples per leaf:** split would create child with < **min_samples_leaf**
- **Pure node:** all samples belong to same class

Stopping Criteria

When to stop splitting and create a leaf

- **Maximum depth reached**: depth $\geq$ **max_depth**
- **Minimum samples:** node has fewer than **min_samples_split** samples
- **Minimum samples per leaf:** split would create child with < **min_samples_leaf**
- **Pure node:** all samples belong to same class
- **No information gain:** best split doesn't improve impurity
- **Maximum leaf nodes:** tree already has **max_leaf_nodes** leaves

Stopping Criteria

When to stop splitting and create a leaf

Hyperparameters

- **Maximum depth reached**: depth $\geq$ **max_depth**
- **Minimum samples:** node has fewer than **min_samples_split** samples
- **Minimum samples per leaf:** split would create child with < **min_samples_leaf**
- **Pure node:** all samples belong to same class
- **No information gain:** best split doesn't improve impurity
- **Maximum leaf nodes:** tree already has **max_leaf_nodes** leaves

Stopping Criteria

Overfitting & Pruning

- Deep trees overfit - they memorize training data, **creating leaves with very few samples**.
- Signs of overfitting:
  - Training accuracy ≈ 100%
  - Test accuracy much lower
  - **Very deep tree with many leaves**

Stopping Criteria

Overfitting & Pruning

- **Pre-Pruning (Early Stopping)**
  - Stop growing before the tree becomes too complex. Use stopping criteria (max_depth, min_samples_leaf, etc.).
  - **Pro:** Simple, fast
  - **Con:** Might stop too early, missing good splits deeper down

Stopping Criteria

Overfitting & Pruning

- **Pre-Pruning (Early Stopping)**
  - Stop growing before the tree becomes too complex. Use stopping criteria (max_depth, min_samples_leaf, etc.).
  - **Pro:** Simple, fast
  - **Con:** Might stop too early, missing good splits deeper down
- **Post-Pruning**
  - Grow the full tree, then remove nodes that don't help.

Stopping Criteria

Overfitting & Pruning

- **Pre-Pruning (Early Stopping)**
    - Stop growing before the tree becomes too complex. Use stopping criteria (max_depth, min_samples_leaf, etc.).
    - **Pro:** Simple, fast
    - **Con:** Might stop too early, missing good splits deeper down
- **Post-Pruning**
    - Grow the full tree, then remove nodes that don't help.
- **Reduced Error Pruning:**
    - Grow complete tree
    - For **each internal node**, consider replacing its subtree with a leaf
    - If validation accuracy doesn't decrease, prune it
    - Repeat until pruning hurts accuracy

Decision Trees

Pros and Cons

Pros

- Highly interpretable (visualize as flowchart)
- No feature scaling required
- Handles numerical and categorical features
- Handles missing values
- Captures non-linear relationships
- Automatic feature selection
- Fast prediction: O(depth)

Cons

- High variance (unstable)
- Prone to overfitting
- Greedy algorithm (no global optimum)
- Axis-aligned splits only (can't capture diagonal boundaries efficiently)
- Biased toward high-cardinality features

Decision Trees

Pros and Cons

Pros

- Highly interpretable (visualize as flowchart)
- No feature scaling required
- Handles numerical and categorical features
- Handles missing values
- Captures non-linear relationships
- Automatic feature selection
- Fast prediction: O(depth)

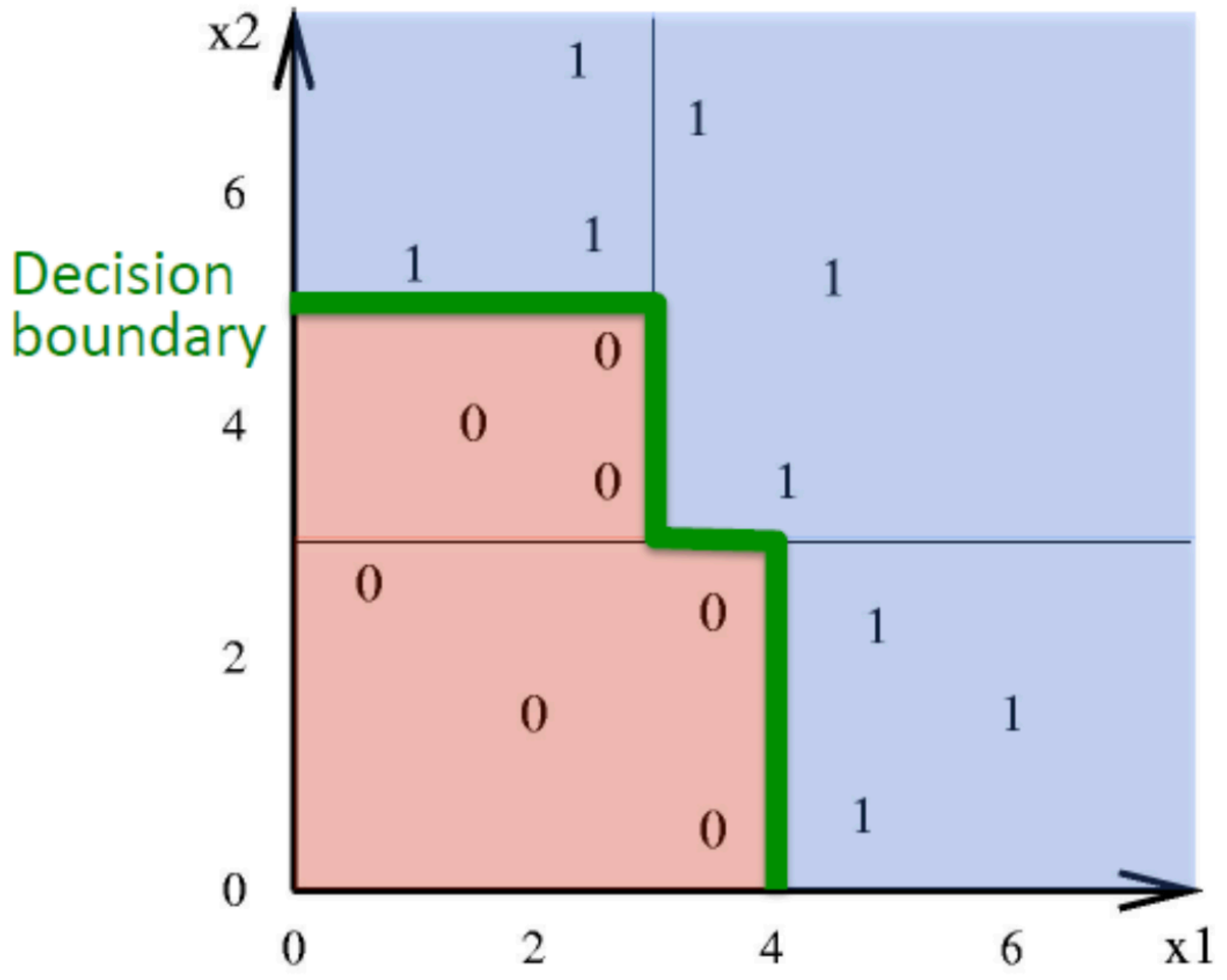
Cons

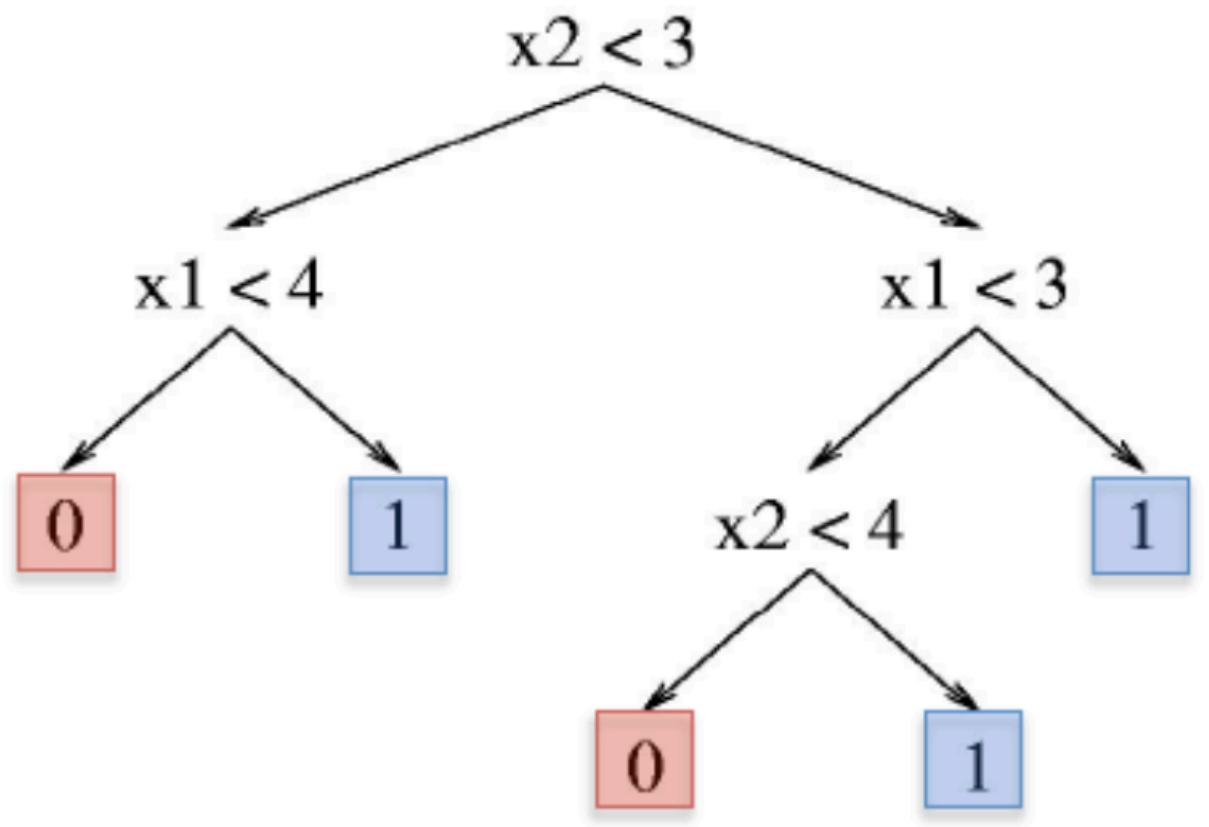
- High variance (unstable)
- Prone to overfitting
- Greedy algorithm (no global optimum)
- **Axis-aligned splits only (can't capture diagonal boundaries efficiently)**
- Biased toward high-cardinality features

Decision Trees

Pros and Cons

- Decision trees divide the feature space into axis parallel rectangles
- Each rectangular region is labeled with one label

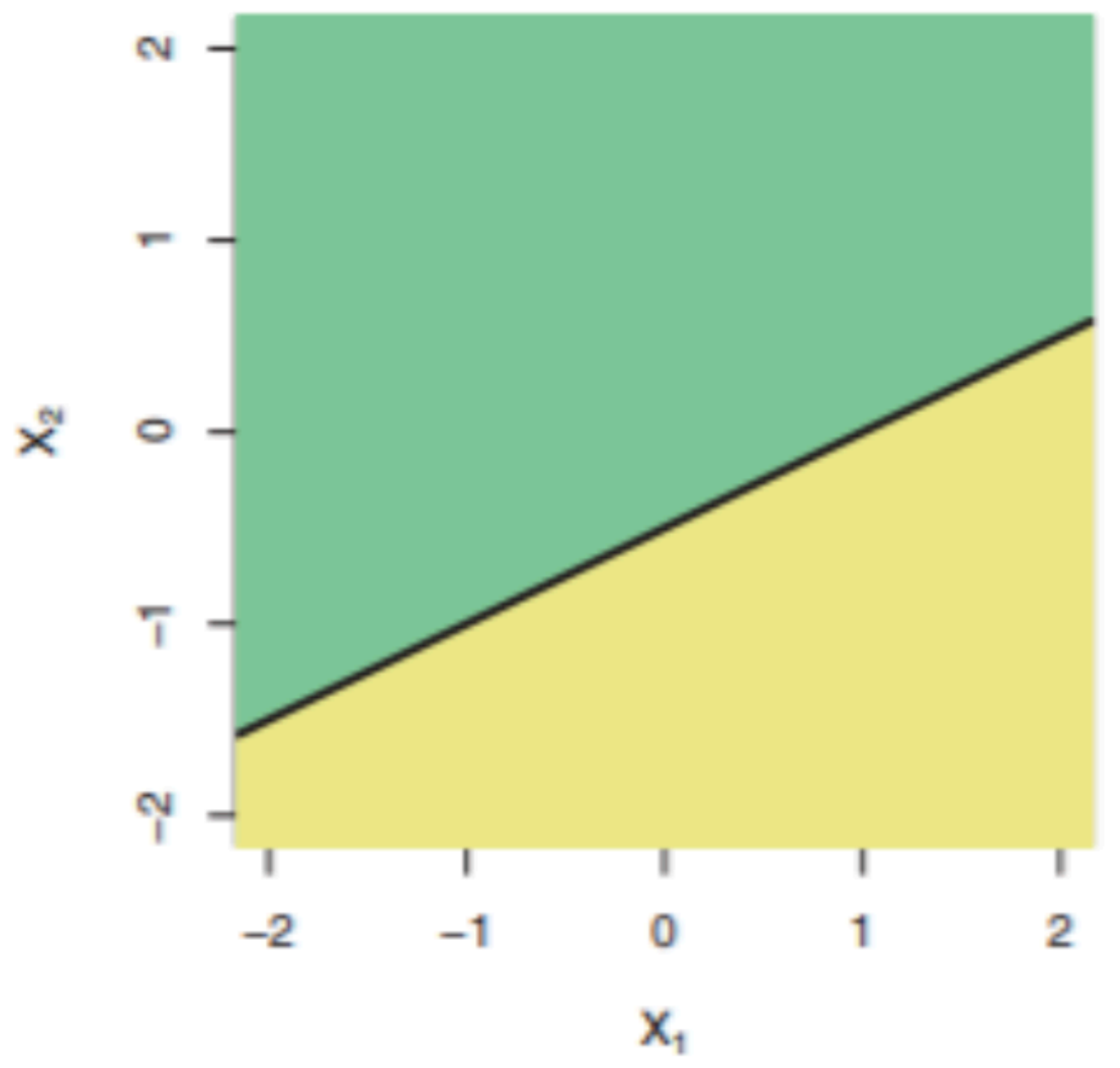




Decision Trees

Pros and Cons

- Decision trees divide the feature space into axis parallel rectangles
- Each rectangular region is labeled with one label

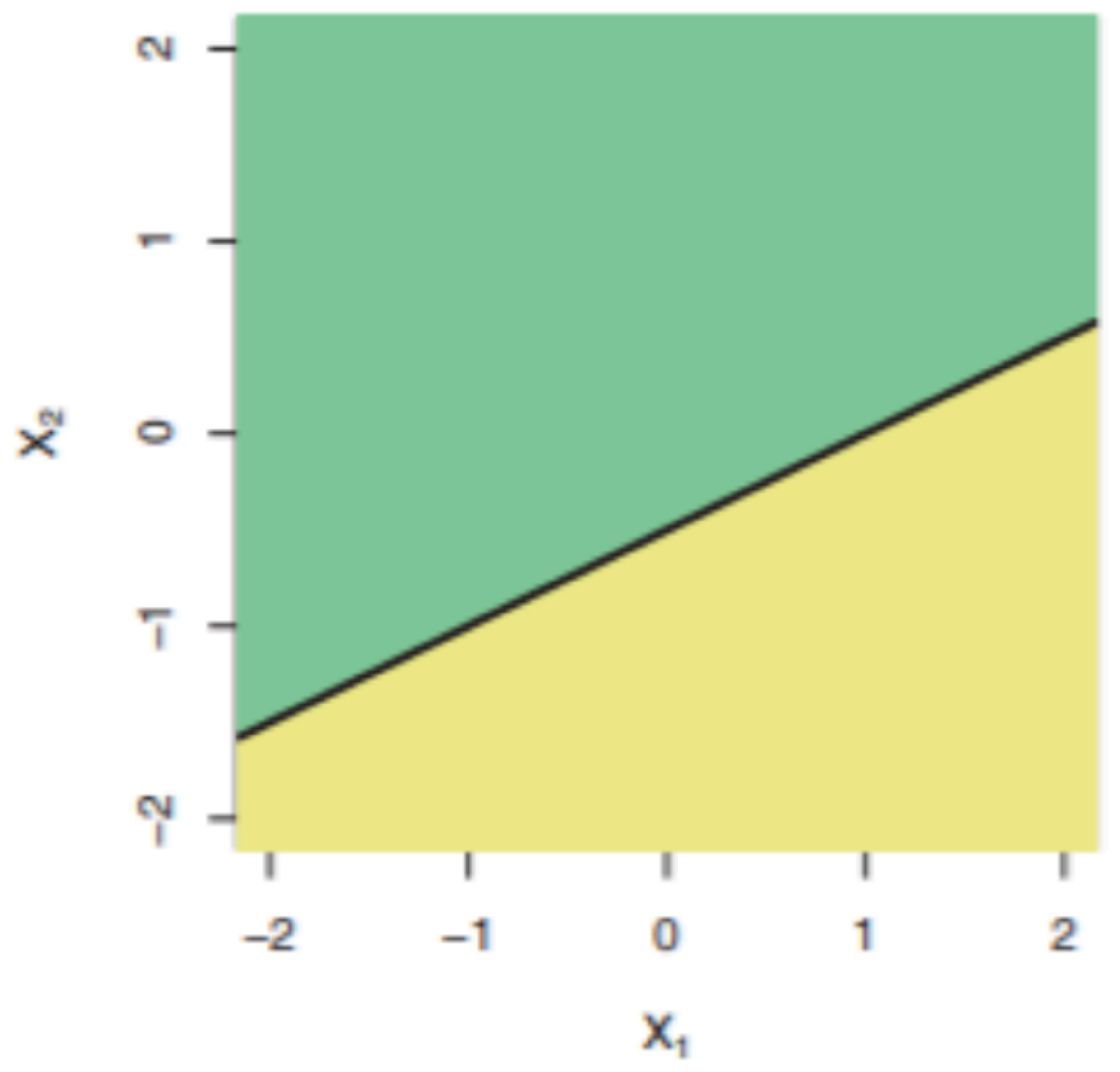


Linear

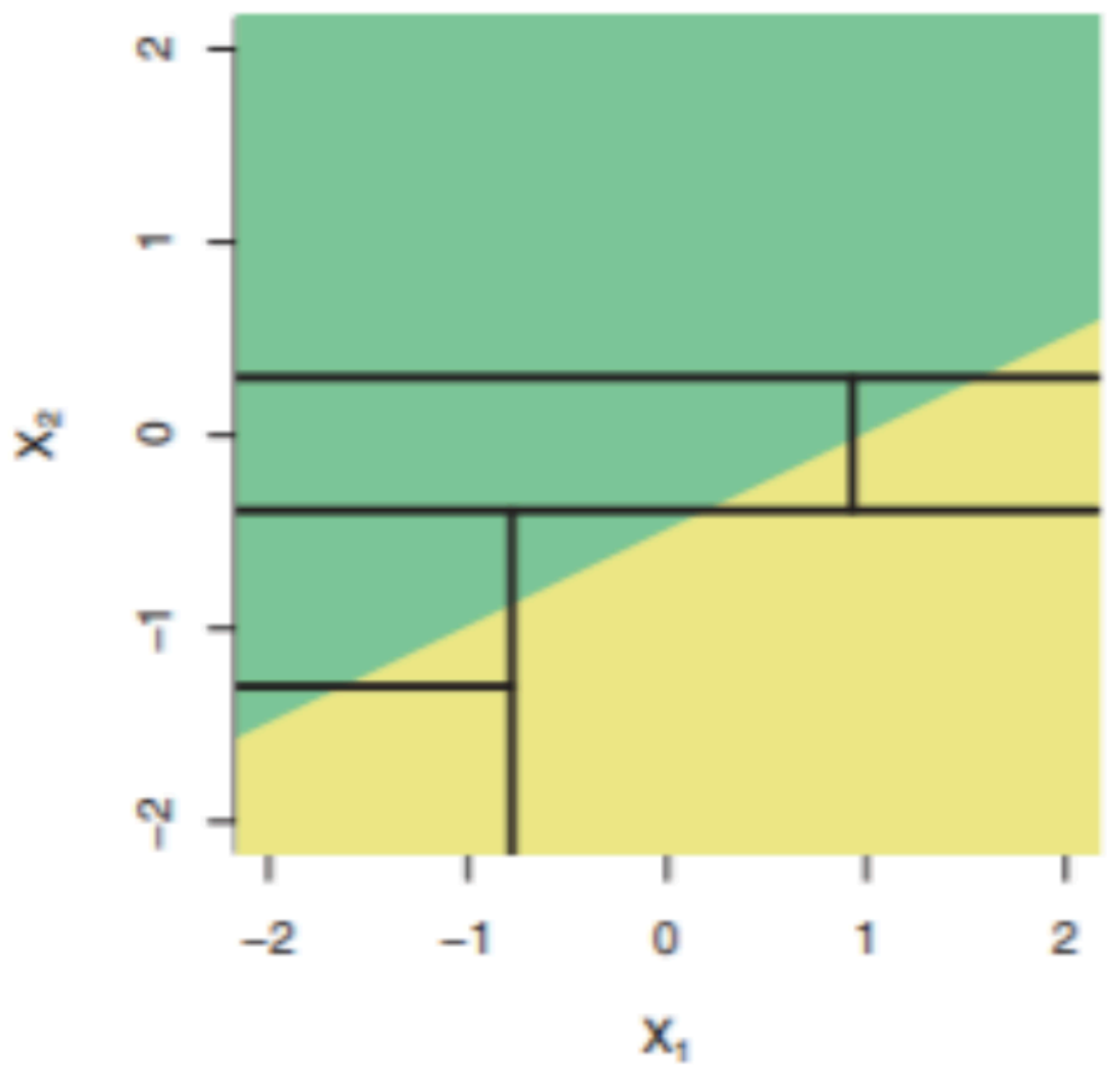
Decision Trees

Pros and Cons

- Decision trees divide the feature space into axis parallel rectangles
- Each rectangular region is labeled with one label



Linear

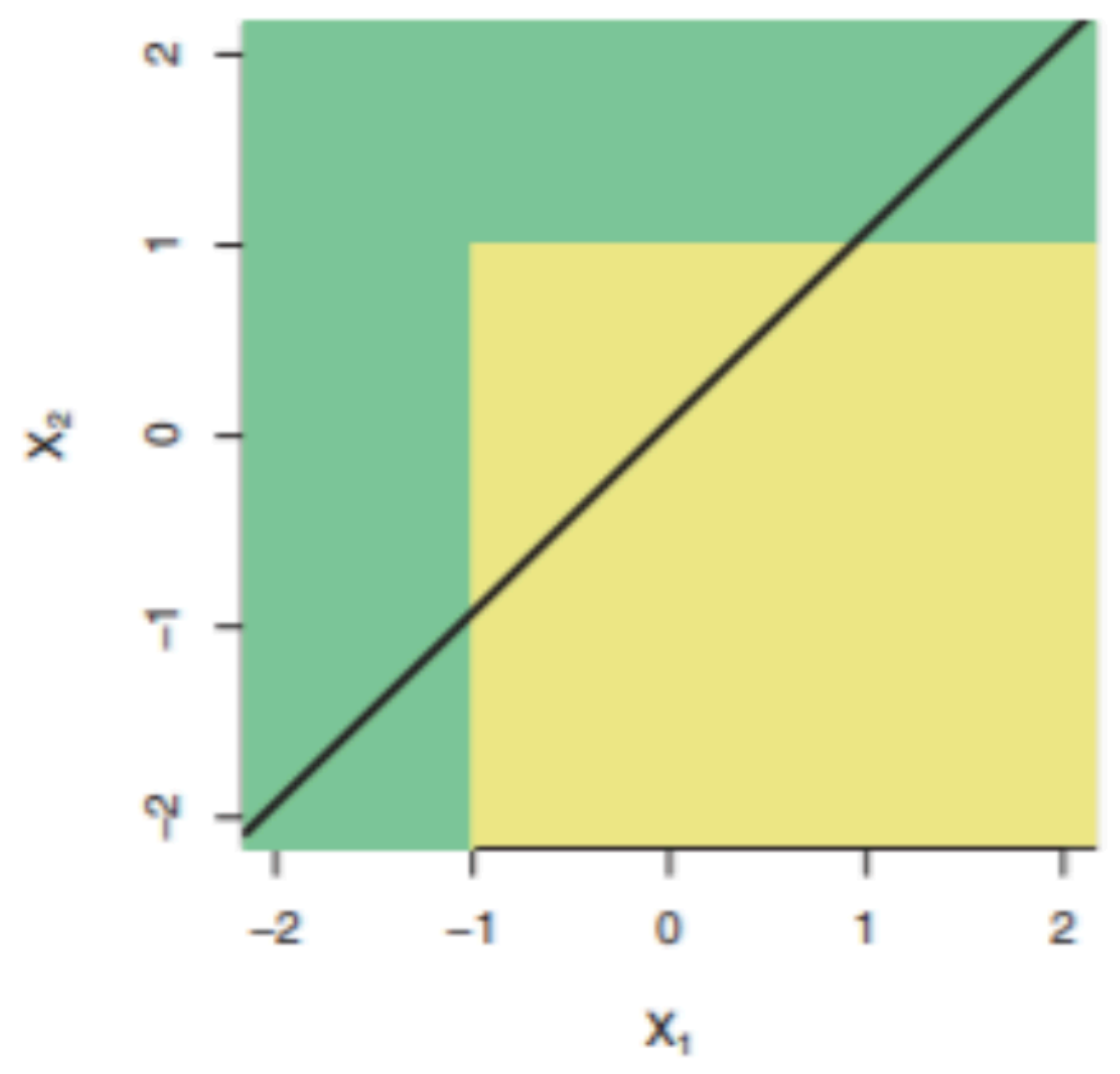


DT

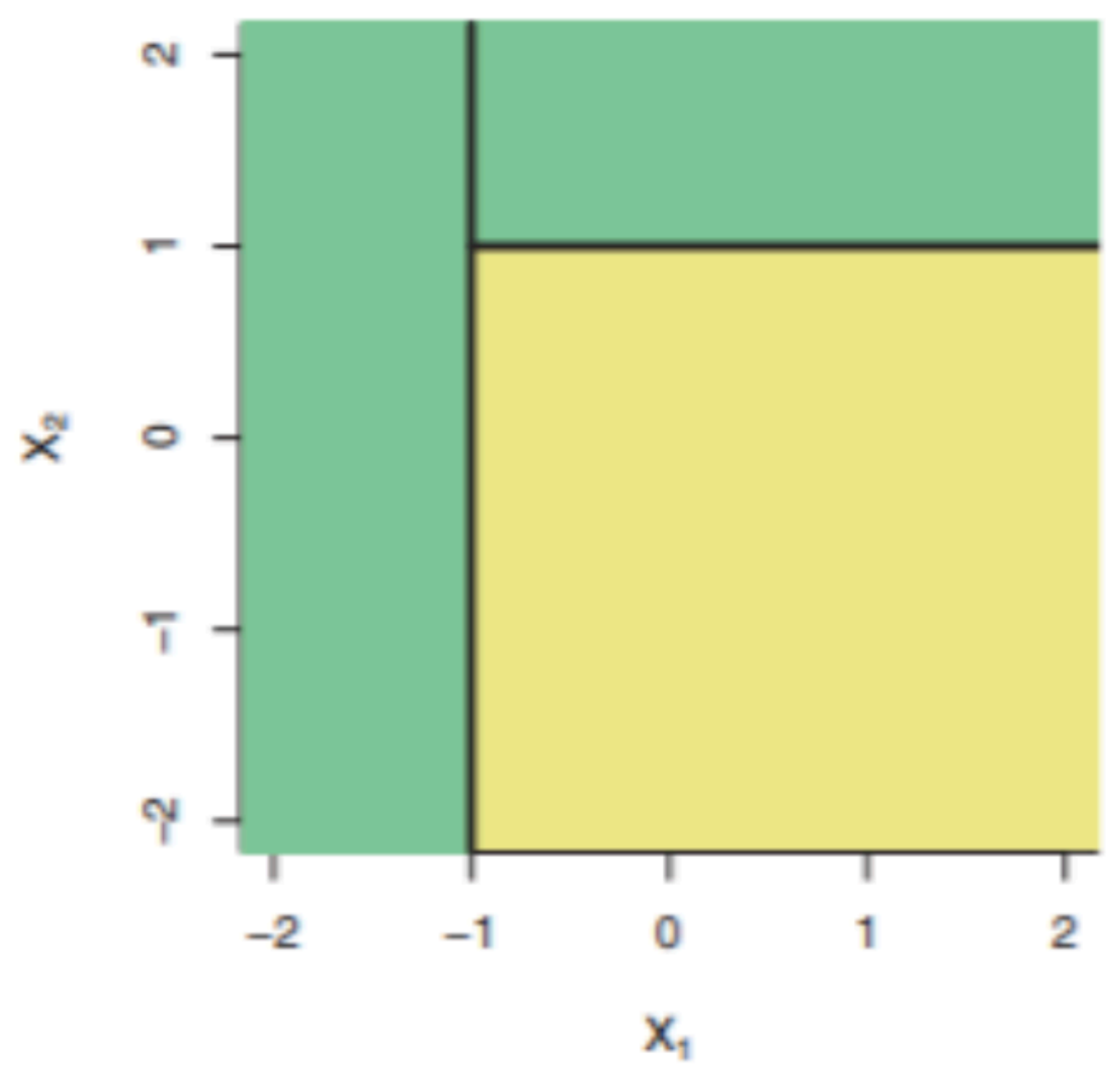
Decision Trees

Pros and Cons

- Decision trees divide the feature space into axis parallel rectangles
- Each rectangular region is labeled with one label



Linear



DT

Today’s Outline

- Decision Trees
- **Ensemble Learning**
  - Bagging
  - Boosting

Ensemble Methods

The Fundamental Idea

- A single decision tree is unstable and prone to overfitting.
- Ensemble methods combine multiple trees to create a more robust model.
- **Key Insight:**
  - Combining many “weak” learners can create a “strong” learner.
- **Analogy:**
  Asking 100 people to estimate something and averaging their answers is often more accurate than asking one expert.

Ensemble Methods

The Fundamental Idea

Method	Strategy	Trees Trained	Key Idea
Bagging	Parallel	Independently	Reduce variance via averaging
Random Forest	Parallel	Independently	Bagging + random feature subsets
Boosting	Sequential	Each corrects previous	Reduce bias via focusing on errors

Bagging

Bootstrap Aggregating

- Bootstrap sampling:
    - Sample **N** points with replacement from a dataset of **N** points.

Bagging

Bootstrap Aggregating

- Bootstrap sampling:
  - Sample **N** points <u>with replacement</u> from a dataset of **N** points.
- **Bagging Algorithm**
  - Create **B** bootstrap samples from the training data
  - Train one decision tree on each bootstrap sample
  - Combine predictions:
    - Classification: Majority vote
    - Regression: Average

Bagging

Why Bagging Works

- If individual trees have variance σ^2 and are independent, then averaging **B** trees reduces variance to $\frac{\sigma^2}{B}$
- Reality:
    - Trees aren't fully independent (trained on **overlapping data**), so the reduction is less dramatic but still substantial.
- Bagging primarily reduces variance without increasing bias.

Bagging

Evaluation - Out-of-Bag (OOB)

- For each sample x_i, find all trees that **did not** include it in their bootstrap sample
- Get predictions from **only those trees**
- Aggregate these predictions
- Compute error across all samples
- OOB error approximates test error <u>without needing a separate validation set</u>.

Random Forests

- Bagging + Feature Randomness
- Random Forest adds another layer of randomization:
    - At each split, consider only a **random subset of <u>features</u>**.

Random Forests

Why does this help?

- Without feature randomness, **all trees tend to use the same strong features at the top**
  - trees are highly correlated
  - averaging provides less benefit.

Random Forests

Why does this help?

- Without feature randomness, **all trees tend to use the same strong features at the top**
  - trees are highly correlated
  - averaging provides less benefit.
- With feature randomness:
  - Trees are de-correlated
  - Different trees capture different patterns
  - Averaging becomes more effective

Random Forests

Hyperparameters

Parameters	Description	General Values
B	Number of trees	100-1000
m	Features per split	$\sqrt{d}$ (classification) d/3 (regression)
max_depth	Max Tree Depth	None (grow fully) or tune
min_samples_leaf	Minimum samples in leaf	1 (classification), 5 (regression)

Feature Importance in Random Forests

- **Mean Decrease in Impurity (MDI)**: Average the impurity reduction from each feature across all trees.

Feature Importance in Random Forests

- **Mean Decrease in Impurity (MDI)**: Average the impurity reduction from each feature across all trees.
- **Permutation Importance**:
  - Compute baseline accuracy on OOB samples
  - For each feature j:
    - Randomly shuffle feature j's values
    - Recompute accuracy
    - Importance (j) = baseline - shuffled accuracy
- Features that hurt accuracy when shuffled are important
- Permutation importance is more reliable but **slower**.

Random Forests

Pros and Cons

Pros

- Excellent accuracy out-of-the-box
- Robust to overfitting
- Handles high-dimensional data
- Provides feature importance
- Built-in OOB validation
- Parallelizable (trees are independent)
- Minimal tuning required

Cons

- Less interpretable than single tree
- Can be slow for very large datasets
- Not optimal for sparse, high-dimensional data (like text)

Boosting

- Unlike bagging (parallel, independent trees), boosting trains trees **sequentially**
    - Each subsequent tree trying to **correct the errors of the previous trees**.
- **Intuition**: Each tree focuses on the samples that **previous trees got wrong**.

AdaBoost - Adaptive Boosting

- **Key idea**:
  - Maintain weights on samples.
  - Increase weights on misclassified samples so the next tree focuses on them.

AdaBoost - Adaptive Boosting

Algorithm

- Initialize weights: $w_i = \dfrac{1}{N}$ for all samples

AdaBoost - Adaptive Boosting

Algorithm

- Initialize weights: $w_i = \frac{1}{N}$ for all samples
- For $t = 1 \to T$:
    1. Train weak learner h_t on weighted data

AdaBoost - Adaptive Boosting

Algorithm

- Initialize weights: $w_i = \frac{1}{N}$ for all samples
- For $t = 1 \to T$:
  1. Train weak learner h_t on weighted data
  2. Compute weighted error: $\epsilon_t = \frac{\sum w_i \cdot I(y_i \neq h_t(x_i))}{\sum w_i}$

AdaBoost - Adaptive Boosting

Algorithm

- Initialize weights: $w_i = \frac{1}{N}$ for all samples
- For $t = 1 \to T$:
  1. Train weak learner h_t on weighted data
  2. Compute weighted error: $\epsilon_t = \frac{\sum w_i \cdot I(y_i \neq h_t(x_i))}{\sum w_i}$
  3. Compute tree weight: $\alpha_t = \frac{0.5 \cdot ln(1 - \epsilon_t)}{\epsilon_t}$

AdaBoost - Adaptive Boosting

Algorithm

- Initialize weights: $w_i = \frac{1}{N}$ for all samples
- For $t = 1 \to T$:
    1. Train weak learner h_t on weighted data
    2. Compute weighted error: $\epsilon_t = \frac{\sum w_i \cdot I(y_i \neq h_t(x_i))}{\sum w_i}$
    3. Compute tree weight: $\alpha_t = \frac{0.5 \cdot ln(1 - \epsilon_t)}{\epsilon_t}$
    4. Update sample weights: $w_i \leftarrow w_i \cdot exp(-\alpha_t \cdot y_i \cdot h_t(x_i))$ (Normalize so weights sum to 1)
- Final prediction: $H(x) = sign(\sum \alpha_t \cdot h_t(x))$

AdaBoost - Adaptive Boosting

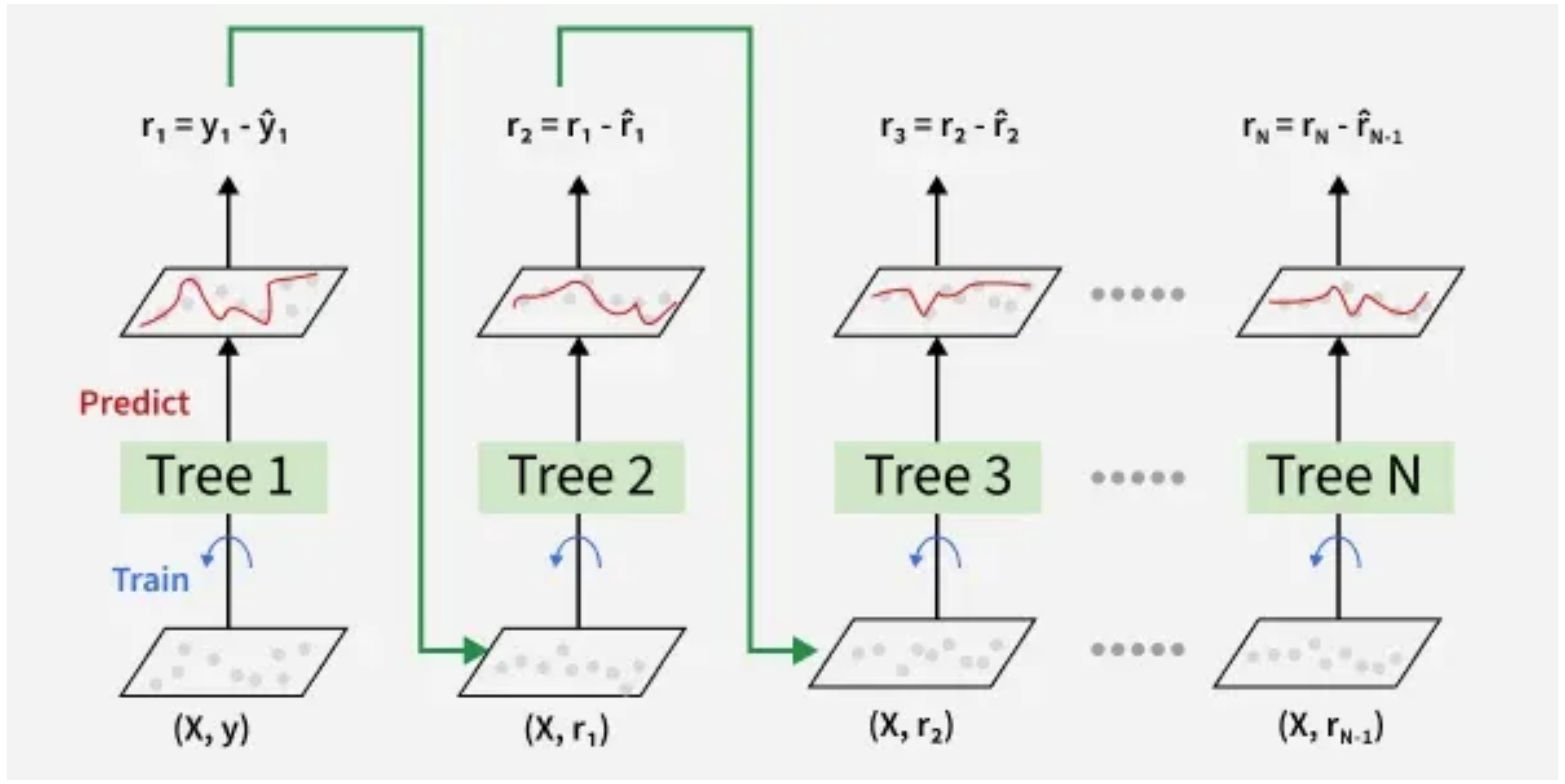
- Intuition Behind the Updates
  - Tree weight α_t :
    - If error ϵ_t = 0.5 (random):
      - α_t = 0 (ignore this tree)
    - If error $\epsilon_t \to 0$ (perfect):
      - $\alpha_t \to \infty$ (trust this tree completely)
    - If error $\epsilon_t \to 1$ (anti-correlated)
      - $\alpha_t \to -\infty$ (flip its predictions)

$$\epsilon_t = \frac{\sum w_i \cdot I(y_i \neq h_t(x_i))}{\sum w_i}$$

$$\alpha_t = \frac{0.5 \cdot ln(1 - \epsilon_t)}{\epsilon_t}$$

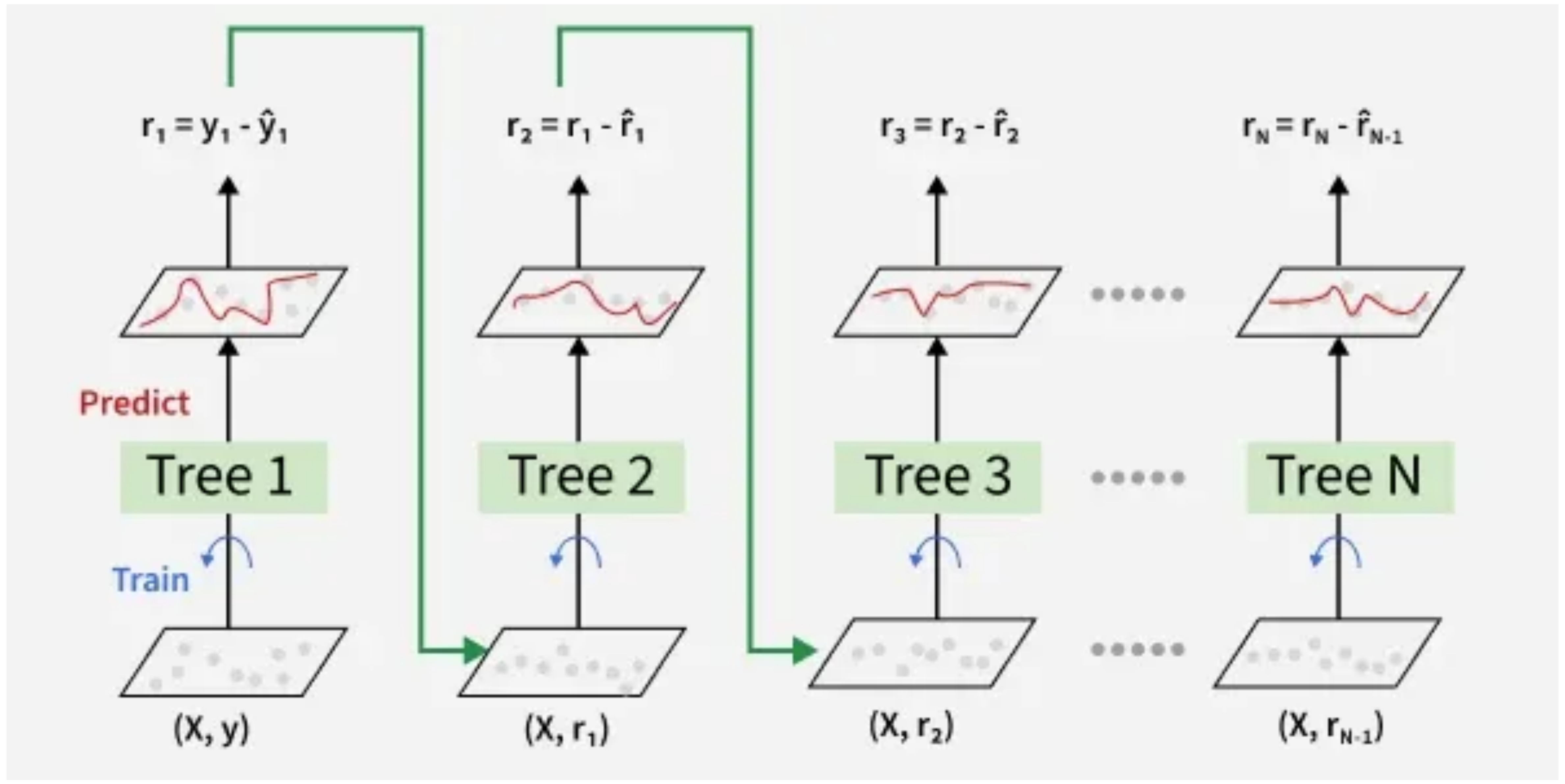
Gradient Boost

- A more general framework that works for **any** differentiable loss function.
- **Key idea:**
    - Each tree fits the **negative gradient of the loss** (the "residuals" for squared error loss).



Gradient Boost

- After each tree is trained its predictions are **shrunk** by multiplying them with the learning rate α which ranges from 0 to 1.
- This prevents overfitting by ensuring each tree has a **smaller impact** on the final model.



XGBoost, LightGBM

- Modern, optimized implementations of gradient boosting:
- **XGBoost (Extreme Gradient Boosting)**
- Key innovations:
  - Regularized objective: adds L1/L2 penalty on leaf weights
  - Second-order gradients: uses Newton-Raphson (Hessian) for better splits
  - Sparsity-aware: efficient handling of missing values
  - Column block structure: parallelized tree building
  - Cache-aware: optimized for CPU cache efficiency

XGBoost, LightGBM

- Modern, optimized implementations of gradient boosting:
- **LightGBM (Light Gradient Boosting Machine)**
- Key innovations:
    - Gradient-based One-Side Sampling (GOSS): **keeps samples with large gradients**
    - Exclusive Feature Bundling: bundles **mutually exclusive features**
    - Histogram-based splitting: bins continuous features for faster splits

When to use what?

Random Forest

- Good default choice
- When interpretability (feature importance) matters
- When you want robustness without tuning
- When you have noisy labels

Gradient Boosting
(XGBoost / LightGBM / CatBoost)

- When you need maximum accuracy
- Structured/tabular data competitions
- When you have time to tune hyperparameters
- When training data is clean

Next Class

- Deep learning