Fundamentals / Foundation of Data Science

Data Science 9CFU

Computer Science 6CFU

Indro Spinelli

Sapienza University of Rome

Parametric vs. Non-Parametric Learning

Parametric Models

- Assume a **fixed functional form** with a **finite number of parameters**:

$$f_\beta,\ \beta \in \mathbb{R}^d$$

- Learning = estimating a small set of parameters
- **Examples:** Linear regression, logistic regression, neural networks

Non-Parametric Models

- Do **not assume a fixed form** for f
- Model capacity grows with the amount of data
- Can represent **highly flexible or "arbitrary" functions**
- **Examples:** k-Nearest Neighbors, decision trees

k-Nearest Neighbors (kNN)

Goal: Classify a new input x based on proximity to training examples.

1) Find the k nearest neighbors of x in the training set

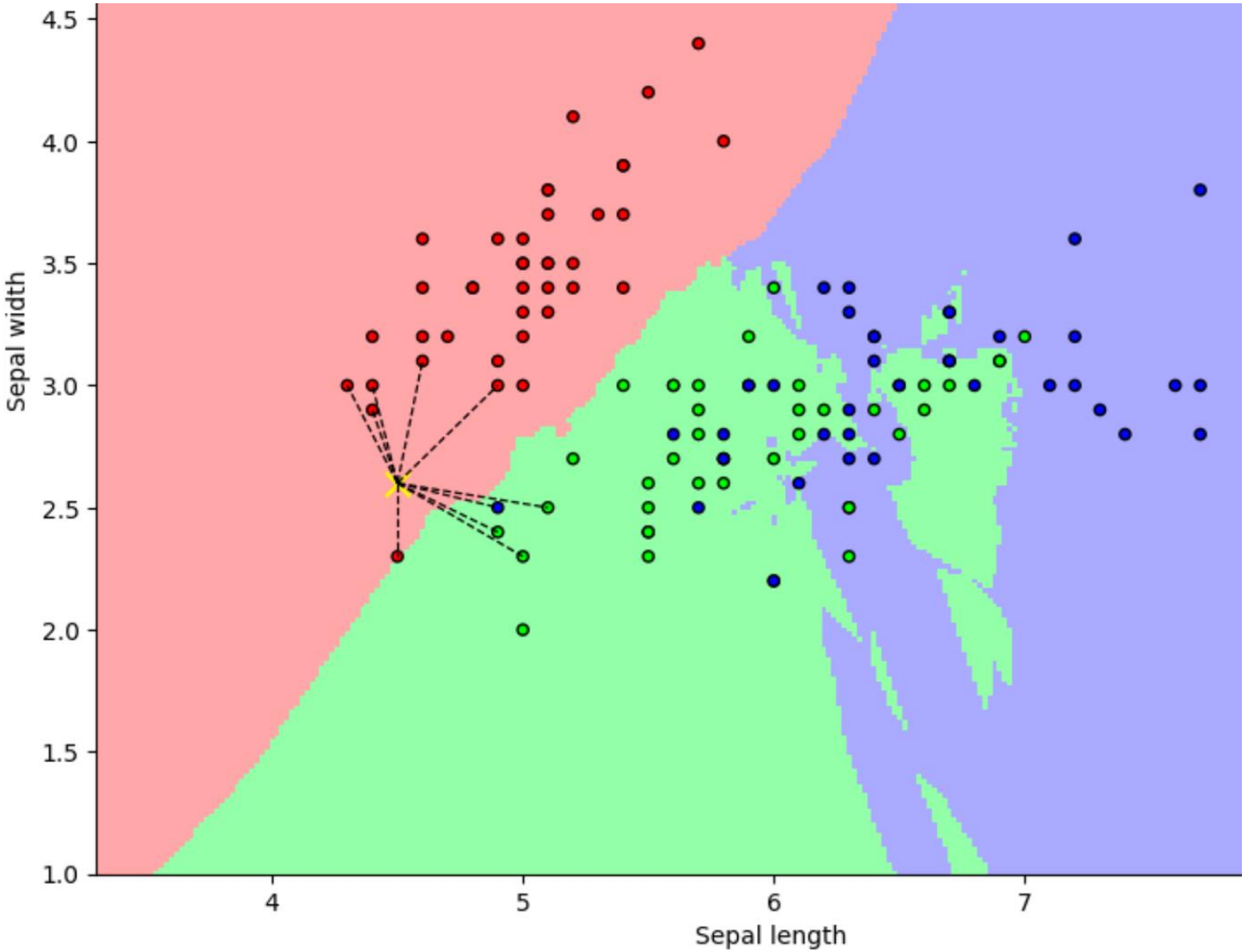
$$i_1, \dots, i_k = k\text{-}ArgMin_{i \in \{1,\dots,n\}} \; dist(x, x_i)$$

where dist(·,·) is typically the Euclidean distance.

2) Predict the label by majority vote among the neighbors:

$$y = Majority\left(y_{i_1}, \dots, y_{i_k}\right)$$

- No training phase, only **store** the data.
- Complexity mainly in the **query time**.
- Sensitive to the **choice of k** and **distance metric**.


4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
Sepal width
4
5
6
7
Sepal length

Loss Minimization Framework

Model Family

- In kNN, the "model" is determined **entirely by the training data** Z.
- The model family = **all functions induced by different training sets** Z.
- The **training examples themselves** act as parameters.
- They grow in complexity as the dataset size n increases.

Loss Function

- kNN does **not explicitly minimize a loss function** during training.
- Learning happens **at prediction time**, not by optimizing parameters.
- Model performance is **evaluated** using standard loss measures:
  - **Classification:** accuracy, error rate
  - **Regression:** mean squared error (MSE)

Advantages of k-Nearest Neighbors (kNN)

Simplicity

- Easy to understand and implement.

Interpretability

- Decisions are based directly on nearby examples, mimicking human reasoning.

Flexibility

- Can adapt over time by adding new examples or adjusting k.

Minimal Assumptions

- No need to assume a specific distribution or functional form for the data.

No Explicit Training Phase

- Most computation happens during prediction.
- Particularly efficient for **dynamic or incrementally growing datasets**.

Disadvantages of k-Nearest Neighbors

Computationally Expensive

- Must compute distances to all training points for each new example.
- Becomes burdensome with large datasets.

Sensitive to Irrelevant or Noisy Features

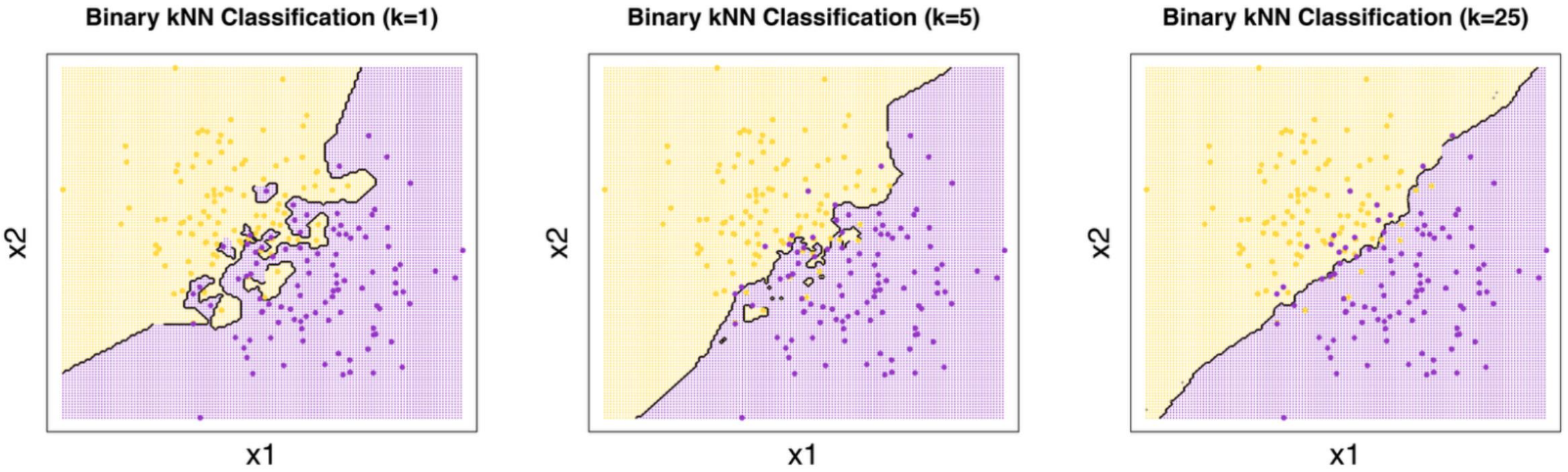
- Uninformative features can distort distance calculations and reduce accuracy.

Biased by Feature Scales or Categories

- Features with larger ranges or many categories can dominate the distance metric.
- Requires careful normalization or weighting.

Choice of k Matters

- Too small → noisy predictions; too large → over-smoothing.


Binary kNN Classification (k=1)
x2
x1
Binary kNN Classification (k=5)
x2
x1
Binary kNN Classification (k=25)
x2
x1

Choosing the Optimal K in KNN

Selecting the hyperparameter K is critical, as it directly controls the model's complexity and its bias-variance trade-off.

Low K (e.g., K=1):

- **Low Bias, High Variance.**
- The model is very flexible and follows the training data closely.
- **Risk:** Prone to **overfitting** and highly sensitive to noise.

High K (e.g., K=N):

- **High Bias, Low Variance.**
- The model is very smooth and ignores local data structure.
- **Risk:** Prone to **underfitting** (will just predict the majority class for the entire dataset).

Goal: Find the "sweet spot" K that balances this trade-off.

Choosing the Optimal K in KNN

The Elbow Method (using a Validation Split)

- This is a visual method for finding a good K.
- **Plot Performance vs. K:** Calculate a performance metric on a separate **validation set** for a range of K values 1 to $\sqrt{n}$ (dataset size).
- **Find the "Elbow":** Choose the K at the "elbow" of the curve a.k.a. the point where the performance improvement begins to plateau

Cross-Validation (The Most Robust Method)

- **Define a K Range:** values 1 to $\sqrt{n}$ (dataset size).
- **Use k-fold Cross-Validation:** Choose the K that provides the **best average performance** across all folds.

Decision Trees

A Practical Non-Parametric Learning Algorithm

- Capable of capturing **complex, nonlinear decision boundaries**.
- Naturally learns **simple, interpretable rules first** before refining them.

Key Idea

- Recursively splits the data into subsets based on **feature values** to reduce uncertainty (e.g., entropy or Gini impurity).
- Each internal node = a decision rule
- Each leaf node = a predicted outcome

Interpretability

- **Highly transparent model:** humans can trace decisions from root to leaf.
- Useful for **explainable AI** and understanding feature importance.

Decision Trees

A **binary tree** composed of **decision nodes** and **leaf nodes**.

Decision Nodes:

- Each internal node applies a **Boolean condition** on the input x.
- Usually tests a **single feature** x_j:
  - **Real-valued feature:** $x_j \geq t$,where $t \in \mathbb{R}$
  - **Categorical feature:** $x_j = t$,where t is one of the categories $\{1, \ldots, k_j\}$

Leaf Nodes:

- Represent the **output** of the model.
- Can output:
  - A **class label** (classification)
  - A **numerical value** (regression)
  - Or a **probability distribution** over classes

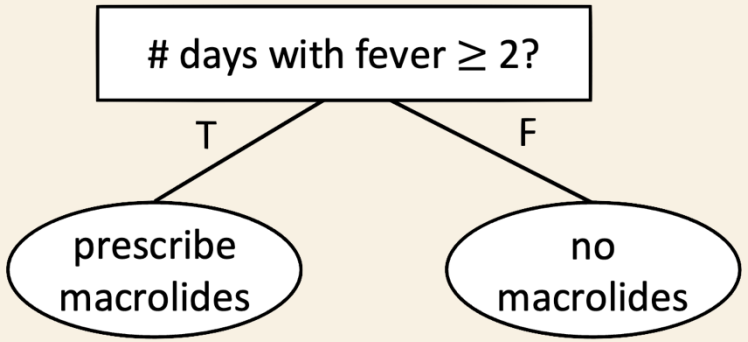
Depth & Decision Boundary

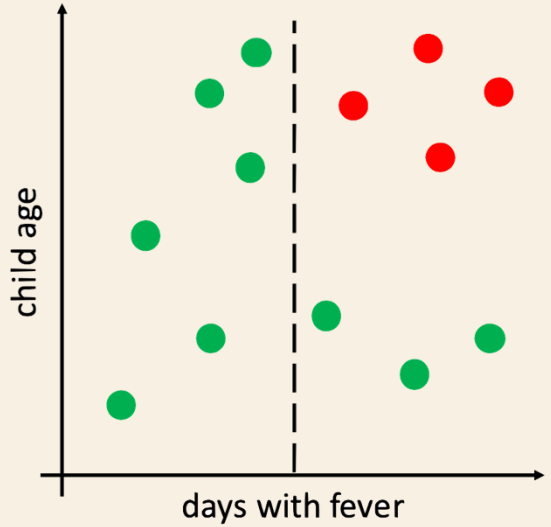
Model Complexity Increases with Tree Depth

Each additional level adds more **splits** → more **decision regions**

Deeper trees can model **complex, nonlinear boundaries**

- **Shallow trees:** high bias, underfit the data
- **Deep trees:** high variance, prone to overfitting





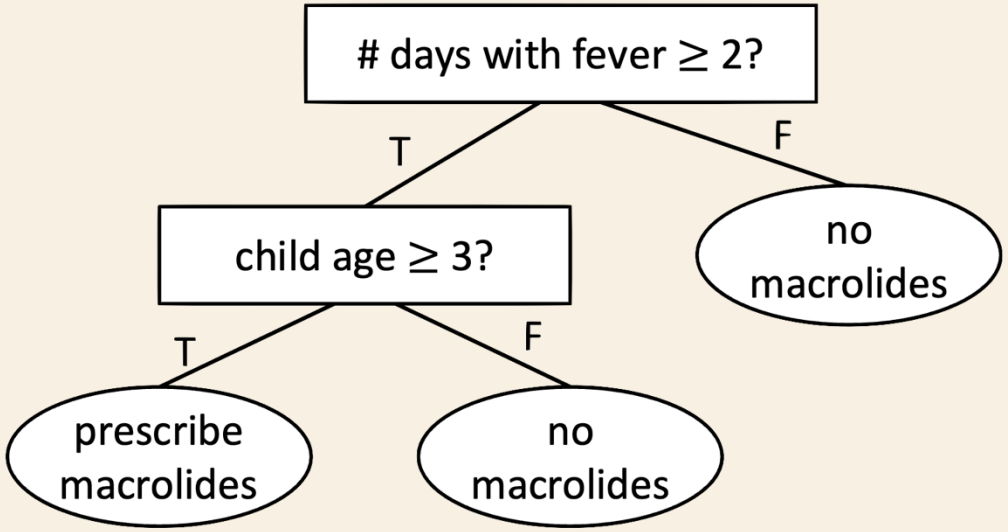
Depth & Decision Boundary

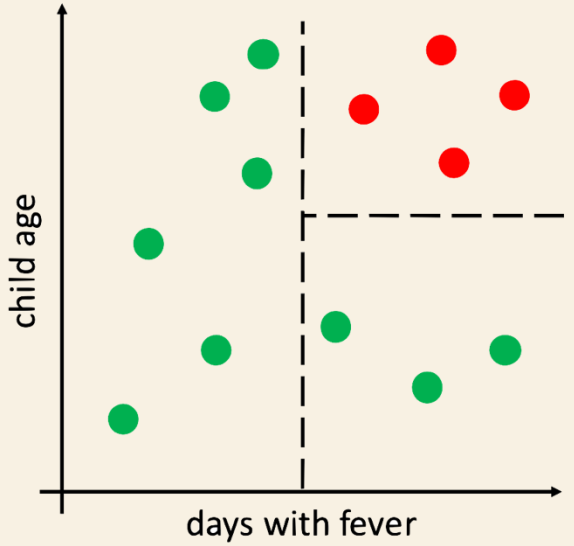
Model Complexity Increases with Tree Depth

Each additional level adds more **splits** → more **decision regions**

Deeper trees can model **complex, nonlinear boundaries**

- **Shallow trees:** high bias, underfit the data
- **Deep trees:** high variance, prone to overfitting





Learning Algorithm for Decision Trees

- **Not a Standard Loss Minimization Approach**
  - Traditional decision tree learning does **not directly minimize a global loss**.
  - Instead, it builds the tree through **greedy, local decisions**.
- **Computational Challenge**
  - Finding the **optimal decision tree** is **NP-complete**.
  - Exact optimization is computationally infeasible for large datasets.
- **Practical Solution**
  - Trees are built **heuristically in a top-down manner** (e.g., ID3, C4.5, CART).
  - At each step, the algorithm selects the **best feature and split** that maximally reduces impurity (e.g., entropy, Gini).

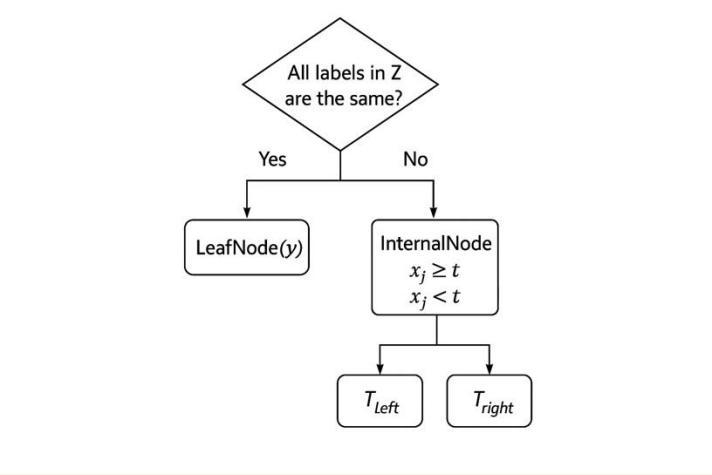
Learning Algorithm for Decision Trees

Let $Z_C = \{(x, y) \in Z \mid C(x, y)\ holds\}$
be the subset of the training data Z satisfying condition C.

Recursive structure: builds the tree top-down by splitting data subsets.

BestSplit(Z): chooses the feature x_j and threshold t that best reduce impurity (e.g., entropy or Gini).

Base case: stops when all examples in a subset share the same label.



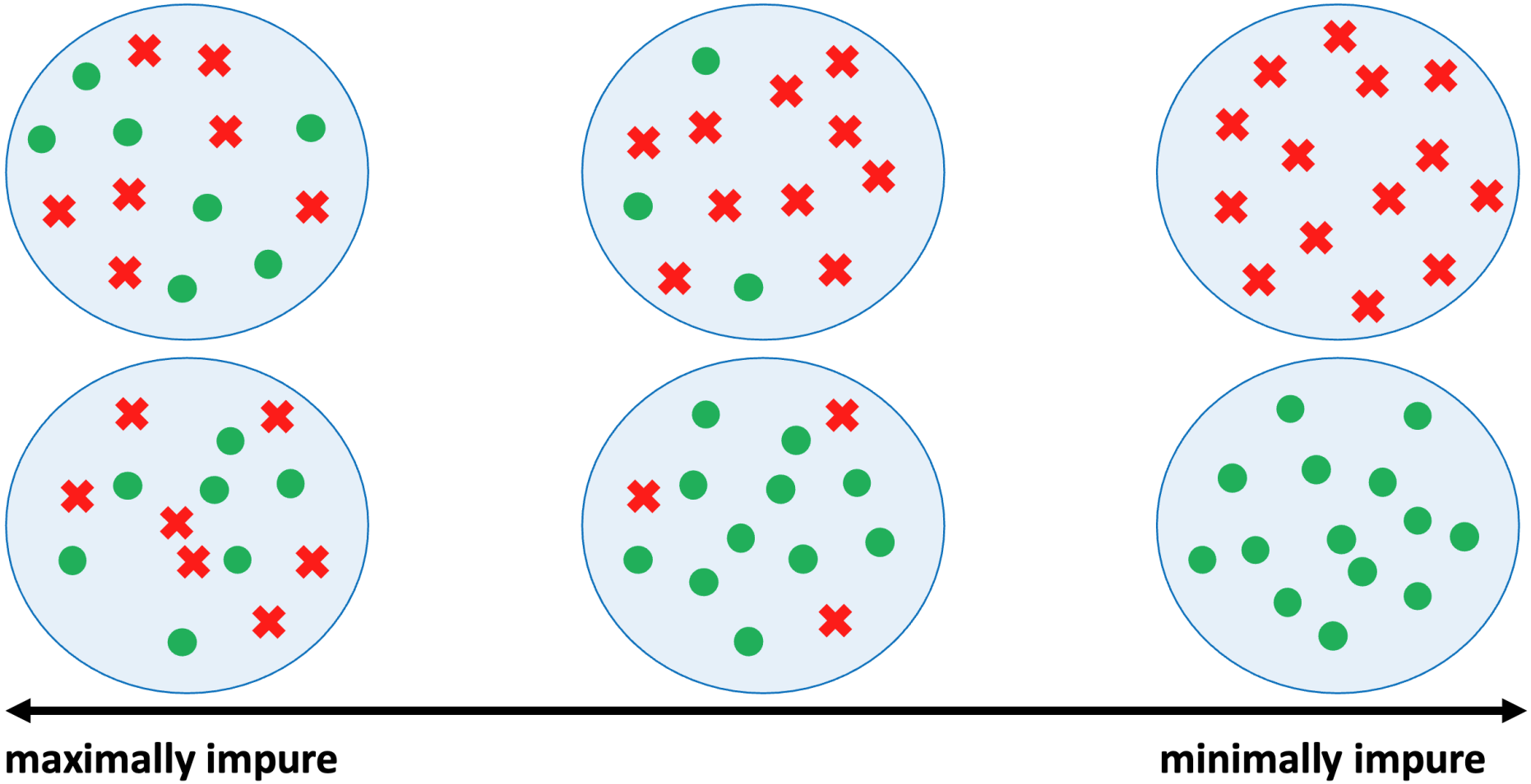

```
def LearnTree(Z):
  # Base case: all examples have the same label
  if all labels in Z are identical (label = y):
    return LeafNode(y)

  # Recursive case: find the best split
  j, t ← BestSplit(Z)
  # Partition data based on the split condition
  Z_left = { (x, y) ∈ Z | x_j ≥ t }
  Z_right = { (x, y) ∈ Z | x_j < t }
  # Recursively build subtrees
  T_left = LearnTree(Z_left)
  T_right = LearnTree(Z_right)

  return InternalNode(j, t, T_left, T_right )
```

Finding the Best Split (Classification)

At any node in the tree, we have a subset of data, Z. Our goal is to find the "best" split (a feature j and a threshold t) that divides Z into two new subsets, Z_{left} and Z_{right} that are as "pure" as possible.

- **"Pure" Node:** A node is perfectly pure if all data points in it belong to the **same class** (e.g., 100% "Yes").
- **"Impure" Node:** A node is maximally impure if it contains a 50/50 mix of two classes.

Our Strategy: Find the split (j, t) that results in the **greatest reduction in impurity** (or "maximizes information gain").


maximally impure
minimally impure

Measuring Impurity

Gini Impurity: Measures the probability of misclassifying a randomly chosen data point if we labeled it according to the node's class distribution. For a set of labels Z with C classes, where p_c is the proportion of class c in the node:

$$I_{Gini}(Z) = 1 - \sum_{c=1}^{C} p_c^2$$

- **Perfect purity:** If all samples belong to a single class c:

  $$p_c = 1.0 \Rightarrow I_{Gini} = 1 - (1)^2 = 0$$

- **Max impurity (binary 50/50):**

  $$p_c = 0.5 \rightarrow I_{\text{Gini}} = 1 - (0.5^2 + 0.5^2) = 0.5$$

Measuring Impurity

Entropy Measures the average "surprise" or uncertainty in the node's labels:

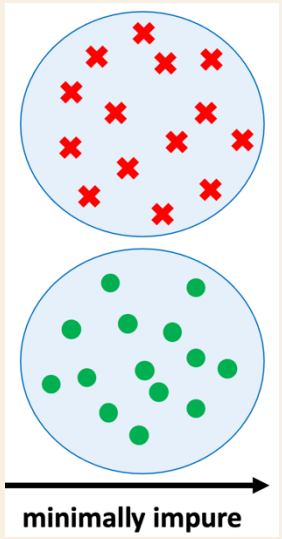
$$I_{Entropy}(Z) = -\sum_{c=1}^{C} p_c \log p_c$$

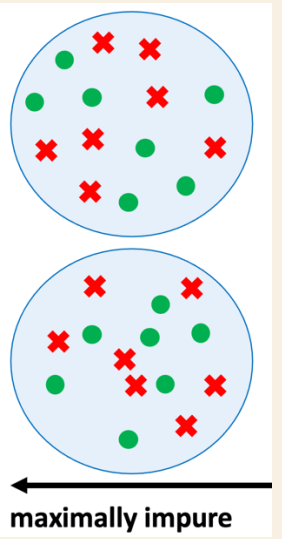
Perfect Purity If all samples belong to a single class c:

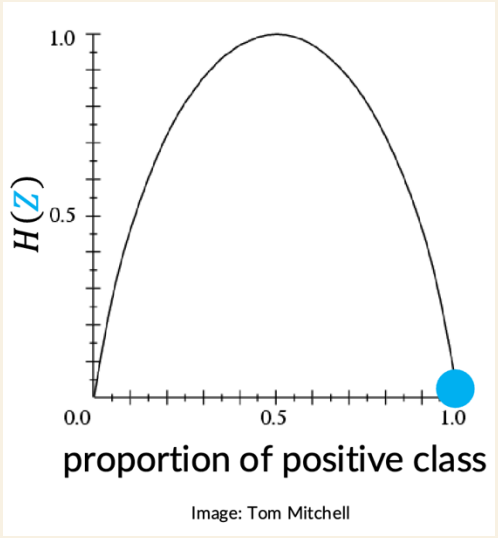
$$p_c = 1.0 \Rightarrow I_{Entropy} = -1 \cdot \log(1) = 0$$

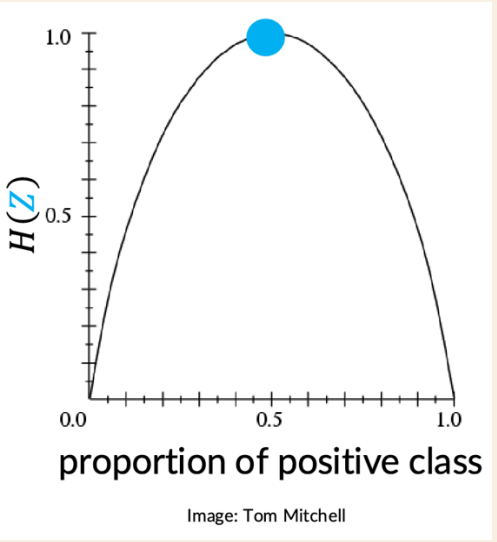
Maximum Impurity (Binary 50/50) if equal mix of two classes:

$$p_1 = p_2 = 0.5 \Rightarrow I_{Entropy} = -[0.5\log(0.5) + 0.5\log(0.5)] = 1$$


minimally impure


maximally impure


1.0
0.5
H(Z)
0.0
0.5
1.0
proportion of positive class
Image: Tom Mitchell


1.0
0.5
H(Z)
0.0
0.5
1.0
proportion of positive class
Image: Tom Mitchell

Measuring Split Quality: Information Gain

We use **Information Gain (IG)**, which is the reduction in impurity achieved by the split. Let:

- Z: parent node with N samples
- Z_L, Z_R :child nodes with N_L, N_R samples

$$\text{IG}(Z, split) = I(Z) - \left(\frac{N_L}{N} I(Z_L) + \frac{N_R}{N} I(Z_R) \right)$$

Best Split:
Choose (j, t) that **maximizes Information Gain**.
Higher IG → child nodes are more homogeneous → better split.

Searching for the Best (j, t)

For each feature $j \in \{1, \ldots, d\}$**:**

Sort unique feature values: $v_1, v_2, \ldots, v_m$

Define candidate thresholds between consecutive values:

$$t \in \left\{\frac{v_1 + v_2}{2}, \ldots, \frac{v_{m-1} + v_m}{2}\right\}$$

For each threshold t

compute the **Information Gain**.

Greedy Strategy

- **Select the Best:** After testing all features j and all their candidate thresholds t, choose the single (j, t) pair that yielded the **maximum IG**.
- This greedy process (finding the *locally* optimal split) is repeated recursively at each new child node until a stopping criterion is met.

Decision Trees for Regression

Classification Trees (Leaf Node): Predict the **majority class** (the **mode**) of the target values $y$in that node.

Regression Trees (Leaf Node): Predict the **average value** (the **mean**) of the target values $y$in that node.

- If a leaf node contains N data points with target values, then the prediction for any new data point that falls into this leaf is:

$$\text{Prediction} = \mu_{node} = \frac{1}{N}\sum_{i=1}^{N} y_i$$

- The resulting model is a **piecewise-constant function** approximating the target function with flat “steps” across regions of the input space.

Split to Reducing Variance (MSE)

In regression trees, our goal is **not** to make nodes “pure” by class, but to make them **homogeneous** in terms of their target values

(i.e., to minimize **variance** within each node).

Impurity Metric: Replace **Gini** or **Entropy** with **Mean Squared.**

For a node Z with N samples and mean target value μ_Z:

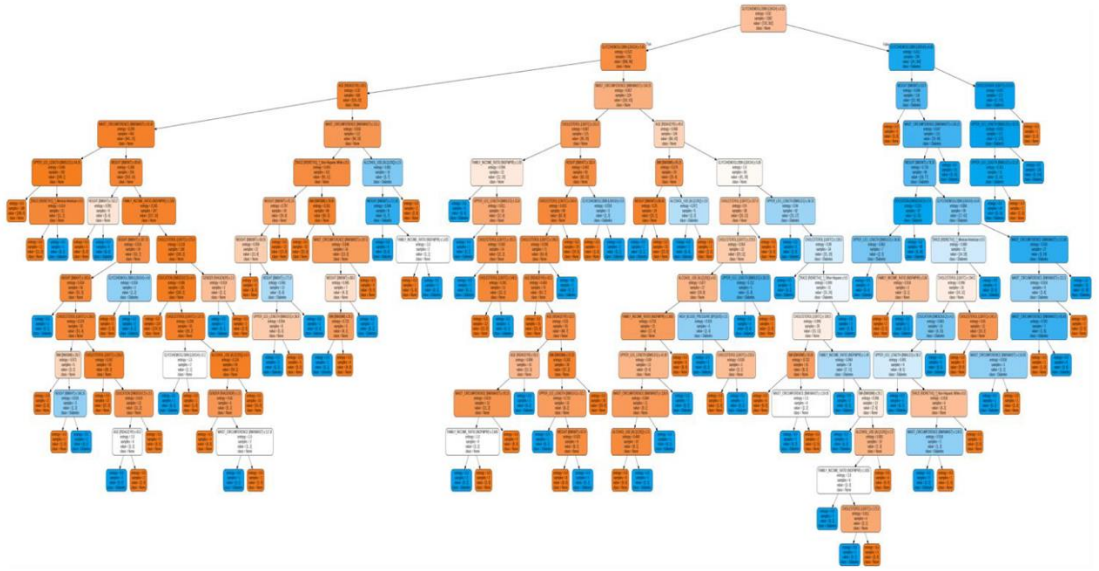
$$\mathrm{MSE}(Z) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mu_Z)^2$$

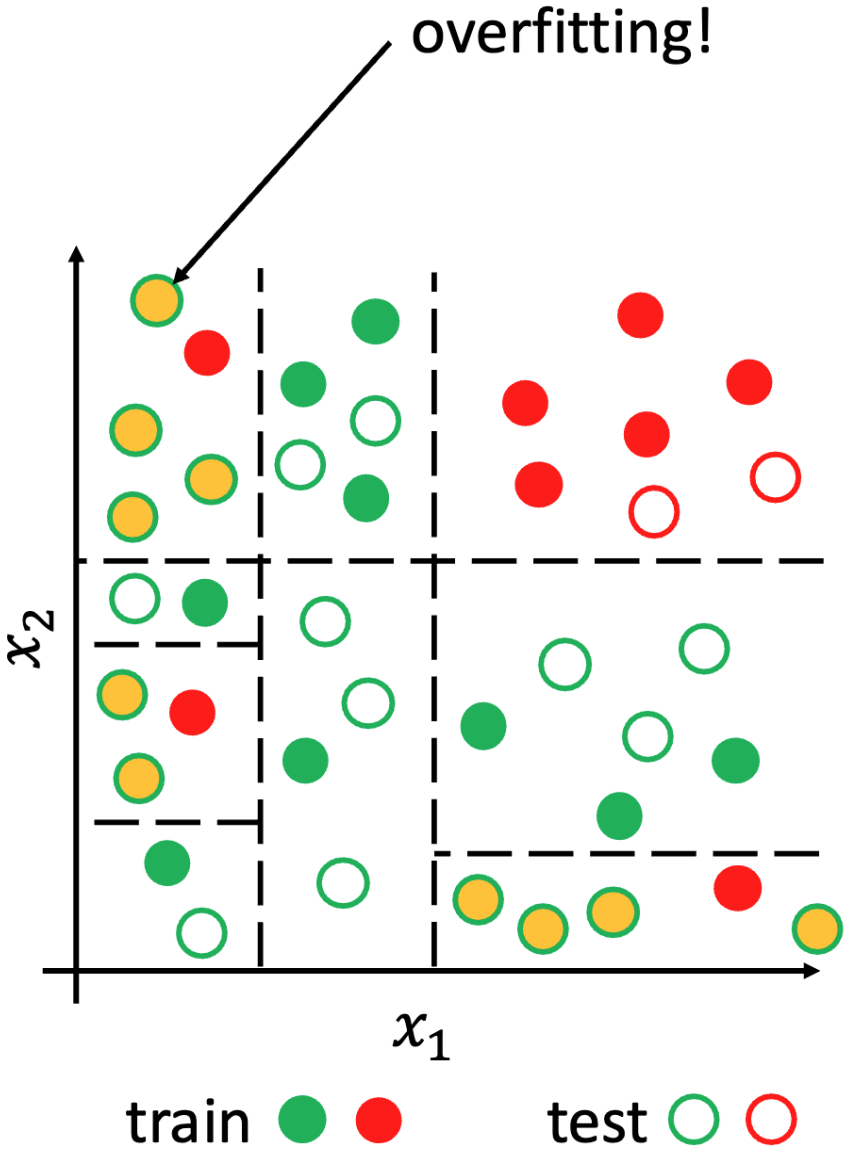
Bias–Variance Tradeoff

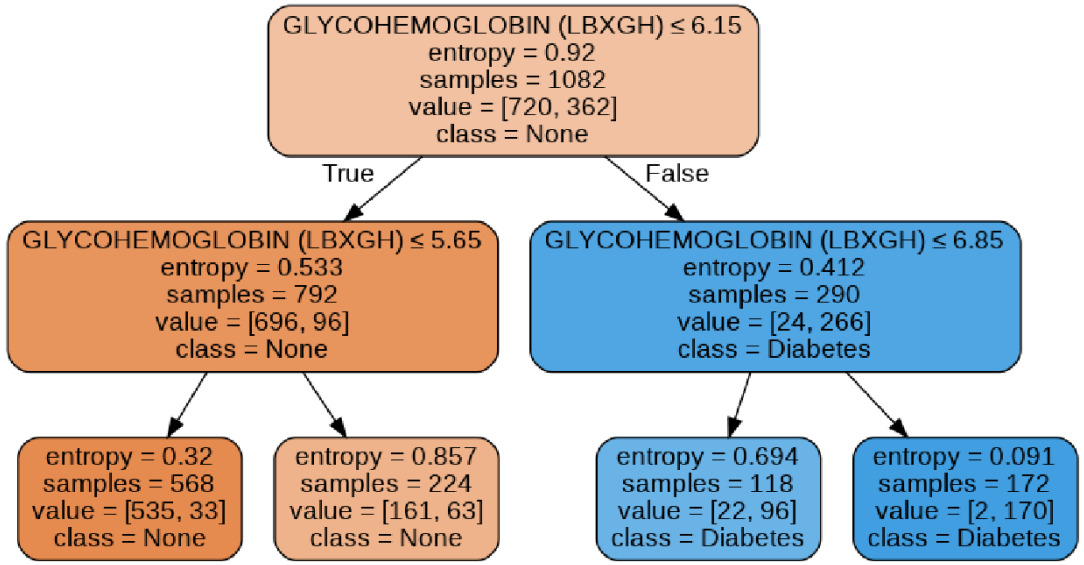
Overfitting occurs when the model starts fitting **noise** rather than the true underlying pattern.

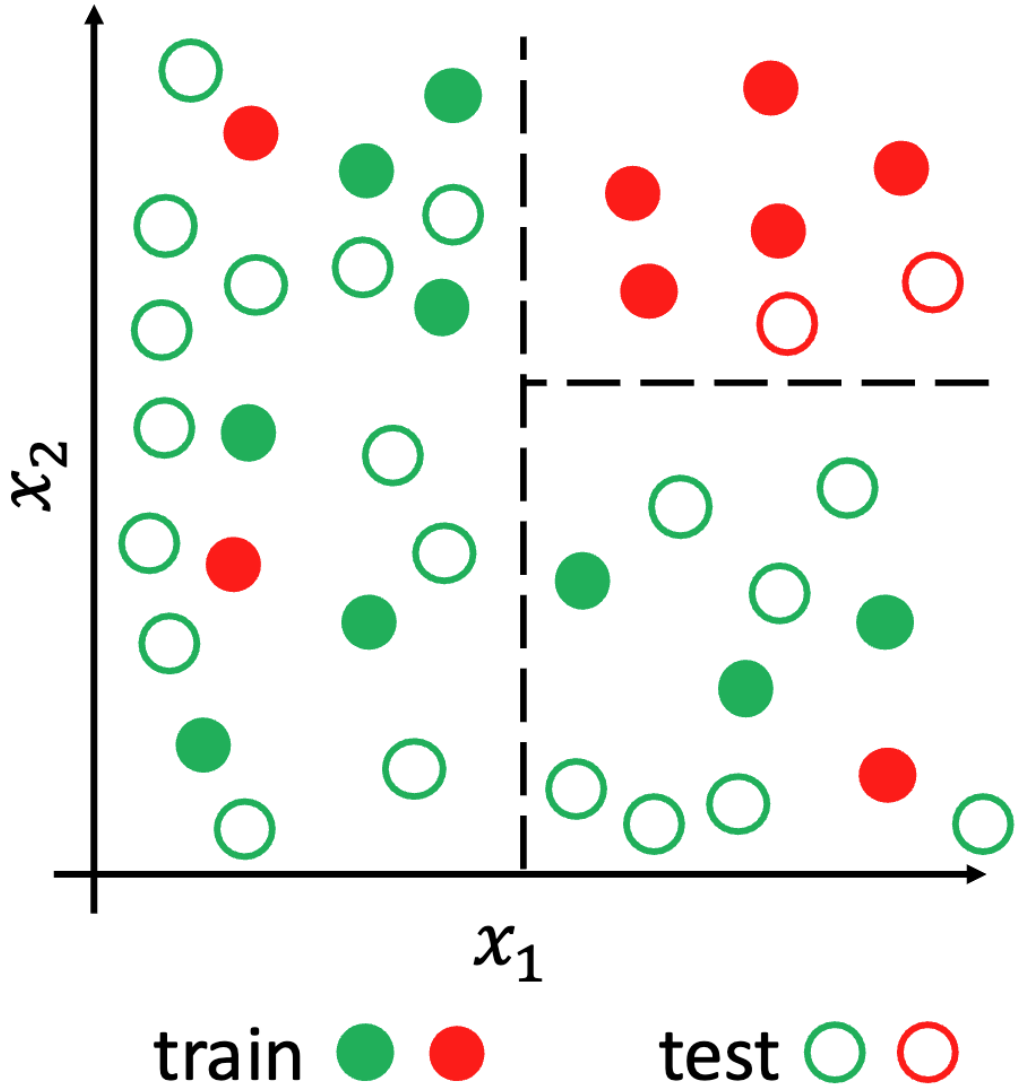
Avoiding Overfitting

- **Gather more data** – improves generalization.
- **Remove irrelevant/noisy features** – simplifies the hypothesis space.
- **Regularization** – introduces a bias toward simpler models.
- **Tree Pruning** – reduce tree depth or remove weak branches.
  - Smaller trees → **fewer parameters**
  - → **lower model capacity**
  - → **better generalization**




overfitting!
x_2
x_1
train
test


GLYCOHEMOGLOBIN (LBXGH) ≤ 6.15
entropy = 0.92
samples = 1082
value = [720, 362]
class = None
True
False
GLYCOHEMOGLOBIN (LBXGH) ≤ 5.65
entropy = 0.533
samples = 792
value = [696, 96]
class = None
GLYCOHEMOGLOBIN (LBXGH) ≤ 6.85
entropy = 0.412
samples = 290
value = [24, 266]
class = Diabetes
entropy = 0.32
samples = 568
value = [535, 33]
class = None
entropy = 0.857
samples = 224
value = [161, 63]
class = None
entropy = 0.694
samples = 118
value = [22, 96]
class = Diabetes
entropy = 0.091
samples = 172
value = [2, 170]
class = Diabetes


x_2
x_1
train
test

Pruning

- **Iterate over internal nodes:** Consider pruning each subtree one at a time.
- **Replace with a leaf node:** Predicts the majority class (or mean value in regression) of its training samples.
- **Evaluate pruning effect:** Measure how pruning affects performance on the validation set.
- **Select the best candidate:** Choose the node whose pruning most reduces validation error.
- **Recursively prune:** Continue until no further improvement is achieved.

def PostPruneTree(T, Z_{train}, Z_{val}):

  # 1. Iterate over internal nodes

  for each internal node N in T :

    # 2. Replace subtree at N with a single leaf node

    $T_N \leftarrow$ Replace(T, N, LeafNode(mode(Z_{train} [N])))

    # 3. Compute the change in validation loss

    $g_N \leftarrow$ Loss(T, Z_{val}) – Loss(T_N, Z_{val})

  # 4. Select the one that reduces validation error

  $N_0 \leftarrow argmax_N g_N$

  if $g_{N^0} > 0$:

    # 5a. Accept pruning and continue recursively

    return PostPruneTree(T_{N^0}, Z_{train}, Z_{val})

  else:

    # 5b. Stop pruning

    return T

Decision Trees: A Summary of Pros and Cons

While standard algorithms like **CART** are popular, fast, and implemented in tools like scikit-learn, it's crucial to understand their fundamental trade-offs.

Strengths

- **Interpretability:** This is their biggest advantage. A trained tree can be visualized and explained as a simple set of "if-then" rules.
- **Flexibility:** They naturally handle both classification (Gini/Entropy) and regression (MSE) tasks.
- **Learns Non-linear Boundaries:** They create complex, axis-aligned, non-linear decision boundaries.

Decision Trees: A Summary of Pros and Cons

While standard algorithms like **CART** are popular, fast, and implemented in tools like scikit-learn, it's crucial to understand their fundamental trade-offs.

Weaknesses

- **High Variance:** This is their biggest drawback. A small change in the training data can result in a completely different tree structure.
- **Tendency to Overfit:** Because they can keep splitting until every leaf is pure, they are highly prone to "memorizing" the training data, which leads to poor performance on new, unseen data.
- **Greedy (Suboptimal) Strategy:** The tree is built using a that finds the "locally best" split at each step.

Build *Many* Trees (Ensemble Methods)

Instead of building one "perfect" tree, build *many different*, imperfect trees (e.g., by training on different subsets of the data).

Aggregate their predictions to get a final, stable, and highly accurate prediction.

- **Pros**
  - Dramatically reduces variance and overfitting.
  - Highly scalable and parallelizable.
  - Often achieves state-of-the-art performance (**Random Forests**, **Gradient Boosted Trees**).
- **Cons**
  - **Completely Loses Interpretability:** It is impossible for a human to understand a model made of 1,000 different trees.

Ensemble Learning

The core idea: **Combining multiple models (even "weak" ones) often creates a single, stronger, and more robust model.**

Instead of relying on a single expert, you ask a committee of experts and combine their answers.

- **Step 1:** Train a set of *base models*
  $f_1, f_2, \dots, f_k$

- **Step 2:** Combine their predictions to form the *ensemble model*
  $F(x) = Combine(f_1(x), f_2(x) \dots f_k(x))$

Train Diverse "Base" Models

We first create a set of individual models, $f_1, f_2, \ldots, f_k$. The key is to make them **diverse**. We want them to make different kinds of errors (independent).

How? Common techniques include:

- **Bagging (e.g., Random Forest):** Train models on different random subsets of the data.
- **Boosting (e.g., Gradient Boosting):** Train models sequentially, where each new model focuses on correcting the mistakes of the previous ones.

Combine Their Predictions

Next, we build a final model, $F(x)$, that aggregates the predictions from all the base models.

Common Combination Methods:

- **Voting (for Classification):** The final prediction is the class receiving the majority of *votes* from the base models.
- **Averaging (for Regression):** The final prediction is the *mean* of all base model outputs.
- **Weighted or Meta-Combination:** Weight each model's prediction by its *validation performance*, or train a **meta-model** to learn the optimal combination or model selection strategy.

Voting (for Classification)

This method, also known as **plurality voting**, outputs the class that receives the most "votes" from the base models.

Formal Formula: Let $C = \{c_1, \dots, c_M\}$ be the set of M possible classes. The final prediction $F(x)$ is the class c_j that maximizes the sum of "votes".

$$F(x) = \arg\max_{c_j \in C} \left(\sum_{i=1}^{k} I\left(f_i(x) = c_j\right) \right)$$

It leverages the "wisdom of the crowd." A single model might make a mistake, but it's much less likely that the *majority* of diverse models will make the *same* mistake.

Averaging (for Regression)

The output is the simple average of all base model predictions. This is the foundation of **Bagging** methods like Random Forest.

$$F(x) = \frac{1}{k}\sum_{i=1}^{k} f_i(x)$$

- **Primary Benefit: Reduces Variance**. It smooths out the "noise" and overfitting of individual models.
- **Best When:** The base models are **unbiased but have high variance** (i.e., they are complex and tend to overfit).

Simple Weighted Combining

We assign a fixed weight α_i to each model, usually based on its overall performance (e.g., its validation accuracy).

- **Regression:** $F(x) = \sum_{i=1}^{k} \alpha_i f_i(x)$
- **Classification:** The "vote" of model f_i counts as α_i.

Limitation: This is **static**. The weights are the same for every single prediction, even if some models are only good at specific *types* of data.

Stacking (or Stacked Generalization)

1. **Train Base Models ($f_1, \ldots, f_k$):** Train k models on the training data.
2. **Create "Meta-Features":** Get the predictions of the base models on the training/validation data. These predictions are the *new features*.
3. **Train Meta-Model ($F(x)$):** Train a final "meta-model" that takes the base models' predictions as input and outputs the final answer.

The meta-model **learns the optimal weights** (and their interactions) from the *predictions* of the base models.

"if model A and B agree, trust them, but if they disagree, trust model C."

Mixture of Experts (MoE)

We train a "gating network" that **dynamically** decides which "expert" model to trust for each specific input.

1. **Experts ($f_1, \ldots, f_k$):** A set of base models, each trained to become a specialist on a *part* of the data.
2. **Gating Network ($g(x)$):** A separate model that looks at the input x and outputs a set of weights ($\alpha_1, \ldots, \alpha_k$) specific to *that input*.
3. **Final Prediction:** The output is a weighted sum, but the weights are **dynamic**.

$$F(x) = \sum_{i=1}^{k} \alpha_i(x) \cdot f_i(x)$$

The Gating Network learns **which expert to trust** based on the input x.

Independent Mistakes

The goal of Bagging is to **reduce variance** (i.e., prevent overfitting) by averaging many "noisy" but (mostly) unbiased models.

Step 1: Bootstrap the Data

- Create k new “bootstrap” datasets.
- Each new dataset is created by sampling n times **with replacement** from the original n training examples.
- Some training examples are **duplicated**, while others are **left out**, the latter are called **Out-of-Bag (OOB)** samples. On average, each bootstrap sample **omits about** $1 - \frac{1}{e} \approx 36.8\%$ of the original data.

Bagging

Step 2: Train Base Models

- Train a separate model f_i on each bootstrap sample.
- Each model learns slightly different patterns due to data variation.

Step 3: Aggregate Combine predictions using **Voting** or **Averaging**.

Step 4: Evaluate

- For each training point x_j find all models f_i that **did not** see x_j during training.
- Use **those models** to generate an **OOB prediction** for x_j and compare to the **true label** y_j.
- The **average error** across all OOB samples gives the is a reliable estimate of the ensemble's generalization performance.

Random Forest

A **Random Forest** is an *ensemble* of many **deep decision trees** trained on different bootstrap samples.

Their predictions are **aggregated (averaged or voted)** to produce a more robust final output.

It builds upon **Bagging (Bootstrap Aggregating)** but introduces **random feature selection** at each tree split which reduces correlation among trees and improves generalization.

Random Features

At **each split** in every tree:

- The algorithm does **not** consider *all* features.
- Instead, it samples a **random subset** of features
  - $\sqrt{d}$ for classification or $\frac{d}{3}$ for regression

This give “weaker” features a chance to influence splits leading to **more diverse** and **less correlated trees with** lower variance.

If a few features are **very predictive**, bagging alone would cause that tree to **split on the same feature** first, making trees **highly correlated.**

Boosting

Boosting is an *ensemble learning* technique that builds a **strong predictor** by **sequentially combining many weak learners**.

- **Train a weak learner** (e.g., a shallow decision tree) on the entire dataset.
- **Evaluate its errors** identify the examples it predicts poorly.
- **Focus the next learner** on these harder cases.
- **Repeat** this process, sequentially refining the ensemble.
- **Combine all learners** into a single **strong model**.

Low Bias Low Variance

Each weak model corrects the predecessor’s errors together, they achieve low bias *and* low variance.

The different boosting algorithms (like Gradient Boosting, XGBoost, and AdaBoost) are just different answers to the question:

"How exactly does the next model 'focus on' the previous model's mistakes?"

AdaBoost

1. **Initialize Weights** Start by assigning **equal importance** (weights) to all training examples.

2. **Train the First Weak Learner** Fit a **simple model** (a tree with one split).

3. **Update Weights**
   - ▲ **Increase** weights of **misclassified** samples (harder cases).
   - ▼ **Decrease** weights of **correctly classified** samples.

4. **Train the Next Learner:** Fit a new weak learner using the **updated weights**, emphasizing the harder data points.
5. **Repeat and Combine:** Iteratively repeat steps 3–4.
   Finally, **combine all weak learners** using a **weighted vote**,
   where stronger models have **greater influence** on the final prediction.

AdaBoost

Given training data $Z = \left\{(x_i, y_i)\right\}_{i=1}^{n}$,where $y_i \in \{-1, +1\}$:

Set initial weights: $w_{1,i} = \frac{1}{n}$ for all $i = 1, \ldots, n$

For each round $t = 1, \ldots, T$**:**

- Train $f_t(\mathrm{w}_t x)$ using the weighted data distribution.
- Compute Weighted Error $\varepsilon_t = \sum_i w_{t,i}\ \mathbf{1}[f_t(x_i) \neq y_i]$
- Compute Model Weight $\beta_t = \frac{1}{2}\ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$
- Update Sample Weights $w_{t+1,i} \propto w_{t,i} \cdot e^{-\beta_t\, y_i\, f_t(x_i)}$
- Normalize so that $\sum_i w_{t+1,i} = 1$.

AdaBoost

Final Strong Classifier

$$F(x) = \text{sign}\left(\sum_{t=1}^{T} \beta_t \ f_t(x)\right)$$

Misclassified examples get **higher weights** in the next round.

Models with **lower training error** get **higher influence** in the final prediction.

AdaBoost combines weak learners into a **high-accuracy ensemble**.

Gradient Boosting

Instead of re-weighting data points like AdaBoost, it sequentially builds new models that predict the *residuals* of the previous models.

Gradient Descent: $\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(\theta_t; Z)$

- *Update parameters in the direction of the negative gradient to minimize loss.*

Boosting: $F_{t+1}(x) = F_t(x) + \beta_{t+1} f_{t+1}(x)$

- *Update the model by adding a new weak learner that moves predictions in the direction of the negative gradient.*

Gradient Boosting

Train $F_0(x)$ and calculate the residuals $r_1 = y - F_0(x)$

Train a new weak learner $h_1(x)$ to predict residuals r_1 (not y).

Add the new weak learner's prediction (scaled by a learning rate ν) to the current ensemble prediction : $F_1(x) = F_0(x) + \nu \cdot h_1(x)$

Repeat for a fixed number of iterations:

- Calculate new residuals $r_2 = y - F_1(x)$.
- Train a new weak learner $h_2(x)$ to predict r_2.
- Update the ensemble $F_2(x) = F_1(x) + \nu \cdot h_2(x)$.

XGBoost

XGBoost (eXtreme Gradient Boosting) is optimized and regularized implementation of Gradient Boosting.

- **Regularization:** Adds **L1** and **L2** penalties to prevent overfitting.
- **Efficient Tree Construction:** Uses approximate split finding and bottom-up pruning.
- **Smart Handling of Missing Data**: Learns the best default split direction (left/right) during training.
- **Column Subsampling:** Randomly samples features per split or tree, reducing overfitting and improving efficiency.

Tom Mitchell – Machine learning chapters 2,3

Machine learning refined chapter 14

The elements of statistical learning chapters 9, 10