Day 2: Machine Learning Algorithms

Naeemullah Khan
naeemullah.khan@kaust.edu.sa

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

KAUST Academy
King Abdullah University of Science and Technology

December 28, 2025

Learning Outcomes

By the end of this session, you will be able to:

1. Navigate the ML Toolbox and distinguish between Linear, Distance-based, Kernel-based, and Tree-based models.
2. Understand Model Intuition and explain how these algorithms learn (e.g., Maximizing Margin in SVM, Greedy Splits in Trees).
3. Understand Ensembling and contrast Bagging (Random Forest) vs. Boosting (Gradient Boosting) and why they reduce error.
4. Understand the advantages and disadvantages of each algorithm.

Recap:

→ What are the 3 pillars of machine learning?

→ What are the 2 tasks of supervised learning?

Table of Contents

The Three Pillars of Learning

أكاديمية كاوست
KAUST ACADEMY

1. The Model (f_θ)

Design Choice:
ML (Linear, Kernel, Trees) & DL (MLP, CNN)

Depends on:
Data Type (Tabular, image) & Complexity

2. Loss Function (J)

Design Choice:
MSE vs. Cross-Entropy

Depends on:
The Task (Regression vs. Classification)

3. Optimizer

Design Choice:
Gradient Descent, Adam

Depends on:
Stability & Convergence Speed (Covered later in Deep Learning)

Today we focus on **1 & 2**.

The Three Pillars of Learning

أكاديمية كاوست
KAUST ACADEMY

1. The Model (f_θ)

Design Choice:
ML (Linear, Kernel, Trees) & DL (MLP, CNN)

Depends on:
Data Type (Tabular, image) & Complexity

2. Loss Function (J)

Design Choice:
MSE vs. Cross-Entropy

Depends on:
The Task (Regression vs. Classification)

3. Optimizer

Design Choice:
Gradient Descent, Adam

Depends on:
Stability & Convergence Speed (Covered later in Deep Learning)

Today we focus on **1 & 2**.
Let's start by discussing the loss functions first.

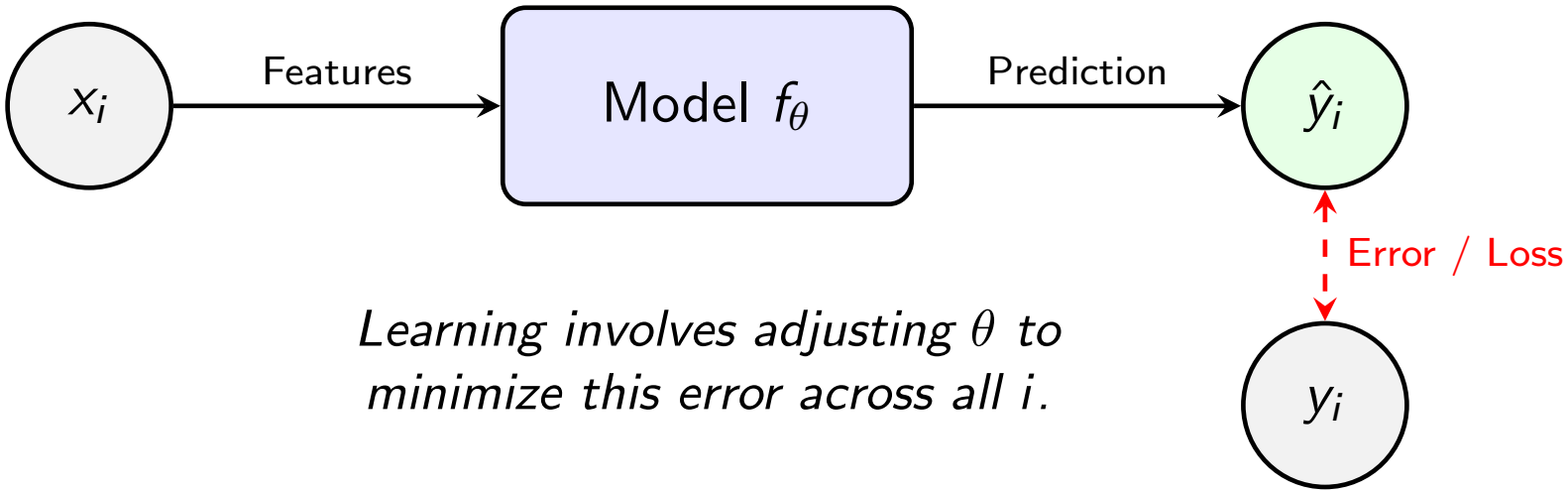
Supervised Learning: The Setup

The Objective

Given a labeled dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$, we aim to learn a mapping function f that generalizes to unseen data:

$$\hat{y} = f(x; \theta) \approx y$$

where θ represents the learnable model parameters.

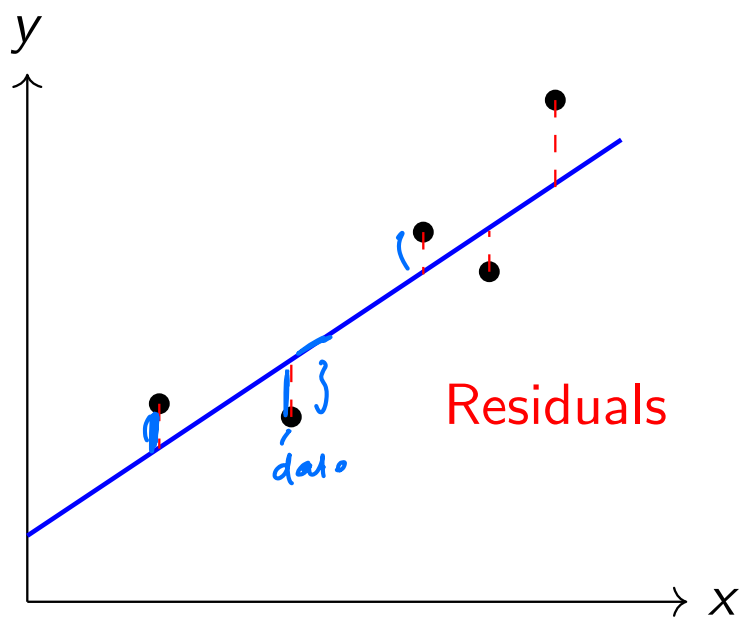


Task Type A: Regression

Definition: The target y is **Continuous** ($y \in \mathbb{R}$).

Examples:

- House Prices ($\$300k, \$500k$)
- Temperature ($24.5°C$)
- Stock Value



Goal: Minimize the sum of red lines.

The Loss: Mean Squared Error

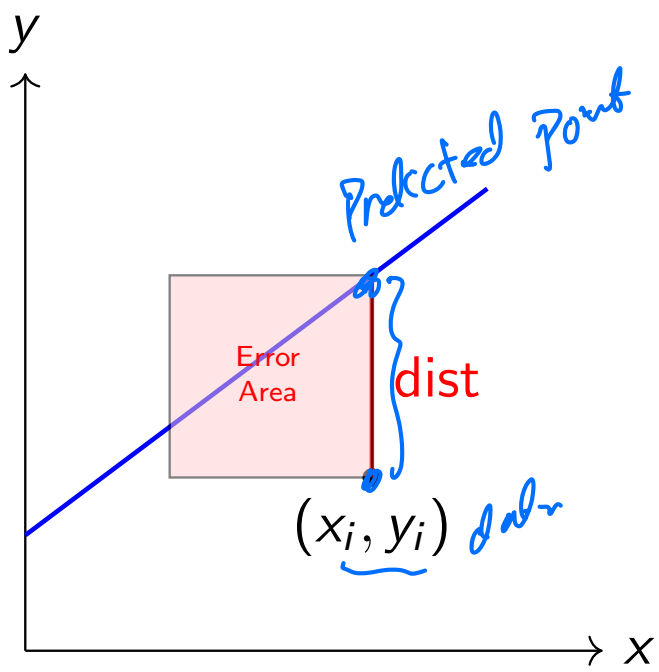
Measures the squared distance between points and the line.

$$\text{MSE} = \frac{1}{N}\sum(y - \hat{y})^2$$

Why Squared Error?

The 3 Steps to MSE Loss:

1. **Residuals ($y - \hat{y}$):** measures the raw distance. Points below the line give negative distances.
2. **The Square $(\dots)^2$:** Squaring ensures all distances are positive so they don't cancel out. It also heavily penalizes **outliers**.
3. **The Average ($\frac{1}{N}\sum$):** We aggregate errors from all N samples to get a single performance score.



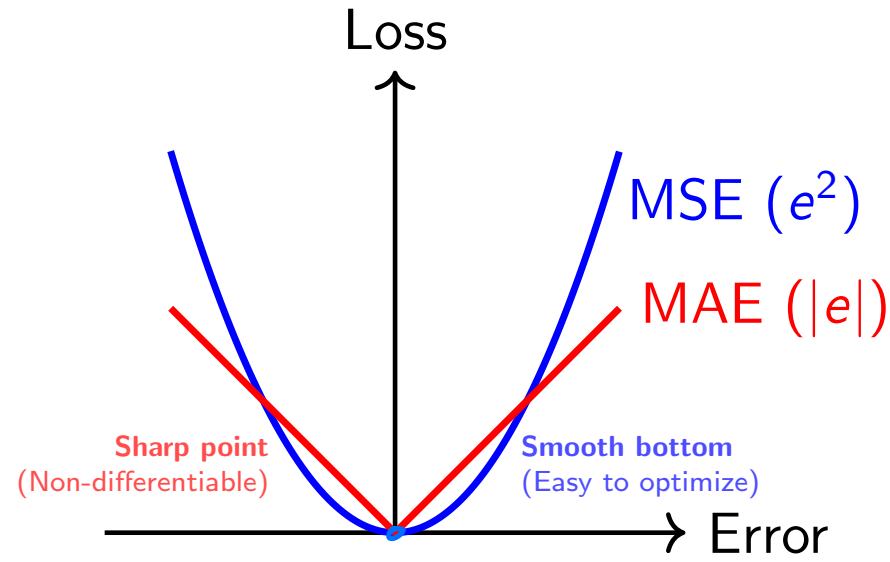
Each "error" acts as a square area we want to shrink to zero.

Why Squared Error?

أكاديمية كاوست
KAUST ACADEMY

Note

If we use the **Absolute** difference $|y - \hat{y}|$ instead of the square, we get **MAE**. It is more robust to outliers, but **not differentiable** at 0.
Mathematically, you can still optimize functions with **finitely** many non-differentiable points using subgradients, though this adds some complexity.



MSE (Blue) is smooth. MAE (Red) is sharp.

Task Type B: Binary Classification

Definition:

Target y is **discrete**: $y \in \{0, 1\}$
(e.g., spam/not spam)

How It Works:

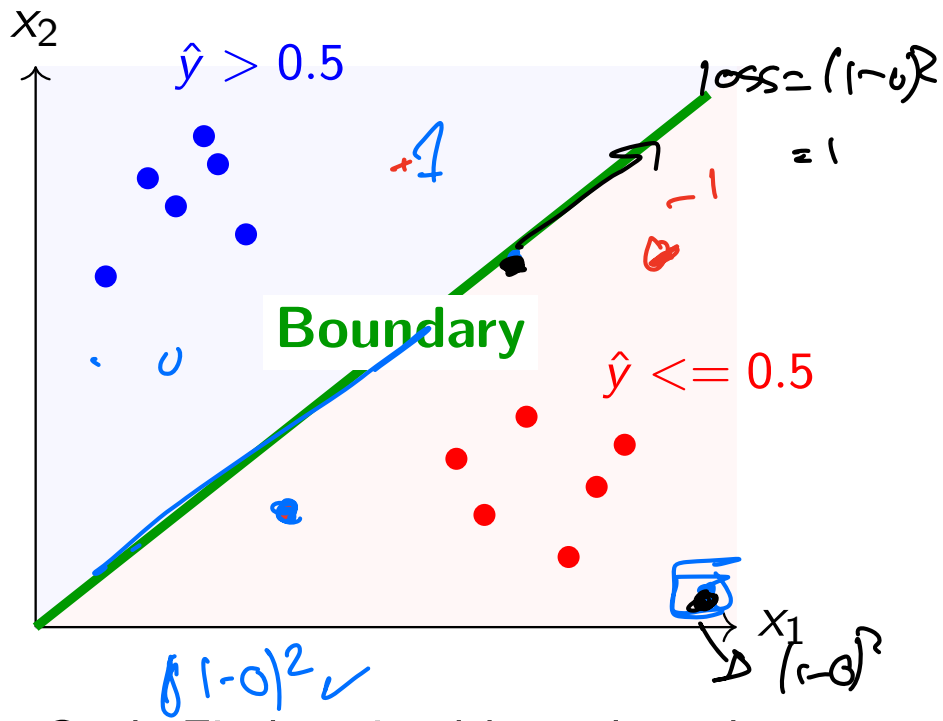
1. Model outputs **probability**: $\hat{y} \in [0, 1]$
2. Apply **threshold** (usually 0.5) to classify
3. If $\hat{y} \geq 0.5 \rightarrow$ Class 1, else Class 0

Example

$$\hat{y} = 0.87 \xrightarrow[\text{(Spam)}]{\text{threshold } 0.5} \textbf{Class 1}$$

What is the loss function?

Decision Boundary Visualization



Goal: Find optimal boundary that maximizes class separation

Can We Use MSE Loss for Classification?

أكاديمية كاوست
KAUST ACADEMY

Problem with MSE

MSE measures **numeric distance**, but in classification, we care about **probability confidence**, not distance!

Can We Use MSE Loss for Classification?

أكاديمية كاوست
KAUST ACADEMY

Problem with MSE

MSE measures **numeric distance**, but in classification, we care about **probability confidence**, not distance!

What We Really Want

A loss function that **heavily penalizes confident wrong predictions**

Example: Classifying an image as a dog with 98% confidence when it's actually a cat should incur massive penalty!

Can We Use MSE Loss for Classification?

أكاديمية كاوست
KAUST ACADEMY

Problem with MSE

MSE measures **numeric distance**, but in classification, we care about **probability confidence**, not distance!

What We Really Want

A loss function that **heavily penalizes confident wrong predictions**

Example: Classifying an image as a dog with 98% confidence when it's actually a cat should incur massive penalty!

Solution: The Logarithm!

The logarithm function has exactly the property we need:
it grows exponentially as confidence in the **wrong** class increases

The Power of the Logarithm

Intuition (True Label $y = 1$):

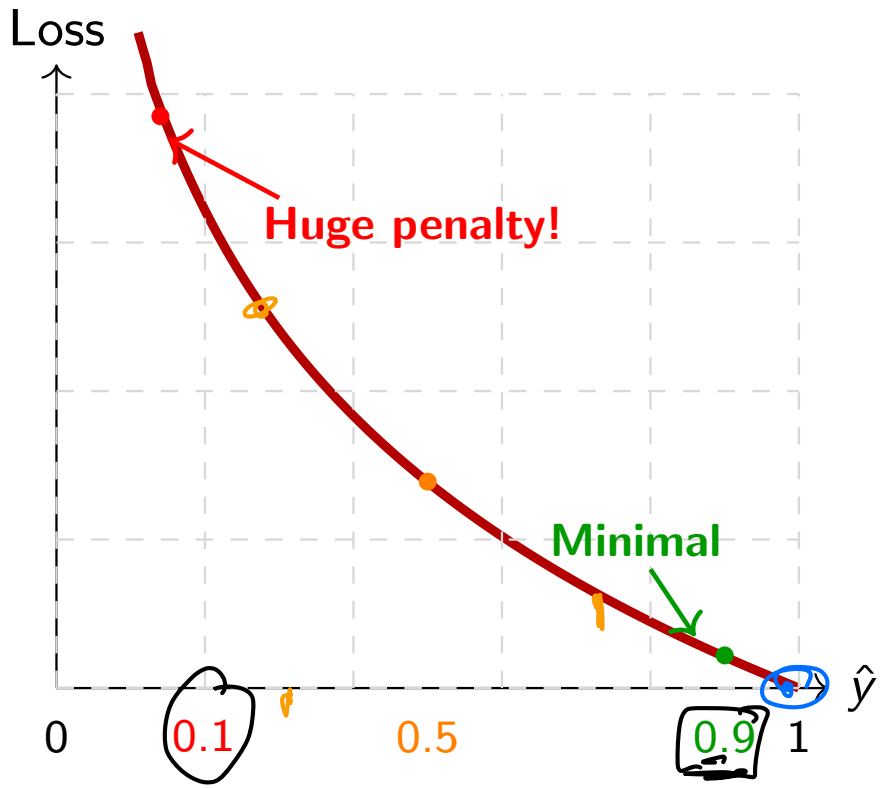
Model outputs probability $\hat{y}$ for class 1

Log Penalty $= -\log(\hat{y})$ behaves as:

- $\hat{y} = 0.99 \rightarrow$ Loss ≈ 0.01 ✓ **Great!**
- $\hat{y} = 0.50 \rightarrow$ Loss ≈ 0.69 **Uncertain**
- $\hat{y} = 0.01 \rightarrow$ Loss ≈ 4.6 ✗ **Disaster!**

The same concept applies when the true label is 0.

Loss vs. Predicted Probability
(when true label = 1)



Curve explodes as $\hat{y} \rightarrow 0$
Forces model to be cautious!

In classification $y = \{0, 1\}$

$\hat{y} = \{0, 1\}$

$y = 0$

$\hat{y} = 1$

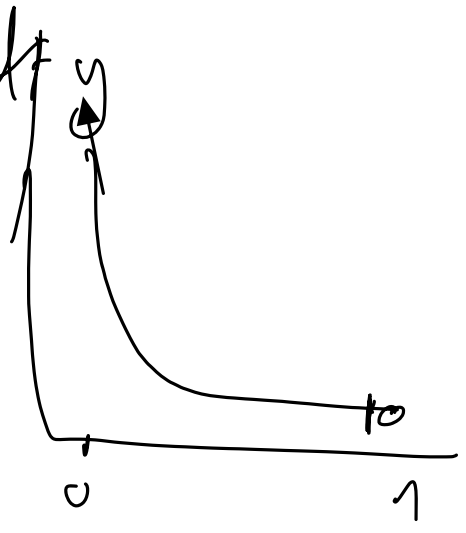
$(1 - 0)^2 = 1$

Class $y=0$

if $y=1$ $\quad -\log(\hat{y}) \rightarrow$

if $y=0$ $\quad \rightarrow -\log(1-\hat{y})$

log perfect



loss funct.

$$L = \begin{cases} -\log(\hat{y}) & \text{if } y=1 \\ -\log(1-\hat{y}) & \text{if } y=0 \end{cases}$$

$L(y,\hat{y})$.

$$\mathbb{1}(x) = \begin{cases} 1 & \text{if } x>0 \\ 0 & \text{o.w} \end{cases}$$

$y=0$

if $y=1$

$$L = y(-\log(\hat{y})) + (1-y)(-\log(1-\hat{y}))$$

$1-0$

$$L = -\left(y \log(\hat{y}) + (1-y) \log(1-\hat{y}) \right)$$

if $y = 1$: $L = -\left(1 (\log(\hat{y})) + (1-1) \log(1-\hat{y}) \right)$

$= -\log(\hat{y})$

if $y = 0$: $L = -\left[0 \log(\hat{y}) + (1-0) \log(1-\hat{y}) \right]$

$= -\log(1-\hat{y})$

Binary Cross-Entropy Loss (Log Loss)

Loss Function: Binary Cross-Entropy (Log Loss)

$$J(\mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

where: $y_i \in \{0, 1\}$ = true label, $\hat{y}_i \in (0, 1)$ = predicted probability, $\mathbf{w}$ = model weights.

- When $y_i = 1$: Loss $= -\log(\hat{y}_i)$ (penalizes low probability for class 1)
- When $y_i = 0$: Loss $= -\log(1 - \hat{y}_i)$ (penalizes high probability for class 1)
- Both terms encourage the model to predict probabilities close to the true labels.

But What If We Have More Than Two Classes?

Binary Classification:

- $y \in \{0, 1\}$
- Two classes only
- Example: Spam / Not Spam

Multiclass Classification:

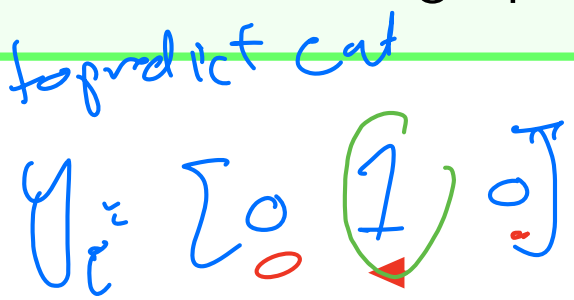
- $y \in \{0, 1, 2, \ldots, K-1\}$
- K classes ($K > 2$)
- Example: Dog / Cat / Bird

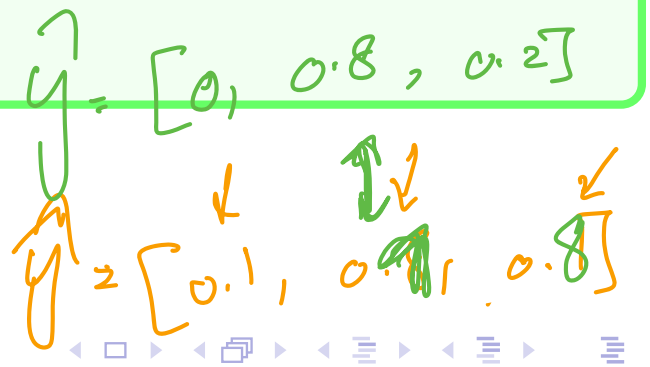
Example: Image Classification

Input: Image $\rightarrow$ Output: $\hat{\mathbf{y}} = [0.1, 0.7, 0.2]$ for 3 classes

Class probabilities: Dog (10%), Cat (70%), Bird (20%)

Model now outputs a **probability distribution** over all K classes instead of a single probability





Multi-class classification

y_i is a vector of length K = # of classes

if impredicting class (2)

$$y_i = [0 \; 1 \; 0], \quad \hat{y} = [0.1, 0.7, 0.2]$$

$$-\sum_{k=1}^{K} y_{i,k} \log(\hat{y}_{i,k})$$

$$= -[0 \log(0.1) + 1 \log(0.7) + 0 \log(0.2)]$$

$k=1$ $k=2$ $k=3$

$$= -\log(0.7)$$

yesterday we argued that the loss for

$\hat{y} = [0.1, 0.7, 0.2]$

is the same as

$\hat{y} = [0.2, 0.7, 0.1]$ (dog, cat, bird)

argmax $\hat{y}$

if $\hat{y}: [0.1, 0.2, 0.7]$ → bird
(x)

$L = -\log(0.7)$

When it's a bird

$y: [0 \quad 0 \quad 1]$

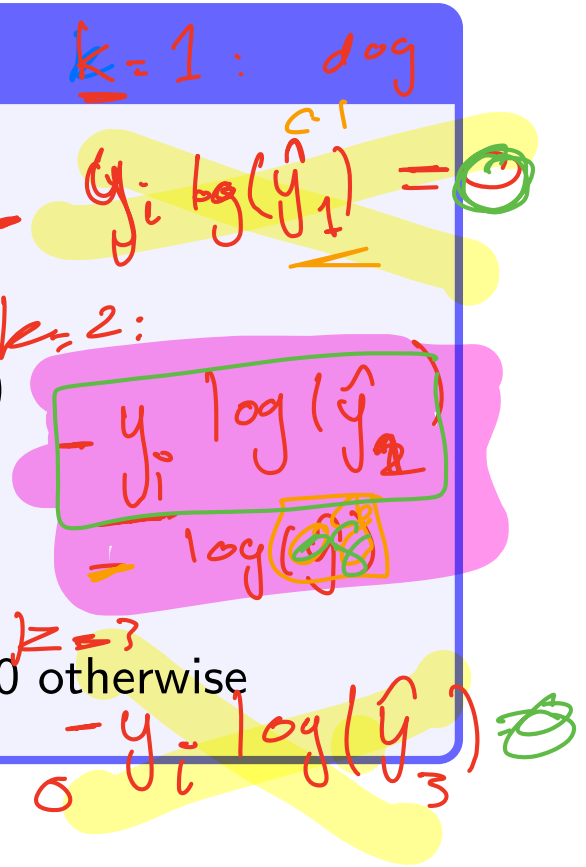
Cross-Entropy (Categorical Cross-Entropy)

Loss Function: Cross-Entropy

Generalization of binary cross-entropy to K classes

$$J(\mathbf{w}) = -\frac{1}{n} \underbrace{\sum_{i=1}^{n}}_{\text{samples}} \underbrace{\left[\sum_{k=1}^{K} y_{i,k} \log(\hat{y}_{i,k}) \right]}_{\text{classes}}$$

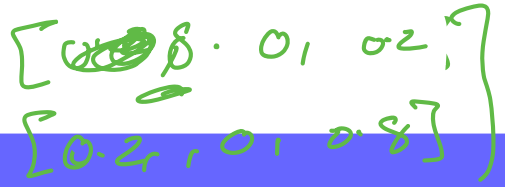
where $y_{i,k} \in \{0, 1\}$ is 1 if sample i belongs to class k, 0 otherwise



How It Works:

- Model outputs probabilities for all K classes: $\hat{\mathbf{y}}_i = [\hat{y}_{i,1}, \hat{y}_{i,2}, \ldots, \hat{y}_{i,K}]$
- True label is one-hot encoded: $\mathbf{y}_i = [0, 1, 0, \ldots, 0]$ if class 2
- Loss penalizes the probability assigned to the **true class**

Cross-Entropy (Categorical Cross-Entropy)



Loss Function: Cross-Entropy

Generalization of binary cross-entropy to K classes

$$J(\mathbf{w}) = -\frac{1}{n} \underbrace{\sum_{i=1}^{n}}_{\text{samples}} \underbrace{\sum_{k=1}^{K}}_{\text{classes}} y_{i,k} \log(\hat{y}_{i,k})$$

where $y_{i,k} \in \{0, 1\}$ is 1 if sample i belongs to class k, 0 otherwise

Example

True class: Cat (class 2), Prediction: $[0.1, 0.7, 0.2]$
Loss $= -\log(0.7) \approx 0.36$ (only the true class matters!)

The Three Pillars of Learning

1. The Model (f_θ)	2. Loss Function (J)	3. Optimizer
Design Choice: ML (Linear, Kernel, Trees) & DL (MLP, CNN)	**Design Choice:** MSE vs. Cross-Entropy	**Design Choice:** Gradient Descent, Adam
Depends on: Data Type (Tabular, image) & Complexity	**Depends on:** The Task (Regression vs. Classification)	**Depends on:** Resources & Scale (Covered later in Deep Learning)

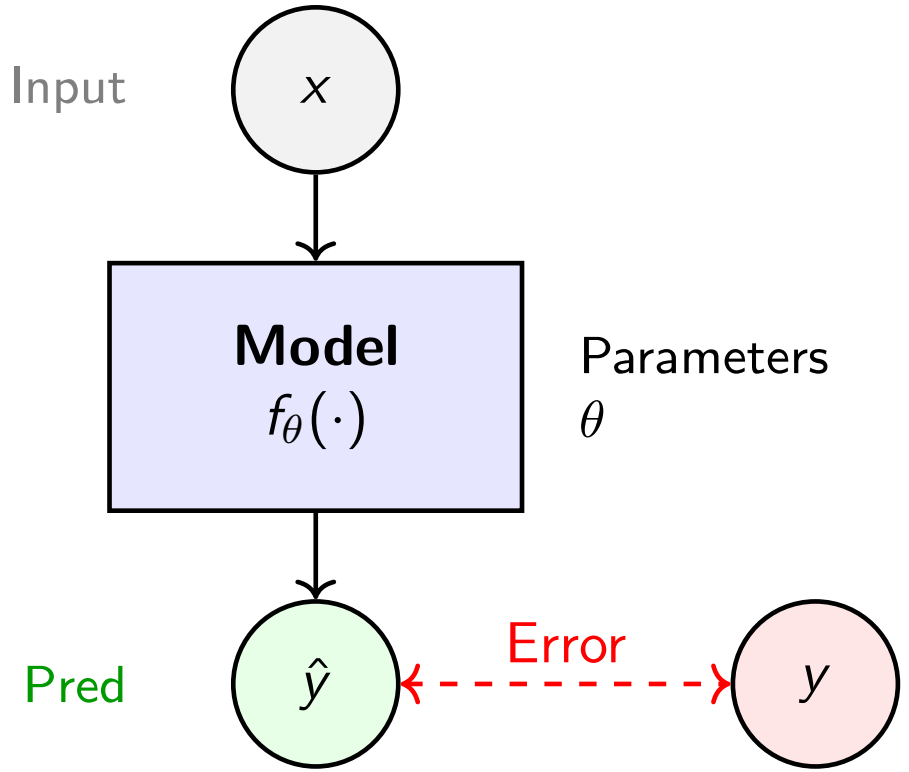
We have covered the **Loss Functions**.
Now, let's move on to the **Models** themselves (f_θ).

What is a Model?

Definition

A **Model** (f_θ) is a simplified mathematical representation of reality. It maps inputs (x) to outputs $(\hat{y})$ using learned parameters (θ).

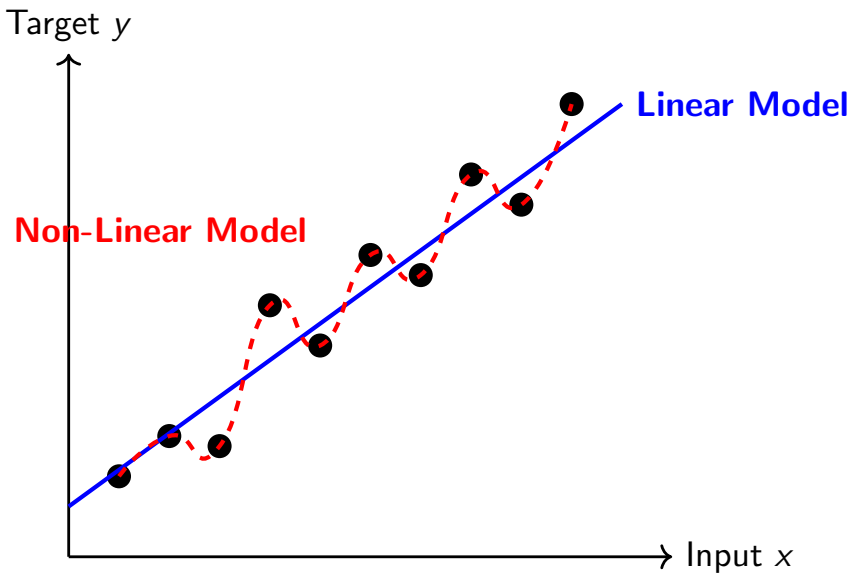
The Goal: Find the "best" function f that minimizes the error between prediction $\hat{y}$ and reality y.



Models Complexity

The relationship between inputs **x** and target y varies wildly. Choosing the correct model family matters to avoid **Underfitting** (too simple) or **Overfitting** (too complex).

- **Simple:** Linear.
  (e.g., House Size → Price)
- **Complex:** Non-linear.
  (e.g., Image Pixels → "Cat")



Machine Learning vs. Deep Learning

أكاديمية كاوست
KAUST ACADEMY

1. Machine Learning (ML)

Algorithms that parse data, learn patterns from it, and apply learned decisions.

Characteristics:

- Best for **Structured Data** (Tables).
- Require **Feature Engineering**.
- Lightweight & Interpretable (mostly).

Examples: *Linear Reg, SVM, Trees*

2. Deep Learning (DL)

A subset of Machine Learning based on artificial neural networks.

Characteristics:

- Best for **Unstructured Data** (Images, Text, Audio).
- **Automated** Feature Extraction.
- Data & Compute Hungry.

Examples: *MLPs, CNNs, Transformers*

Machine Learning vs. Deep Learning

1. Machine Learning (ML)

Algorithms that parse data, learn patterns from it, and apply learned decisions.

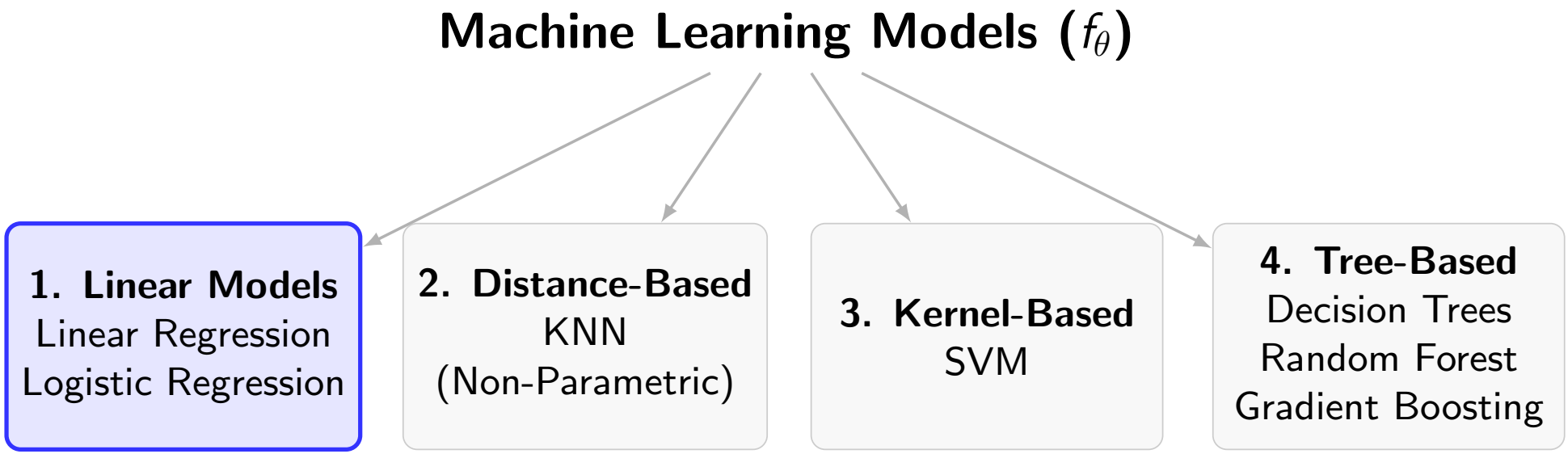
Characteristics:

- Best for **Structured Data** (Tables).
- Require **Feature Engineering**.
- Lightweight & Interpretable (mostly).

Examples: *Linear Reg, SVM, Trees*

Let's start with Machine Learning first.

The ML Toolbox: Types of Models



↑ **We Start Here!**

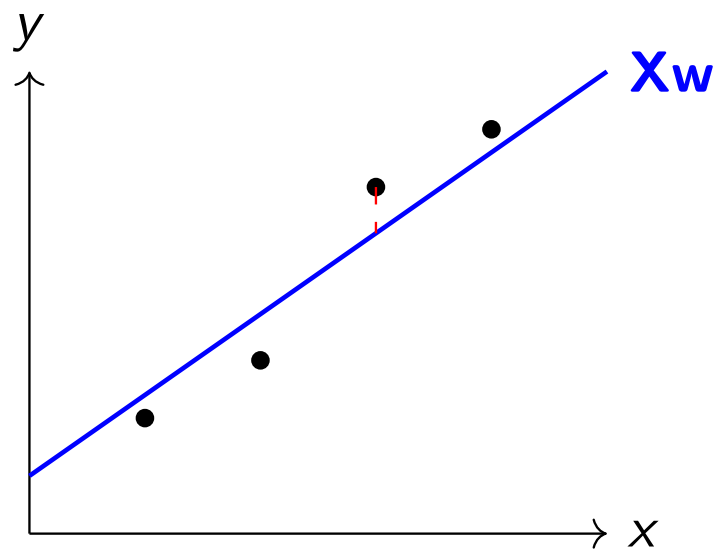
Linear Regression

We start with the simplest assumption: The target y is a **linear combination** of the input features.

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$$

The Components:

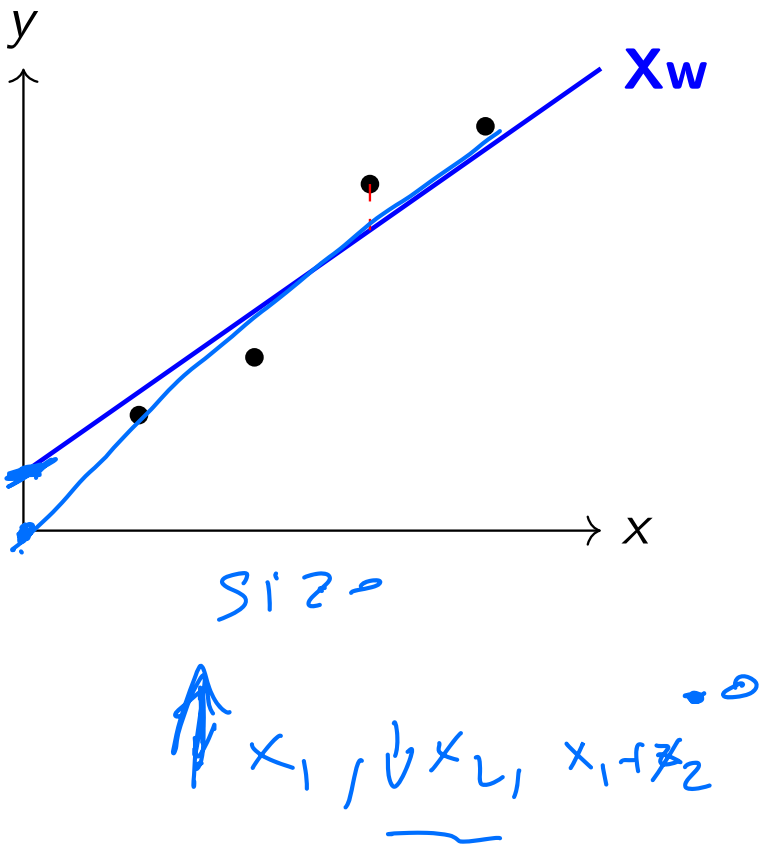
- $\hat{y}$: predicted value
- $x_1, \ldots, x_d$: the input features.
- $w_1, \ldots, w_d$: the weights. How much each feature matters.
- w_0: the bias (Intercept).



We start with the simplest assumption: The target y is a **linear combination** of the input features.

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$$

$\hat{price} = w_0 + w_{area} \cdot area + w_{bdroom} \cdot bdrooms + \cdots$



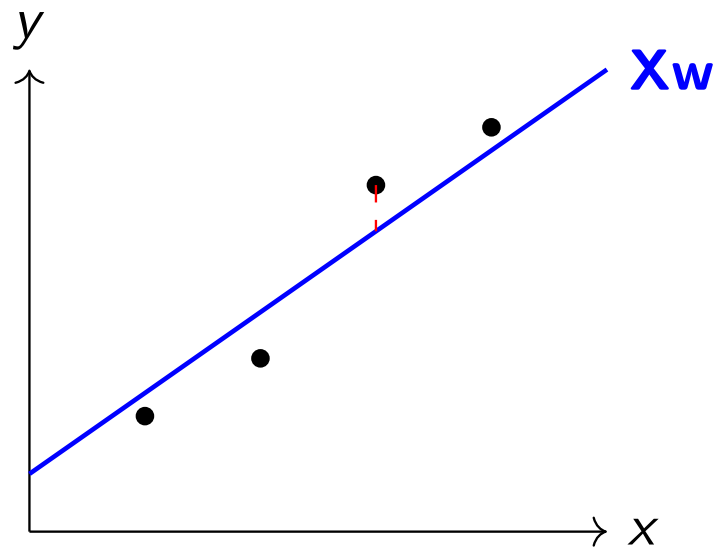
The Components:

- $\hat{y}$: predicted value
- $x_1, \ldots, x_d$: the input features.
- $w_1, \ldots, w_d$: the weights. How much each feature matters.
- w_0: the bias (Intercept).

Linear Regression

We start with the simplest assumption: The target y is a **linear combination** of the input features.

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$$



The Components:

- $\hat{y}$: predicted value
- $x_1, \ldots, x_d$: the input features.
- $w_1, \ldots, w_d$: the weights. How much each feature matters.
- w_0: the bias (Intercept).

Writing long sums ($\sum$) is tedious. We use Linear Algebra to represent Linear Regression in matrices notation.

Any linear (affine) function can be represented using a matrix & vector

Translating to Vectors (Vectorization)

Step 1: The Bias Trick: We add a "dummy" feature $x_0 = 1$ to the input so we have each weight multiplied by a x feature.

$$\hat{y} = w_0(1) + w_1 x_1 + \cdots + w_d x_d$$

Step 2: Vector Notation (For one sample): We rewrite this equation in a vector dot product form:

$$\hat{y} = \begin{bmatrix} w_0 & w_1 & \dots & w_d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} = \mathbf{w}^T \mathbf{x}$$

Step 3: Matrix Notation (For all data): For N samples, prediction becomes a simple Matrix-Vector multiplication:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$$

$X \in \mathbb{R}^{N \times d}$

Finding the Best Line

Now we have the equation. But how do we choose the best $\mathbf{w}$?

Finding the Best Line

$\hat{y} = Xw$ MSE

Now we have the equation. But how do we choose the best **w**?
We need the loss and the optimizer!

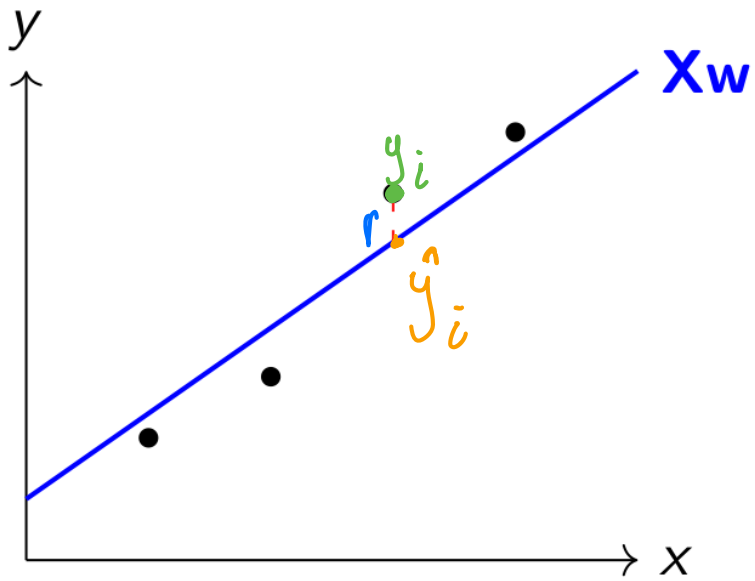
$$L = \frac{1}{N} \sum_{n=1}^{N} (y - \hat{y})^2$$

Finding the Best Line

Now we have the equation. But how do we choose the best **w**?
We need the loss and the optimizer!

Let's wrap our model in the Mean Squared Error:

$$J(\mathbf{w}) = \frac{1}{N}||\mathbf{Xw} - \mathbf{y}||_2^2$$



Finding the Best Line

Now we have the equation. But how do we choose the best $\mathbf{w}$?
We need the loss and the optimizer!

Let's wrap our model in the Mean Squared Error:

$$J(\mathbf{w}) = \frac{1}{N}||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2$$

Now, how do we find the $\mathbf{w}$ that minimizes this $J(\mathbf{w})$?

$$w^* = \underset{w}{\operatorname{argmin}}\, J(w)$$

Finding the Best Line

Now we have the equation. But how do we choose the best **w**?
We need the loss and the optimizer!

Let's wrap our model in the Mean Squared Error:

$$J(\mathbf{w}) = \frac{1}{N}||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2$$

Now, how do we find the **w** that minimizes this $J(\mathbf{w})$?

Option A: Analytical
(Solve for zero gradient)

OR

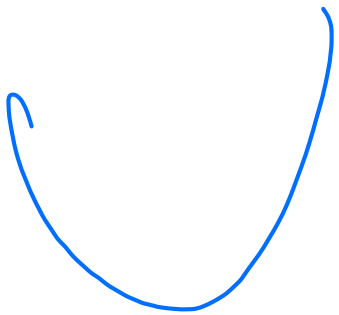
Option B: Iterative
(Gradient Descent)

Method A: The Normal Equation

For linear models, MSE is **convex**. **Q:** What does that tell you?

Method A: The Normal Equation

For linear models, MSE is **convex**. **Q:** What does that tell you?
Guaranteed Global Minimum!.



Method A: The Normal Equation

For linear models, MSE is **convex**. **Q:** What does that tell you? *Guaranteed Global Minimum!*.

Step 1: The Gradient Derived via Chain Rule:

$$\nabla J(\mathbf{w}) = \frac{2}{N}\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y})$$

Method A: The Normal Equation

For linear models, MSE is **convex**. **Q:** What does that tell you? *Guaranteed Global Minimum!*.

Step 1: The Gradient Derived via Chain Rule:

$$\nabla J(\mathbf{w}) = \frac{2}{N}\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y})$$

Step 2: Set to Zero

$$\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y} = 0$$

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{y}$$

Method A: The Normal Equation

For linear models, MSE is **convex**. **Q:** What does that tell you? *Guaranteed Global Minimum!*.

Step 1: The Gradient Derived via Chain Rule:

$$\nabla J(\mathbf{w}) = \frac{2}{N}\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y})$$

Step 2: Set to Zero

$$\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y} = 0$$

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{y}$$

Step 3: Solve for w Multiply both sides by inverse $(\mathbf{X}^T\mathbf{X})^{-1}$:

$O(N^3)$

The Normal Equation

$$\mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

→ unique if X^TX is invertible

Cons: Inverting large matrices is very slow ($O(d^3)$). Doesn't work if X is singular.

Method B: Gradient Descent

Instead of jumping to the solution, we take small steps.

Step 1: Calculate Gradient Derived via Chain Rule:

$$\nabla J(\mathbf{w}) = \frac{2}{N}\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y})$$

Step 2: Update Rule Move opposite to the gradient to reduce error.

Iterative Update

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \alpha \nabla J$$

α: Learning Rate (Step size)

Pros: Very efficient for large datasets.

Revisiting Overfitting

Imagine we train two linear models on the same data. Both achieve **Training MSE = 0.10**. Which one is better?

Model A

$$y = 0.5x_1 + 0.3x_2$$

Model B

$$y = 500.5x_1 - 429.7x_2$$

Revisiting Overfitting

Imagine we train two linear models on the same data. Both achieve **Training MSE = 0.10**. Which one is better?

Model A (Small Weights)

$\hat{y} = 0.5x_1 + 0.3x_2$

Val MSE: 0.12

Stable. Why?
$\rightarrow$ Small changes in x cause **Small** changes in y.

Model B (Huge Weights)

$\hat{y} = 500.5x_1 - 429.7x_2$

Val MSE: 15.0 (!!!)

Unstable. Why?
$\rightarrow$ Small changes in x cause **Massive** changes in y.

Revisiting Overfitting

Imagine we train two linear models on the same data. Both achieve **Training MSE = 0.10**. Which one is better?

Model A (Small Weights)

$y = 0.5x_1 + 0.3x_2$

Val MSE: 0.12

Stable. Why?
$\rightarrow$ Small changes in x cause **Small** changes in y.

Model B (Huge Weights)

$y = 500.5x_1 - 429.7x_2$

Val MSE: 15.0 (!!!)

Unstable. Why?
$\rightarrow$ Small changes in x cause **Massive** changes in y.

Observation: Large weights are often a symptom of **Overfitting**. The model is trying too hard to wiggle through every training point.

The Fix: Constraining the Weights

We must force the model to keep weights small. We do this by adding a **Constraint** (Penalty) to the loss function.

Minimize Error + Minimize Weights

$$J_{final} = \overbrace{J_{MSE}}^{\text{Fit the Data}} + \overbrace{\lambda \cdot R(\mathbf{w})}^{\text{Keep Weights Small}}$$

What is λ (Lambda)?

Regularization Strength (Hyperparameter):

- **High λ:** Huge penalty $\rightarrow$ Tiny weights (Model becomes simple) $\rightarrow$ Increases underfitting.
- **Low λ:** Small penalty $\rightarrow$ Weights can grow (Model fits data closely) $\rightarrow$ Increases overfitting.
- $\lambda = 0$**:** Standard Linear Regression (No penalty).

Regularization Example

Let's set $\lambda = 1$ and Penalty $R(\mathbf{w}) = w^2$. Compare two models that have similar error, but different weights.

Model A (Overfit)

Weight $w = 100$
MSE Error $= 0$ (Perfect fit)

Total Loss:

$$J = 0 + 1 \cdot (100)^2$$

$$\mathbf{J = 10,000}$$

HUGE LOSS!

Model B (Balanced)

Weight $w = 2$
MSE Error $= 5$ (Some error)

Total Loss:

$$J = 5 + 1 \cdot (2)^2$$

$$\mathbf{J = 9}$$

WINNER!

Regularization Types

How do we define the penalty $R(\mathbf{w})$? We have two main flavors:

- **Ridge (L2):** $R(\mathbf{w}) = \sum w^2$ (Squared) → even weight distribution
- **Lasso (L1):** $R(\mathbf{w}) = \sum |w|$ (Absolute) → concentrate weight on small set of features

least absolute shrinkage and selection operator

Ridge Regression (L2)

We penalize the **Squared Magnitude** of the weights.

Objective Function

$$J(\mathbf{w}) = \text{MSE} + \lambda \sum w^2$$

Effect: "Weight Decay"

- Shrinks weights smoothly toward 0 (stability).
- Usually does not make weights exactly 0 (not sparse).
- Works well with correlated features.

Lasso Regression (L1)

We penalize the **Absolute Value** of the weights.

Objective Function

$$J(\mathbf{w}) = \text{MSE} + \lambda \sum |w|$$

Effect: "Sparsity"

- Encourages **sparsity**: many weights become exactly 0.
- Acts as **feature selection**.
- Good when we believe **only a few features matter**.

How do we find the optimal weights w?

1. Ridge (L2): Has a Closed-Form Solution

Ridge Normal Equation

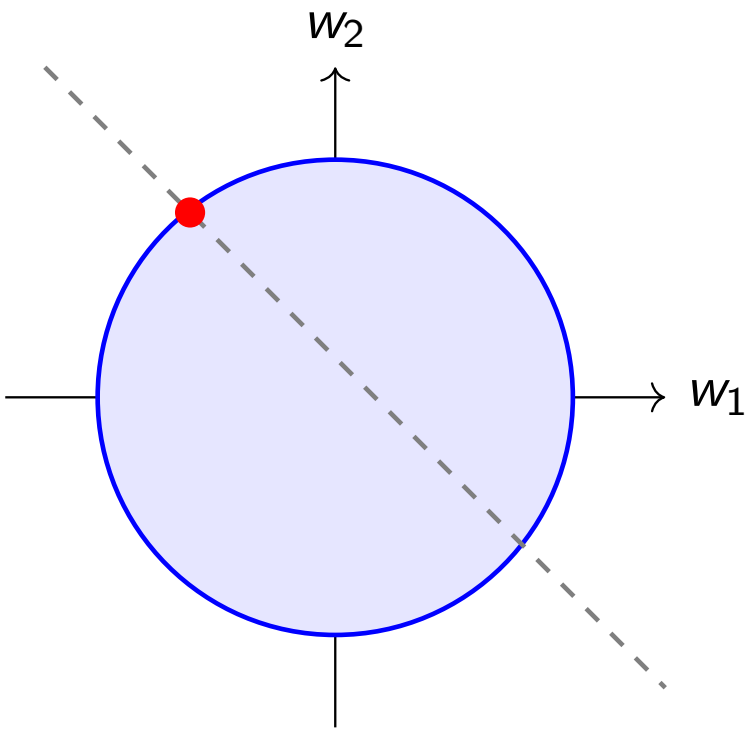
$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

**Note: Adding $\lambda\mathbf{I}$ ensures the matrix is always invertible.*

2. Lasso (L1): No Closed-Form Solution

- Because the absolute value function $|w|$ is not differentiable at 0, we cannot derive a simple formula.
- We must use **Iterative Algorithms** to find the minimum.

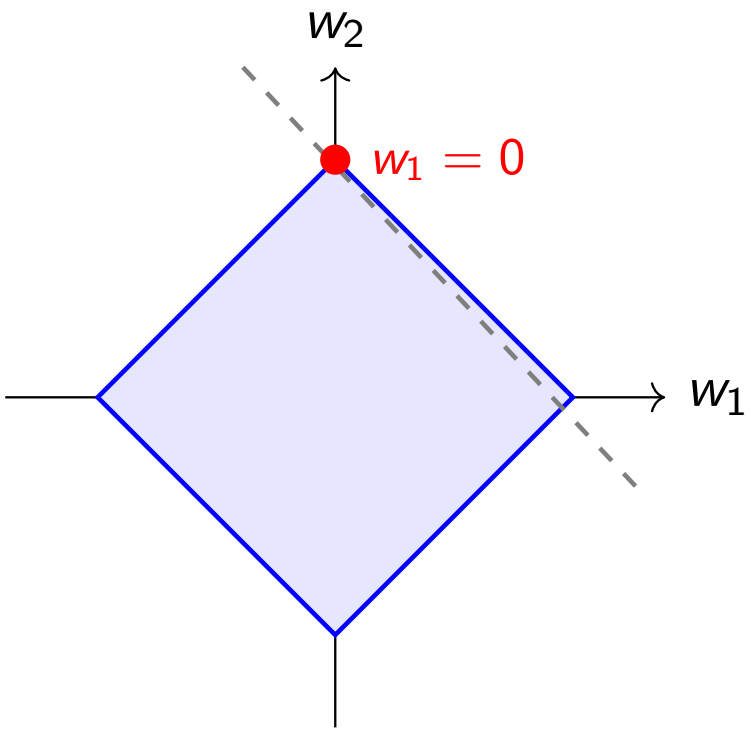
L1 vs L2: Geometric Intuition



Loss Contour + L2 Constraint

Ridge: Smooth solution, rarely hits axis

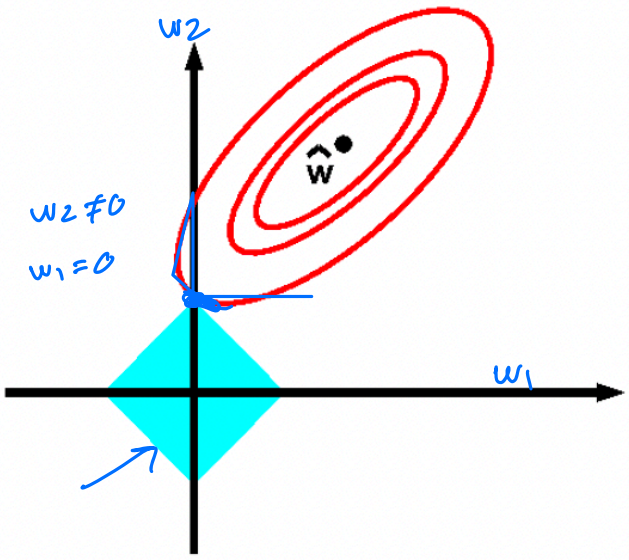
$w_1 = -1.5, w_2 = 2.0$



Loss Contour + L1 Constraint

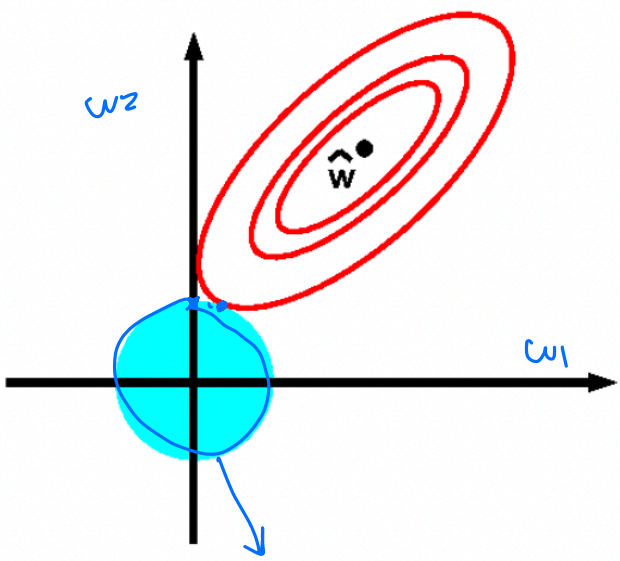
Lasso: Sharp corners promote sparsity

$w_1 = 0, w_2 = 2.5$



$\lambda \|w\|_1$

$= \lambda(|w_1| + |w_2|)$



$\lambda \|w\|_2^2$

$= \lambda(w_1^2 + w_2^2)$

Adapting Linear Regression for Classification

$y \in \mathbb{R}$

The Challenge: We want to classify data (e.g., Spam vs Not Spam).

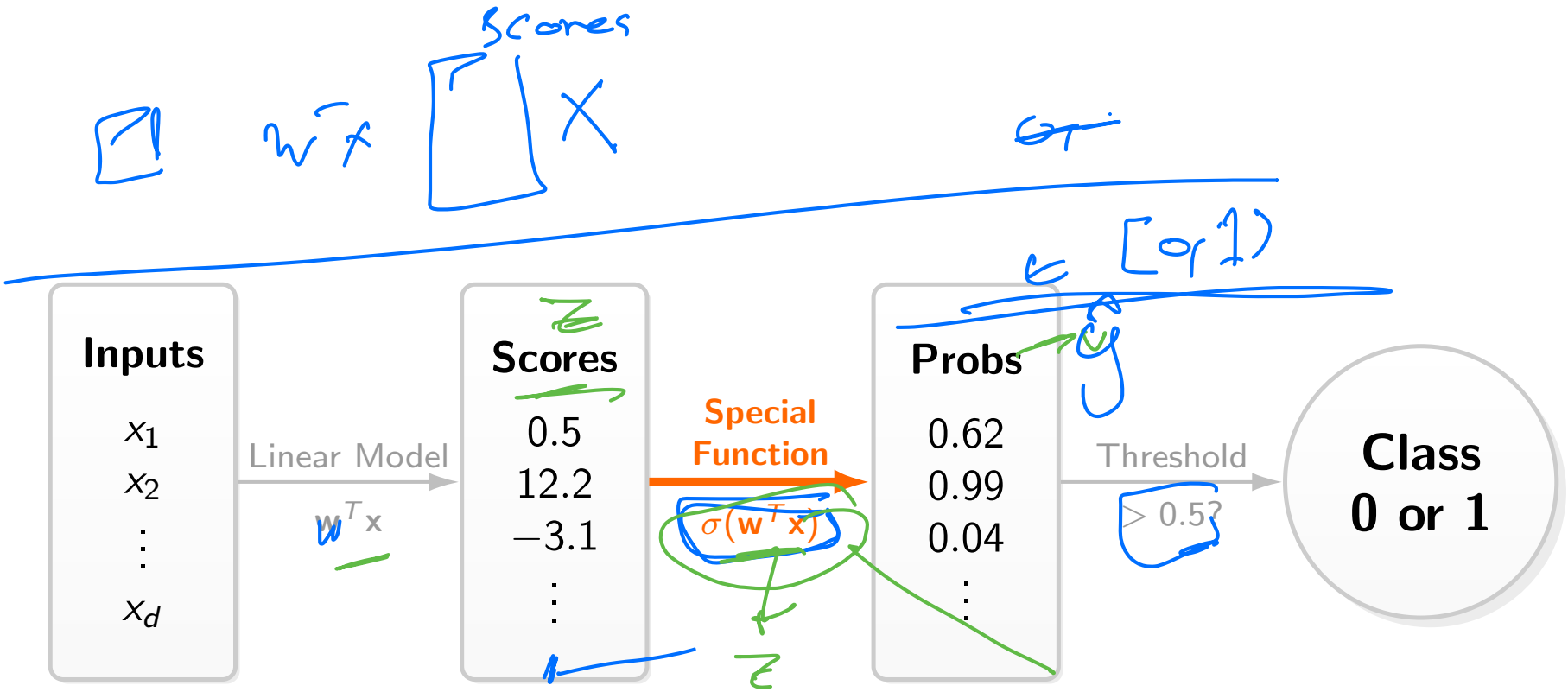
However, our Linear Model $f(x) = \mathbf{w}^T\mathbf{x}$ outputs numbers in $(-\infty, \infty)$.

Problem

If model outputs 500 or -20, how do we interpret that as a probability?

We need a mechanism to map these raw "scores" into a valid probability range $[0, 1]$.

The Classification Pipeline

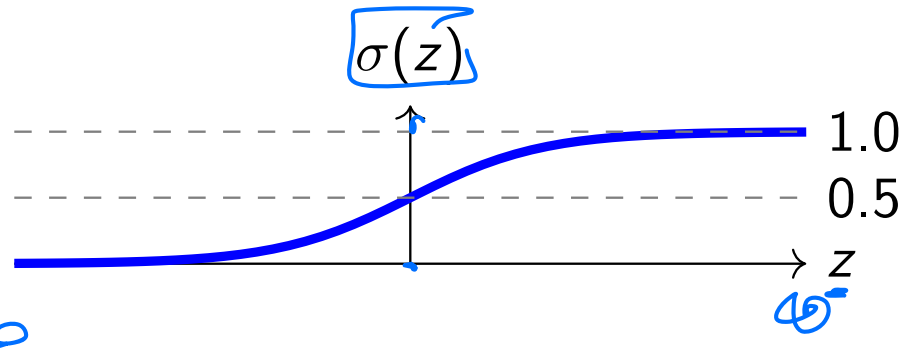


The Solution: The Sigmoid Function

The "Special Function" we need is the **Sigmoid** (Logistic Function).

Sigmoid Function

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Properties:

- Squashes any z into $(0, 1)$.
- $z = 0 \implies \hat{y} = 0.5$.
- Differentiable everywhere (Smooth).

$\hat{y} = \sigma(z)$, $z = W^T X$

The Gradient: Chain Rule Breakdown

To train, we need to minimize the Binary Cross-Entropy Loss:

$$J = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

Let's compute $\frac{\partial J}{\partial w}$ using the Chain Rule:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

The Three Components:

1. **Loss Derivative:** $\frac{\hat{y} - y}{\hat{y}(1-\hat{y})}$
2. **Sigmoid Derivative:** $\hat{y}(1 - \hat{y})$
3. **Linear Derivative:** x

(Can you derive this?)

The Gradient "Magic"

Now, multiply them together:

$$\frac{\partial J}{\partial w} = \left(\frac{\hat{y} - y}{\hat{y}(1-\hat{y})} \right) \cdot \hat{y}(1-\hat{y}) \cdot x$$

The complicated Sigmoid derivative perfectly cancels the denominator of the Log Loss!

Final Gradient Update

$$\nabla J(\mathbf{w}) = \frac{1}{N} \mathbf{X}^T (\hat{\mathbf{y}} - \mathbf{y})$$

Observation

This formula is **identical** to Linear Regression (ignore constants)! The only difference is how we calculate $\hat{y}$ (Sigmoid vs. Identity).

What about multi-class GD update?

$$J(\mathbf{w}) = -\frac{1}{n}\underbrace{\sum_{i=1}^{n}}_{\text{samples}}\underbrace{\sum_{k=1}^{K}}_{\text{classes}} y_{i,k}\log(\hat{y}_{i,k})$$

z_{ik}

$$\hat{y}_{i,k} = \frac{e^{z_{ik}}}{\sum_{j=1}^{K} e^{z_{ij}}}$$

$\frac{\partial z}{\partial w} = x$

We want $\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$

$-\frac{1}{n}\left(\frac{y}{\hat{y}}\right)$

$\frac{\partial J}{\partial \hat{y}_{ik}} = \frac{-1}{n}\frac{y_{ik}}{\hat{y}_{ik}}$

$\frac{\partial \hat{y}_{ik}}{\partial z_{im}} = \hat{y}_{ik}\left(\delta - \hat{y}_{im}\right)$ (delta function if $k=m$)

$k=m$: $= \hat{y}_{ik}(1-\hat{y}_{ik})$

$k\neq m$: $= -\hat{y}_{ik}\hat{y}_{im}$

Let's work for example i:

$$\frac{\partial J}{\partial z_{im}} = \sum_{k=1}^{K}\frac{\partial J_i}{\partial \hat{y}_{ik}}\frac{\partial \hat{y}_{ik}}{\partial z_{im}}$$

$$= \sum_{k=1}^{K}\frac{-y_{ik}}{\hat{y}_{ik}}\left(\hat{y}_{ik}(\delta_{km} - \hat{y}_{im})\right)$$

$$= \sum_{k=1}^{K} -y_{ik}\left(\delta_{km} - \hat{y}_{im}\right)$$

$$= \sum_{k} -y_{ik}\delta_{km} + \sum_{k=1}^{K} y_{ik}\hat{y}_{im}$$

because of delta: $\sum_k y_{ik}\delta_{km} = y_{im}$

y_{ik} is one-hot vec.

$$= -y_{im} + \hat{y}_{im} \qquad = \hat{y}_{im}$$

$$= \hat{y}_{im} - y_{im}$$

Generalization: Multiclass (Softmax)

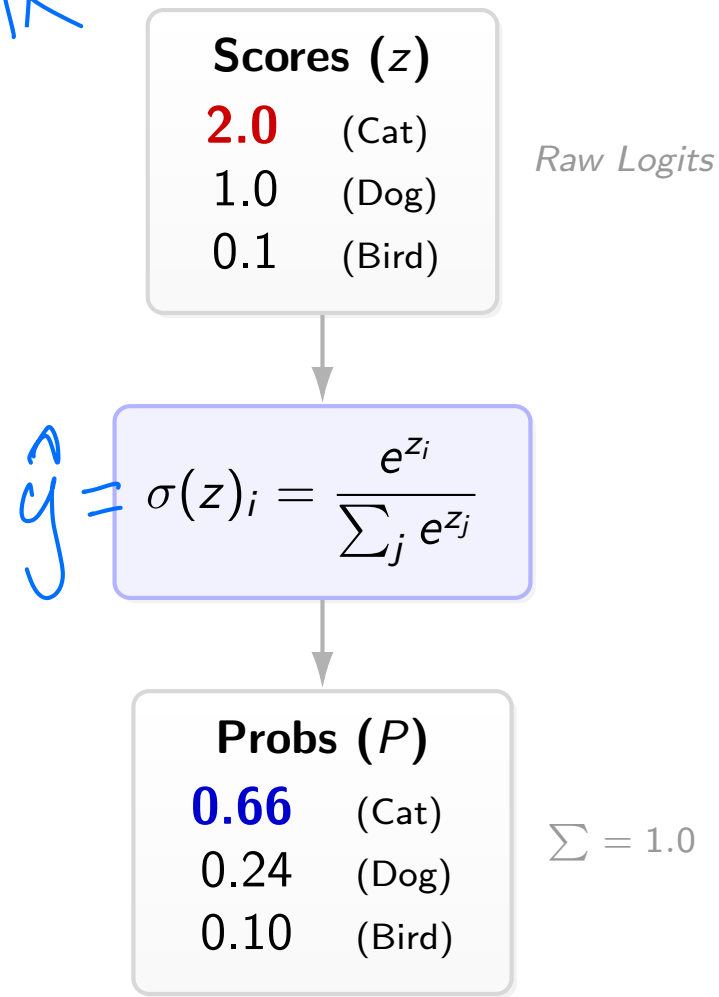
What if we have $K > 2$ classes? (e.g., Cat, Dog, Bird). We generalize Sigmoid to **Softmax**.

Softmax Function

$$P(y = k) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

- **Exponent (e^z):** Forces all scores to be positive.
- **Sum ($\sum$):** Normalizes so all probabilities sum to 1.0.
- **Loss:** Cross-Entropy.

$$J = -\frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} y_{i,k} \log(\hat{y}_{i,k})$$



Linear Models: Advantages & Disadvantages

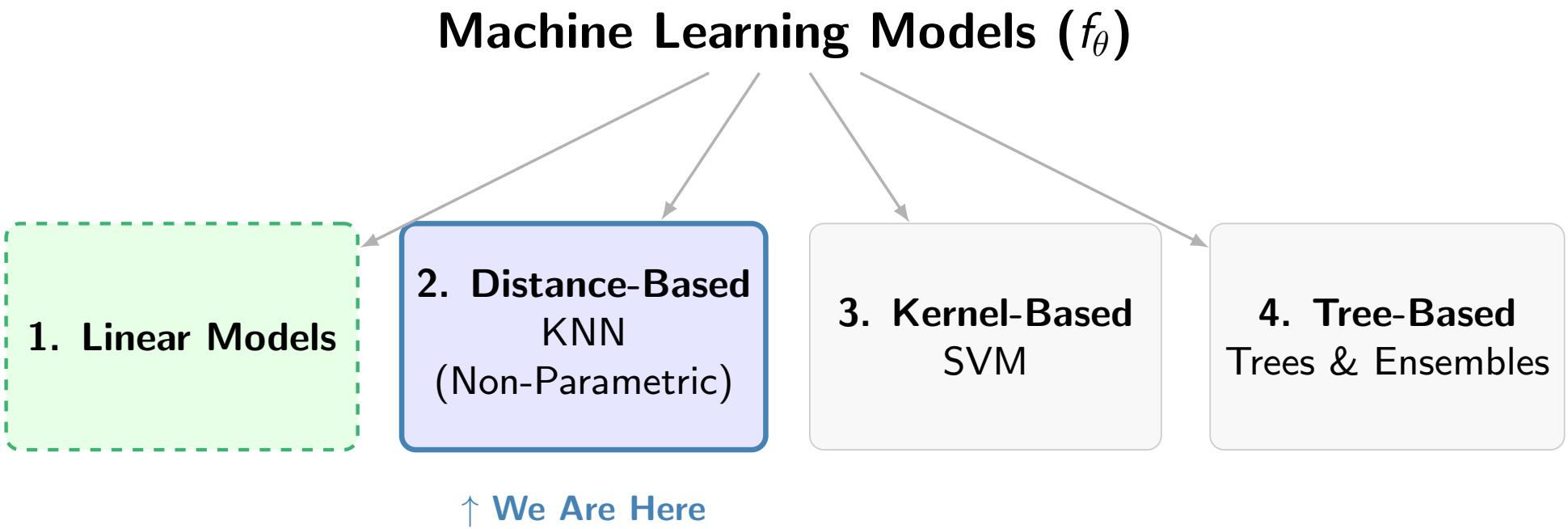
Advantages

- **Interpretable:** The weights **w** directly tell us feature importance.
- **Fast & Efficient:** Training is simple matrix math. Inference is just a dot product.

Disadvantages

- **High Bias (Underfitting):** Cannot model complex, non-linear relationships.
- **Sensitive to Outliers.** Especially when using MSE.
- **Scaling Required:** Sensitive to feature magnitude (Requires Standardization).

The ML Toolbox: Roadmap Update

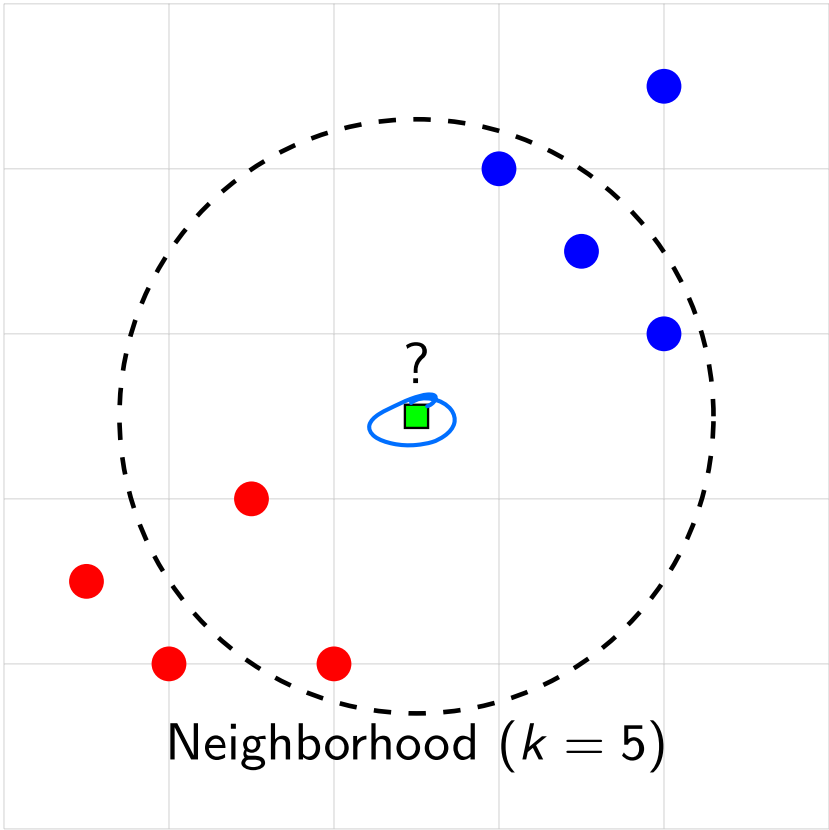


k-Nearest Neighbors (k-NN): Concept

Linear models try to learn a global formula ($\mathbf{w}^T\mathbf{x}$). **k-NN** is different: it is a **Lazy Learner**. It memorizes the training data and makes predictions based on local similarity. So, it has **no parameters**.

The Algorithm:

1. Choose a distance metric $d(\cdot, \cdot)$ and an integer k.
2. For a new point x_{new}, find the k training points closest to it.
3. **Classification:** Take the majority vote.
4. **Regression:** Take the average.



k-NN: Mathematics of Distance

The performance of k-NN depends heavily on how we define "closeness."

1. **Euclidean Distance (L_2 Norm)**: Most common

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2} = ||\mathbf{x} - \mathbf{y}||_2$$

2. **Manhattan Distance (L_1 Norm)**: Good for high dimensions

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{d} |x_i - y_i| = ||\mathbf{x} - \mathbf{y}||_1$$

k-NN: Advantages and Disadvantages

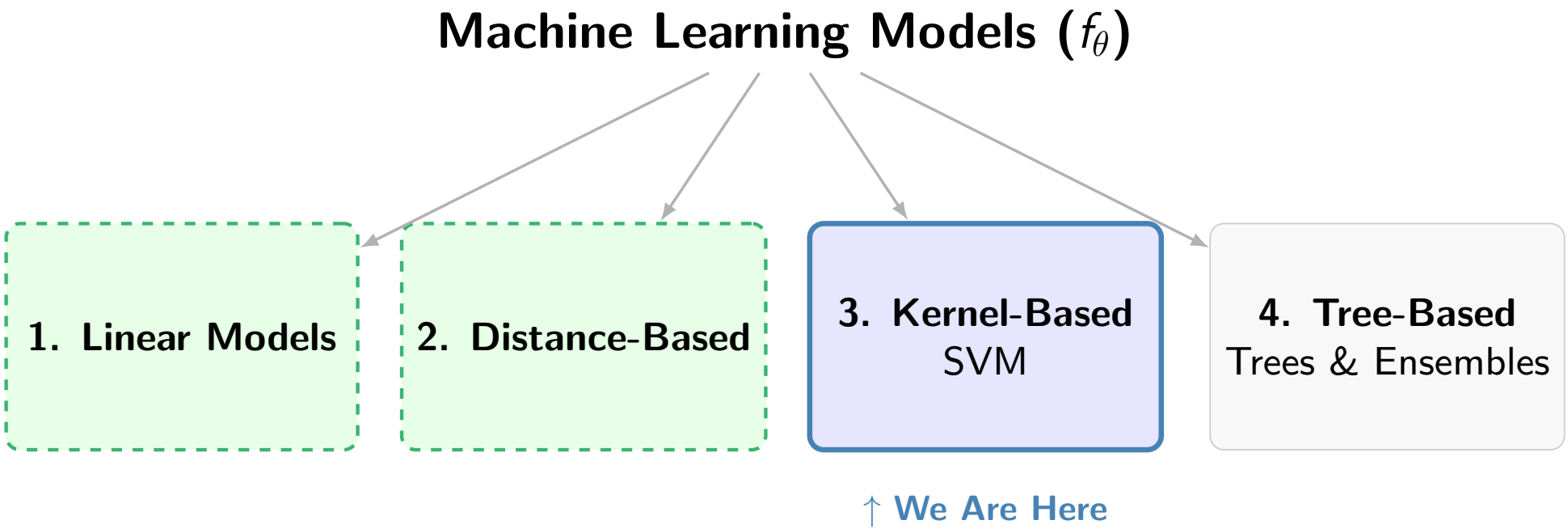
Advantages

- **Simple:** Easy to understand and implement.
- **No Training:** Training time is effectively zero ($O(1)$).

Disadvantages

- **Slow Inference:** Prediction is $O(N)$ (must scan all data).
- **Memory Intensive:** Must store the entire dataset.
- **Scaling Required:** Sensitive to feature magnitude (Requires Standardization).
- **Curse of Dimensionality:** Performance degrades noticeably in high dimensions.

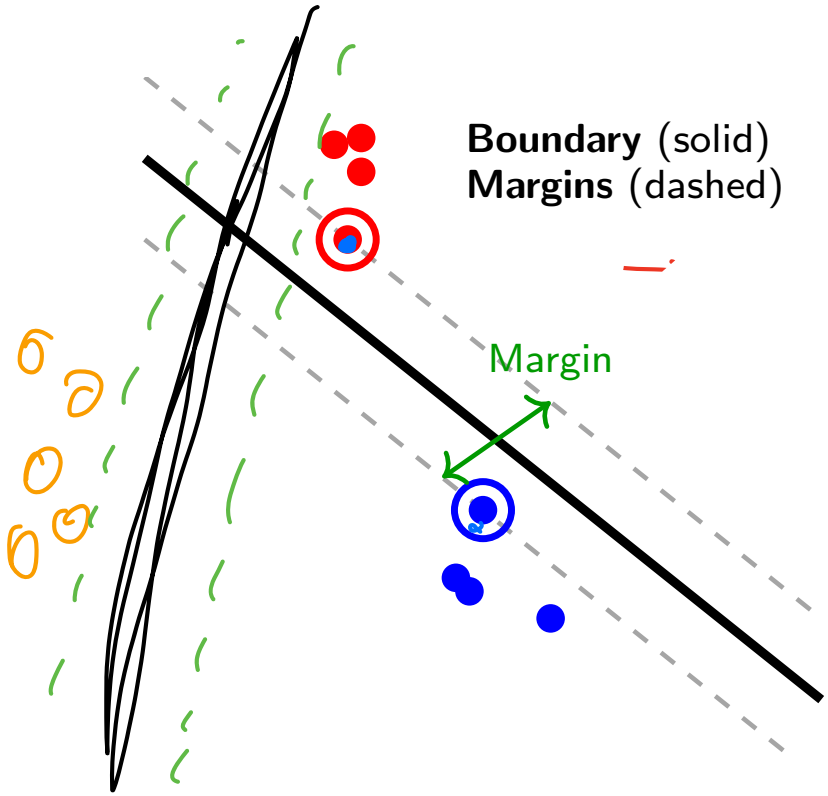
The ML Toolbox: Roadmap Update



Support Vector Machines (SVM)

"Success isn't about barely crossing the line. It's about having some room to spare."

Logistic Regression learns a boundary that fits probabilities. SVM chooses the boundary with the **largest safety gap (Margin)**.



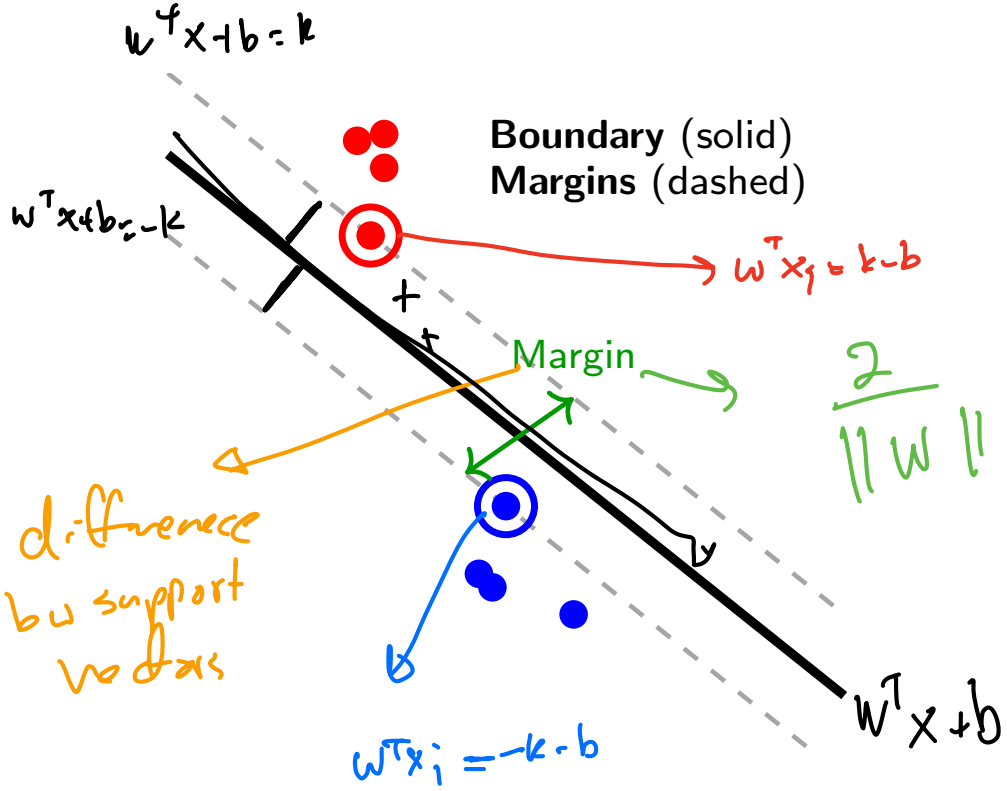
- **Hyperplane:** boundary ($\mathbf{w}^T\mathbf{x} + b = 0$).
- **Margin:** "safety gap" to the closest points.
- **Support Vectors:** closest points that **define** the boundary.

Bigger margin ⇒ more robust to noise.

Support Vector Machines (SVM)

"Success isn't about barely crossing the line. It's about having some room to spare."

Logistic Regression learns a boundary that fits probabilities. SVM chooses the boundary with the **largest safety gap (Margin)**.



- **Hyperplane:** boundary ($\mathbf{w}^T\mathbf{x} + b = 0$).
- **Margin:** "safety gap" to the closest points.
- **Support Vectors:** closest points that **define** the boundary.

Bigger margin ⇒ more robust to noise.

Hard Margin SVM

We established that we want the **widest possible margin**.
Mathematically, maximizing width $\iff$ minimizing weights $||\mathbf{w}||$.

The "Hard Margin" Objective

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2$$

$$\text{Subject to: } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$$

What does this mean?

- **Minimize Objective:** Keep the weights small.
- **Subject to Constraint:** Strictly forbid ANY data point from touching the margin (i.e., being misclassified).

**Note: This assumes data is perfectly linearly separable (Hard Margin).*

Soft Margin: When Perfection is Impossible

Real data is noisy (never perfectly linearly separable). A "Hard Margin" would crash if just one outlier existed in the data.

Solution: We relax the rules. We allow some "violations" (ξ_i) but penalize them using a hyperparameter **C**.

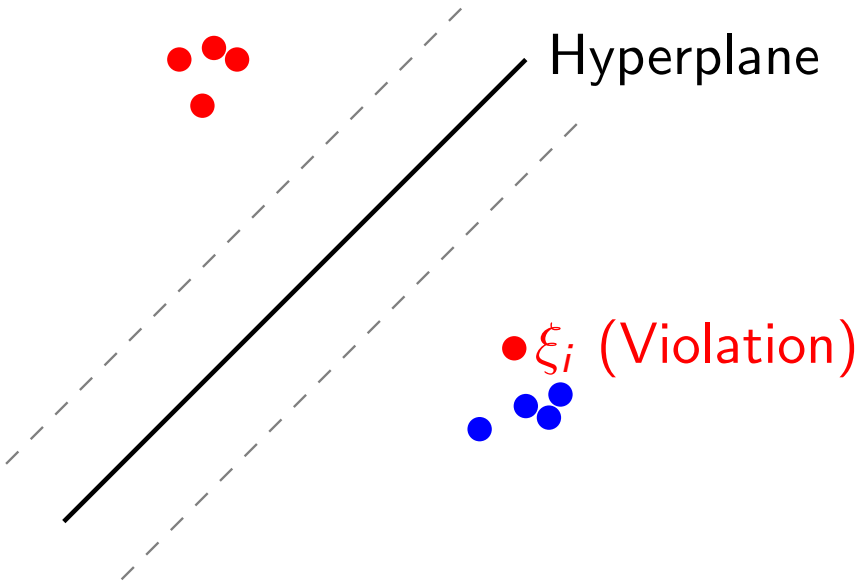
The Hyperparameter C

Large C (Strict)

- Penalizes errors heavily.
- Result: Narrow margin, risk of overfitting.

Small C (More Violations)

- Ignores small errors.
- Result: Wider margin, better generalization.

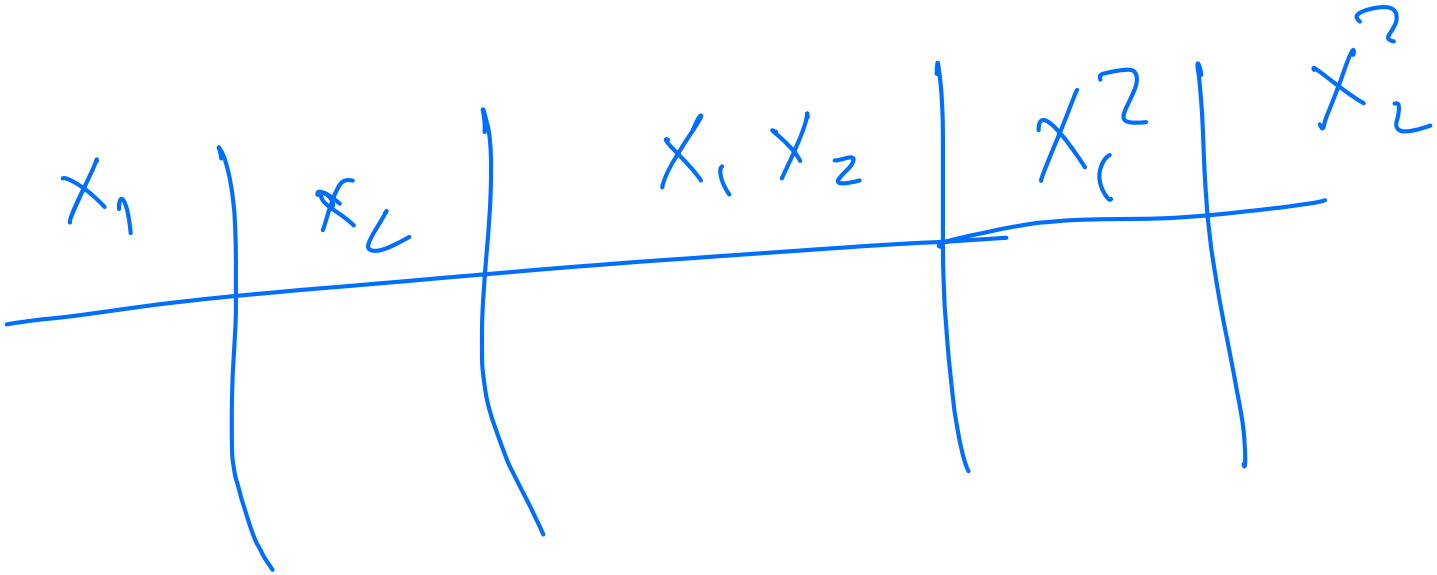


"Ideally, separate perfectly. If not, minimize the sum of violations."

$$\min \frac{1}{2}\|w\|^2 + C\sum \xi_i$$

Handling Non-Linearity: The Kernel Trick

Problem: Standard SVM finds a linear boundary. What if data is not linearly separable (e.g., a donut)?



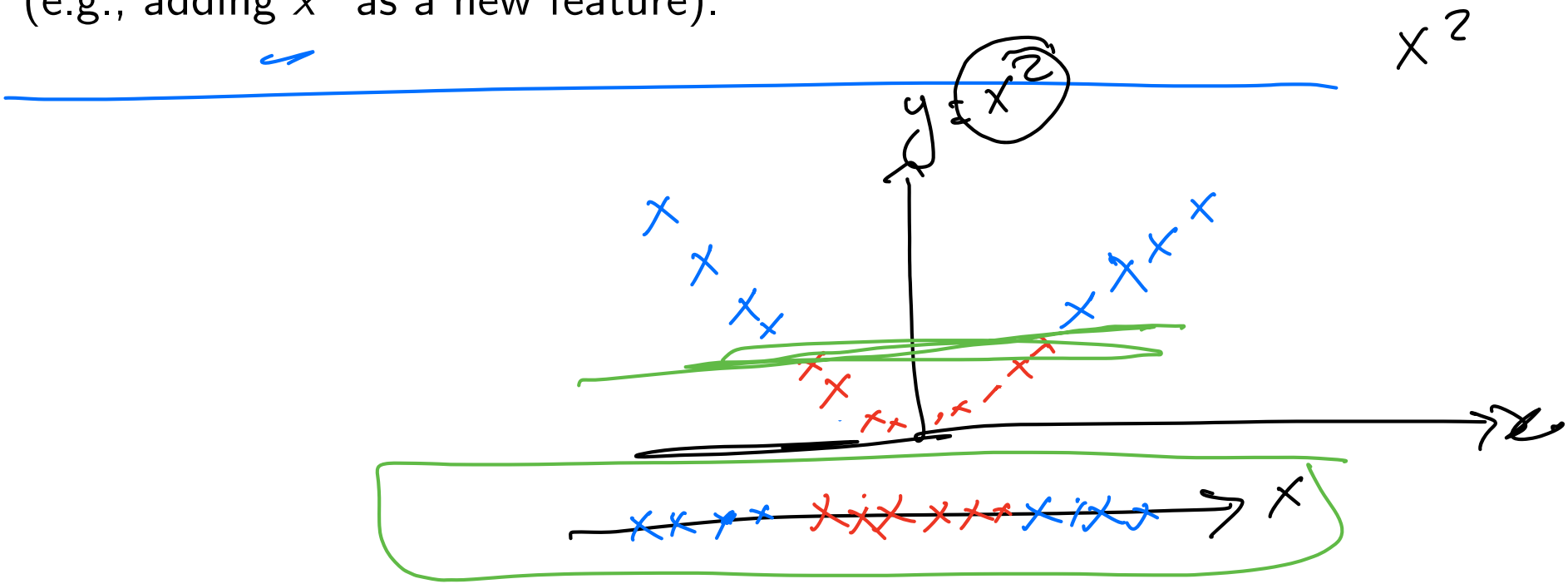
Handling Non-Linearity: The Kernel Trick

Problem: Standard SVM finds a linear boundary. What if data is not linearly separable (e.g., a donut)?

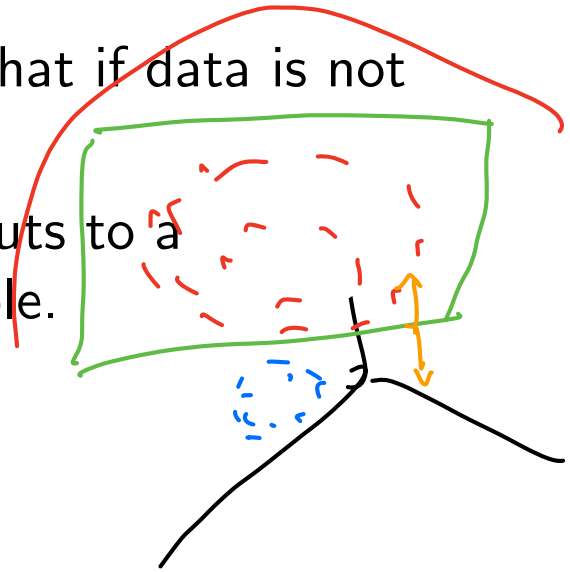
The Solution (Feature Engineering): We map inputs to a higher-dimensional space where they *become* separable.

$$\mathbf{x} \rightarrow \phi(\mathbf{x})$$

(e.g., adding x^2 as a new feature).



Handling Non-Linearity: The Kernel Trick



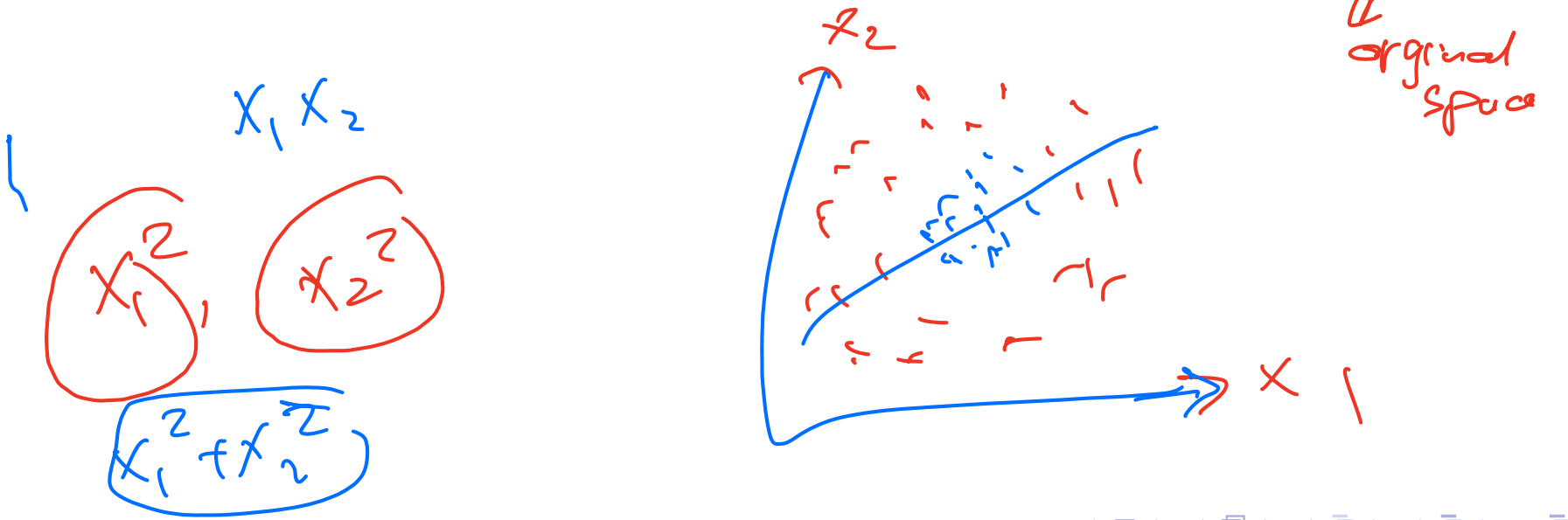
Problem: Standard SVM finds a linear boundary. What if data is not linearly separable (e.g., a donut)?

The Solution (Feature Engineering): We map inputs to a higher-dimensional space where they *become* separable.

$$\mathbf{x} \rightarrow \phi(\mathbf{x})$$

(e.g., adding x^2 as a new feature).

Issue: Computing $\phi(\mathbf{x})$ explicitly is computationally expensive.



Handling Non-Linearity: The Kernel Trick

Problem: Standard SVM finds a linear boundary. What if data is not linearly separable (e.g., a donut)?

The Solution (Feature Engineering): We map inputs to a higher-dimensional space where they *become* separable.

$$\mathbf{x} \rightarrow \phi(\mathbf{x})$$

(e.g., adding x^2 as a new feature).

Issue: Computing $\phi(\mathbf{x})$ explicitly is computationally expensive.

The "Trick" (Implicit Computation): We use special type of functions called **Kernel Functions** which compute the high-dim **similarity directly** without explicitly building them!

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$$

This is called the **Kernel Trick**.

Common SVM Kernels: A Menu of Similarity

The choice of Kernel tells the SVM "how to measure similarity."

- **1. Linear Kernel** (Good for Linear Problems)
  - $K(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\mathbf{y}$
  - **What it finds:** Straight lines/planes.
- **2. Polynomial Kernel**
  - $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T\mathbf{y} + c)^d$
  - **What it finds:** Curved boundaries and feature interactions.
- **3. Radial Basis Function (RBF)** (The Default)
  - $K(\mathbf{x}, \mathbf{y}) = \exp\left(-\gamma||\mathbf{x} - \mathbf{y}||^2\right)$
  - **What it finds:** "Local" similarity. Points are similar only if they are close in distance.

Rule of Thumb: Start with **RBF** if you don't know the structure.

SVM: Strengths and Weaknesses

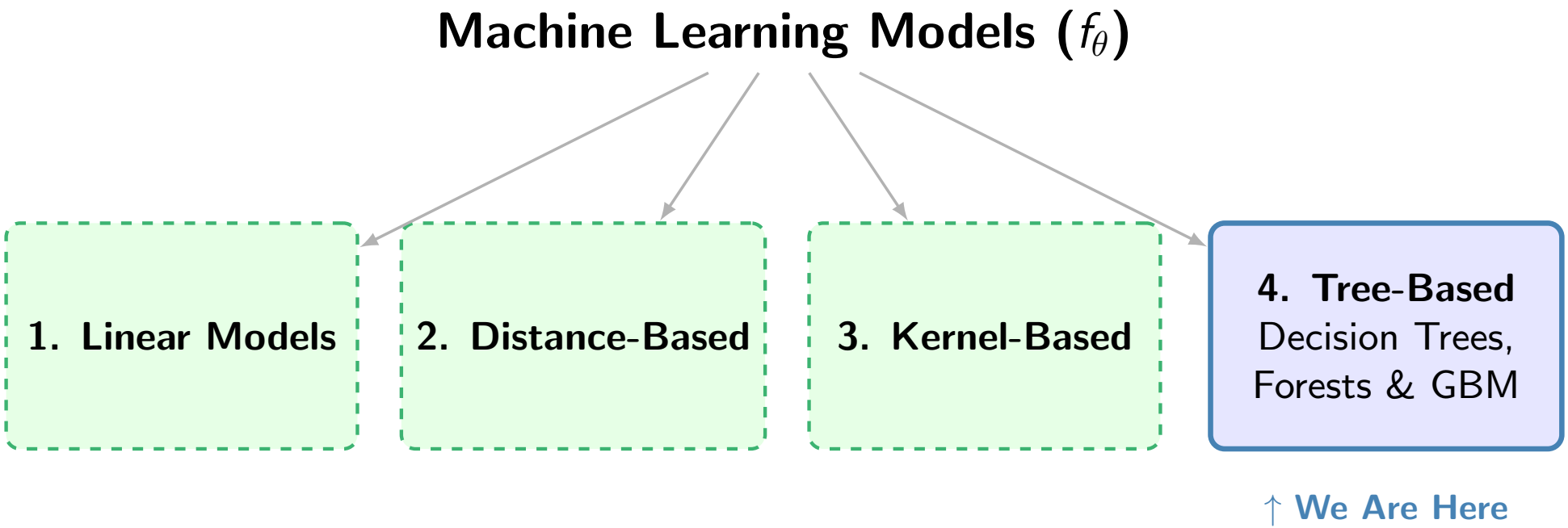
Advantages

- **High Dimensionality:** Works very well when *features* $\gg$ *samples*.
- **Powerful:** Different kernels allow for modeling complex non-linear relationships.

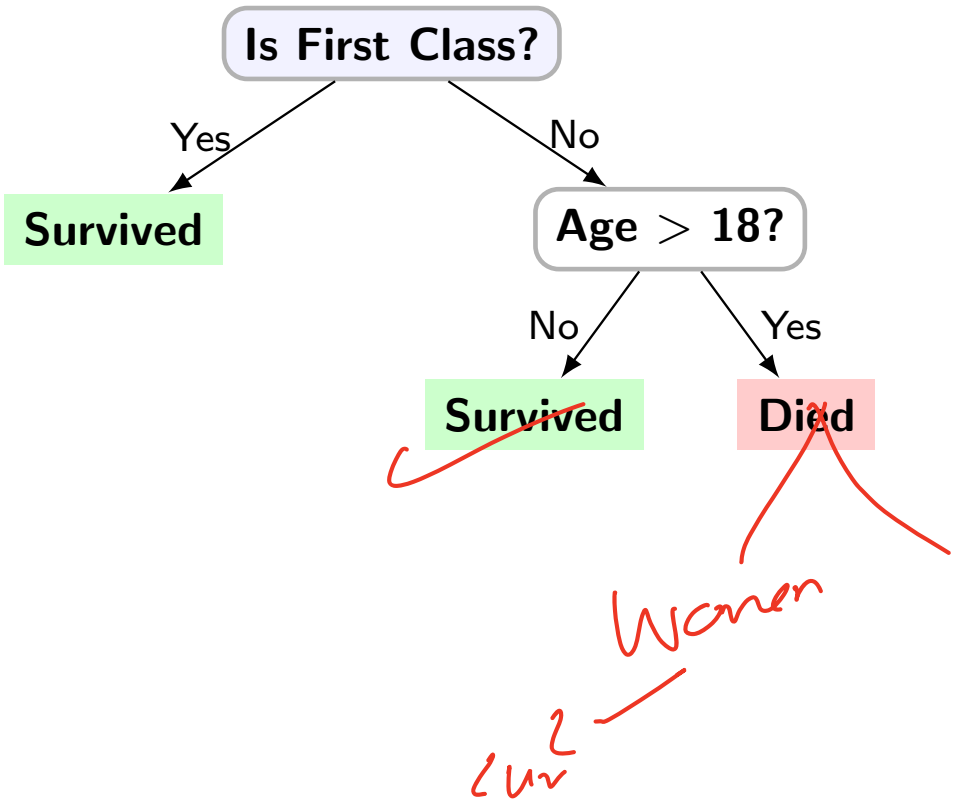
Disadvantages

- **Noise Sensitivity:** Sensitive to noise and outliers (if C is large).
- **Scalability:** Slow for large datasets ($> 10k$ samples).
- **Preprocessing:** Requires feature scaling.

The ML Toolbox: Roadmap Update



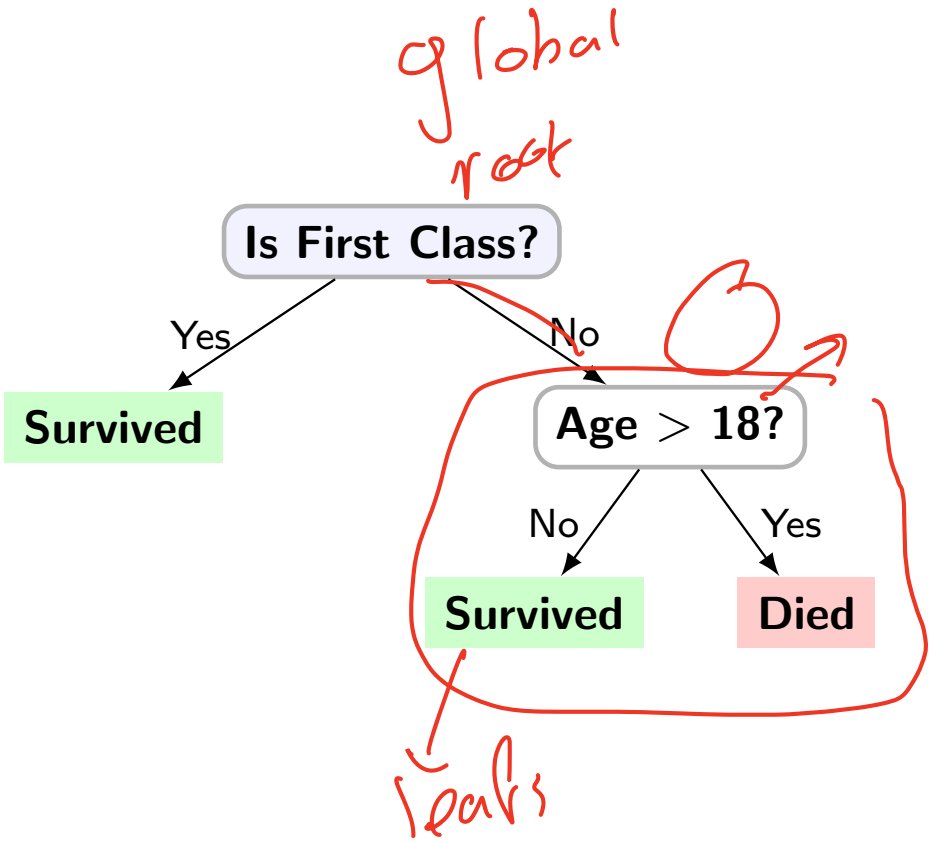
Intuition: We split data by asking a sequence of Yes/No questions to separate classes.



Example: Titanic Survival

- **Goal:** Predict who survived.
- **Root Question:** "Was the passenger in First Class?" (Best separator).
- **Next Question:** If not, "Were they a child?"

Intuition: We split data by asking a sequence of Yes/No questions to separate classes.



Example: Titanic Survival

- **Goal:** Predict who survived.
- **Root Question:** "Was the passenger in First Class?" (Best separator).
- **Next Question:** If not, "Were they a child?"

The Core Question

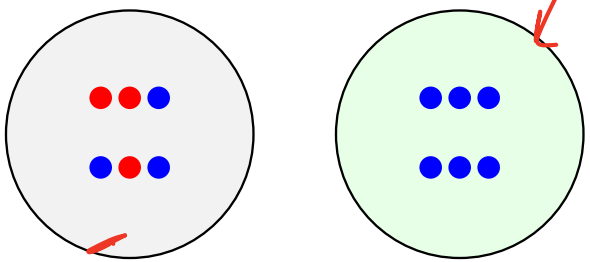
We have many features (Age, Ticket, Gender...).

How do we decide which question to ask first?

How to Choose the Best Split? (Purity)

We want questions that lead to **"Pure" nodes** (containing only one class). We measure "disorder" using **Impurity Metrics**.

Visualizing Impurity



High Impurity (Mixed) | Zero Impurity (Pure)

The Algorithm's Goal: Pick the split that maximizes the drop in Impurity (Information Gain).

The "Impurity Metrics":

1. **Gini Impurity (Default):**

$$G = 1 - \sum p_i^2$$

   - $G = 0.5$: Max chaos (50/50).
   - $G = 0.0$: Pure (One class).

2. **Entropy (Information):**

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

What is p_i?
p_i is the **proportion** of samples belonging to class i in the current node.

Wait... Where is Gradient Descent?

Q: Do Decision Trees use Gradient Descent to find the best split?

Wait... Where is Gradient Descent?

Q: Do Decision Trees use Gradient Descent to find the best split?

The Answer: NO!

Standard Decision Trees are **Non-Differentiable**. You cannot calculate a gradient ∇J because splits are discrete "hard" choices (Left vs. Right).

Wait... Where is Gradient Descent?

Comparison of Learning Algorithms:

Non-Trees Models

Method: Gradient Descent

- Loss surface is smooth (convex).
- We slide down the curve.
- Update weights **w** iteratively.

Decision Trees

Method: Greedy Search

- Try **every possible split** for every feature.
- Pick the one with max Information Gain.
- **Repeat recursively until:**
  - The node is **Pure** (100% one class).
  - Or we hit a **Stop Condition** (e.g., Max Depth).

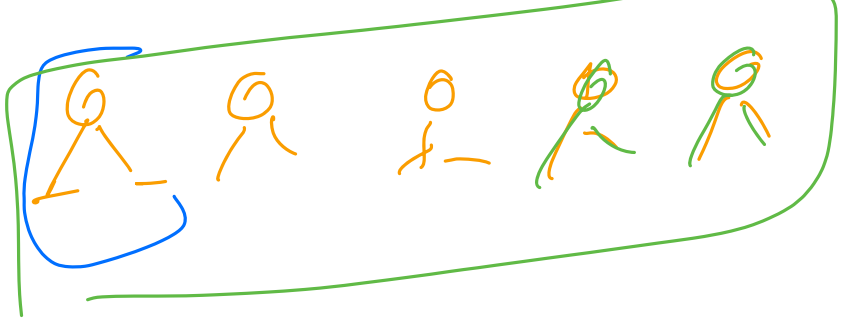
Decision Trees: Advantages and Disadvantages

The Advantages

- **Simple:** Easy to understand.
- **Interpretable:** Easy to explain to non-experts ("If X $>$ 5, then Y").
- **No Preprocessing Required:** No feature scaling or encoding required.

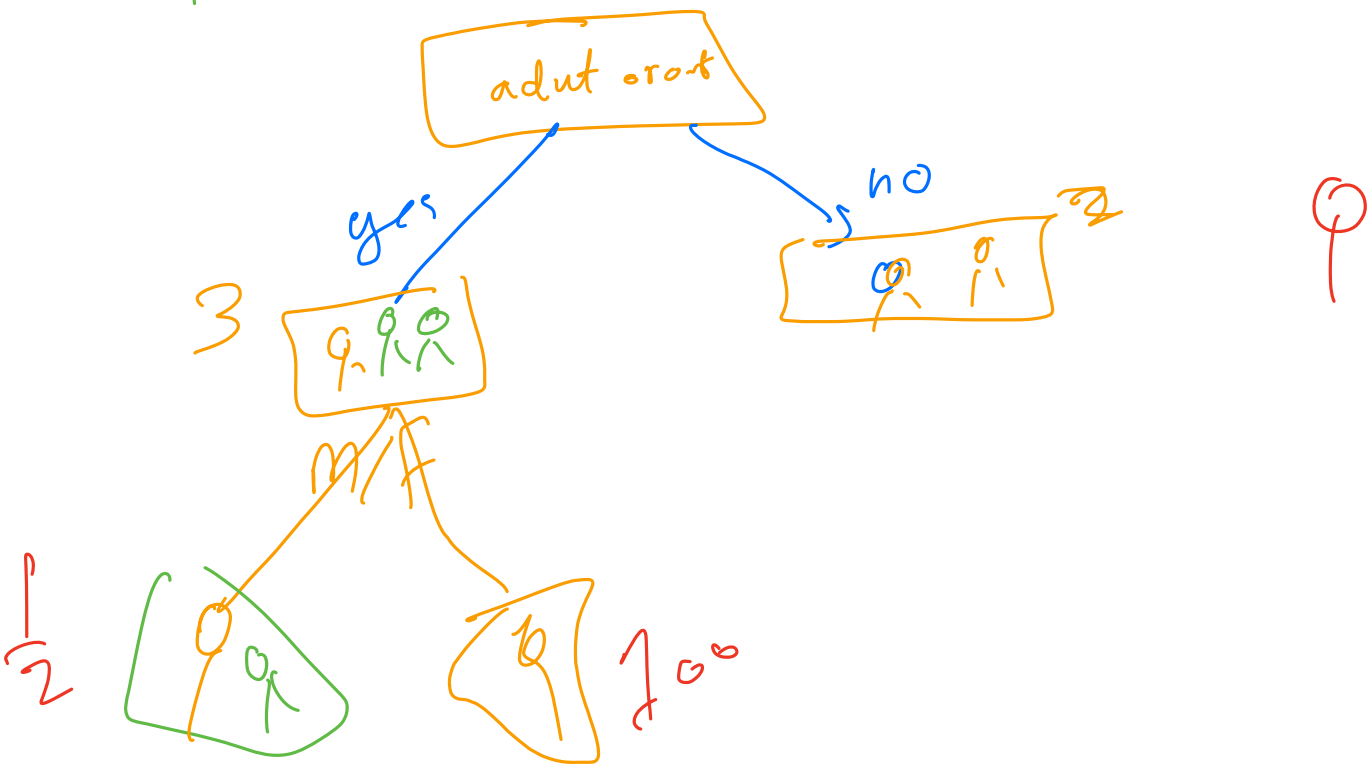
The Disadvantages

- **High Variance:** A small change in data = completely different tree.
- **Overfitting:** Not setting the stop conditions (e.g., Max Depth) guarantees overfitting.



~~Rent~~ Sold

not Sold

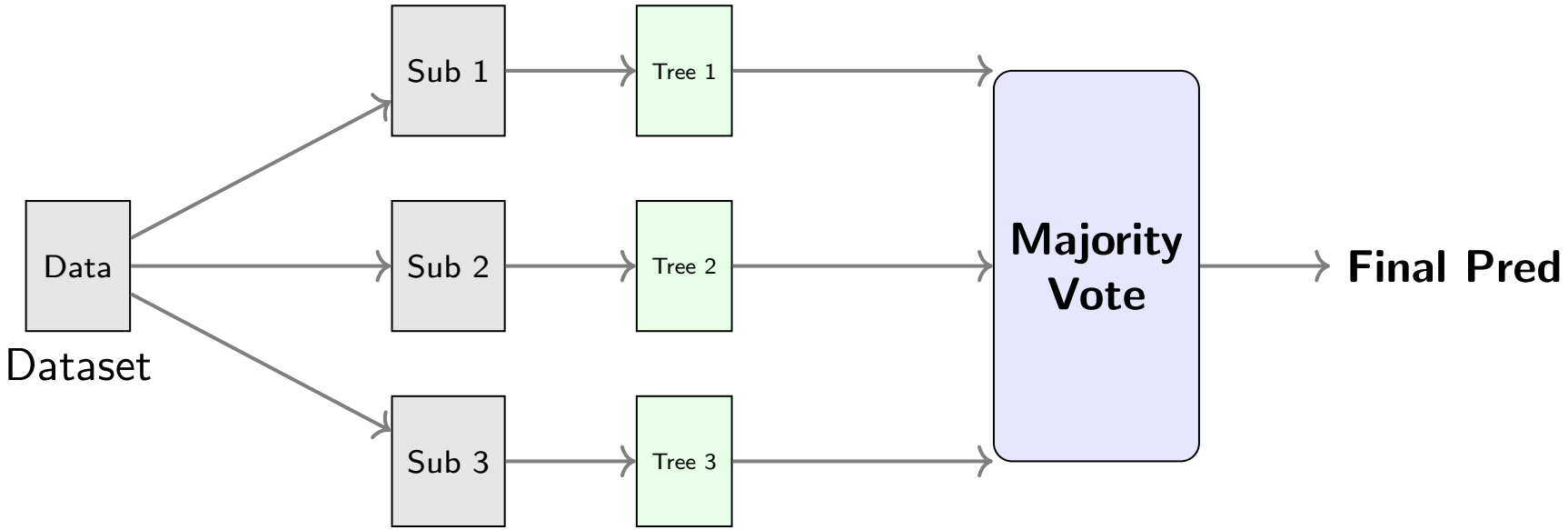


Random Forest

Problem: Single trees are unstable (High Variance).

Problem: Single trees are unstable (High Variance).

Solution: Train many trees on random subsets and average them (**Bagging**).



"Many uncorrelated weak models $\rightarrow$ One strong model."

Random Forest: Advantages and Disadvantages

Advantages

- **Robust:** Very hard to overfit.
- **High Performance:** Very good baseline.
- **Low Hyperparameters Sensitivity:** Works well with default hyperparameters.
- **Interpretability:** Tells you which features matter most (Feature Importance).

Disadvantages

- **Slow Inference:** Must run data through every single tree.
- **Memory Intensive:** Stores all trees in RAM.

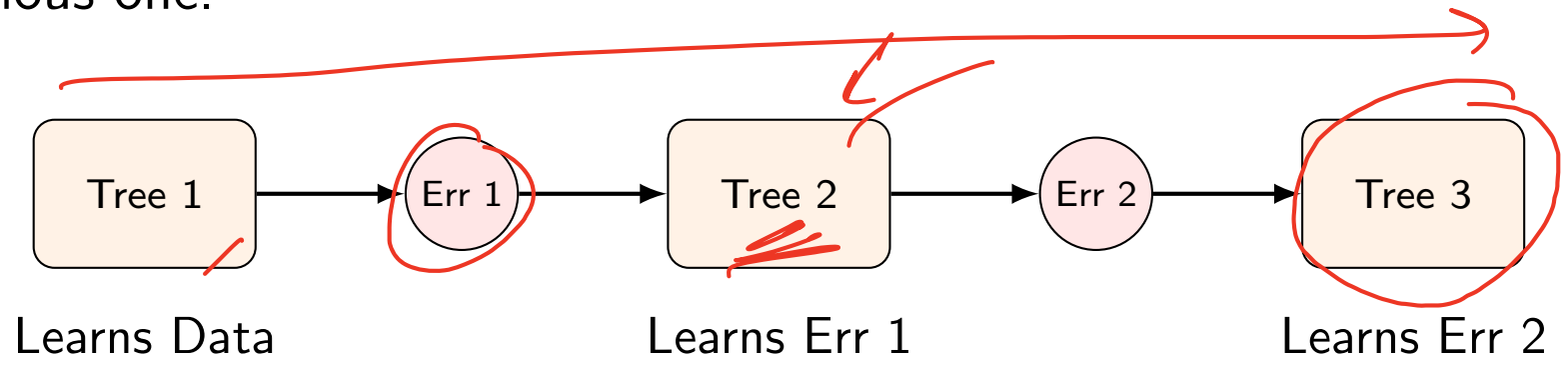
Gradient Boosting: Correcting Mistakes

Problem: Random Forest just averages opinions. It doesn't learn from mistakes.

Gradient Boosting: Correcting Mistakes

Problem: Random Forest just averages opinions. It doesn't learn from mistakes.

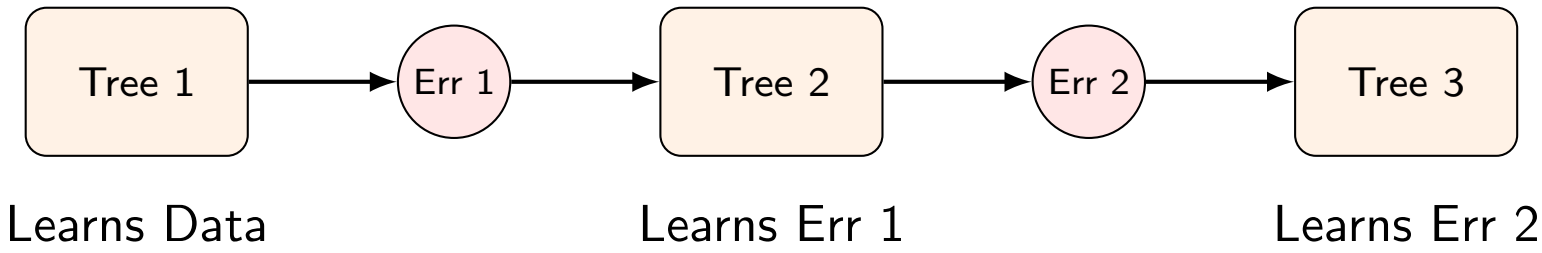
Solution: Train trees **sequentially**. Each tree fixes the errors of the previous one.



Gradient Boosting: Correcting Mistakes

Problem: Random Forest just averages opinions. It doesn't learn from mistakes.

Solution: Train trees **sequentially**. Each tree fixes the errors of the previous one.



Common Implementations:

- **XGBoost:** Best Default
- **LightGBM:** Lightweight
- **CatBoost:** Requires minimal hyperparamters tuning.

Why is it called *Gradient* Boosting?

We are not just "fixing errors." We are performing **Gradient Descent**, but in a different way.

1. Gradient Descent

In Linear Regression, we update **parameters** (w) to minimize Loss.

$$\mathbf{w}_{new} = \mathbf{w}_{old} \underbrace{-\alpha \cdot \nabla Loss}_{\text{Step down the hill}}$$

2. Gradient Boosting

Here, we update the **function** (F) by adding new trees.

$$F_{new}(x) = F_{old}(x) + \underbrace{\alpha \cdot \text{Tree}(x)}_{\text{Step down the hill}}$$

- If we use **Squared Error** Loss: $L = \frac{1}{2}(y - \hat{y})^2$
- The Gradient (derivative) is: $\nabla L = -(y - \hat{y})$
- Therefore: **The Residual (which we fit trees on)** $(y - \hat{y})$ **is effectively the Negative Gradient!**
- Final Model = Base Tree + $\sum$ (Error-Correcting Trees).

Gradient Boosting

Advantages

- **SOTA Performance:** Best accuracy for tabular data.
- **Flexibility:** Customizable Loss functions.
- **No Preprocessing Required:** Handles Missing Values natively. No need for scaling.

Disadvantages

- **Sensitive:** Harder to tune (learning rate, depth).
- **Overfitting:** Easy to overfit if not careful (too many trees).
- **Serial:** Slow training (hard to parallelize).
- **Data Hungry:** Can overfit easily if data is small.

The "No Free Lunch" Theorem

"Is there one Master Algorithm to rule them all?"

The Theorem (Wolpert, 1996)

Averaged over all possible data generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points.

$$\mathbb{E}[Error(\text{Model}_A)] = \mathbb{E}[Error(\text{Model}_B)]$$

In Plain English:

- No model works best for *every* problem.
- A model that is great at Image Recognition might be terrible at Spam Classification.

The Implication for us

Since we cannot mathematically prove which model will work best beforehand:

1. We must make **assumptions** about our data (e.g., "it looks linear").
2. We must **experiment** with multiple models from our toolbox.

References

- Aurélien Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*
- Andrew Ng, *Machine Learning Specialization* (Coursera/DeepLearning.AI)

Slides contributed by Mohamed Eltayeb

Suggested

https://reemali.com/blog/resources/ai/