Machine Learning Course Summary

Master in Data Science and Advanced Analytics

BA and DS | *NOVA IMS*

Instructor: Roberto Henriques

Lecture 1: The Big Picture

How Everything Connects Together

This overview shows how all the concepts in this course fit together. Use these diagrams as your mental map while studying.

1. The Machine Learning Pipeline

Every ML project follows this workflow:

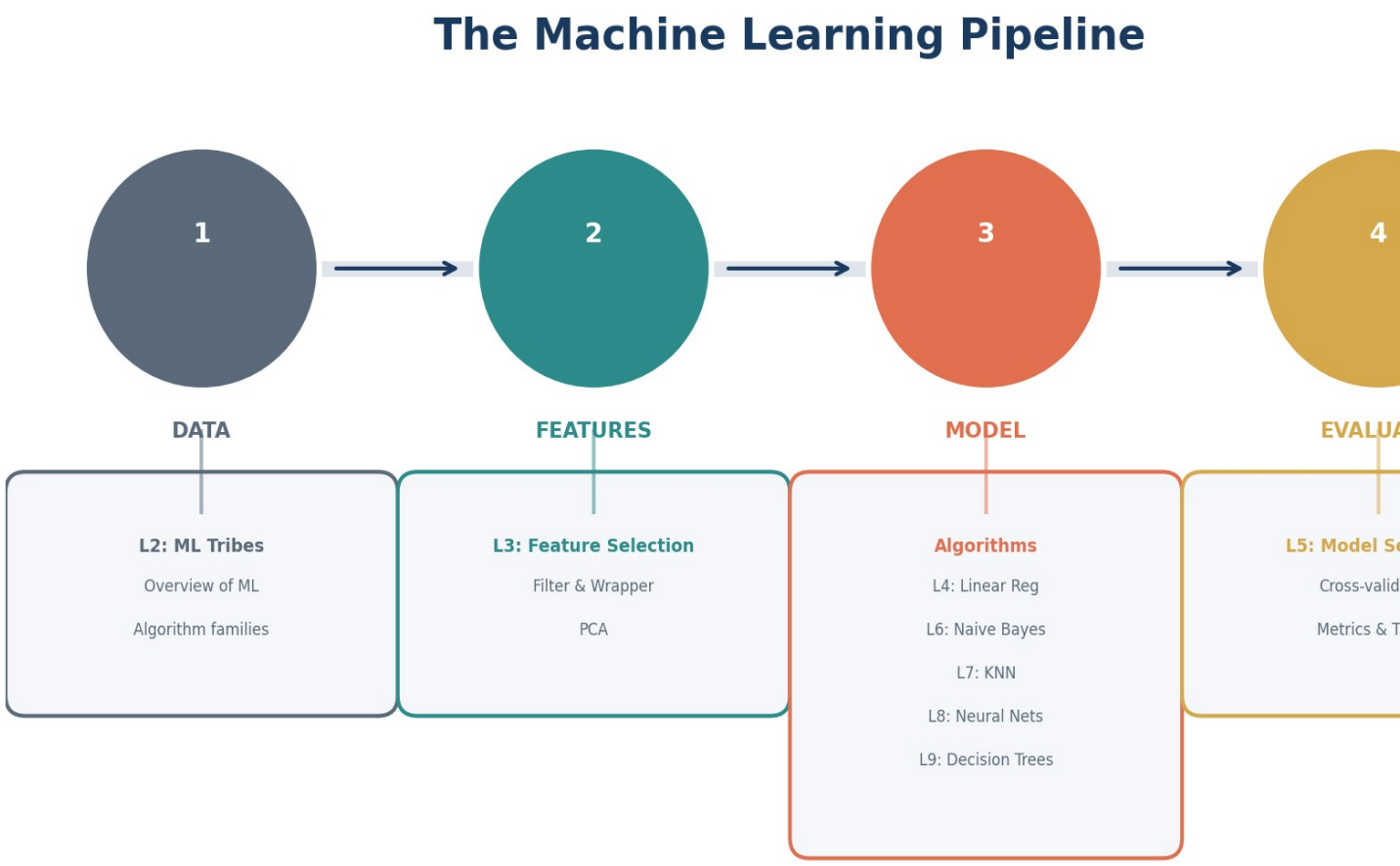


Figure 1: The Machine Learning Pipeline

2. Algorithm Selection Guide

Follow the flowchart to choose the right algorithm:

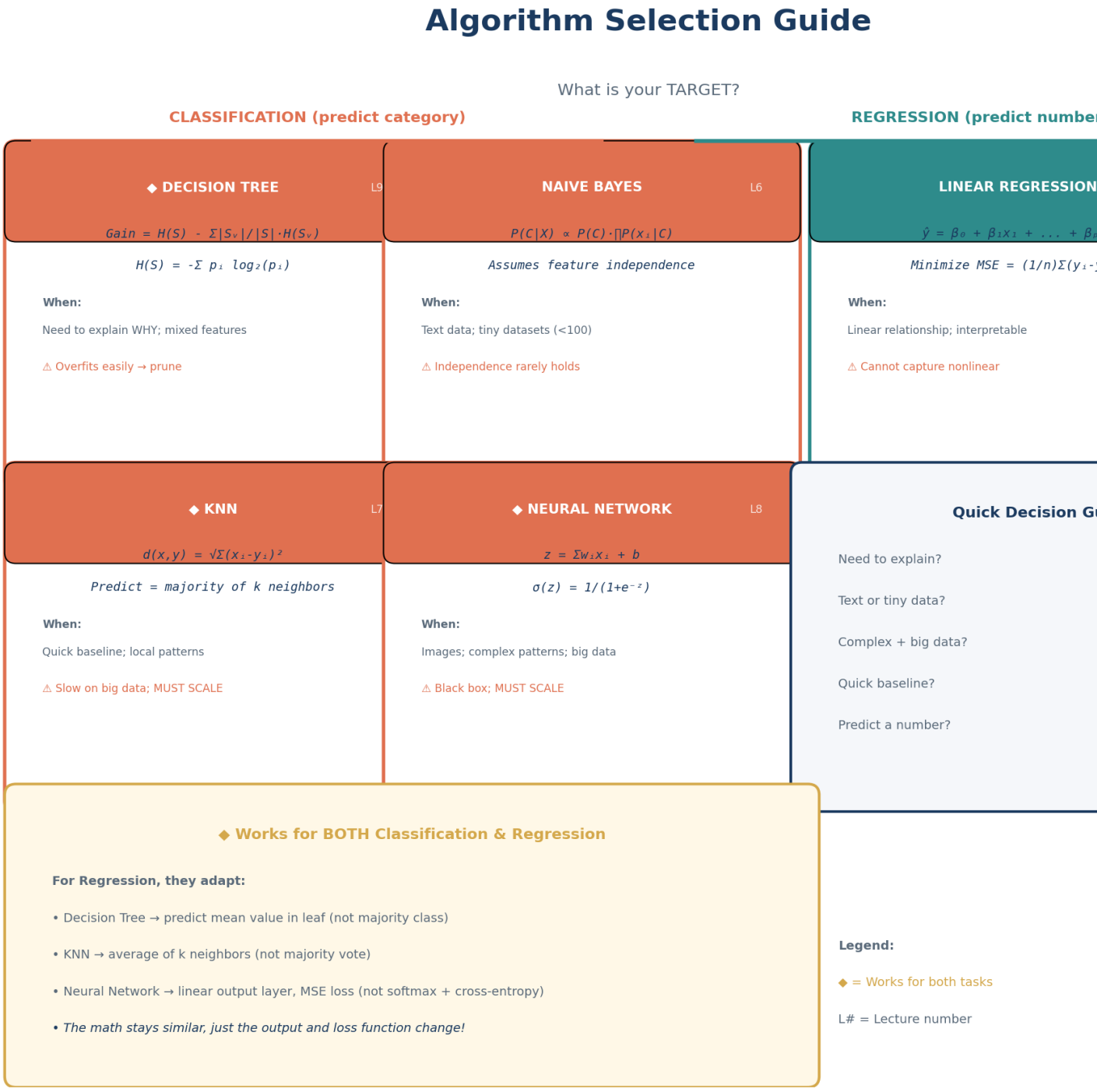
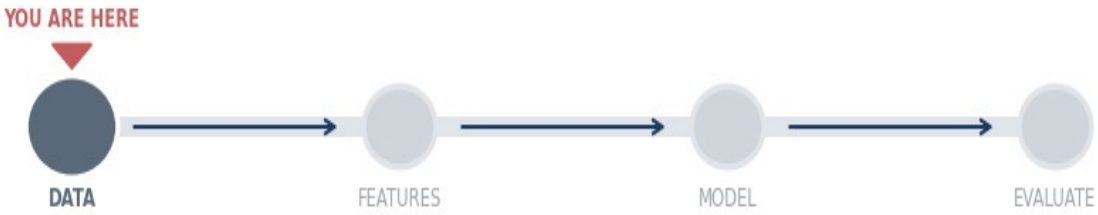


Figure 2: Algorithm Selection Guide

Lecture 2: Machine Learning Tribes



1. How Do Computers Get New Knowledge?

Computers can acquire new knowledge through five fundamental approaches:

1. **Fill in gaps in existing knowledge** – Using logical rules to infer missing information
2. **Emulate the brain** – Modeling biological neural networks
3. **Simulate evolution** – Using natural selection and genetic processes
4. **Reduce uncertainty** – Applying probabilistic reasoning
5. **Check for similarities between old and new** – Learning through analogy

Each approach corresponds to a distinct school of thought or "tribe" in machine learning, as described in Pedro Domingos' book *The Master Algorithm* (2015).

2. The Five Tribes of Machine Learning

Tribe	Origins	Master Algorithm
Symbolists	Logic, Philosophy	Inverse Deduction
Connectionists	Neuroscience	Backpropagation
Evolutionaries	Evolutionary Biology	Genetic Programming
Bayesians	Statistics	Probabilistic Inference
Analogizers	Psychology	Kernel Machines (SVM)

2.1 Symbolists

Origins: Logic and Philosophy

Core Concept: Inverse deduction (induction). Just as subtraction is the inverse of addition, induction is the inverse of deduction.

Understanding Deduction, Induction, and Abduction:
Deduction (general to specific): Given rule + premise, derive conclusion.

Example: "Socrates is human" + "All humans are mortal" = "Socrates is mortal"
Induction (specific to general): Given examples, infer the general rule.
Example: "Socrates is human" + "Socrates is mortal" = "All humans are mortal" (infer rule)
Abduction (inference to best explanation): Given rule + observation, infer premise.
Example: "All humans are mortal" + "Socrates is mortal" = "Socrates is probably human"
ML primarily uses INDUCTION: learning general rules from specific training examples.

Key Techniques:

- **Logic Programming:** Prolog, Inductive and Abductive logic programming
- **Expert Systems:** Knowledge bases and Inference Engines
- **Decision Trees:** C4.5, regression trees, random forests
- **Functional Programming:** Lisp

Components (Symbolists):
Representation: Rules, logic programs, decision trees
Evaluation: Accuracy, information gain, coverage
Optimization: Inverse deduction, rule refinement
Master Formula: IF (conditions) THEN (conclusion)

2.2 Connectionists

Origins: Neuroscience

Core Concept: Learning modeled on biological neurons, using backpropagation to adjust connection weights.

Key Milestones:

- 1943: McCulloch and Pitts model neural cells
- 1957: Perceptron invented by Frank Rosenblatt
- 1986: Backpropagation reinvented
- 2006: Deep Learning revival with Hinton's deep belief nets
- 2017: AlphaGo defeats Go world champion

Components (Connectionists):
Representation: Neural networks (layers of weighted connections)
Evaluation: Error/Loss function (e.g., squared error, cross-entropy)
Optimization: Backpropagation + Gradient Descent
Master Formula: output = activation(sum of weight_i x input_i + bias)

2.3 Evolutionaries

Origins: Evolutionary Biology

Core Concept: Models natural selection, crossover, and mutation. Solutions evolve through generations.

Key algorithms: Genetic Algorithms, Genetic Programming, Ant Colony Optimization, Particle Swarm Optimization

Components (Evolutionaries):
Representation: Programs, variable-length strings, trees

Evaluation: Fitness function (how well solution solves problem)
Optimization: Genetic search (selection, crossover, mutation)
Master Formula: new_population = select(fittest) + crossover + mutate

2.4 Bayesians

Origins: Statistics

Core Concept: Everything is uncertainty, so compute probabilities using Bayes' theorem.

Key algorithms: Naive Bayes, Bayesian Belief Networks, Hidden Markov Models, MCMC

Components (Bayesians):
Representation: Probabilistic graphical models (Bayesian networks)
Evaluation: Posterior probability P(hypothesis|data)
Optimization: Probabilistic inference (exact or MCMC sampling)
Master Formula: P(H|D) = P(D|H) x P(H) / P(D) - Bayes Theorem

2.5 Analogizers

Origins: Psychology

Core Concept: Learning through analogy—solving new problems by finding similar past cases.

Key algorithms: k-Nearest Neighbors (kNN), Support Vector Machines (SVM)

Components (Analogizers):
Representation: Instances, similarity metrics, kernel functions
Evaluation: Margin (SVM), classification accuracy (KNN)
Optimization: Constraint optimization (SVM), lazy evaluation (KNN)
Master Formula: prediction = f(similarity to known examples)

3. Unifying the Tribes

Domingos proposes a unified "Master Algorithm" combining: Probabilistic logic for representation, Posterior probability for evaluation, Genetic Programming for formula discovery, and Backpropagation for weight learning.

The Master Algorithm Components:
Representation: Probabilistic logic (Markov Logic Networks) - each rule has a weight
Evaluation: Posterior probability + user-defined objective function
Optimization: Formula discovery via Genetic Programming + Weight learning via Backpropagation
This combines the strengths of all five tribes into one unified framework.

Lecture 3: Feature Selection

Section A: What is Feature Selection?

Feature selection methods reduce the number of input variables to those most useful for predicting the target variable.

Why Perform Feature Selection?

- **Improved Interpretability:** Simpler models are easier to understand
- **Shorter Training Time:** Fewer features = less computation
- **Enhanced Generalization:** Reduces overfitting
- **Variable Redundancy:** Removes duplicate information

Section B: Types of Feature Selection Methods

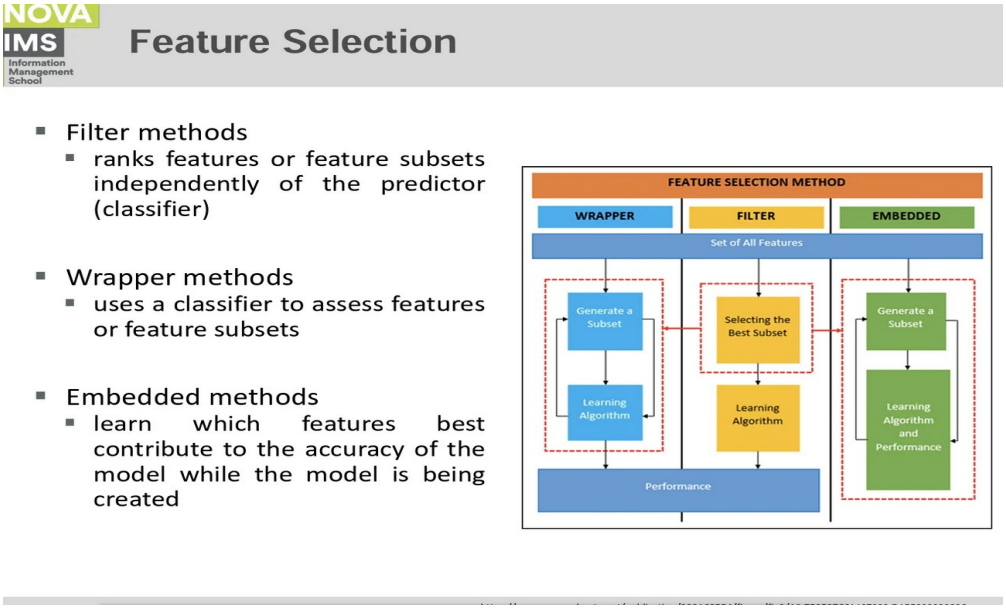


Figure 1: Overview of Feature Selection Methods

B.1 Filter Methods

[METHOD TYPE: FILTER - Statistical scoring independent of any ML model]
What they are: Filter methods rank features independently of any ML algorithm using statistical techniques.

Process: Input Features → Statistical Scoring → Feature Subset Selection → Learning Algorithm

Advantages: Fast, computationally efficient, model-agnostic

Limitation: May miss feature interactions since features are evaluated independently

Section C: Statistical Tests for Filter Methods

Understanding Univariate vs Bivariate Analysis:
Univariate: Examines ONE variable at a time. In feature selection, this means testing each feature INDIVIDUALLY against the target - asking "does this single feature relate to the target?" Features are evaluated in isolation, ignoring interactions between features.
Bivariate: Examines the relationship between TWO variables simultaneously. This is what correlation coefficients measure - how two variables move together.
Multivariate: Examines relationships among THREE or more variables together, capturing interactions that univariate methods miss.
Filter methods are typically univariate - they test each feature independently. This is fast but may miss features that are only useful in combination with others.

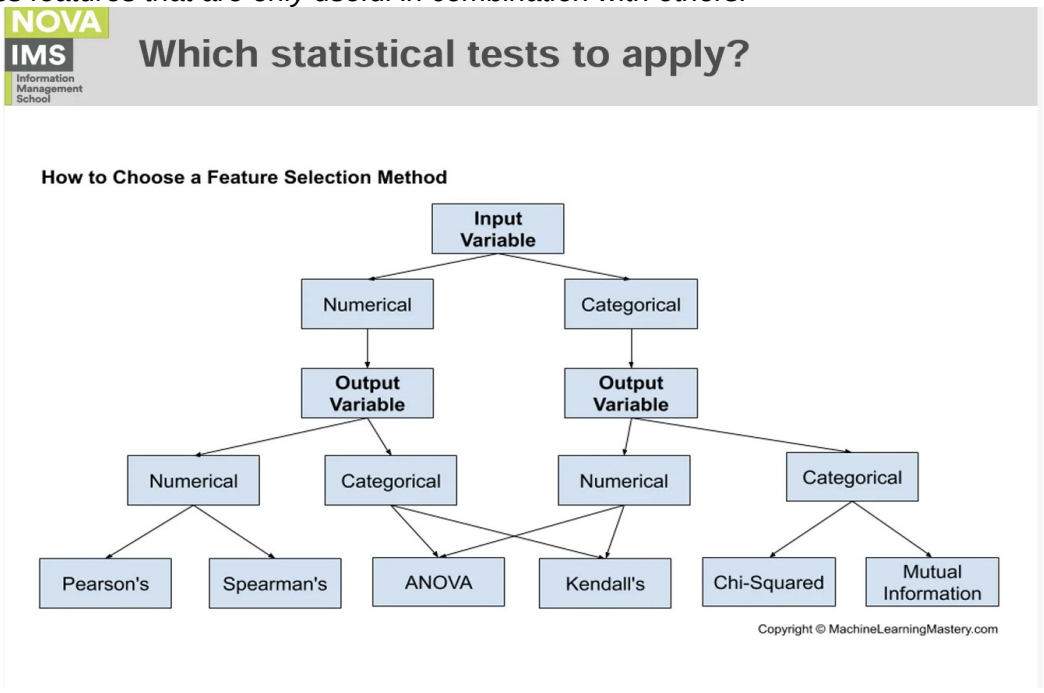


Figure 2: How to Choose a Feature Selection Method

Input \ Output	Numerical Output	Categorical Output
Numerical Input	Pearson's, Spearman's	ANOVA

Categorical Input	Kendall's	Chi-Squared, Mutual Info

3.1 Understanding Correlation Indices

Correlation indices are statistical measures that quantify the **strength and direction** of the relationship between two variables. In Feature Selection, they serve two purposes:

6. **Select relevant features:** High correlation with target → feature is likely useful
7. **Exclude redundant features:** High correlation between features → one can be removed

Important: Correlation does NOT imply causation! A high correlation simply indicates a relationship exists, not that one variable causes changes in the other.

C.2 Pearson's Correlation Coefficient

Use case: Two continuous variables with a **linear relationship**

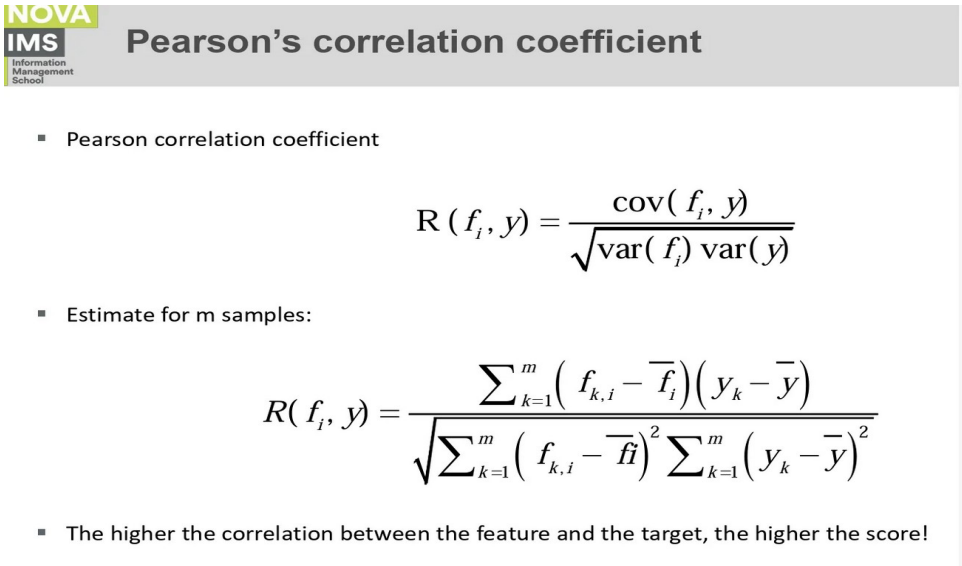
Assumptions: Both variables are normally distributed

What it measures: The strength of the linear relationship between two continuous variables.

Mathematical Formula:

$$r = \Sigma[(x_i - \bar{x})(y_i - \bar{y})] / \sqrt{[\Sigma(x_i - \bar{x})^2 \times \Sigma(y_i - \bar{y})^2]}$$

Or equivalently: $r = Cov(X,Y) / (\sigma_x \times \sigma_y)$



Interpretation:

- **r = +1:** Perfect positive linear relationship
- **r = -1:** Perfect negative linear relationship
- **r = 0:** No linear relationship (but could still have nonlinear relationship!)
- **|r| > 0.7:** Generally considered strong correlation

Limitation: Only detects linear relationships. A perfect parabola would have r ≈ 0!

Deep Intuition: Why Pearson Only Detects Linear Relationships
The Pearson formula measures how much X and Y deviate from their means TOGETHER:
r = sum of [(xi - mean_x)(yi - mean_y)] / normalizing factor
When X is above its mean and Y is also above its mean: product is POSITIVE.
When X is below its mean and Y is also below its mean: product is also POSITIVE.
If this happens consistently, r approaches +1 (positive linear relationship).
The Problem with Non-Linear Data (e.g., Y = X squared):
When X = -3, Y = 9. When X = +3, Y = 9. Same Y for opposite X values!
The positive products (right side of parabola) CANCEL the negative products (left side).
Result: r is near 0, even though Y = X squared is a PERFECT mathematical relationship!
Practical Rule - When to Use Pearson:
1. ALWAYS scatter plot your data first
2. If points form a roughly straight band, Pearson works
3. If you see curves, U-shapes, or clusters, use Spearman or Mutual Information
Spearman converts to RANKS first - it only asks "as X increases, does Y tend to increase?"
This captures ANY monotonic relationship, not just straight lines.

C.3 Spearman's Rank Correlation

Use case: Two continuous and/or ordinal variables with a **monotonic relationship**

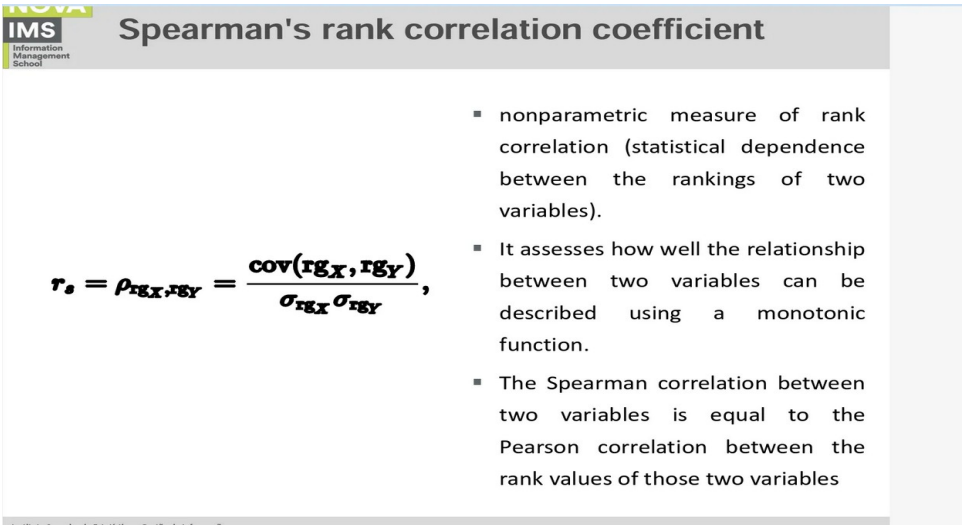
Assumptions: Non-parametric (no assumptions about data distribution)

What it measures: How well the relationship can be described by a monotonic function (always increasing or always decreasing).

Mathematical Formula:

$$\rho = 1 - (6 \times \Sigma d_i^2) / [n(n^2 - 1)]$$

Where $\mathbf{d_i = rank(x_i) - rank(y_i)}$ is the difference between ranks for each observation.



Key insight: Spearman's ρ = Pearson's r applied to rank values. This makes it robust to outliers and able to detect monotonic (not just linear) relationships.

Range: -1 to +1, with 0 indicating no monotonic relationship.

C.4 Kendall's Tau Correlation

Use case: Two continuous and/or ordinal variables

What it measures: Very similar to Spearman, but uses a different mathematical approach based on concordant and discordant pairs.

Mathematical Formula:

$$\tau = (\text{concordant pairs} - \text{discordant pairs}) / \text{total pairs}$$

Concordant pair: Both x and y increase together. **Discordant pair:** One increases while the other decreases.

Range: -1 to +1. Non-parametric, often preferred for small samples.

C.5 Cramer's V (for Categorical Variables)

Use case: Two **categorical variables**

What it measures: The strength of association between two categorical variables, based on Chi-Square statistic.

Mathematical Formula:

$$V = \sqrt{[\chi^2 / (n \times (\min(r,c) - 1))]}$$

Where χ^2 is the Chi-Square statistic, n is sample size, r is number of rows, c is number of columns.

Range: 0 to 1, where 0 = no association, 1 = perfect association.

C.6 Chi-Square Test for Categorical Feature Selection

Purpose: To check if a categorical feature is **independent** of the target variable. If they're independent, the feature is not useful for prediction.

How It Works:

8. **Create contingency table:** Cross-tabulate the feature and target
9. **Calculate expected frequencies:** What we'd expect if variables were independent
10. **Compute Chi-Square statistic:** $\chi^2 = \Sigma[(\text{Observed} - \text{Expected})^2 / \text{Expected}]$
11. **Get p-value:** If $p < 0.05$, reject independence → feature is useful

Practical Example (Titanic Dataset):

Using Chi-Square to test which categorical features predict survival:

- **Cabin:** $p < 0.05$ → IMPORTANT for prediction (keep)
- **Sex:** $p < 0.05$ → IMPORTANT for prediction (keep)
- **Title:** $p < 0.05$ → IMPORTANT for prediction (keep)
- **Last Name:** $p > 0.05$ → NOT important (discard)
- **Ticket:** $p > 0.05$ → NOT important (discard)
- **First Name:** $p > 0.05$ → NOT important (discard)

Key insight: The Chi-Square test automates feature selection for categorical variables by testing statistical independence.

C.7 Summary: Choosing a Correlation Measure

Correct Statistical Test Selection:
The choice of statistical test depends on the variable types of your INPUT and OUTPUT:
Numerical Input + Numerical Output: Pearson (linear), Spearman (monotonic), Kendall (ordinal)
Numerical Input + Categorical Output: ANOVA F-test, Mutual Information
Categorical Input + Numerical Output: ANOVA F-test, Kendall
Categorical Input + Categorical Output: Chi-Squared, Mutual Information, Cramer's V
Key insight: ANOVA works whenever you have a MIX of numerical and categorical - it tests whether the numerical variable differs significantly across categorical groups.

Method	Variable Types	Relationship	Range
Pearson	Continuous	Linear only	-1 to +1
Spearman	Continuous/Ordinal	Monotonic	-1 to +1
Kendall	Continuous/Ordinal	Monotonic	-1 to +1
Cramer's V	Categorical	Association	0 to 1
Chi-Square	Categorical	Independence test	p-value

Section D: Information Theory and Mutual Information

Deep Intuition: Information Theory from First Principles

The Core Problem Information Theory Solves

Before Shannon, we had no mathematical way to answer: "How much information is in a message?"

Think about why this is hard:

"The sun rose today" → low information (you already expected this)

"A meteor will hit Earth tomorrow" → high information (completely unexpected)

Shannon's insight: Information is the resolution of uncertainty.

Surprise: The Building Block

Start with a single event. How "surprising" is it when event X occurs?

Intuition requirements for a "surprise" function:

1. Certain events (P=1) should have zero surprise

2. Impossible events ($P \to 0$) should have infinite surprise

3. Independent events should have ADDITIVE surprise: Surprise(A and B) = Surprise(A) + Surprise(B)

The ONLY function satisfying all three properties:

Surprise(x) = $-\log_2 P(x)$

Why logarithm? The additivity requirement forces it.

We want: Surprise(A and B) = Surprise(A) + Surprise(B)

But probabilities multiply: P(A and B) = P(A) × P(B) for independent events

We need a function f where: f(a × b) = f(a) + f(b)

That is the definition of logarithm: log(a × b) = log(a) + log(b)

Concrete Example: Two coin flips

Coin 1: P(heads) = 0.5, so Surprise = $-\log_2(0.5)$ = 1 bit

Coin 2: P(heads) = 0.5, so Surprise = $-\log_2(0.5)$ = 1 bit

Both heads: P(both) = 0.5 × 0.5 = 0.25

Surprise(both) = $-\log_2(0.25)$ = 2 bits

Check: 1 bit + 1 bit = 2 bits. The logarithm automatically converts multiplication into addition.

Why negative? Why base 2?

Negative: P(x) is between 0 and 1, so log of it is negative. We negate to get positive surprise values.

Base 2: Gives us units of "bits." One bit = information from a fair coin flip.

Entropy: Expected Surprise

For a random variable X with multiple possible outcomes:

Entropy = average surprise weighted by probability

$H(X) = -\sum P(x) \log_2 P(x)$

What does entropy measure?

"On average, how many yes/no questions do I need to determine the outcome?"

Example: Fair coin

P(H) = 0.5, P(T) = 0.5

$H(X) = -[0.5 \times \log_2(0.5) + 0.5 \times \log_2(0.5)]$

$H(X) = -[0.5 \times (-1) + 0.5 \times (-1)] = 1$ bit

One yes/no question ("Is it heads?") determines the outcome.

Example: Biased coin (99% heads)

P(H) = 0.99, P(T) = 0.01

H(X) ≈ 0.08 bits

Almost no questions needed — just guess heads. Low entropy = low uncertainty.

Key insight: Entropy is maximized when all outcomes are equally likely, and minimized when one outcome is certain.

The Problem Entropy Alone Cannot Answer

Entropy tells you: How uncertain is X?

But often you care about: How much does knowing Y reduce uncertainty about X?

Examples:

• How much does knowing someone's age tell you about their income?

• How much does knowing email content tell you about whether it's spam?

• How much does knowing a pixel tell you about the object class?

That quantity is mutual information.

Three Cases That Build Intuition

Case A: Independent variables

X = coin flip, Y = dice roll (unrelated)

Knowing Y tells you nothing about X

$H(X) = 1$ bit, $H(X|Y) = 1$ bit $\rightarrow I(X;Y) = 0$

No shared information.

Case B: Perfectly dependent

X = temperature in Celsius, Y = same temperature in Fahrenheit

They contain the same information

$H(X|Y) = 0 \rightarrow I(X;Y) = H(X)$

All information is shared.

Case C: Partially related (real life)

X = exam grade, Y = hours studied

Y gives some information about X, but not perfect

$H(X|Y) < H(X) \rightarrow 0 < I(X;Y) < H(X)$

Partial information is shared.

Why Symmetry? I(X;Y) = I(Y;X)

$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$

How much X tells you about Y = How much Y tells you about X

That's why it's called MUTUAL information.

The KL-Divergence Form (measuring distance from independence)

MI can also be written as:

$$\mathbf{I(X;Y) = \sum P(x,y) \log[P(x,y) / (P(x)P(y))]}$$

This compares what actually happens jointly vs. what would happen if independent.

If $P(x,y) = P(x)P(y)$, then $\log[P(x,y)/(P(x)P(y))] = \log(1) = 0$

MI measures how far the joint distribution is from independence.

Geometric Intuition: The Fog Metaphor

Think of uncertainty as fog:

• Entropy H(X) = how thick the fog is

• Conditioning on Y removes some fog

• Mutual information = how much fog disappears

If Y tells you nothing → fog stays

If Y tells you everything → fog clears completely

Why ML Cares About Mutual Information

MI answers critical ML questions:

• Which feature is most informative about the label?

• Which word tells me most about sentiment?

• Which pixel tells me most about the object?

• Which variable should I split on in a decision tree?

Used in: Feature selection, decision trees, representation learning, deep learning theory.

The One Formula to Remember

$I(X;Y) = H(X) - H(X|Y)$

Mutual information is the reduction in uncertainty of X when Y is known.

Even Cleaner Intuition

If entropy is: Average surprise

Then mutual information is: How much surprise disappears when you observe another variable

Joint Entropy: Two Variables Together

$H(X,Y) = -\sum\sum P(x,y) \log_2 P(x,y)$

Measures: "How many bits to describe both X and Y together?"

Critical property: $H(X,Y) \leq H(X) + H(Y)$

Equality ONLY when independent. If related, knowing one tells you about the other.

Conditional Entropy: Remaining Uncertainty

$H(Y|X) = -\sum\sum P(x,y) \log_2 P(y|x)$

Measures: "If I know X, how much uncertainty remains about Y?"

The Chain Rule: $H(X,Y) = H(X) + H(Y|X)$

Total = Uncertainty in X + Remaining uncertainty in Y after knowing X

Mutual Information: The Payoff for ML

$I(X;Y) = H(Y) - H(Y|X)$

= Uncertainty in Y MINUS Uncertainty in Y after knowing X

= The REDUCTION in uncertainty about Y gained by learning X

Equivalent formulations:

$I(X;Y) = H(Y) - H(Y|X)$

$I(X;Y) = H(X) - H(X|Y)$

$I(X;Y) = H(X) + H(Y) - H(X,Y)$

Venn Diagram Model:

H(X) = left circle, H(Y) = right circle, I(X;Y) = overlap

H(X|Y) = X not in Y, H(Y|X) = Y not in X, H(X,Y) = total area

For Feature Selection:

$I(X;Y) = 0 \rightarrow$ X useless (tells nothing about Y)

$I(X;Y) > 0 \rightarrow$ X useful (reduces uncertainty in Y)

$I(X;Y) = H(Y) \rightarrow$ X perfect (determines Y completely)

Why MI Beats Correlation:

Correlation detects LINEAR/monotonic relationships only

MI detects ANY statistical dependency

Example: X=[1,2,3,4,5,6,7,8], Y=[1,0,1,0,1,0,1,0]

Correlation ≈ 0 but MI > 0 (Y determined by X odd/even)

Connection to Classifiers:

Decision Trees: Information Gain = $H(Y) - H(Y|X) = I(X;Y)$

Best split = feature with maximum MI with target

Naive Bayes: $P(Y|X_1,...,X_n) \propto P(Y) \times \prod P(X_i|Y)$

Features with high I(X;Y) are informative; those with I(X;Y) ≈ 0 can be dropped

One Sentence Summary:

Entropy $H(X) = -\sum P(x)\log_2 P(x)$ measures average surprise; Mutual Information $I(X;Y) = H(Y) - H(Y|X)$ measures shared information; both detect ANY dependency (not just linear), powering decision tree splits and feature selection.

Information theoretic criteria provide a powerful way to measure relationships between variables. Unlike correlation, they can detect **any type of dependency**—including nonlinear relationships.

D.1 What is Mutual Information?

Intuitive explanation: Mutual Information (MI) measures "how much information (in terms of entropy) two random variables share." In other words, it quantifies how much knowing one variable reduces uncertainty about the other.

Think of it this way: If you know someone's height, how much does that tell you about their weight? If knowing height gives you a good guess about weight, then height and weight have high mutual information.

Mathematical Formula:

Information Theory

- Information Theoretic Criteria
- Most approaches use (empirical estimates of) **mutual information** between features and the target:

$$I(x_i, y) = \int_{x_i} \int_{y} p(x_i, y) \log \frac{p(x_i, y)}{p(x_i)\, p(y)}\, dxdy$$

- Case of discrete variables:

$$I(x_i, y) = \sum_{x_i} \sum_{y} P(X = x_i, Y = y) \log \frac{P(X = x_i, Y = y)}{P(X = x_i) P(Y = y)}$$

- (probabilities are estimated from frequency counts)

Figure 3: Mutual Information Formulas (Continuous and Discrete Cases)

Understanding the Formula:

The formula compares the **joint probability** p(x,y) with what we'd expect if the variables were **independent** p(x)×p(y). When these differ a lot, the variables share information.

- **If X and Y are independent:** MI = 0 (knowing X tells you nothing about Y)
- **If X determines Y completely:** MI is maximized
- **For discrete variables:** Probabilities are estimated from frequency counts in your data

Visual Understanding with Entropy:

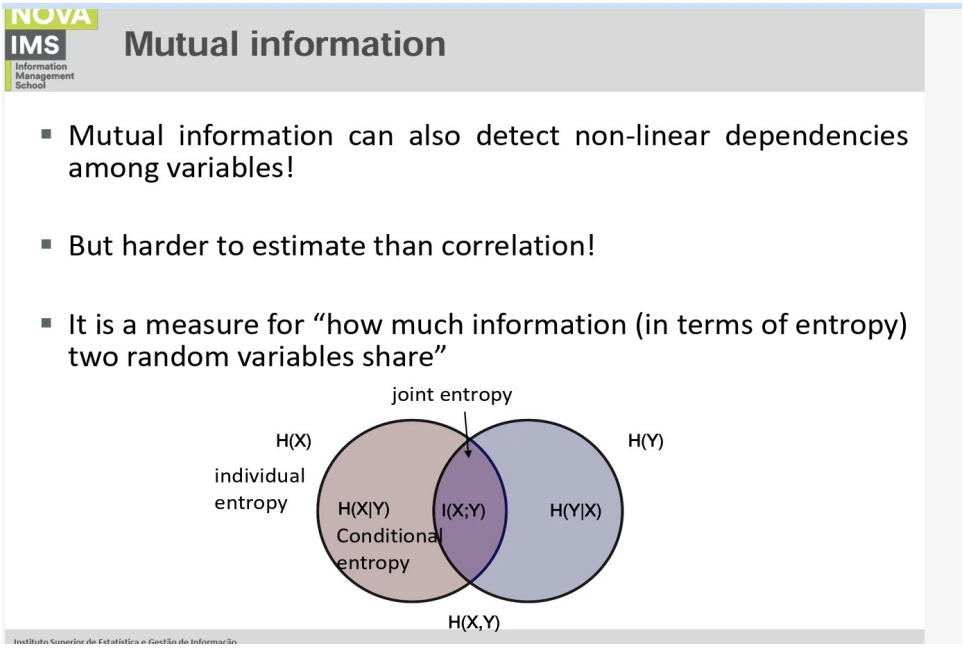


Figure 4: Mutual Information as Shared Entropy

Reading the Venn Diagram:

- **H(X):** Total entropy (uncertainty) of variable X
- **H(Y):** Total entropy of variable Y
- **I(X;Y):** The overlap—mutual information. This is the shared information between X and Y
- **H(X|Y):** Conditional entropy—uncertainty remaining in X after knowing Y
- **H(X,Y):** Joint entropy—total uncertainty in both variables together

D.2 Advantages of Mutual Information

- **Detects non-linear dependencies:** Unlike Pearson correlation (which only finds linear relationships), MI can find ANY type of relationship
- **More general:** Works for both numerical and categorical variables

Example: Imagine X is "time of day" and Y is "traffic level." Traffic might be high in the morning, low at noon, high in evening—a complex nonlinear pattern. Pearson correlation might show low correlation (because it's not a straight line), but MI would correctly identify that X strongly predicts Y.

D.3 Limitations of Mutual Information

- **Not normalized:** MI values can be incomparable between different datasets. A value of 0.5 in one dataset might mean something different than 0.5 in another.
- **Hard to compute for continuous variables:** Continuous variables need to be discretized (binned), and the results can be sensitive to how you choose the bins.
- **Harder to estimate than correlation:** Requires more data for reliable estimates

D.4 Maximal Information Coefficient (MIC)

MIC was developed to address the limitations of standard mutual information. It solves two key problems:

- **Normalization:** MIC lies in the range [0, 1], making it comparable across datasets
- **Optimal binning:** MIC automatically searches for the best way to bin continuous variables

Formula: $MIC(x, y) = \max\{I(x, y) / \log_2 \min\{n_x, n_y\}\}$

Interpretation:

- **MIC = 0:** Variables are independent
- **MIC = 1:** One variable completely explains the other
- **MIC represents:** The percentage of variable Y that can be explained by variable X

Section E: Wrapper Methods

[METHOD TYPE: WRAPPER - Uses ML model performance to evaluate feature subsets]

While filter methods rely on statistical properties of the data, wrapper methods **"cheat" at identifying relevant variables** by creating multiple models and using their performance as a proxy for feature relevance.

Key concept: The **base estimator** is a model we use just to "peek" at which features are relevant—it's not necessarily the model we'll use for final predictions.

E.1 How Wrapper Methods Work

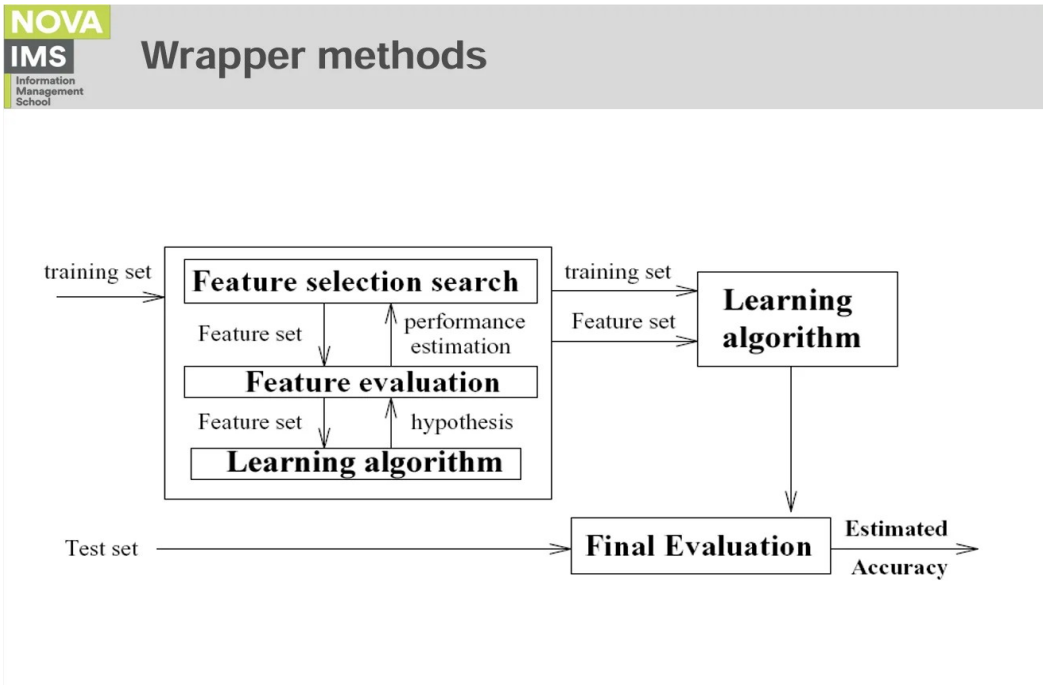


Figure 5: Wrapper Methods Workflow

The General Process:

12. **Select a subset of features**
13. **Train a model** using only these features
14. **Evaluate the model's performance** according to a pre-defined metric
15. **Repeat steps 1-3** on different feature subsets
16. **Select the feature subset** that resulted in the best model performance

E.2 Types of Wrapper Methods

The main difference between wrapper methods is **how they select feature subsets**. Here are the main approaches:

Exhaustive Feature Selection

- **How it works:** Tries ALL possible combinations of features and selects the one with best performance
- **Advantage:** Guaranteed to find the optimal subset
- **Disadvantage:** Computationally infeasible for large feature sets (2^n combinations!)

Forward Selection

- **How it works:** Starts with NO features, adds one feature at a time
- **At each step:** Add the feature that improves model performance the MOST
- **Stop when:** Adding more features doesn't improve performance
- **Best for:** When you expect few features to be relevant

Backward Elimination

- **How it works:** Starts with ALL features, removes one feature at a time
- **At each step:** Remove the feature that decreases model performance the LEAST
- **Stop when:** Removing any feature significantly hurts performance
- **Best for:** When you expect most features to be relevant

Stepwise Selection

- **How it works:** Combination of forward and backward
- **At each step:** Features can be ADDED or REMOVED based on performance impact
- **Advantage:** More flexible than pure forward/backward, can correct earlier mistakes

Recursive Feature Elimination (RFE)

- **How it works:** A special case of Backward Selection
- **Key difference:** Uses the base estimator's feature importance to rank features
- **Process:** Recursively removes the LEAST important feature based on model coefficients/importance scores

Section F: Recursive Feature Elimination (RFE)

RFE is one of the most popular wrapper methods. It works by **recursively removing the least important features** until the desired number remains.

F.1 How RFE Works

17. **Start with all features:** Train a model using all predictors
18. **Rank features:** Each predictor is ranked by its importance to the model
19. **Remove worst features:** At each iteration, keep only the S_i top-ranked features
20. **Retrain and evaluate:** Refit the model with fewer features, measure performance
21. **Select optimal subset:** Choose the number of features that gives best performance

Algorithm Pseudocode:

Algorithm 1: Recursive feature elimination

```
Tune/train the model on the training set using all predictors
Calculate model performance
Calculate variable importance or rankings
for Each subset size S_i, i = 1 ... S do
    Keep the S_i most important variables
    [Optional] Pre–process the data
    Tune/train the model on the training set using S_i predictors
    Calculate model performance
    [Optional] Recalculate the rankings for each predictor
end
Calculate the performance profile over the S_i
Determine the appropriate number of predictors
Use the model corresponding to the optimal S_i
```

Figure 6: Recursive Feature Elimination Algorithm

Concrete Example:

Suppose you have 100 features and want to find the optimal subset:

- S = {100, 80, 60, 40, 20, 10, 5} — candidate subset sizes
- Train with all 100 features → Accuracy = 85%
- Keep top 80 features → Accuracy = 86%
- Keep top 60 features → Accuracy = 87%
- Keep top 40 features → Accuracy = 88% ← Best!
- Keep top 20 features → Accuracy = 86%
- Final model uses 40 features

F.2 The Selection Bias Bias Problem

Warning: RFE can suffer from **selection bias** (Ambroise and McLachlan, 2002). This happens when:

- You have many uninformative features and one happens to correlate with the outcome by chance
- RFE gives that spurious feature a high rank
- On new data, that feature is useless

The core problem: Training data is being used for THREE purposes simultaneously:

22. Predictor selection (choosing which features)
23. Model fitting (training the model)
24. Performance evaluation (measuring accuracy)

Unless you have a very large dataset, this leads to overly optimistic performance estimates and features that don't generalize.

Section G: RFE with Cross-Validation (RFECV)

Solution: Wrap the entire RFE process inside cross-validation. This gives honest performance estimates that account for the variation due to feature selection.

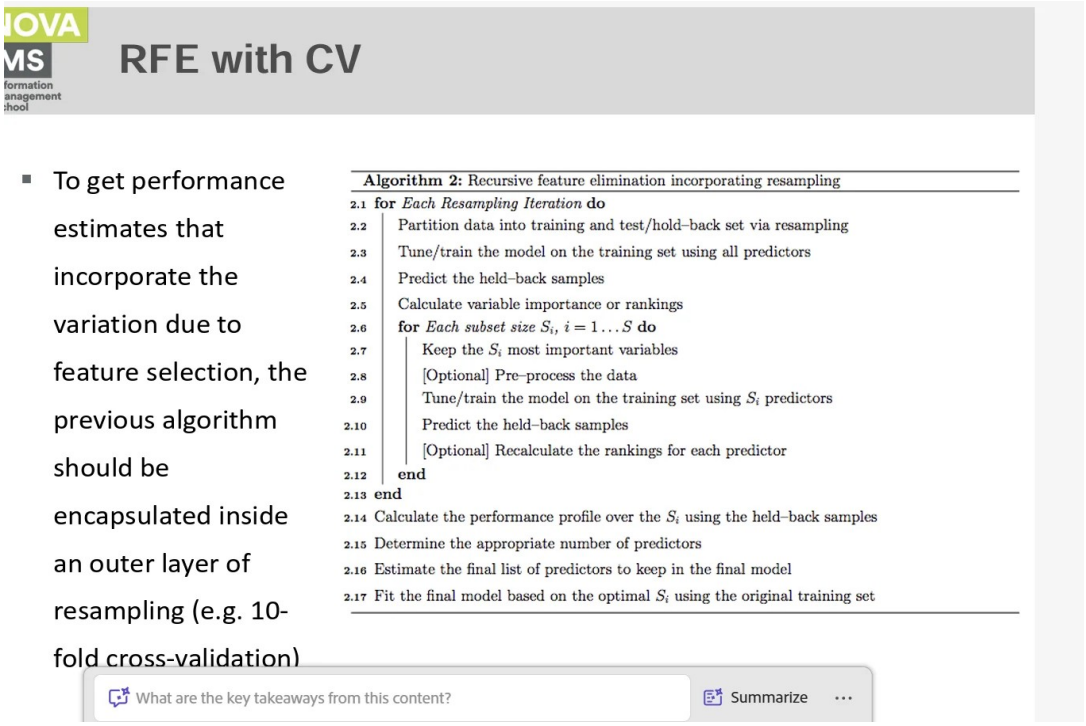


Figure 7: RFE with Cross-Validation Algorithm

G.1 How RFECV Works

The key insight is that the entire feature selection process happens **inside each fold**:

25. **Split data into folds:** e.g., 10-fold cross-validation
26. **For each fold:** Use training portion to run RFE, predict on held-back portion
27. **Aggregate results:** Calculate performance across all folds for each subset size
28. **Determine optimal size:** Find the number of features with best cross-validated performance
29. **Final model:** Fit using the optimal S_i on the full training set

G.2 Benefits of RFECV

- **Better performance estimates:** Accounts for variability from feature selection
- **Probabilistic feature importance:** Each fold may select slightly different features, giving a more robust ranking
- **Consensus ranking:** At the end, features consistently selected across folds are more trustworthy

G.3 Trade-offs

- **Computationally expensive:** You're now training many models (folds × subset sizes)

- **Can be parallelized:** The outer resampling loop can run on multiple processors

Section H: Embedded Methods

[METHOD TYPE: EMBEDDED - Feature selection happens during model training (regularization)]
Embedded methods **learn which features best contribute to accuracy while the model is being created**. Feature selection happens as part of the training process, not as a separate step.

H.1 What Are Embedded Methods?

The most common type are **regularization methods** (also called **penalization methods**). They:

- Introduce additional constraints into the optimization
- Bias the model toward lower complexity (fewer coefficients)
- Automatically shrink unimportant feature weights toward zero

Key examples: Ridge Regression (L2), LASSO (L1), and Elastic Net (L1+L2)

Why Regularization Works:

Think of it like a budget constraint. Without regularization, the model can freely assign any coefficient value. With regularization, there's a "cost" to having large coefficients, forcing the model to be more selective about which features get non-zero weights.

H.2 Ridge Regression (L2 Regularization)

What is it? Ridge regression adds a penalty based on the **squared** magnitude of coefficients to the standard least squares cost function.

First, recall Ordinary Least Squares:

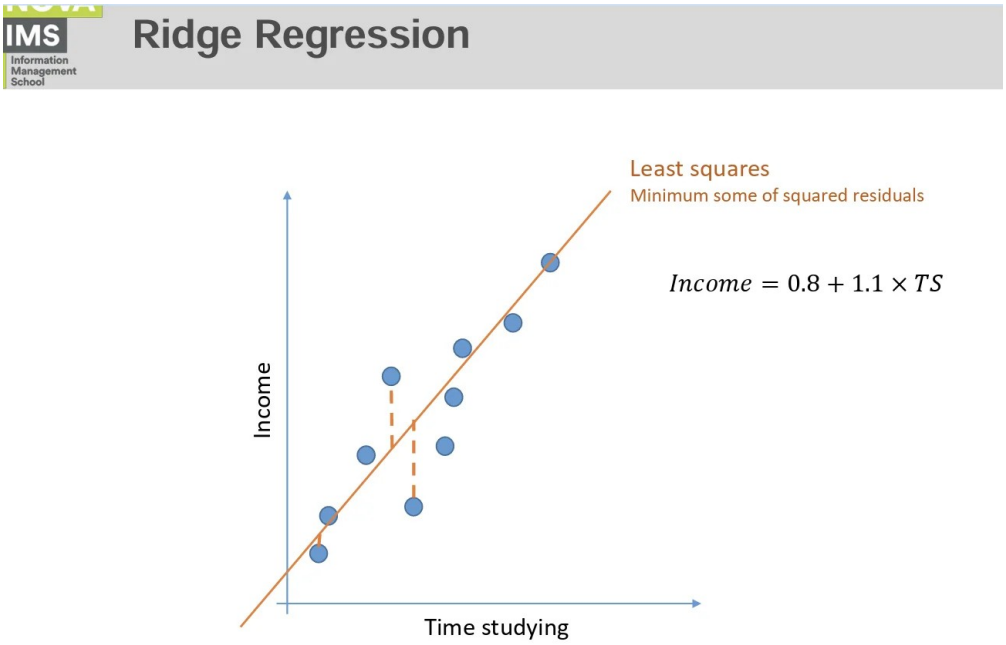


Figure 8: Ordinary Least Squares - Minimizing Squared Residuals

In ordinary least squares, we find the line that minimizes the sum of squared distances (residuals) between predictions and actual values. The equation Income = 0.8 + 1.1 × TimeStudying means each additional hour of studying predicts $1.10 more income.

Ridge Regression Cost Function:

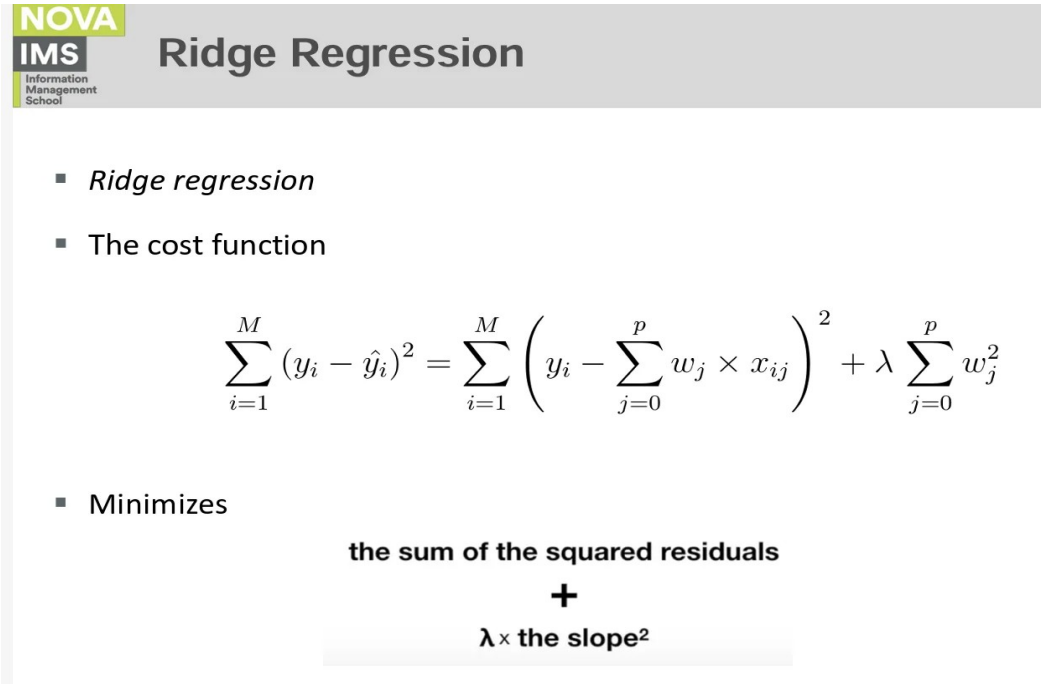


Figure 9: Ridge Regression Cost Function

Understanding the Formula:

Ridge minimizes: **Sum of Squared Residuals + λ × Sum of Squared Coefficients**

- **First part:** Standard least squares (fit the data)
- **Second part (λΣw²):** Penalty for large coefficients
- **λ (lambda):** Controls the trade-off between fit and simplicity

The λ Parameter:

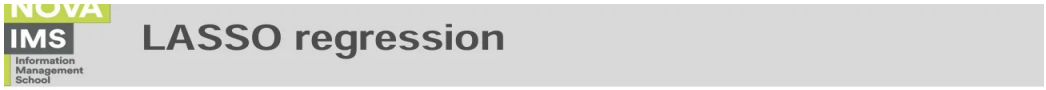
- **λ = 0:** Ridge = Ordinary Least Squares (no penalty)
- **λ → ∞:** All coefficients shrink toward zero
- **Increasing λ:** Promotes smaller slopes (asymptotically to zero)
- **Choosing λ:** Use cross-validation to find value with lowest variance

When to Use Ridge:

- When you have multicollinearity (correlated features)
- When you have more features than samples
- **Note:** Ridge shrinks coefficients but does NOT set them to exactly zero

H.3 LASSO Regression (L1 Regularization)

LASSO stands for **Least Absolute Shrinkage and Selection Operator**. The key difference from Ridge: LASSO uses the **absolute value** of coefficients instead of squared.



- The cost function for Lasso (Least Absolute Shrinkage and Selection operator) regression can be written as:

$$\sum_{i=1}^{M}(y_i-\hat{y}_i)^2=\sum_{i=1}^{M}\left(y_i-\sum_{j=0}^{p}w_j\times x_{ij}\right)^2+\lambda\sum_{j=0}^{p}|w_j|$$

- Difference is instead of taking the square of the coefficients, magnitudes are taken into account.
- This type of regularization (L1) can lead to zero coefficients i.e. some of the features are completely neglected for the evaluation of output

Figure 10: LASSO Regression Cost Function

Understanding the Difference:

LASSO minimizes: **Sum of Squared Residuals + λ × Sum of |Coefficients|**

- **Ridge (L2):** Penalty = λΣw² (squared coefficients)
- **LASSO (L1):** Penalty = λΣ|w| (absolute value of coefficients)

Why Does This Matter?

The crucial difference: L1 regularization can drive coefficients to **exactly zero**, effectively removing features from the model!

Intuitive explanation: Imagine you're trying to reduce costs. With L2 (squared penalty), halving a coefficient reduces the penalty by 75%. With L1 (absolute penalty), halving a coefficient only reduces the penalty by 50%. This means L1 makes it more worthwhile to eliminate features entirely rather than just shrinking them.

LASSO as Feature Selection:

This is why LASSO is an **embedded feature selection method**—it automatically selects features by setting unimportant ones to zero.

Example:

Suppose you have 100 features predicting house prices. After LASSO regression with appropriate λ:

- 70 features might have coefficient = 0 (eliminated)
- 30 features have non-zero coefficients (selected)
- The model automatically determined which 30 features matter most

H.4 Ridge vs LASSO: When to Use Which?

Aspect	Ridge (L2)	LASSO (L1)
Penalty	Σw^2 (squared)	$\Sigma \lvert w \rvert$ (absolute)
Feature Selection	No (shrinks, never zero)	Yes (can set to exactly 0)
Correlated Features	Handles well (keeps all)	Picks one arbitrarily
Best For	Many small effects	Sparse models (few features)

Elastic Net: Best of Both Worlds

Elastic Net combines L1 and L2 penalties: it can select features (like LASSO) while handling correlated features better (like Ridge). Use when you want feature selection but have groups of correlated features.

Section I: Dimensionality Reduction (PCA)

So far, we've learned how to **select** which features to keep. But there's another approach: **create entirely NEW features** that combine the information from your original features into fewer, more powerful ones.

This is what PCA does! It's one of the most important techniques in data science.

I.1 The Problem: Too Many Correlated Features

Imagine you have a dataset with 10 features. Several problems can arise:

- **Redundancy:** Features are often correlated (height & weight, income & spending)
- **Multicollinearity:** Correlated features cause problems for regression models
- **Curse of dimensionality:** More features require exponentially more data
- **Visualization:** Can't visualize more than 3 dimensions

I.2 What is PCA? (Intuitive Explanation)

Key idea: Your 10 features are like 10 different directions in which your data can vary. Many of these directions are correlated—they "move together." This creates redundancy.

PCA finds NEW directions (called **principal components**) that:

- Capture the **maximum possible variance** (information/spread) in the data
- Are **completely uncorrelated** with each other (perpendicular)
- Usually the first 2-4 components explain **most of the story** in the data

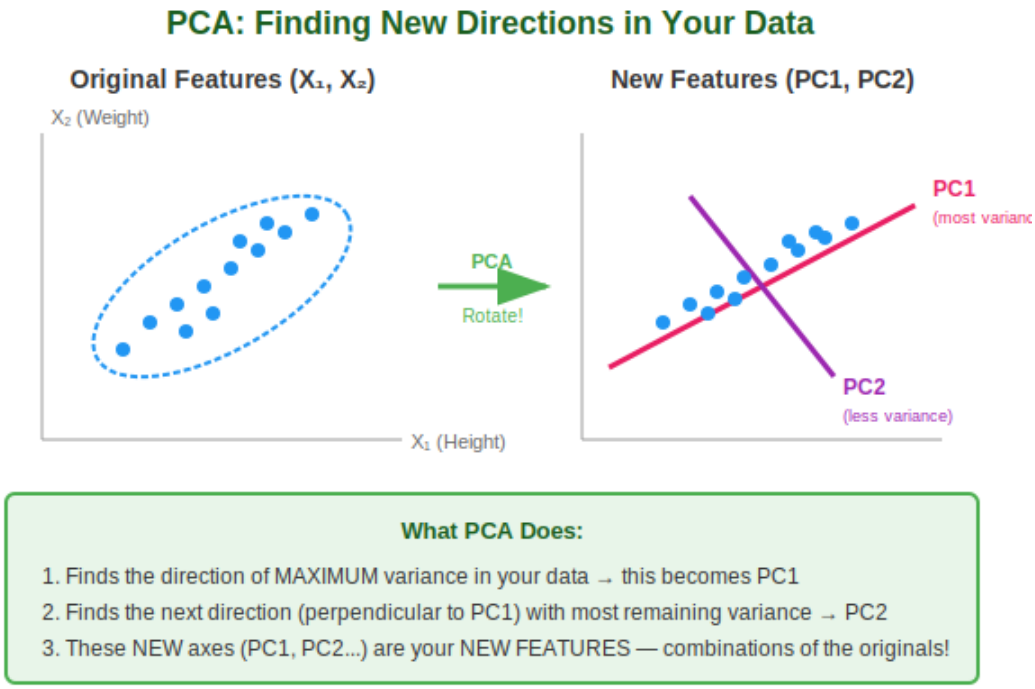


Figure 25: PCA Rotates the Coordinate System to Find Directions of Maximum Variance

The Cloud Analogy

Imagine your data points form a **cloud in 10-dimensional space**. PCA rotates the coordinate system so that: Axis 1 (PC1) goes along the **longest direction** of the cloud, Axis 2 (PC2) goes along the **second longest perpendicular direction**, and so on. Then you can keep just the first few axes and project everything onto them—a much simpler picture!

I.3 How PCA Creates New Features

This is the key insight: Each Principal Component is a **weighted combination of ALL your original features**.

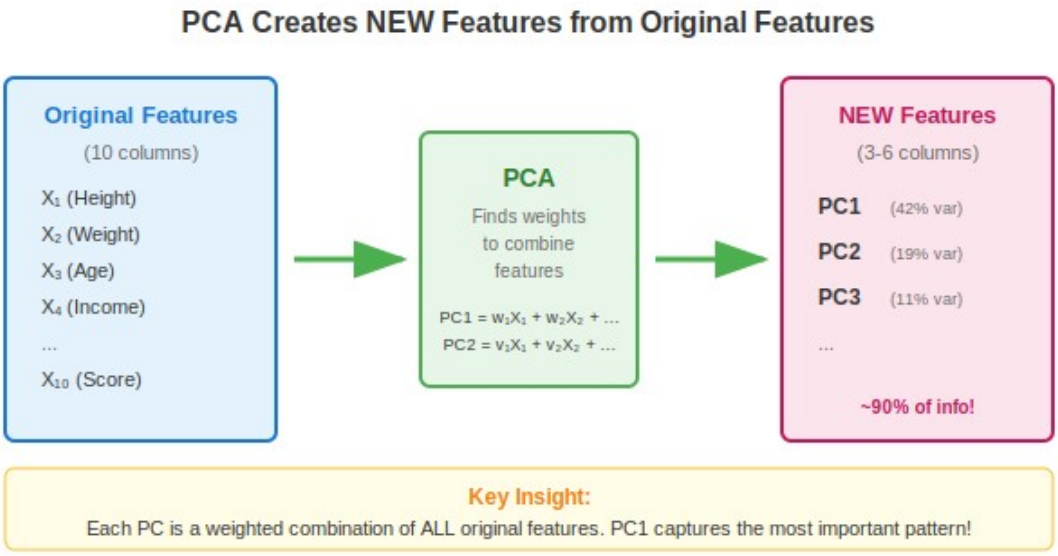


Figure 26: PCA Transforms 10 Original Features into Fewer Principal Components

The Math (Simplified)

Each principal component is calculated as:

$$PC_1 = w_{11}X_1 + w_{12}X_2 + w_{13}X_3 + \ldots + w_{1n}X_n$$

$$PC_2 = w_{21}X_1 + w_{22}X_2 + w_{23}X_3 + \ldots + w_{2n}X_n$$

Where the **weights (w)** are called **loadings**. They tell you how much each original feature contributes to each PC.

Concrete Example:

If you have features: Height, Weight, Age, Income, then PC1 might be:

$$PC_1 = 0.6 \times Height + 0.7 \times Weight + 0.1 \times Age + 0.3 \times Income$$

This PC1 is a **new feature** that captures the combined effect of all original features, with Height and Weight contributing most!

I.4 How Many Components to Keep?

PCA ranks components by how much **variance** they explain:

- **PC1:** Explains the MOST variance (e.g., 42%)
- **PC2:** Explains the second most (e.g., 19%)
- **PC3:** Explains less (e.g., 11%)
- ...and so on, always decreasing

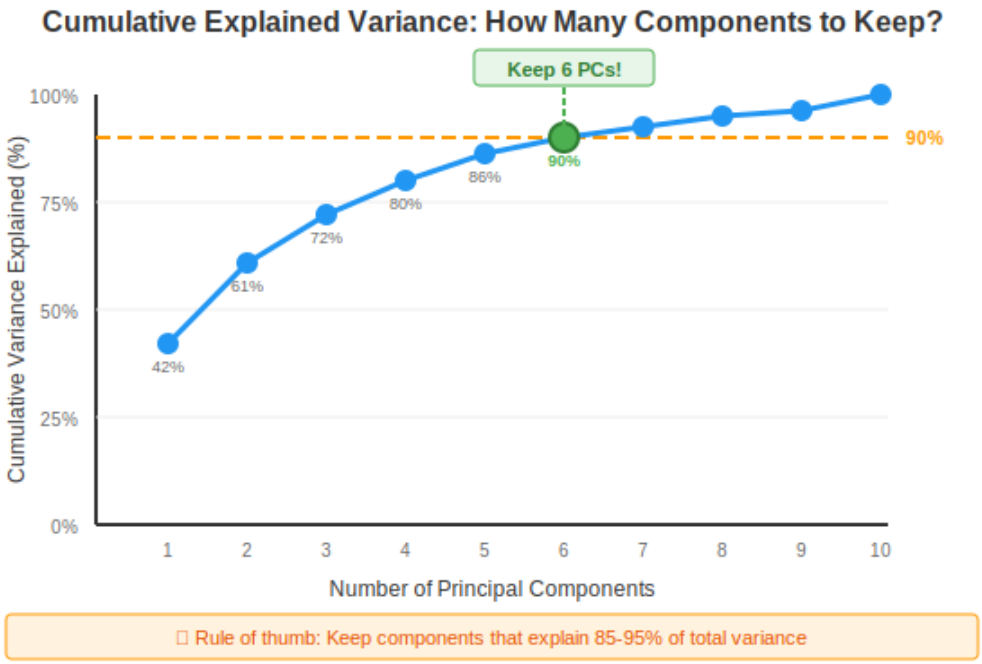


Figure 27: Cumulative Explained Variance Plot (Scree Plot)

Rule of thumb: Keep enough components to explain **85-95% of total variance**. In the example above, 6 components capture 90% of the information from the original 10 features!

I.5 When to Use PCA

Your Goal	What to Do with PCA
Visualize data	Keep 2 or 3 components → scatter plot (very powerful!)
Feed better features to ML model	Use X_pca instead of original X (fewer columns)
Remove redundancy/noise	Keep components that explain ≥ 85-95% variance
Understand the data	Look at loadings (pca.components_) to see which original features matter most in each PC

I.6 Important: Standardize Before PCA!

⚠️ **Critical:** PCA is very sensitive to the **scale** of your features!

Problem: If Income ranges from 20,000-200,000 and Age ranges from 18-80, PCA will think Income is more important just because the numbers are bigger.

Solution: Always **standardize your features first** (subtract mean, divide by standard deviation) so all features have mean=0 and std=1.

I.7 PCA Summary

- **What it does:** Creates NEW features (PCs) that are weighted combinations of original features
- **Why it works:** PCs are uncorrelated and ordered by importance (variance explained)
- **How many to keep:** Enough to explain 85-95% of variance
- **Key requirement:** Standardize features first!
- **Result:** Compress 10 features → 3-6 features while keeping ~90% of information

Section J: Summary - Choosing a Feature Selection Method

Method	Speed	Interactions	Overfitting Risk	Best For
Filter	Fast	No	Low	Large datasets, quick baseline
Wrapper	Slow	Yes	High	Best accuracy needed
RFE	Medium	Yes	Medium	Feature ranking
RFECV	Slow	Yes	Low	Robust selection

Lecture 4: Linear Regression

Deep Intuition: Understanding Linear Regression
What Type of Problem Is This?
This is a SUPERVISED REGRESSION problem. You have input features X and a continuous numerical output Y. You want to predict Y given new X values.
The Five Components:
1. Task: Predict a continuous value (regression, not classification)
2. Model: A linear function: $y = w_0 + w_1x_1 + w_2x_2 + ...$
3. Loss: Sum of Squared Errors
4. Optimization: Find weights minimizing loss (closed-form or gradient descent)
5. Evaluation: Performance on NEW unseen data (R^2, RMSE)
Why Squared Errors?
Positive and negative errors cancel if just summed. Squaring also penalizes large errors MORE - wrong by 10 is FOUR times as bad as wrong by 5.
One Sentence Mastery: "Linear regression finds weights minimizing squared errors, assuming the true relationship is approximately linear."
[Supervised Learning — Regression]

1. What is Linear Regression?

The Big Picture: Linear regression is a method for predicting a continuous numerical value (like price, temperature, or sales) based on one or more input variables. It's one of the most fundamental and widely-used techniques in statistics and machine learning.

Real-World Examples:

- Predicting house prices based on size, location, bedrooms
- Forecasting sales based on advertising spend
- Estimating salary based on years of experience
- Predicting economic growth based on various indicators

1.1 Simple vs Multiple Linear Regression

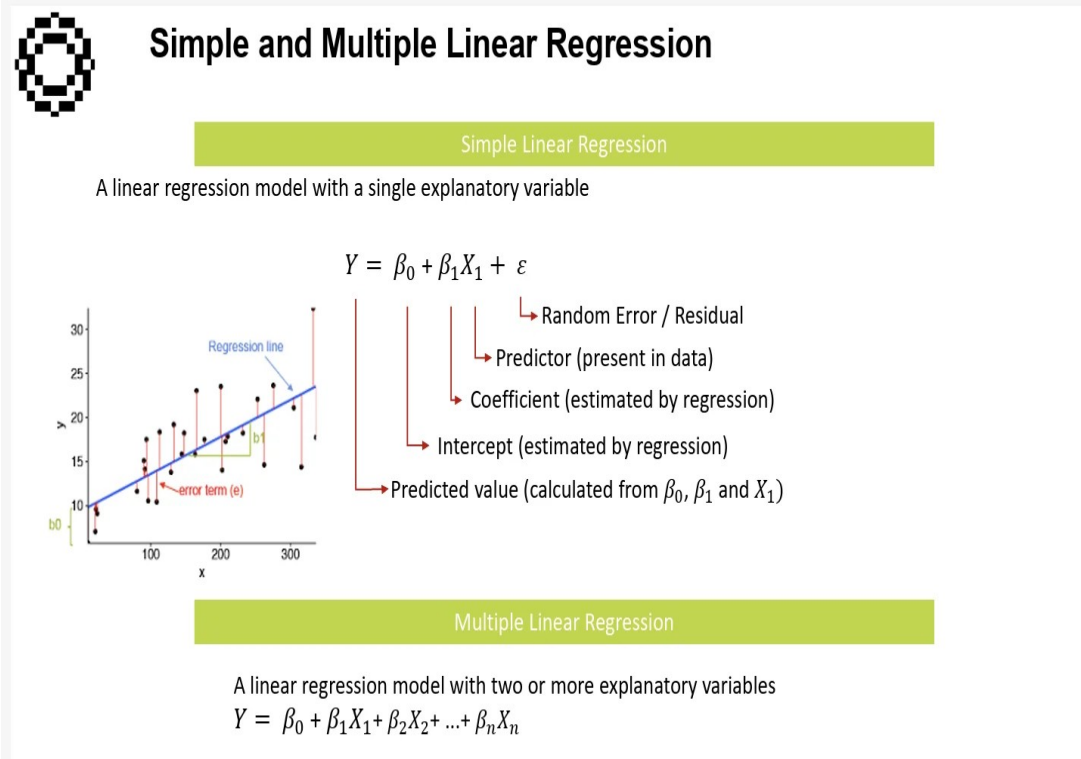


Figure 11: Simple and Multiple Linear Regression

Simple Linear Regression (One Predictor):

Formula: $Y = \beta_0 + \beta_1 X_1 + \varepsilon$

Let's break down each part:

- **Y:** The predicted value (what we're trying to predict)
- **β_0 (beta-zero):** The intercept — the value of Y when X = 0
- **β_1 (beta-one):** The coefficient/slope — how much Y changes when X increases by 1
- **X_1:** The predictor variable (what we use to make predictions)
- **ε (epsilon):** The error/residual — the difference between predicted and actual values

Multiple Linear Regression (Multiple Predictors):

Formula: $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_n X_n$

Same idea, but now we have multiple X variables, each with its own coefficient.

1.2 Concrete Example: House Prices

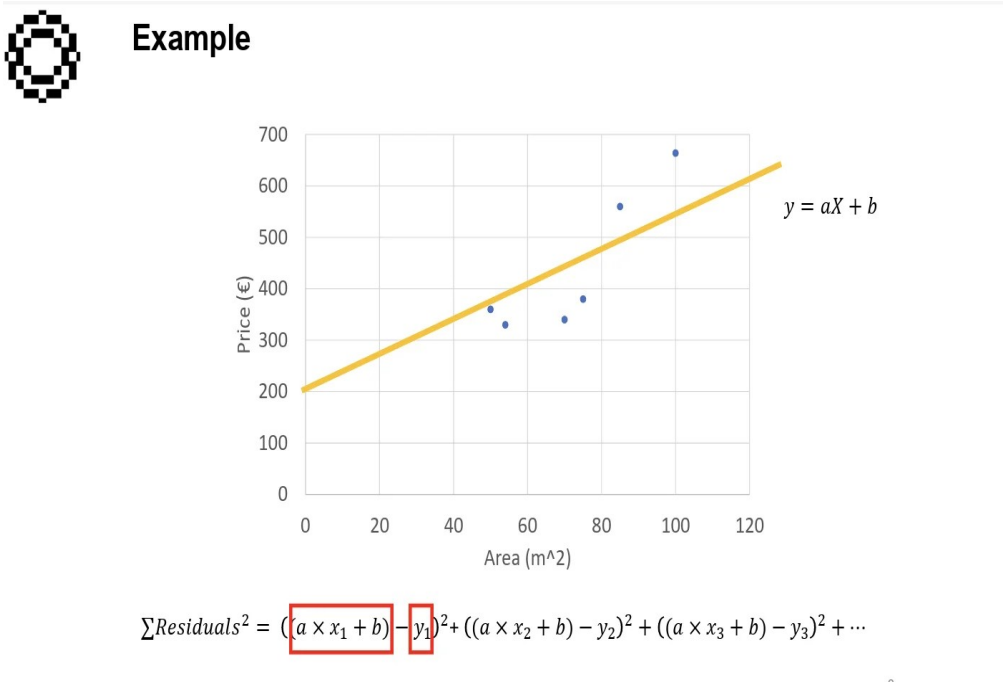


Figure 12: Predicting House Price from Area

The Question: Can we predict house price (€) from its size (m²)?

The Model: Price = a × Area + b

How it works: We draw a line that best fits the data points. The line gives us our predictions.

What Are Residuals?

A **residual** is the difference between what the model predicted and what actually happened:

Residual = Actual Value − Predicted Value

In the graph, residuals are shown as vertical lines from each data point to the regression line. We want these to be as small as possible.

1.3 How Do We Find the Best Line? (Least Squares)

The Goal: Find the line that minimizes the total error (residuals).

Why squared? We square the residuals because:

- It makes all errors positive (negative errors would cancel out positive ones)
- It penalizes large errors more than small ones
- It has nice mathematical properties for optimization

The Formula:

$\Sigma Residuals^2 = (predicted_1 - actual_1)^2 + (predicted_2 - actual_2)^2 + ...$

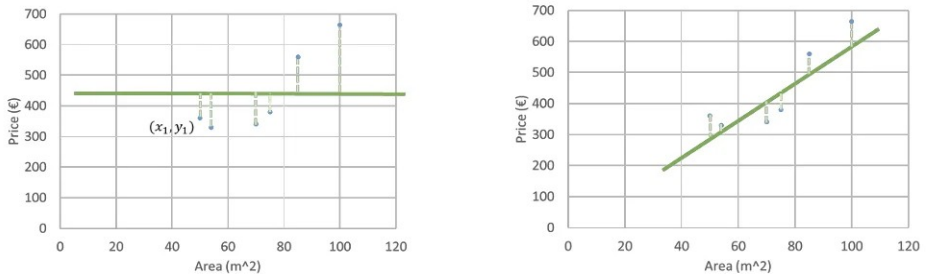
The "least squares" method finds the values of β_0 and β_1 that make this sum as small as possible.

2. Understanding R² (R-Squared)

The Big Question: "How good is our model?" R² gives us a way to answer this.

2.1 What Does R² Actually Mean?

$$R^2$$



- There is less variation around the LS line compared to the raw variation of prices
- So, we can say that some of the variation in the prices is "explained" by taking house size into account
  - Bigger houses are more expensive and smaller houses are cheaper
- R^2 **tells us how much of the variation** in the price can be explained by taking its size into account

Figure 13: R² - Comparing Variance Around Mean vs Around Regression Line

R² tells us: "What percentage of the variation in Y is explained by our model?"

Think of it this way:

Imagine you're trying to predict house prices. Without any information, your best guess would be the **average price** (the horizontal line in the left graph). All houses would get the same prediction, and you'd have a lot of error.

Now, if you know the house SIZE, you can make better predictions using the **regression line** (the diagonal line in the right graph). The errors are smaller because bigger houses get higher predictions.

R² measures how much better the regression line is compared to just using the average.

2.2 How to Calculate R²

- R^2 **tells us how much of the variation** in the price can be explained by taking its size into account

$$R^2 = \frac{Var(mean) - Var(fit)}{Var(mean)} \quad \text{or} \quad R^2 = \frac{SS(mean) - SS(fit)}{SS(mean)}$$

$$SS(mean) = 16111.67$$
$$SS(fit) = 3478.1$$
$$R^2 = 0.784$$

So, 78% of the price of the houses can be explained by its size

Figure 14: R² Calculation with House Price Example

The Formula:

R² = (SS(mean) - SS(fit)) / SS(mean)

Or equivalently: **R² = 1 - (SS(fit) / SS(mean))**

Let's break this down:

- **SS(mean) = Total Sum of Squares:** How much the actual values vary from the mean (average)
- **SS(fit) = Residual Sum of Squares:** How much the actual values vary from the regression line predictions
- **SS(mean) - SS(fit) = Explained Variation:** How much better the model is than just using the mean

Concrete Example from the Slides:

- SS(mean) = 16,111.67 (total variance around the average price)
- SS(fit) = 3,478.1 (remaining variance around the regression line)
- **R² = (16,111.67 - 3,478.1) / 16,111.67 = 0.784**
- **Interpretation:** 78.4% of the variation in house prices can be explained by house size!

2.3 Interpreting R² Values

R² Value	Meaning	Interpretation
R² = 0	0% explained	Model is no better than guessing

		the mean
$R^2 = 0.5$	50% explained	Half of variation explained by model
$R^2 = 0.78$	78% explained	Good! Most variation captured
$R^2 = 1$	100% explained	Perfect fit (rare, may be overfitting!)

Important: A "good" R^2 depends on the field. In physics, $R^2 > 0.99$ might be expected. In social sciences, $R^2 = 0.3$ might be excellent because human behavior is inherently unpredictable.

2.4 The Problem with R^2: Why We Need Adjusted R^2

The Problem: R^2 **always increases** (or stays the same) when you add more variables, even if those variables are useless!

Why? Adding any variable gives the model more flexibility to fit the training data, even if it's just fitting noise.

Example:

Suppose you're predicting house prices with: Size, Bedrooms, Location. $R^2 = 0.78$

Now add a completely random variable (like "day of the week you signed the contract")

R^2 might go up to 0.79! But that extra 1% is fake—it won't help on new data.

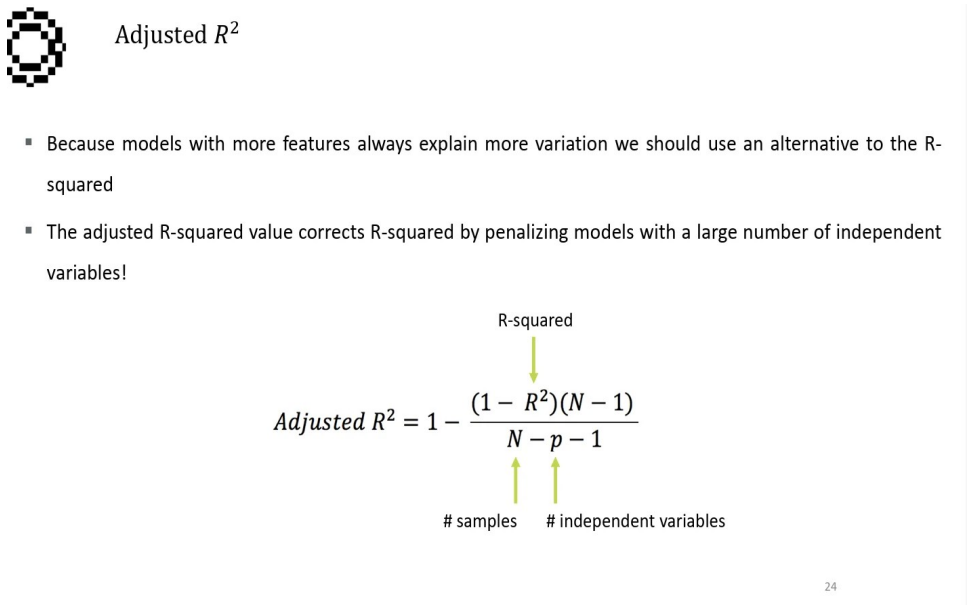


Figure 15: Adjusted R^2 Formula

The Solution: Adjusted R^2

Adjusted R^2 **penalizes** models for having too many variables. It only increases if the new variable improves the model more than expected by chance.

Formula components:

- **N:** Number of samples (data points)
- **p:** Number of independent variables (features)
- **Key insight:** As p increases, the penalty increases, so Adjusted R^2 can actually DECREASE if you add useless variables.

Rule of thumb: Always report Adjusted R^2 when comparing models with different numbers of variables.

3. Understanding P-Values and Statistical Significance

This is where many students get confused, so let's build the intuition step by step.

3.1 The Core Question: Is This Variable Actually Useful?

When we fit a regression model, each variable gets a coefficient (β). But here's the problem: **is that coefficient real, or just random noise?**

Example:

Suppose you're predicting salary, and your model says:

$$\text{Salary} = 30{,}000 + 2{,}500 \times \text{YearsExperience} + 50 \times \text{ShoeSize}$$

The coefficient for ShoeSize is 50. Does shoe size *really* affect salary? Or did this just happen by random chance in our data?

The p-value helps us answer this question.

3.2 What is a P-Value? (Plain English)

Definition: The p-value is the probability of seeing a coefficient this large (or larger) **if the variable actually had NO effect**.

Think of it like a court trial:

- **Null Hypothesis (H_0):** "The defendant is innocent" = "This variable has no effect (coefficient = 0)"
- **Evidence:** The coefficient we calculated from our data
- **P-value:** "If the defendant were innocent, what's the probability we'd see this much evidence against them?"

Interpreting P-Values:

- **Low p-value (< 0.05):** "It's very unlikely we'd see this coefficient by chance. The variable probably has a real effect."
- **High p-value (> 0.05):** "We could easily see this coefficient just by random chance. We can't conclude the variable has an effect."

Key insight: A low p-value means we **reject** the null hypothesis (we believe the variable matters). A high p-value means we **fail to reject** it (not enough evidence).

3.3 How P-Values Are Calculated: Standard Error and t-Value

To get a p-value, we first need two things:

Standard Error of the Coefficient:

What it measures: How **uncertain** we are about our coefficient estimate. A large standard error means our estimate is imprecise.

Analogy: If you measure your height once, you get a number. If you measure it 100 times, you'd get slightly different numbers each time. The standard error is like the spread of those measurements.

t-Value (t-statistic):

$$\textbf{t-value} = \textbf{Coefficient} / \textbf{Standard Error}$$

What it means: How many standard errors away from zero is our coefficient?

- **Large |t-value|:** Coefficient is far from zero relative to its uncertainty → probably real
- **Small |t-value|:** Coefficient is close to zero relative to its uncertainty → might be noise

Rule of thumb: |t-value| > 1.96 typically corresponds to p-value < 0.05 (statistically significant at 95% confidence level).

3.4 Summary: Model Evaluation Statistics

Statistic	What You Want
R-squared	Higher is better (0 to 1)
Adjusted R^2	Higher is better (use when comparing models)
Std. Error	Closer to zero is better (more precise estimate)
t-statistic	|t| > 1.96 for p < 0.05 (statistically significant)
p-value	p < 0.05 means variable is statistically significant

3.5 Reading Regression Output (Practical Example)

Here's how to interpret typical regression output:

Example output for predicting medical costs:

Variable	Coefficient	t-value	p-value
age	256.8	21.59	**< 0.001** ✓
sex_male	-131.3	-0.40	**0.693** ✗
bmi	339.3	11.86	**< 0.001** ✓
smoker_yes	23847.5	57.72	**< 0.001** ✓
region_northwest	-352.8	-0.74	**0.459** ✗

How to read this:

- **age (p < 0.001):** Highly significant! Each year of age adds $256.80 to medical costs.
- **bmi (p < 0.001):** Highly significant! Each BMI point adds $339.30 to costs.
- **smoker_yes (p < 0.001):** Highly significant! Smokers pay $23,847.50 more on average.
- **sex_male (p = 0.693):** NOT significant! We can't conclude sex affects medical costs.
- **region_northwest (p = 0.459):** NOT significant! This region isn't different from the baseline.

4. Key Takeaways

30. **R^2** tells you what % of variation your model explains. Higher is better, but context matters.
31. **Adjusted R^2** penalizes extra variables. Use this when comparing models with different numbers of features.
32. **P-value** tells you if a variable's effect is real or just noise. $p < 0.05$ is typically considered significant.
33. **t-value** = coefficient / standard error. $|t| > 1.96$ usually means $p < 0.05$.
34. **Standard Error** measures uncertainty in coefficient estimates. Smaller is better.
35. **Coefficients** tell you the direction and magnitude of each variable's effect.

5. Logistic Regression

What if we want to predict a **category** instead of a number? That's where logistic regression comes in.

5.1 Linear vs Logistic Regression: Key Differences

The fundamental difference: Linear regression predicts continuous values, logistic regression predicts probabilities of belonging to a class.

Aspect	Linear Regression	Logistic Regression
Outcome	Continuous (price, weight)	Categorical (yes/no, pass/fail)
Output Range	-∞ to +∞	0 to 1 (probability)
Estimation	OLS (minimize squared errors)	MLE (maximize likelihood)
Error Distribution	Normal (Gaussian)	Not normal (binomial)
Interpretation	Direct: β = change in Y	Indirect: β = change in log-odds

5.2 Why Can't We Use Linear Regression for Classification?

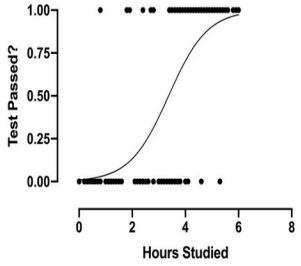
The Problem: Imagine we want to predict whether a student passes (1) or fails (0) based on hours studied.

Linear vs. Logistic Regression
Qualitative Target

Let's consider we want to estimate a model where the dependent variable is limited to a binary response

$$y_i = \begin{cases} 1, if\ student\ passed\ the\ test \\ 0, otherwise \end{cases}$$

If we apply a scatter diagram to this dataset, we are going to obtain a completely distinct visualization from the ones where the response is continuous!



Reflects that the dependent variable is limited to two possible outcomes

32

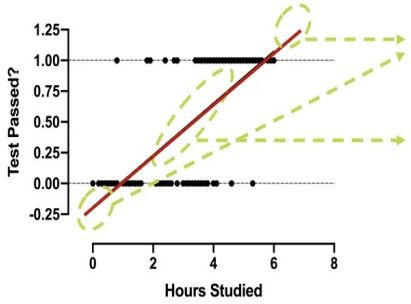
Figure 16: Binary Classification - Pass/Fail Based on Hours Studied

Notice that the outcome (y) is limited to just **two possible values: 0 or 1**. The data points cluster at the top (passed) and bottom (failed).

What happens if we fit a straight line?

Linear vs. Logistic Regression
Qualitative Target

How to determine the best fit model through data where the dependent variable is limited to a set of outcomes?



Predictions above 1 and bellow 0 don't make sense

OLS performs well in average observations, but it does a poor job of "fitting" or "describing" the data points

OLS is not an appropriated estimation method if our target is qualitative

Main drawbacks of using linear models in these cases:

1. Having probabilities above 1 or bellow 0, which are impossible outcomes
2. Assuming that the probability changes linearly with the explanatory variables

Figure 17: Why Linear Regression Fails for Classification

Two Major Problems:

36. **Impossible predictions:** Linear regression can predict values above 1 or below 0, which don't make sense as probabilities!
37. **Wrong assumption:** It assumes probability changes linearly with X, but real probability often follows an S-shaped curve.

Conclusion: OLS is NOT appropriate when our target variable is qualitative (categorical). We need a different approach!

5.3 The Solution: The Logistic (Sigmoid) Function

Key idea: Instead of predicting Y directly, we predict the **probability** that Y = 1, and we use a special function that keeps predictions between 0 and 1.

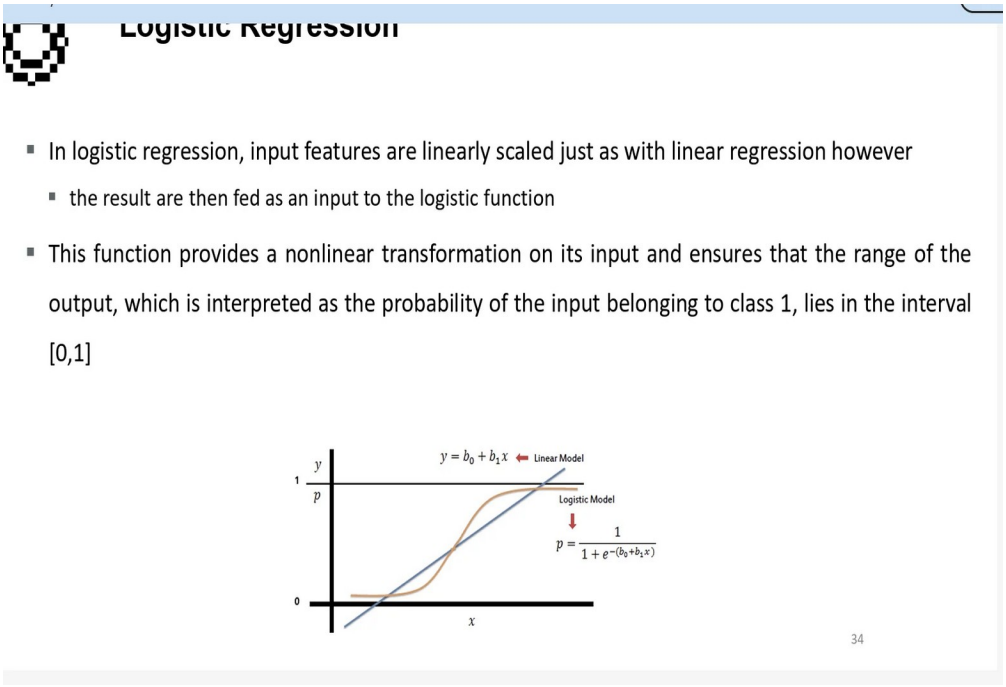


Figure 18: The Logistic Function - From Linear to S-Shaped

How it works:

38. **Start with linear combination:** $y = b_0 + b_1x$ (just like linear regression)
39. **Feed into logistic function:** $p = 1 / (1 + e^{(-(b_0 + b_1x))})$
40. **Output is probability:** The result is always between 0 and 1

The S-shaped curve: Notice how the blue line (linear) extends beyond 0 and 1, while the orange curve (logistic) stays within the valid probability range [0, 1].

Key insight: The logistic function performs a **nonlinear transformation** that "squashes" any input into the range [0, 1], making it perfect for probabilities.

5.4 Understanding Odds and Log-Odds (Logit)

To really understand logistic regression, we need to understand **odds** and **log-odds**.

What are Odds?

Probability: "What's the chance of success?" → p

Odds: "How many successes for each failure?" → p / (1-p)

Example:

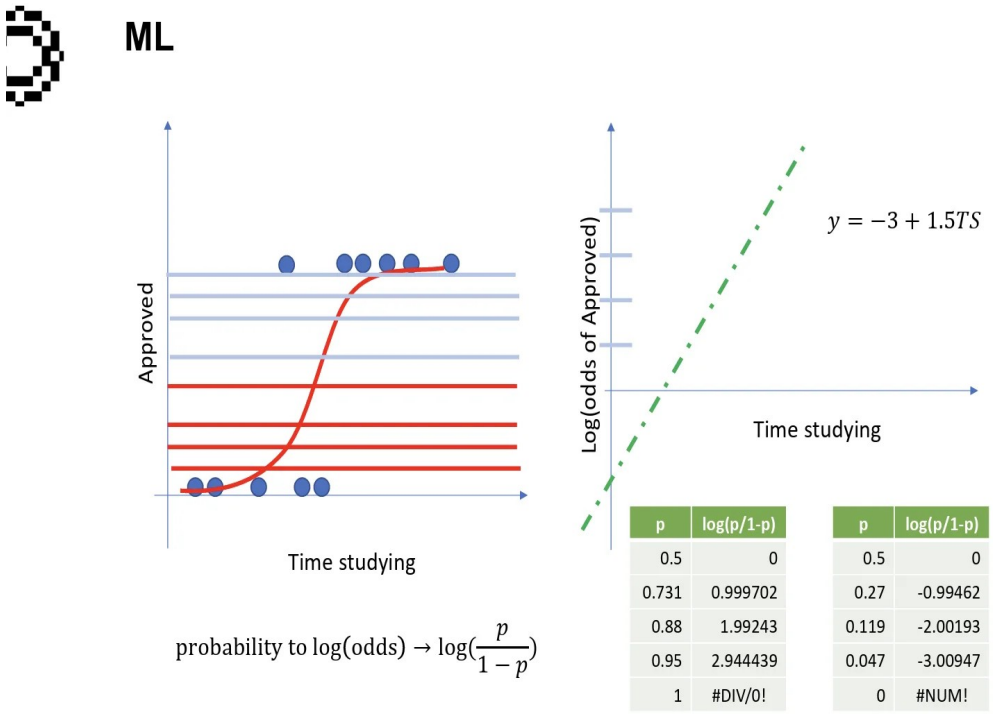
- If p = 0.5 (50% chance), odds = 0.5/0.5 = 1 ("even odds")
- If p = 0.75 (75% chance), odds = 0.75/0.25 = 3 ("3 to 1 odds")
- If p = 0.9 (90% chance), odds = 0.9/0.1 = 9 ("9 to 1 odds")

Why Log-Odds?

The problem with odds is they're bounded: odds go from 0 to infinity. But we want something that can be any real number (like the output of a linear equation).

Solution: Take the logarithm! Log-odds (also called **logit**) can range from -∞ to +∞.

$$\textbf{log(odds) = log(p / (1-p)) = } \beta_0 + \beta_1 X$$



p	log(p/1-p)
0.5	0
0.731	0.999702
0.88	1.99243
0.95	2.944439
1	#DIV/0!

p	log(p/1-p)
0.5	0
0.27	-0.99462
0.119	-2.00193
0.047	-3.00947
0	#NUM!

Figure 19: Transforming Probability to Log-Odds

The transformation: The left graph shows the S-shaped probability curve. The right graph shows that when we transform to log-odds, we get a **straight line**!

This is the magic of logistic regression: We can use linear regression on the log-odds scale, then convert back to probabilities.

5.5 From Log-Odds Back to Probability

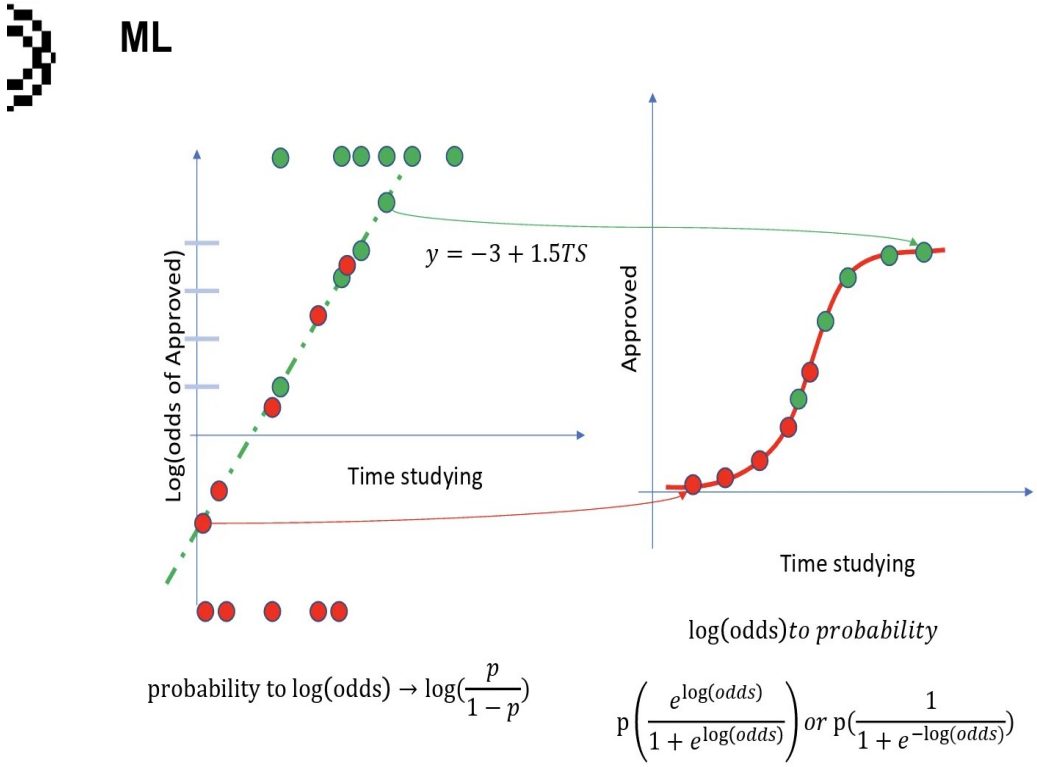


Figure 20: Converting Log-Odds Back to Probability

The Complete Process:

41. **Fit a linear model:** log(odds) = $\beta_0 + \beta_1 X$ (e.g., y = -3 + 1.5 × TimeStudying)
42. **This gives log-odds:** A straight line in the right graph
43. **Convert to probability:** p = e^(log-odds) / (1 + e^(log-odds))
44. **Get the S-curve:** The red S-shaped curve in the left graph

5.6 Maximum Likelihood Estimation (MLE)

The big question: How do we find the best coefficients (β_0, β_1, etc.) for our logistic model?

In linear regression: We minimize the sum of squared errors (OLS)

In logistic regression: We use **Maximum Likelihood Estimation (MLE)**

What is MLE? (Plain English)

Goal: Find the parameters that make our observed data **most likely** to have occurred.

Intuitive Example:

Suppose we have 10 students: 7 who studied 5+ hours all passed, 3 who studied < 2 hours all failed.

- **Bad model:** Predicts everyone has 50% chance of passing → This data would be very unlikely
- **Good model:** Predicts high probability for 5+ hours, low for <2 hours → This data would be very likely

MLE finds the model that makes our observed data most probable.

Key Points about MLE:

- **No closed-form solution:** Unlike OLS, there's no simple formula. We need iterative algorithms.
- **Uses log-likelihood:** It's easier to maximize the log of the likelihood
- **Computer-based:** The algorithm starts with arbitrary values and iterates until convergence

5.7 Interpreting Logistic Regression Coefficients

Warning: Interpretation is NOT as straightforward as linear regression!

What the coefficients mean:

- **Sign of β_k:** Positive = probability increases as X increases; Negative = probability decreases
- **Magnitude $|\beta_k|$:** Larger values = steeper S-curve = faster change in probability
- **Exact interpretation:** A 1-unit increase in X changes the log-odds by β_k (not the probability!)

Odds Ratio Interpretation:

To get a more intuitive interpretation, we can exponentiate the coefficient:

$$\text{Odds Ratio} = e^{\beta_k}$$

Meaning: For each 1-unit increase in X, the odds are multiplied by e^{β_k}

Example:

If $\beta = 0.5$ for "hours studied":

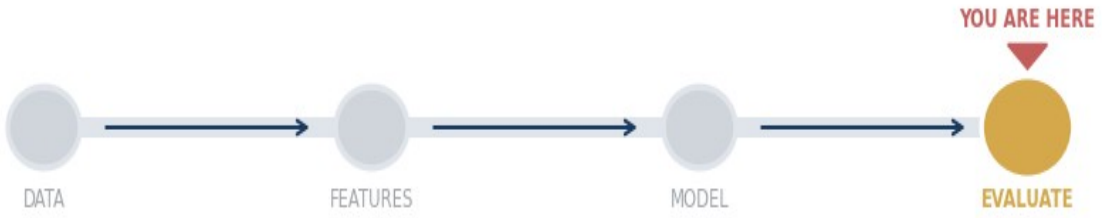
- Odds ratio = $e^{0.5} \approx 1.65$
- **Interpretation:** Each additional hour of studying multiplies the odds of passing by 1.65 (or increases odds by 65%)

5.8 Summary: Linear vs Logistic Regression

Use...	When...
Linear Regression	Predicting continuous values (price, temperature, sales)
Logistic Regression	Predicting categories or probabilities (yes/no, pass/fail, churn/stay)

Lecture 5: Model Selection

Deep Intuition: The Real Goal of Machine Learning
The Fundamental Misunderstanding
Most beginners think: "Fit training data as well as possible." WRONG. A lookup table would be best then. We reject such models because...
The Real Goal
MINIMIZE GENERALIZATION ERROR - performance on NEW, UNSEEN data. Training error is just a proxy.
Why Training Error Misleads
Training data = signal + noise. Complex models fit BOTH, memorizing noise. This is OVERFITTING.
Bias-Variance Tradeoff
Simple models: high bias, low variance. Complex models: low bias, high variance. Sweet spot minimizes TOTAL error.
One Sentence Mastery: "Model selection finds the right complexity - simple enough to avoid noise, complex enough to capture patterns."



1. The Problem of Overfitting

The core question: How do we know if our model will work well on **new, unseen data**?

1.1 What is Overfitting?

Overfitting occurs when a model learns the training data **too well**—it memorizes the noise and peculiarities of the training set instead of learning the general patterns.

Signs of Overfitting:

- **Excellent performance on training data**
- **Poor performance on new data**
- The model is too complex for the amount of data available

Analogy: Imagine a student who memorizes the exact answers to practice problems instead of understanding the concepts. They'll ace practice tests but fail when the real exam has different questions.

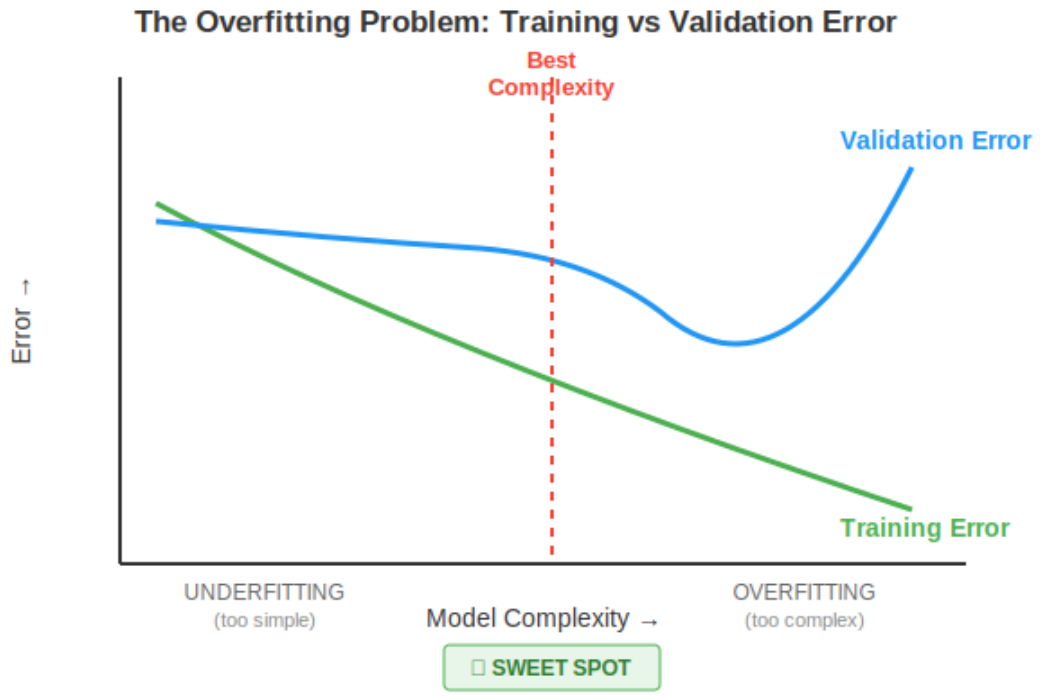


Figure 21: The Overfitting Problem - Training vs Validation Error

Understanding the Graph:

- **Training Error (green):** Always decreases as model complexity increases—the model fits training data better and better
- **Validation Error (blue):** First decreases, then INCREASES—the model starts fitting noise instead of patterns
- **Best Complexity:** The point where validation error is minimum—not too simple, not too complex
- **The gap:** When training error is much lower than validation error, you're overfitting!

2. The Solution: Keep Some Data Aside

Key insight: To know how well our model generalizes, we need to test it on data it has **never seen before**.

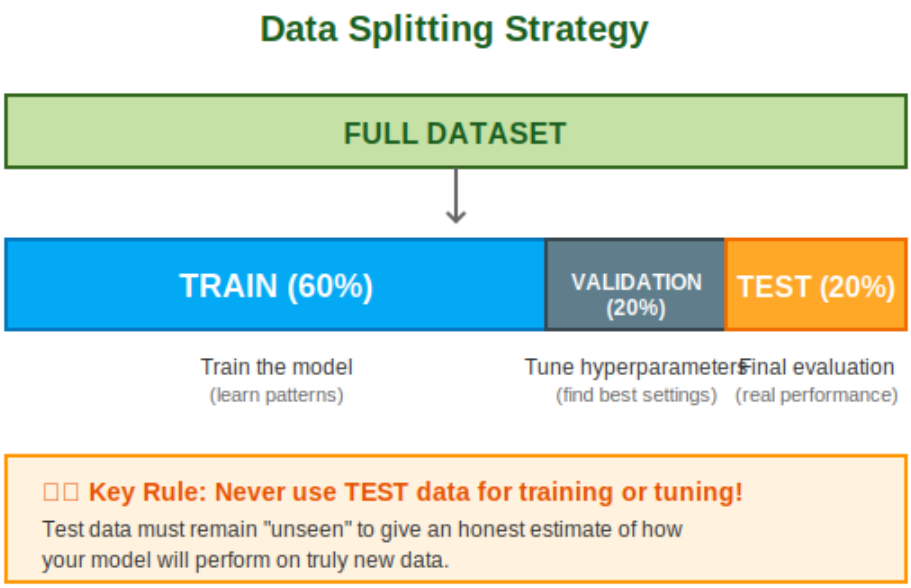


Figure 22: Train / Validation / Test Split Strategy

2.1 The Three Data Sets

Training Set (~60-70%)

- **Purpose:** Train the model (learn the patterns)
- **The model sees this data and adjusts its parameters**
- Performance here will be optimistic (potentially overfit)

Validation Set (~15-20%)

- **Purpose:** Tune hyperparameters and choose the best model
- **Used to compare different models or settings**
- Helps detect overfitting during development
- Can be used multiple times (but becomes less "independent" each time)

Test Set (~15-20%)

- **Purpose:** Final, unbiased estimate of model performance
- **ONLY used ONCE at the very end!**
- Simulates how the model will perform in the real world
- **Never use for training or tuning decisions**

⚠️ **Critical Rule:** If you use test data to make ANY decisions about your model (which features to use, which hyperparameters, etc.), you've **"leaked" information** and your performance estimate will be too optimistic.

3. K-Fold Cross Validation

The problem with hold-out: A single train/validation split might be "lucky" or "unlucky." Different splits can give very different results!

The solution: Use **multiple splits** and average the results. This is called **K-Fold Cross Validation**.

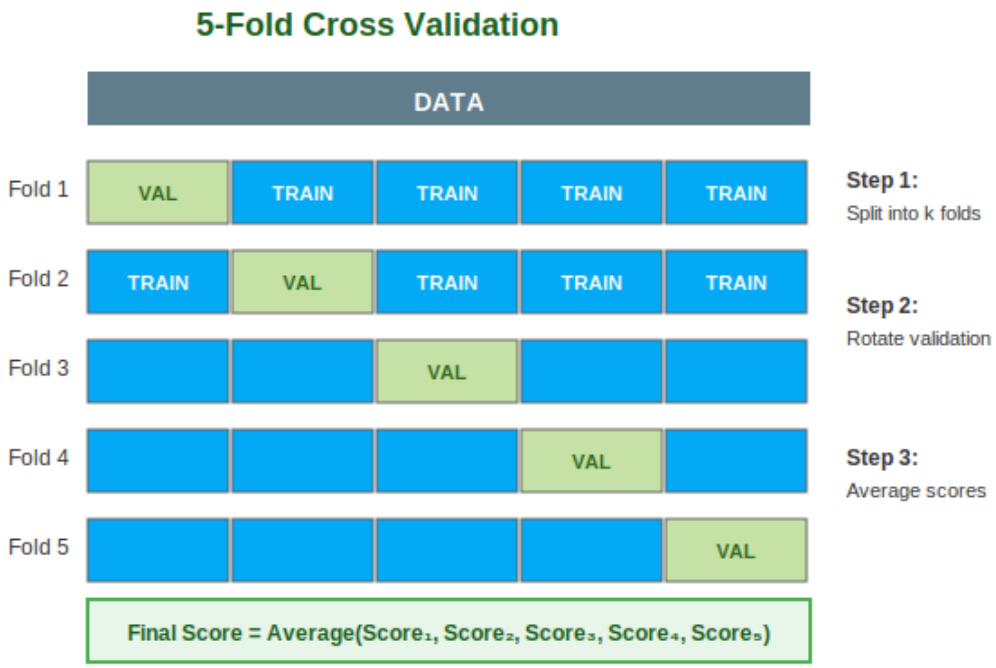


Figure 23: 5-Fold Cross Validation

3.1 How K-Fold CV Works

45. **Split the data into k equal parts ("folds")** — typically k=5 or k=10
46. **For each fold:**
    - Use that fold as validation data
    - Use the remaining k-1 folds as training data
    - Train the model and record the validation score
47. **Average all k scores** to get the final performance estimate

Why This Works:

- **Every data point gets used for validation exactly once**
- Every data point gets used for training k-1 times
- The average reduces variance from "lucky" or "unlucky" splits
- You also get the **standard deviation** of scores, showing how stable your model is

4. Comparing Two Models

When you have two candidate models, how do you decide which is better?

4.1 The 5-Step Process

48. **Choose your metric:** Accuracy? F1-score? RMSE? (depends on your problem)
49. **Choose an evaluation method:** Hold-out, K-fold CV, etc.
50. **Run the evaluation for both models** and collect metrics

51. **Compare metrics:** Look at both mean and standard deviation
52. **Pick the model with the best performance**

But wait! How to be sure one is ALWAYS better?

Problem: Using different data splits will lead to slightly different results. Model A might win on one split, Model B on another.

Solution: Cross-validation gives you multiple scores. Calculate the **mean AND standard deviation**. If the means are close but standard deviations are large, the difference might not be meaningful!

5. Choosing the Right Validation Strategy

The best validation method depends on your **dataset size**:

Dataset Size	Recommended Method	Why?
Large (>10,000)	Hold-out (Train/Val/Test)	Enough data for reliable single-split estimate
Medium (1,000-10,000)	K-Fold CV (k=5 or 10)	More stable estimates, uses data efficiently
Small (<1,000)	Leave-One-Out (k=n)	Maximizes training data, tests every point

5.1 Leave-One-Out Cross Validation (LOOCV)

Special case: When k = n (number of samples). Each "fold" is exactly one data point.

- **Advantage:** Maximizes training data (n-1 samples used for training each time)
- **Disadvantage:** Very computationally expensive (must train n models)
- **Best for:** Small datasets where every data point is precious

5.2 Stratified Sampling

Important for classification: Make sure each fold has the same proportion of each class as the full dataset.

Why it matters:

Imagine you have 90% class A and 10% class B. Without stratification, one fold might accidentally have 0% class B—your model wouldn't learn to recognize class B at all!

Rule: Always use stratified sampling for classification problems.

6. Key Takeaways for Model Selection

53. **Never evaluate on training data** — always use held-out data for performance estimates
54. **Use test sets and hold-out** for large datasets (>10,000 samples)
55. **Use K-fold cross-validation** for medium-sized datasets (more reliable estimates)
56. **Use leave-one-out** for small datasets (maximize training data)

57. **Always use stratified sampling** for classification to maintain class proportions
58. **Don't use test data for parameter tuning** — use separate validation data
59. **Report both mean AND standard deviation** when comparing models

Lecture 5 (Part 2): Model Assessment

1. What are Evaluation Metrics?

Definition: An evaluation metric **quantifies the performance** of a predictive model. It tells us "how good" our model is.

Different metrics for different problems:

- **Classification:** Accuracy, Error Rate, Precision, Recall, F1, ROC-AUC...
- **Regression:** MAE, MSE, RMSE, R^2, Adjusted R^2...

2. Metrics for Classification

2.1 The Confusion Matrix

Everything starts with the confusion matrix. In classification, predictions are either correct or wrong. The confusion matrix organizes all possible outcomes:

		True Class	
		Positive	**Negative**
Predicted Class	**Positive**	**TP** (True Positive)	**FP** (False Positive)
	Negative	**FN** (False Negative)	**TN** (True Negative)

Understanding Each Cell:

- **True Positive (TP):** Predicted positive, actually positive ✓
- **True Negative (TN):** Predicted negative, actually negative ✓
- **False Positive (FP):** Predicted positive, actually negative ✗ — *Type I Error*
- **False Negative (FN):** Predicted negative, actually positive ✗ — *Type II Error*

Concrete Example: Goat Detection

Suppose we built a model to identify goats in images:

		True Class	
		Goat	**Not Goat**
Predicted Class	**Goat**	**30**	**4**
	Not Goat	**6**	**20**

Total samples = 30 + 4 + 6 + 20 = 60. Now let's calculate various metrics.

2.2 Accuracy

Definition: Proportion of **all events correctly identified** (both positive and negative).

$$\textbf{Accuracy} = (TP + TN) / (TP + TN + FP + FN)$$

Example: Accuracy = (30 + 20) / (30 + 20 + 4 + 6) = 50/60 = **0.833 (83.3%)**

⚠️ **Limitation:** Accuracy can be misleading with imbalanced datasets! If 95% of emails are not spam, a model that predicts "not spam" for everything gets 95% accuracy but is useless.

2.3 Error Rate (Misclassification Rate)

Definition: Proportion of all events **incorrectly identified**. Simply 1 - Accuracy.

$$\textbf{Error Rate} = (FP + FN) / (TP + TN + FP + FN)$$

Example: Error Rate = (4 + 6) / 60 = 10/60 = **0.167 (16.7%)**

Interpretation: The lower the error rate, the better!

2.4 Precision

Question: "When the model predicts positive, how often is it correct?"

$$\textbf{Precision} = TP / (TP + FP)$$

Example: Precision = 30 / (30 + 4) = 30/34 = **0.882 (88.2%)**

When to use Precision: When the **cost of False Positives is high**.

Example: Email spam detection. A false positive means a legitimate email gets sent to spam — the user might miss important messages!

2.5 Recall (Sensitivity / True Positive Rate)

Question: "Of all actual positives, how many did the model catch?"

$$\textbf{Recall} = TP / (TP + FN)$$

Example: Recall = 30 / (30 + 6) = 30/36 = **0.833 (83.3%)**

When to use Recall: When the **cost of False Negatives is high**.

Example: Disease detection. A false negative means a sick patient is told they're healthy — they won't get treatment!

2.6 Specificity (True Negative Rate)

Question: "Of all actual negatives, how many did the model correctly identify?"

$$\textbf{Specificity} = TN / (TN + FP)$$

Example: Specificity = 20 / (20 + 4) = 20/24 = **0.833 (83.3%)**

Think of it as: "Recall for the negative class."

2.7 F1 Score

The problem: Precision and recall often trade off against each other. How do we balance them?

$$F1 = 2 \times (Precision \times Recall) / (Precision + Recall)$$

The harmonic mean: F1 is the harmonic mean of precision and recall. It's high only when BOTH precision and recall are high.

Example: F1 = 2 × (0.882 × 0.833) / (0.882 + 0.833) = **0.857**

When to use F1: When you need a **balance between precision and recall**, especially with imbalanced datasets.

2.8 Which Metric to Choose?

Metric	Use When...	Example
Precision	Cost of False Positive is high	Spam detection
Recall	Cost of False Negative is high	Disease detection
F1 Score	Balance between Precision & Recall	Imbalanced datasets
Accuracy	FP & FN costs are equal, balanced data	General classification

2.9 Classification Threshold (Cutoff)

How classification works: Most algorithms predict a **probability**, then convert it to a class using a threshold.

60. Compute probability of belonging to class "1"
61. Compare to cutoff value, classify accordingly

Default cutoff = 0.5:

- If probability ≥ 0.5 → classify as "1" (positive)
- If probability < 0.5 → classify as "0" (negative)

Key insight: You can adjust the cutoff to optimize for precision or recall! Lower cutoff = more positives (higher recall, lower precision). Higher cutoff = fewer positives (higher precision, lower recall).

2.10 ROC Curve and AUC

ROC Curve: Plots True Positive Rate (Recall) vs False Positive Rate at different thresholds.

AUC (Area Under Curve): Single number summarizing the ROC curve. Higher = better.

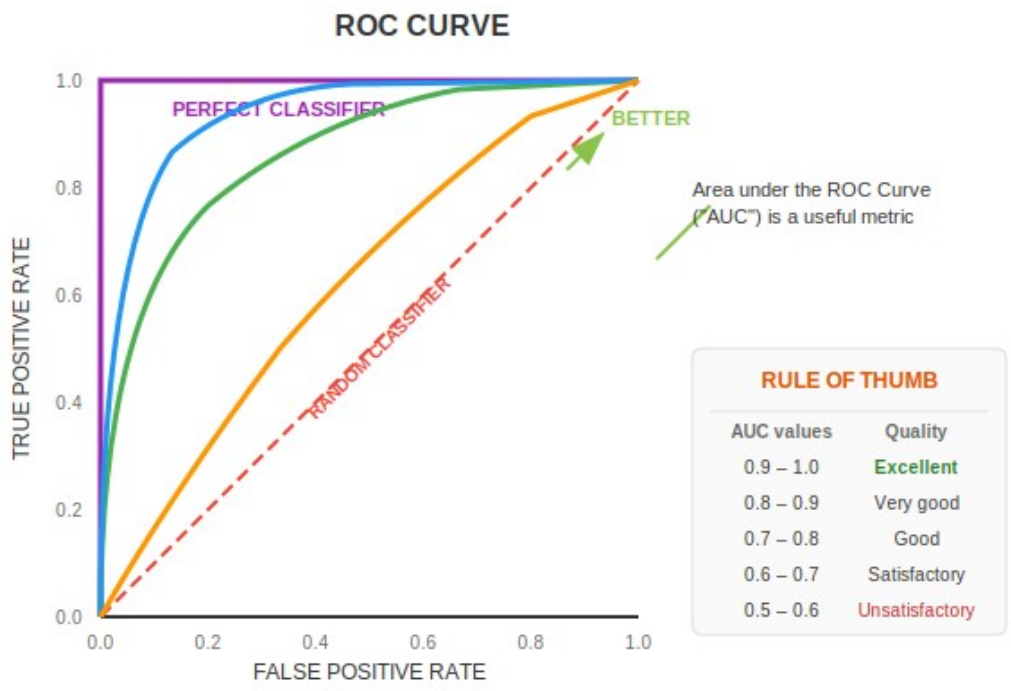


Figure 24: ROC Curve - Comparing Different Classifiers

Understanding the ROC Curve:

- **Perfect Classifier (purple):** Goes straight up then right — catches all positives with zero false positives
- **Random Classifier (dashed red diagonal):** AUC = 0.5 — no better than flipping a coin
- **Good classifiers:** Curve above the diagonal — the more "bulging" toward top-left, the better
- **Direction:** Higher and more to the left = BETTER

AUC Interpretation:

AUC Value	Quality
0.9 – 1.0	**Excellent**
0.8 – 0.9	Very Good
0.7 – 0.8	Good
0.6 – 0.7	Satisfactory
0.5 – 0.6	Unsatisfactory

Why use AUC? It measures model performance across ALL thresholds, making it useful when you haven't decided on a specific cutoff yet.

2.11 Precision-Recall Curve

What it shows: Plots Precision vs Recall at different thresholds.

Finding the Best F1 Score:

62. Generate the Precision-Recall Curve
63. At each threshold, calculate F1 score
64. Find the threshold with maximum F1
65. Use that threshold for predictions

3. Metrics for Regression

For regression problems, we measure **how far predictions are from actual values**.

3.1 Mean Absolute Error (MAE)

$$\mathbf{MAE = (1/n) \times \Sigma|y_i - \hat{y}_i|}$$

What it measures: Average absolute difference between predicted and actual values.

Interpretation: "On average, predictions are off by X units."

When to use: When all errors matter equally. Being off by 10 is exactly twice as bad as being off by 5.

3.2 Mean Squared Error (MSE)

$$\mathbf{MSE = (1/n) \times \Sigma(y_i - \hat{y}_i)^2}$$

What it measures: Average **squared** difference between predicted and actual values.

When to use: When large errors are especially bad. Being off by 10 is FOUR times as bad as being off by 5 (10^2 vs 5^2).

3.3 Root Mean Squared Error (RMSE)

$$\mathbf{RMSE = \sqrt{MSE} = \sqrt{[(1/n) \times \Sigma(y_i - \hat{y}_i)^2]}}$$

Why use RMSE over MSE? RMSE is in the same units as the target variable, making it more interpretable.

Example: If predicting house prices in dollars, RMSE = $15,000 means "on average, predictions are about $15,000 off."

3.4 R-Squared (Coefficient of Determination)

$$\mathbf{R^2 = 1 - [\Sigma(y_i - \hat{y}_i)^2 / \Sigma(y_i - \bar{y})^2]}$$

What it measures: The percentage of variance in the target explained by the model.

Example: $R^2 = 0.85$ means "85% of the variance in the dependent variable is explained by the model."

Range: 0 to 1 (can be negative for very bad models). Higher is better.

3.5 Adjusted R-Squared

The problem with R^2: It **always increases** when you add more variables, even useless ones!

$$\textbf{Adjusted } R^2 = 1 - [(1-R^2)(N-1) / (N-p-1)]$$

Where N = sample size, p = number of predictors.

When to use: When comparing models with **different numbers of variables**. Adjusted R^2 penalizes adding useless predictors.

3.6 Which Regression Metric to Choose?

Metric	Use When...	Note
MAE	All errors matter equally (linear penalty)	Robust to outliers
MSE / RMSE	Large errors should be penalized more	RMSE more interpretable
R^2	Want % of variance explained	Range: 0 to 1
Adjusted R^2	Comparing models with different # of predictors	Penalizes extra vars

4. Key Takeaways for Model Assessment

66. **The confusion matrix** is the foundation for all classification metrics
67. **Precision** — use when false positives are costly (spam detection)
68. **Recall** — use when false negatives are costly (disease detection)
69. **F1 Score** — use for balance between precision and recall
70. **Accuracy** — only reliable with balanced datasets
71. **AUC** — threshold-independent, good for model comparison
72. **MSE/RMSE** — penalize large errors more; RMSE is more interpretable
73. **MAE** — when all errors matter equally
74. **Adjusted R^2** — use when comparing models with different numbers of predictors

Lecture 6: Probabilistic Classifiers

Deep Intuition: Probabilistic Thinking
Why Probabilities Matter
A doctor wants "90% cancer" vs "51% cancer" - confidence changes decisions.
The Core Idea: Flipping the Question
We want P(class|features). We can measure P(features|class). Bayes theorem FLIPS between them.
The "Naive" Assumption
Features assumed INDEPENDENT given class. Unrealistic but works - only needs correct ranking.
One Sentence Mastery: "Naive Bayes flips P(features|class) to P(class|features) using Bayes theorem with independence assumption."
[Supervised Learning — Classification]

Bayes' Theorem & Naive Bayes

1. Why Probabilistic Classifiers?

So far, we've seen classifiers like Logistic Regression that give us a prediction (Class A or Class B). But what if we want to know **how confident** the model is in its prediction?

The Motivation:

- **Uncertainty matters:** A doctor wants to know "80% likely cancer" vs "51% likely cancer"
- **Ranking predictions:** Email spam filter can rank emails by spam probability
- **Decision making:** Different actions for 90% vs 60% confidence
- **Combining evidence:** Update beliefs as new information arrives

Probabilistic classifiers give us **probabilities** for each class, not just a hard prediction. The foundation for this is **Bayes' Theorem**.

2. Bayes' Theorem: The Foundation

2.1 The Core Idea: Updating Beliefs

Bayes' Theorem answers a fundamental question: ***"How should I update my beliefs when I see new evidence?"***

Everyday example: You hear your neighbor's car alarm. Initially, you might think "probably just a false alarm" (prior belief). But then you see a broken window (evidence). Now you update your belief: "much more likely someone tried to break in" (posterior belief).

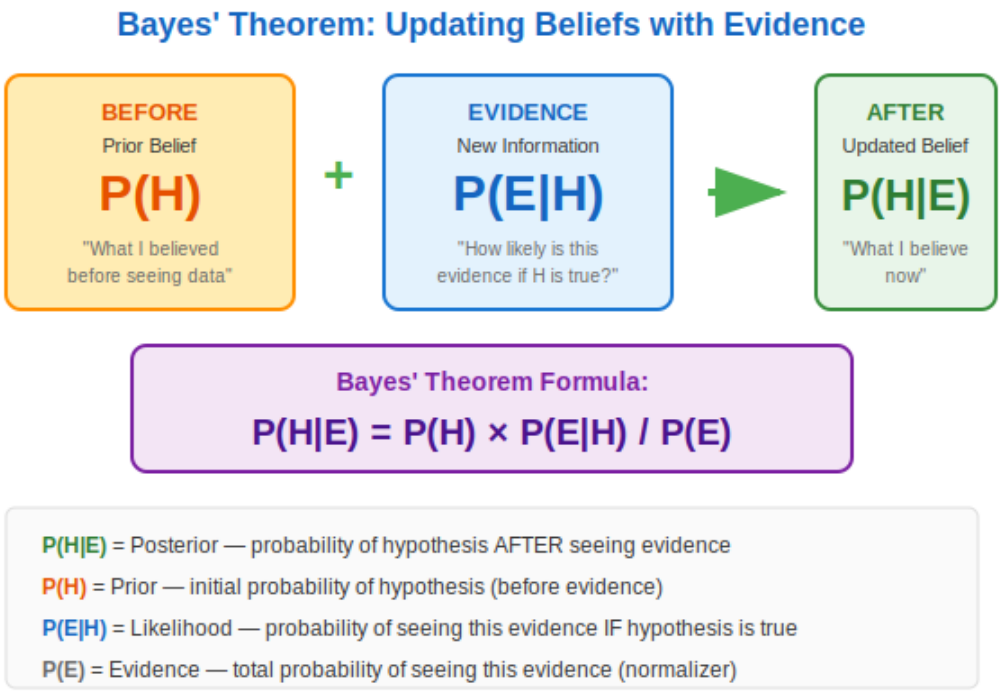


Figure 28: Bayes' Theorem - Updating Beliefs with Evidence

2.2 The Formula Explained

$$P(H|E) = P(H) \times P(E|H) / P(E)$$

Posterior = (Prior × Likelihood) / Evidence

Breaking Down Each Term:

- **P(H|E) — Posterior:** Probability of hypothesis H **AFTER** seeing evidence E. This is what we want!
- **P(H) — Prior:** What we believed **BEFORE** seeing any evidence. Our initial belief.
- **P(E|H) — Likelihood:** How likely is this evidence **IF** our hypothesis is true?
- **P(E) — Evidence:** Total probability of seeing this evidence under any circumstance. Acts as a normalizer.

2.3 Derivation (Where Does This Come From?)

Start with the probability of A and B occurring together:

$$P(A \text{ and } B) = P(A) \times P(B|A)$$

$$P(A \text{ and } B) = P(B) \times P(A|B)$$

Since both equal P(A and B), we can set them equal:

$$P(A) \times P(B|A) = P(B) \times P(A|B)$$

Solve for P(A|B):

$$P(A|B) = P(A) \times P(B|A) / P(B)$$

2.4 Calculating P(E): The Total Probability

Sometimes we can't observe P(E) directly. We calculate it by considering **all possible hypotheses**:

$$P(E) = P(H_1)\times P(E|H_1) + P(H_2)\times P(E|H_2) + \ldots = \Sigma\, P(H_i)\times P(E|H_i)$$

Example: P(Positive Test) = P(Has Disease)×P(Positive|Has Disease) + P(No Disease)×P(Positive|No Disease)

3. Concrete Example: Medical Diagnosis

This famous example shows why Bayes' Theorem is so important and often counterintuitive!

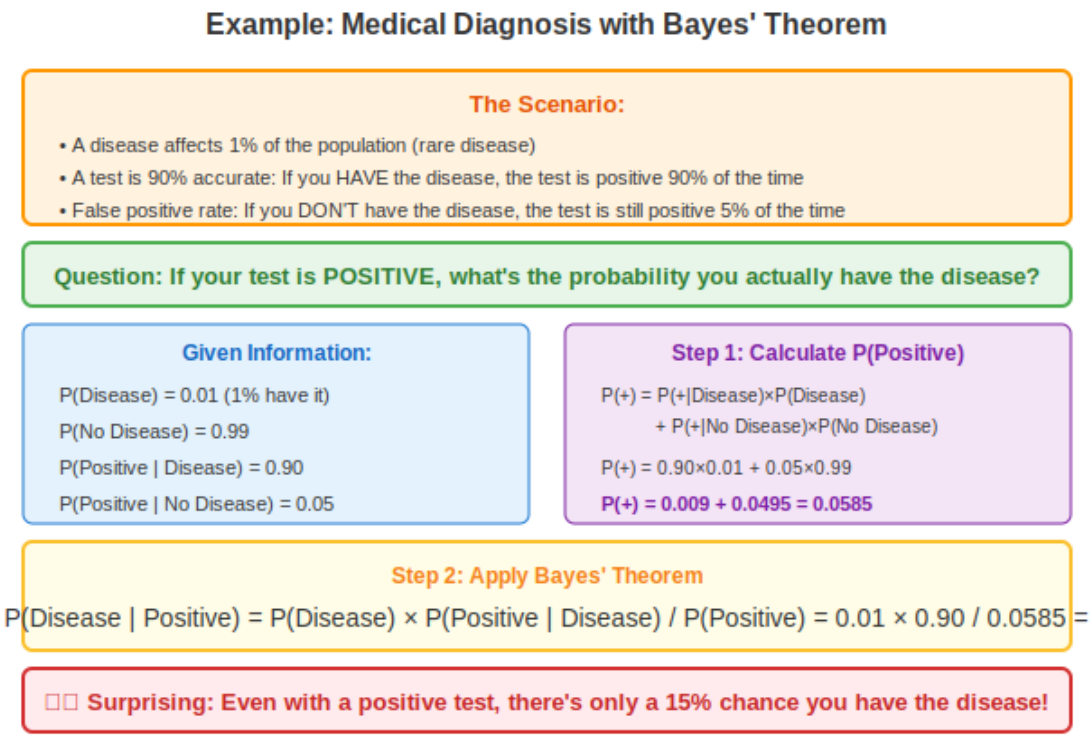


Figure 29: Medical Diagnosis Example - The Base Rate Fallacy

Why is the result so surprising?

The Base Rate Fallacy: We tend to ignore the **prior probability** (how rare the disease is). Even though the test is 90% accurate, the disease is so rare (1%) that most positive tests are actually false positives!

Key insight: In a population of 1000 people: ~10 have the disease (9 test positive), but ~990 don't have it (50 false positives). So positive results: 9 true + 50 false = 59 total. Only 9/59 ≈ 15% actually have the disease!

4. Applying Bayes to Classification

4.1 The Setup

In machine learning, we want to classify a data sample **X** (with features x_1, x_2, ..., x_n) into one of several classes C_1, C_2, ..., C_m.

Example: Customer X has features (Age=35, Income=€4000, Student=No). We want to predict: Will they buy a computer?

Using Bayes' Theorem:

$$\mathbf{P(Class|X) = P(X|Class) \times P(Class) / P(X)}$$

What each term means in classification:

- **P(Class|X) — Posterior:** Probability customer belongs to "Buys Computer" class given their features
- **P(Class) — Prior:** Overall probability of any customer buying a computer (e.g., 30% of customers buy)
- **P(X|Class) — Likelihood:** Probability of seeing these exact features among customers who DO buy computers
- **P(X) — Evidence:** Probability of seeing a customer with these exact features

4.2 Maximum A Posteriori (MAP) Classification

The classification rule: Assign X to the class with the **highest posterior probability**.

$$\mathbf{Predicted\ Class = argmax\ P(C_i|X) = argmax\ P(X|C_i) \times P(C_i)}$$

Key simplification: Since P(X) is the same for all classes, we don't need to calculate it! We just compare $P(X|C_i) \times P(C_i)$ for each class and pick the maximum.

5. The Naive Bayes Classifier

5.1 The Problem: Too Many Combinations

To calculate P(X|Class), we need the probability of seeing **this exact combination of features**. With many features, there are too many combinations to estimate from data!

Example: With 10 binary features, there are $2^{10} = 1024$ possible combinations. We'd need thousands of examples to estimate each probability!

5.2 The "Naive" Solution: Assume Independence

Naive Bayes makes a **simplifying assumption**: All features are **conditionally independent** given the class.

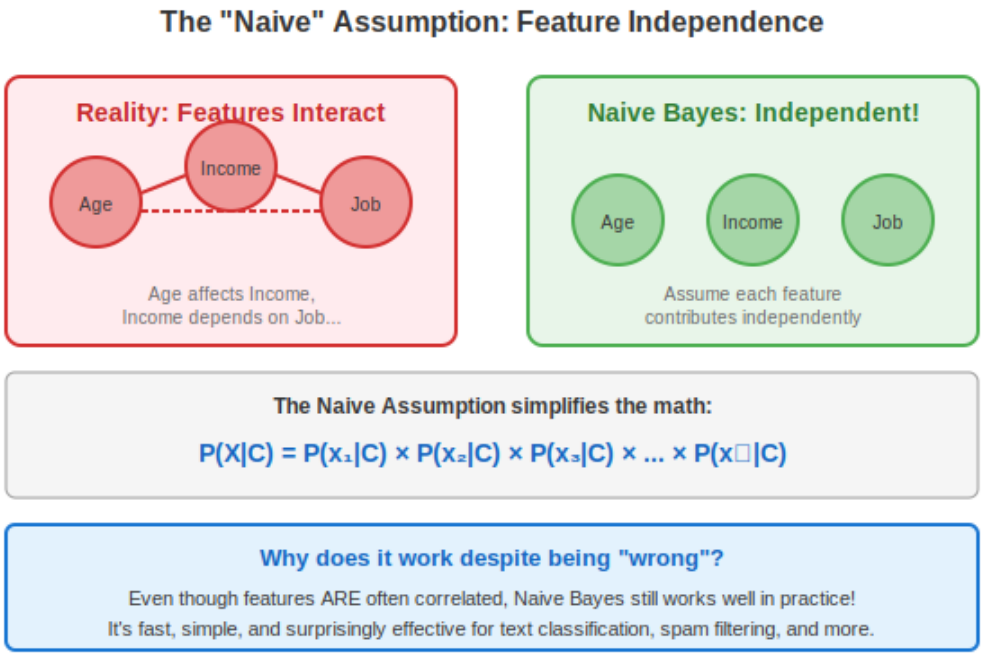


Figure 30: The Naive Assumption - Features are Treated as Independent

This lets us break down the likelihood into a simple product:

$$P(X|C) = P(x_1|C) \times P(x_2|C) \times P(x_3|C) \times ... \times P(x_n|C)$$

Now we only need to estimate P(each feature | class) separately! Much easier with limited data.

5.3 Why is it Called "Naive"?

The assumption is called "naive" because it's usually **wrong**! Features in real data are often correlated.

Example: Age and Income are correlated (older people tend to earn more). But Naive Bayes pretends they're independent.

Surprisingly, it still works well! Even though the probability estimates may be off, the **ranking of classes** is often correct. That's all we need for classification.

5.4 The Complete Classification Process

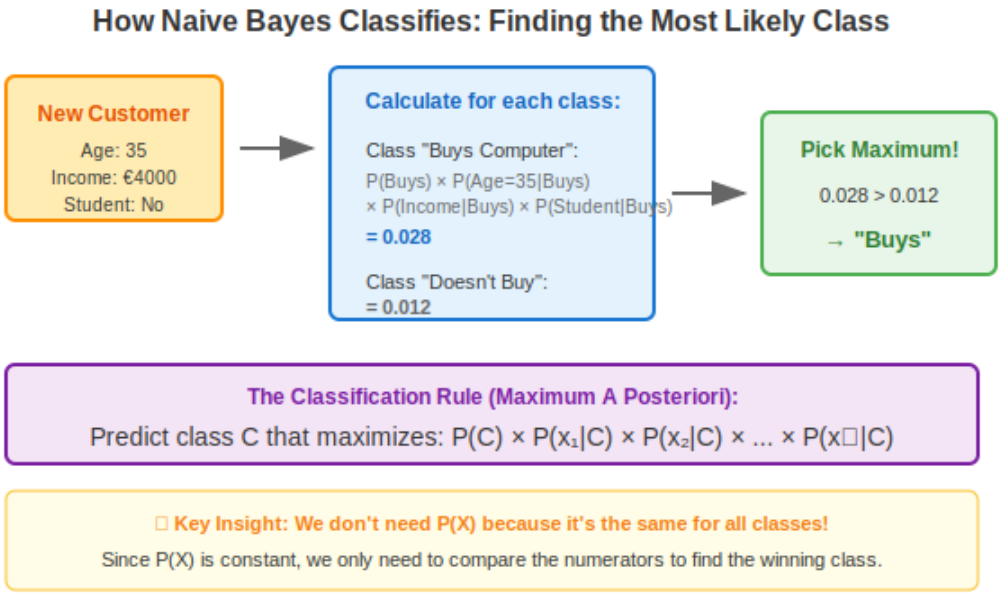


Figure 31: How Naive Bayes Classifies a New Sample

Step-by-step process:

75. **Calculate priors:** $P(C_i)$ = count of class i / total samples
76. **Calculate likelihoods:** For each feature, P(feature value | class)
77. **Multiply together:** Score = $P(C) \times P(x_1|C) \times P(x_2|C) \times ...$
78. **Pick the winner:** Class with highest score

5.5 Handling Unknown Class Priors

If we don't know the class prior probabilities, we have two options:

- **Option 1:** Assume all classes are **equally likely**: $P(C_1) = P(C_2) = ... = 1/m$
- **Option 2:** Estimate from training data: $P(C_i)$ = (samples in class i) / (total samples)

6. Types of Naive Bayes Classifiers

Type	Feature Type	Use Case
Gaussian NB	Continuous (assumes normal distribution)	Real-valued features like height, weight, temperature
Multinomial NB	Discrete counts	Text classification (word counts), document categorization
Bernoulli NB	Binary (0/1)	Text (word present/absent), binary features

7. Naive Bayes: Pros and Cons

✅ Advantages:

- **Fast and simple:** Training and prediction are very fast
- **Works with small data:** Needs less training data than many other methods
- **Handles many features:** Scales well to high-dimensional data
- **Probabilistic outputs:** Gives probability estimates, not just predictions
- **Great for text:** Excellent for text classification, spam filtering

❌ Disadvantages:

- **Independence assumption:** Often violated in practice
- **Probability estimates:** The actual probabilities may be poorly calibrated
- **Zero frequency problem:** If a feature-class combination never appears in training, probability = 0 (use Laplace smoothing to fix)

8. Key Takeaways

79. **Bayes' Theorem** lets us update beliefs with evidence: Posterior = Prior × Likelihood / Evidence
80. **The Prior matters!** Rare events stay rare even with strong evidence (base rate fallacy)
81. **MAP Classification:** Pick the class with highest P(Class) × P(Features|Class)
82. **Naive Bayes** assumes features are independent — makes computation tractable
83. **"Naive" but effective:** Works well in practice despite the unrealistic assumption
84. **Best for text classification:** Spam filtering, sentiment analysis, document categorization
85. **P(X) can be ignored** for classification since it's the same for all classes

15. Bayesian Belief Networks: Deep Dive

Now let's explore Bayesian Belief Networks in more detail, including how to build them and use them for probabilistic inference.

15.1 Key Terminology

- **Nodes** represent variables (e.g., Season, Location, Sales)
- **Arcs (edges)** represent directed dependencies among variables
- **Parent:** Node X is a parent of Node Z if there exists a directed arc from X to Z
- **Descendant:** Node Z is a descendant of Node X if there is a directed path from X to Z

15.2 The Markov Property (Conditional Independence)

The fundamental property of Bayesian networks:
Each variable in a Bayesian network is conditionally independent of its non-descendants in the network, given its parents.
This property allows us to factorize the joint probability distribution:
$P(X_1, X_2, ..., X_m) = \prod P(X_i \mid parents(X_i))$
This is MUCH more efficient than computing the full joint probability table!

15.3 Building a Bayesian Network

To build a Bayesian network, there are two main considerations:

86. **Structure:** What is the dependence relationship among the variables of interest?
87. **Parameters:** What are the associated "local" probabilities (Conditional Probability Tables)?

- To build a Bayesian network, there are two main considerations:
  1. What is the dependence relationship among the variables of interest?
  2. What are the associated "local" probabilities?

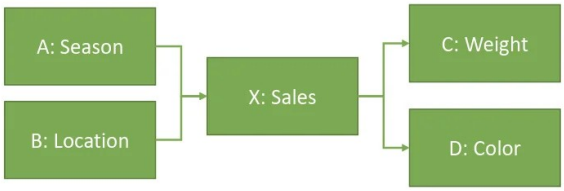


Figure 37: Clothing Retail BBN – Season and Location Influence Sales, Which Influences Weight and Color

15.4 Clothing Retail Example: Building the Structure

Consider a clothing retailer trying to model factors affecting their sales:

88. **Season** does not depend on any other variables → placed at the top of the network
89. **Location** does not depend on other variables → also at the top of the network
90. **Sales (X)** depends on both Season and Location → has arcs from both
91. **Fabric Weight and Color** are only known after purchase → depend on Sales (what was actually sold)

15.5 Specifying the Local Probabilities

For each node, we specify a Conditional Probability Table (CPT):

- **Season node:** Sales are uniform throughout seasons (P = 0.25 each for Spring, Summer, Fall, Winter)
- **Location node:** 60% from Los Angeles store, 40% from New York store
- **Sales node:** P(clothing type | season, location) — e.g., P(shorts | winter, NY) = 0.05
- **Weight and Color nodes:** P(weight | clothing type) and P(color | clothing type)

Note: Root nodes (Season, Location) only need marginal probabilities, not conditional ones, because they have no parents.

15.6 Using the BBN for Probabilistic Inference

Example question: What is the probability that a purchase involved light-fabric, neutral-colored Bermuda shorts in New York during winter?

Using the bayesian network to find probabilities

- Find the probability that a given purchase involved **light-fabric**, **neutral-colored Bermuda shorts** was made in **New York** in the **winter**.

Using:

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_m = x_m) = \prod_{i=1}^{m} P(X_i = x_i | parents(X_i))$$

$$P(A = a_4, B = b_1, C = c_1, D = d_2, X = x_3)$$

$$= P(A = a_4).P(B = b_1).P(X = x_3 | A = a_4 \cap B = b_1).P(C = c_1 | X = x_3).P(D = d_2 | X = x_3)$$

$$= \mathrm{P}(seas = wint) \times P(Loc = NY) \times \mathrm{P}(Cloth = shorts | seas = Wint\ and\ Loc = NY) \times P(fab = light | cloth = shorts) \times P(color = neutr | cloth = shorts)$$

$$= 0.25 \times 0.4 \times 0.05 \times 0.5 \times 0.4 = 0.001$$

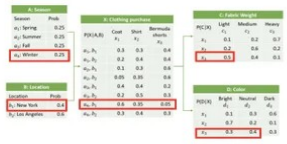


Figure 38: Computing Joint Probability Using the BBN Factorization

Step-by-step calculation:

P(Season=Winter, Location=NY, Clothing=Shorts, Fabric=Light, Color=Neutral)
= P(Season=Winter) × P(Location=NY) × P(Shorts | Winter, NY) × P(Light | Shorts) × P(Neutral | Shorts)
= 0.25 × 0.4 × 0.05 × 0.5 × 0.4 = 0.001

Result: There's a 0.1% chance of this specific combination occurring.

15.7 How Are Bayesian Networks Constructed?

There are three main approaches to building Bayesian networks:

1. Subjective Construction (Expert Knowledge)

- Humans identify direct causal relationships from domain knowledge
- **Markovian assumption:** Each variable becomes independent of its non-effects once its direct causes are known
- **Hidden Markov Models (HMMs):** Special case for dynamic systems where states are not observable but outputs are

2. Synthesis from Other Specifications

- Derived from formal system designs: block diagrams, information flow charts
- Useful in engineering and systems analysis

3. Learning from Data

- Learn from data like medical records, student admissions, customer transactions
- Can learn just the parameters (given structure) OR learn both structure and parameters
- **Maximum Likelihood Principle:** Choose the network that maximizes the probability of observing the given dataset

16. Final Summary: Lecture 6

This lecture covered probabilistic classifiers, from the foundation of Bayes' Theorem to practical applications:

- **Bayes' Theorem:** Foundation for updating beliefs with evidence
- **Naive Bayes:** Simple, fast classifier assuming feature independence
- **Practical issues:** Laplace smoothing, missing values, numeric attributes

- **Bayesian Belief Networks:** Model actual dependencies using DAGs and CPTs
- **BBN Construction:** Expert knowledge, system specifications, or learning from data

Lecture 6 Practical: Model Selection and Hyperparameter Tuning

This section covers the practical implementation of model selection concepts using scikit-learn, including cross-validation techniques, model comparison, hyperparameter tuning, and pipelines.

1. Cross-Validation Techniques in Scikit-Learn

Scikit-learn provides multiple cross-validation strategies. Here are the most important ones:

1.1 Basic Splitting and K-Fold Variants

- **train_test_split:** Simple one-time split into train and test sets
- **KFold:** Standard K-Fold cross-validation
- **StratifiedKFold:** K-Fold that preserves class proportions in each fold (ALWAYS use for classification!)
- **RepeatedKFold:** Runs K-Fold multiple times with different random splits for more stable estimates
- **RepeatedStratifiedKFold:** Best of both worlds — stratified AND repeated

1.2 Leave-One-Out Variants

- **LeaveOneOut:** Each sample is used once as test set (K = n). Best for very small datasets.
- **LeavePOut:** Leave P samples out as test set each iteration

1.3 Group-Based Splitting

Use when samples belong to groups that shouldn't be split (e.g., multiple samples from same patient):

- **GroupKFold:** Ensures all samples from one group stay in the same fold
- **LeaveOneGroupOut:** Each group is used once as the test set

1.4 Special Cases

- **TimeSeriesSplit:** For time series data — always trains on past, tests on future (respects temporal order)
- **ShuffleSplit:** Random permutation cross-validator with control over train/test size

2. Comparing Models with Cross-Validation

One of the main reasons to use cross-validation is to compare different models fairly on the same data with the same validation strategy.

2.1 Example: Logistic Regression vs Decision Tree

Using RepeatedKFold (n_splits=6, n_repeats=2) to compare models:

```
from sklearn.model_selection import RepeatedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
rkf = RepeatedKFold(n_splits=6, n_repeats=2)
```

Key point: The cross-validation functions are generic — you can pass ANY model to compare them fairly.

3. Hyperparameter Tuning

3.1 Parameters vs Hyperparameters

Parameters: Learned during training (e.g., weights in linear regression, coefficients in logistic regression)
Hyperparameters: Set BEFORE training — they control how the algorithm learns. Different hyperparameters lead to very different models!

3.2 Examples of Hyperparameters

Logistic Regression:

- `C:` Regularization strength (smaller = stronger regularization)
- `penalty:` Type of regularization ('l1', 'l2', 'elasticnet')
- `solver:` Optimization algorithm

Decision Tree:

- `max_depth:` Maximum depth of the tree (controls overfitting)
- `min_samples_split:` Minimum samples needed to split a node
- `min_samples_leaf:` Minimum samples required at a leaf node
- `criterion:` How to measure split quality ('gini' or 'entropy')

3.3 Grid Search vs Random Search

When tuning hyperparameters, we need to search through different combinations to find the best performing model. There are two main approaches:
Grid Search (GridSearchCV)

- **How it works:** Systematically tests ALL combinations from a predefined grid of parameter values
- **Pros:** Exhaustive — guaranteed to find the best combination in your grid
- **Cons:** Computationally expensive! With 3 hyperparameters each having 5 values = 125 combinations to try

Random Search (RandomizedSearchCV)

- **How it works:** Randomly samples parameter combinations from a grid or distribution
- **Pros:** Much faster! You control how many combinations to try (n_iter parameter)
- **Cons:** Might miss the optimal combination (but often finds "good enough" quickly)

When to use which: Use Grid Search for small search spaces. Use Random Search when you have many hyperparameters or continuous ranges.

3.4 Grid Search with Holdout Validation

When performing hyperparameter tuning with holdout validation, follow these critical rules:

92. **Fit preprocessing only on X_train:** Scaling, NaN filling, encoding — all fitted on training data only
93. **Evaluate on X_val:** Validation set wasn't used for training or preprocessing decisions
94. **NEVER touch the test set:** The test set is never used during hyperparameter selection — only for final evaluation

Example parameter grid:

```
param_grid = {
    'C': [0.01, 0.1, 1, 10],
    'penalty': ['l1', 'l2']
}
```

4. Pipelines: Combining Preprocessing and Modeling

4.1 What is a Pipeline?

A Pipeline chains multiple steps (preprocessing + model) into a single object that can be fit, transformed, and used for prediction as one unit.

Why use pipelines?

- **Prevents data leakage:** Preprocessing is correctly applied within each CV fold
- **Cleaner code:** One object instead of manually chaining fit/transform calls
- **Easy deployment:** Save and load the entire workflow as one object
- **Works with GridSearchCV:** Can tune hyperparameters of any step in the pipeline

4.2 Basic Pipeline Example

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
pipe = Pipeline([
    (‘scaler’, StandardScaler()),
    (‘classifier’, LogisticRegression())
])
pipe.fit(X_train, y_train)  # Fits scaler AND classifier
pipe.predict(X_test)        # Scales AND predicts
```

4.3 Pipeline with GridSearchCV

You can tune hyperparameters of any step in the pipeline using the naming convention: stepname__parametername

```
from sklearn.model_selection import GridSearchCV
param_grid = {
    ‘classifier__C’: [0.1, 1, 10],
    ‘classifier__penalty’: [‘l1’, ‘l2’]
}
grid_search = GridSearchCV(pipe, param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

Key benefit: The scaler is fit ONLY on the training fold in each CV iteration, preventing data leakage!

4.4 The Data Leakage Problem (Why Pipelines Matter)

Without a pipeline (WRONG):

```
scaler.fit(X)  # Fitted on ALL data including test!
X_scaled = scaler.transform(X)
cross_val_score(model, X_scaled, y)  # Leakage!
```

With a pipeline (CORRECT):

```
cross_val_score(pipe, X, y)  # Scaler fitted only on train fold each time
```

Data leakage occurs when information from the test/validation set “leaks” into training. This leads to overly optimistic performance estimates that won’t generalize!

5. Practical Workflow Summary

The complete workflow for model selection and tuning:

95. **Split data:** Train / Validation / Test (or use CV for train/val)
96. **Create pipeline:** Preprocessing steps + model
97. **Define parameter grid:** Hyperparameters to search

98. **Run GridSearchCV or RandomizedSearchCV:** With appropriate CV strategy
99. **Select best model:** Based on validation performance
100. **Final evaluation:** Test best model on held-out test set (ONLY ONCE!)

Lecture 7: Instance-Based Learning (K-Nearest Neighbors)

Deep Intuition: Learning by Analogy
The Core Philosophy
Similar inputs → similar outputs. KNN REMEMBERS all examples, looks up similar ones. No model built - data IS the model.
Lazy vs Eager
Eager (neural nets): Heavy training, fast prediction. Lazy (KNN): No training, slow prediction.
Critical Decisions
K=1 noisy, K=100 over-smooths. ALWAYS NORMALIZE features. Curse of dimensionality: KNN fails in high dimensions.
One Sentence Mastery: "KNN predicts by finding K similar examples and voting - requires normalized features and few dimensions."
[Supervised Learning — Classification]

1. What is Instance-Based Learning?

Instance-based learning is fundamentally different from other machine learning approaches we've studied. Instead of building an explicit model during training, it simply stores all training examples and defers all computation until prediction time.

1.1 Lazy Learning vs Eager Learning

Eager Learning (most algorithms we've seen):
- Training phase: Build an explicit model (learn weights, build tree, estimate probabilities)
- Prediction phase: Apply the model (fast!)
- Examples: Linear Regression, Logistic Regression, Decision Trees, Naive Bayes, Neural Networks

Lazy Learning (instance-based):
- Training phase: Just store the data (essentially instant!)
- Prediction phase: Do ALL the work — compare new instance to stored examples (slow!)
- Examples: K-Nearest Neighbors, Case-Based Reasoning

Why "lazy"? Because the algorithm procrastinates! It doesn't do any real work during training — it waits until you actually need a prediction, then scrambles to figure out the answer by looking at stored examples. It's like a student who doesn't study until the night before the exam!

1.2 The Rote Learner: Simplest Instance-Based Approach

The simplest instance-based classifier is a "rote learner":

- Memorizes the entire training dataset
- Only classifies a new instance if it EXACTLY matches a training example

Problem: What if there's no exact match? This is almost always the case with continuous features!
Solution: Use the "closest" point(s) instead — this is the Nearest Neighbor algorithm!

2. K-Nearest Neighbors (KNN) Algorithm

2.1 The Basic Algorithm

Given a new instance x to classify:

101. Calculate the distance from x to ALL training instances
102. Select the k nearest (closest) training instances
103. For classification: Return the majority class among the k neighbors
104. For regression: Return the average of the target values of the k neighbors

Special case (k=1): 1-Nearest Neighbor simply assigns the class of the single closest training example.

2.2 Distance Metrics

To find the k closest neighbors, we need a way to measure "closeness" between two points. Here are the most common distance metrics:

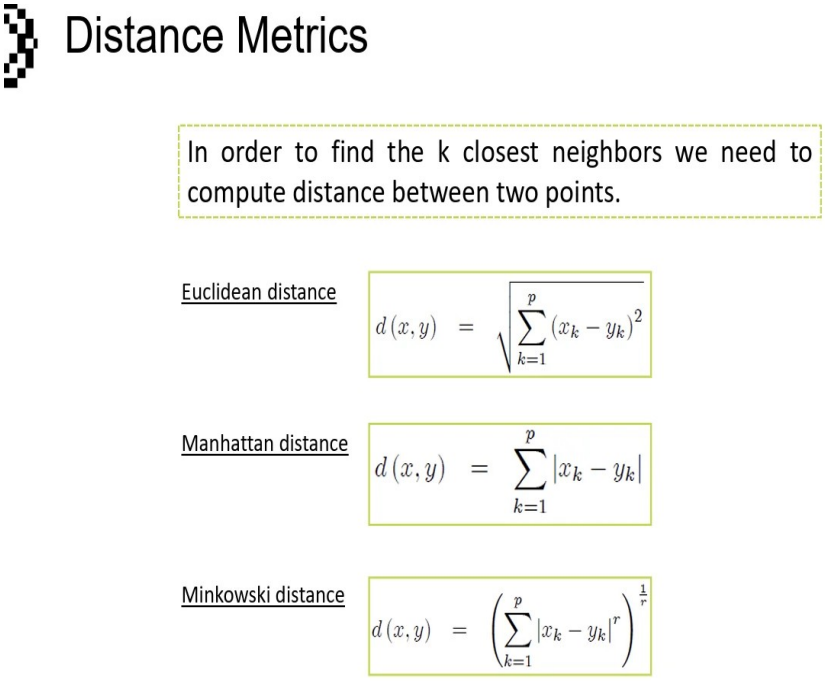


Figure 39: Common Distance Metrics for KNN

Euclidean Distance (L2 norm): The "straight line" distance. Most commonly used. d(x,y) = $\sqrt{(\sum(x_k - y_k)^2)}$
Manhattan Distance (L1 norm): "City block" distance — sum of absolute differences. d(x,y) = $\sum|x_k - y_k|$
Minkowski Distance: Generalization: d(x,y) = $(\sum|x_k - y_k|^n)^{(1/r)}$. When r=2, it's Euclidean; when r=1, it's Manhattan.

CRITICAL: Feature scaling matters! If one feature ranges from 0-1000 and another from 0-1, the first will dominate the distance calculation. ALWAYS standardize or normalize features before using KNN!

2.3 Determining the Class: Voting Schemes

Standard approach (Majority Vote): Each of the k neighbors gets one vote. The class with the most votes wins.
Distance-Weighted Voting: Closer neighbors should have more influence! Weight each vote by the inverse of distance squared: $w = 1/d^2$
Distance-Weighted KNN formula:
$f(x_k) = (\sum w_i \times f(x_i)) / (\sum w_i)$ where $w_i = 1 / d(x_k, x_i)^2$
Shepard's method: With distance weighting, it can make sense to use ALL training examples (not just k), since far-away points will have negligible weight anyway.

3. The Effect of k: Decision Boundaries

The choice of k dramatically affects the decision boundary and model behavior:

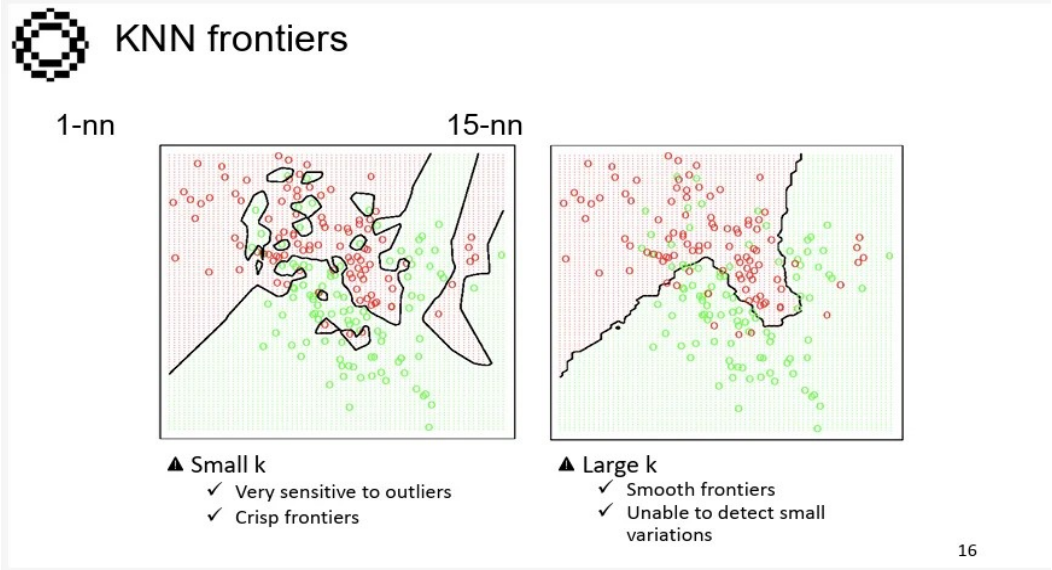


Figure 40: Effect of k on KNN Decision Boundaries (1-NN vs 15-NN)

3.1 Why Small k Gives Jagged Boundaries (1-NN Case)

With k=1, every point is classified based on its SINGLE nearest neighbor. This creates a Voronoi diagram:

- **Each training point "owns" a region** — the set of all points closer to it than to any other training point
- **Boundaries are the perpendicular bisectors** between adjacent training points
- **Sensitive to outliers:** A single mislabeled point or outlier creates an "island" of the wrong class
- **High variance, low bias:** The model is very flexible and will overfit to noise in the training data

3.2 Why Large k Gives Smooth Boundaries (15-NN Case)

With large k, each prediction is based on voting among many neighbors:

- **Smoothing effect:** Outliers get outvoted by the majority of neighbors

- **More robust:** Less affected by noise and mislabeled points
- **BUT:** Unable to capture small local variations in the data
- **Low variance, high bias:** The model is more rigid and may underfit

3.3 The Extreme Case: k → n

If k equals the total number of training points (k = n), then EVERY point gets to vote on EVERY prediction. This means:

- Every new point gets classified as the majority class in the entire dataset
- For balanced binary classification, the decision boundary becomes approximately linear (like logistic regression!)
- The model becomes maximally simple — essentially just predicting the most common class

Practical advice: Use cross-validation to find the optimal k. Common starting point: k = √n (square root of training set size). Use odd k for binary classification to avoid ties.

4. KNN for Regression (Continuous Targets)

KNN can also predict continuous values! Instead of voting, we average:
Simple average: $\hat{c}(x_k) = (1/k) \sum c(x_i)$ for the k nearest neighbors
Weighted average: $\hat{c}(x_k) = (\sum w_i\, c(x_i)) / (\sum w_i)$ where $w_i = 1/d^2$
Example: Predicting house prices. Find k similar houses, average their prices.

5. The Computational Problem: K-d Trees

5.1 The Problem with Naive KNN

The basic KNN algorithm has a serious scalability problem:

- **Training time:** O(1) — just store the data
- **Prediction time:** O(n × d) for EACH prediction — compare to all n training points across d features

With millions of training points, computing distances to ALL of them for each prediction is prohibitively slow!
Solution: Use spatial data structures to quickly find nearby points without checking every single one. The most famous is the K-d Tree.

5.2 What is a K-d Tree?

A K-d Tree (k-dimensional tree) is a binary tree that recursively partitions the feature space. Think of it as a smart index that lets you quickly eliminate large portions of the data when searching for neighbors.
Key properties:

- Each node represents a training instance
- At each level, we split on a different feature (cycling through features)
- The split point is the MEDIAN value of that feature
- Left subtree: points with feature value < median
- Right subtree: points with feature value ≥ median

5.3 Building a K-d Tree: Step-by-Step Example

Let's build a K-d tree with a simple 2D dataset of 7 students with their Math and Science scores:

```
Student  | Math | Science
---------|------|--------
```

A	7	2
B	5	4
C	9	6
D	2	3
E	4	7
F	8	1
G	3	8

Step 1: Split on Math (first feature)

- Sort by Math: D(2), G(3), E(4), B(5), A(7), F(8), C(9)
- Median is B (Math=5) → B becomes the ROOT
- Left subtree: {D, G, E} (Math < 5)
- Right subtree: {A, F, C} (Math ≥ 5)

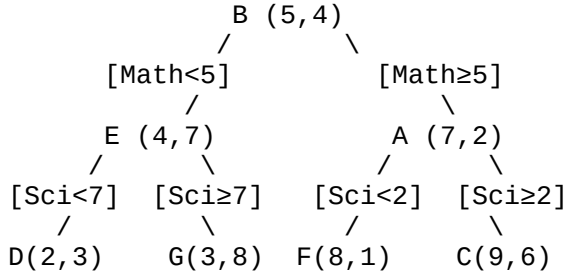
Step 2: Split left subtree {D, G, E} on Science (second feature)

- Sort by Science: D(3), E(7), G(8)
- Median is E (Science=7) → E becomes B's left child
- E's left child: D (Science < 7)
- E's right child: G (Science ≥ 7)

Step 3: Split right subtree {A, F, C} on Science

- Sort by Science: F(1), A(2), C(6)
- Median is A (Science=2) → A becomes B's right child
- A's left child: F (Science < 2)
- A's right child: C (Science ≥ 2)

Final K-d Tree structure:



5.4 Finding Nearest Neighbor Using the K-d Tree

Now let's find the nearest neighbor to a new student X with scores (6, 5):

Step 1: Traverse down to a leaf

- At B: X's Math=6 ≥ 5 → go RIGHT to A
- At A: X's Science=5 ≥ 2 → go RIGHT to C
- C is a leaf. Distance to C(9,6): $\sqrt{((6-9)^2+(5-6)^2)} = \sqrt{10} \approx 3.16$
- **Best distance so far:** 3.16 (candidate: C)

Step 2: Backtrack and check if we need to explore other branches

- Back at A(7,2): Distance to A = $\sqrt{((6-7)^2+(5-2)^2)} = \sqrt{10} \approx 3.16$ (same!)
- Could F's branch have a closer point? Distance from X to the splitting hyperplane (Science=2) is |5-2|=3 < 3.16 → YES, we must check F!
- Distance to F(8,1): $\sqrt{((6-8)^2+(5-1)^2)} = \sqrt{20} \approx 4.47$ (worse, keep C or A)

Step 3: Continue backtracking to B

- Distance to B(5,4): $\sqrt{((6-5)^2+(5-4)^2)} = \sqrt{2} \approx 1.41$ → NEW BEST!
- Check left subtree? Distance to hyperplane (Math=5) is |6-5|=1 < 1.41 → YES
- Explore E subtree... (checking D, E, G — none closer than B)

Result: Nearest neighbor to X(6,5) is B(5,4) with distance $\sqrt{2} \approx 1.41$

Key insight: We only had to compute distances to 5-6 points instead of all 7. For large datasets, this savings is enormous!

5.5 When K-d Trees Work Best

K-d trees are most effective when:

- **Many more instances than features:** n >> 2^m (where m is number of features)
- **Rule of thumb:** Need at least 2^m instances for m features
- **Low dimensionality:** Works great for d < 20 features

Curse of dimensionality: In high dimensions, almost all points are far from each other, and K-d trees degrade to brute-force search. For high-dimensional data, use approximate nearest neighbor methods (LSH, FAISS).

6. KNN: Advantages and Disadvantages

Advantages:

- Simple to understand and implement
- No training phase (instant!)
- Naturally handles multi-class problems
- Can capture complex, non-linear decision boundaries
- Easy to add new training data (just store it!)

Disadvantages:

- Slow prediction (must compare to all training points)
- High memory usage (store entire training set)
- Sensitive to irrelevant features and feature scaling
- Suffers from curse of dimensionality
- No insight into which features are important

7. Key Takeaways for Lecture 7

- **Instance-based = Lazy learning:** No model built during training, all work done at prediction time
- **KNN algorithm:** Find k nearest neighbors, take majority vote (classification) or average (regression)
- **Distance matters:** Euclidean most common, always scale features first!
- **Choice of k is crucial:** Small k = overfit (jagged boundaries), Large k = underfit (smooth boundaries)
- **Distance weighting:** Use $w=1/d^2$ to give closer neighbors more influence
- **K-d Trees:** Binary tree index to speed up neighbor search — splits feature space recursively
- **K-d Tree works best when:** n >> 2^m (many instances, few features)
- **Curse of dimensionality:** KNN struggles in high dimensions — all points become “far” from each other

8. Other Measures of Similarity

So far we’ve focused on Euclidean and Manhattan distances for continuous features. But what about other types of data? KNN can use many different similarity/distance measures depending on your data type.

8.1 Similarity Measures for Binary Features

When features are binary (True/False, 0/1), we compare instances by counting matches and mismatches. For two instances, we define four quantities:

- **CP (Co-Presence):** Both instances have True (1-1 match)

- **CA (Co-Absence):** Both instances have False (0-0 match)
- **PA (Presence-Absence):** First instance True, second instance False (1-0 mismatch)
- **AP (Absence-Presence):** First instance False, second instance True (0-1 mismatch)

Note: CP + CA + PA + AP = total number of binary features

8.2 Binary Similarity Indices

Russel-Rao Similarity Index:
Sim^{RR} = CP / (total number of binary features)
Uses only co-presence (1-1 matches). Good when only "positive" matches matter.
Sokal-Michener Similarity Index (Simple Matching):
$Sim^{\chi M}$ = (CP + CA) / (total number of binary features)
Uses both co-presence AND co-absence. Treats 0-0 matches as equally important as 1-1 matches.
Jaccard Similarity Index:
$Sim^{\check{}J}$ = CP / (CP + PA + AP)
Uses all EXCEPT co-absence. Ignores 0-0 matches entirely! This is important when absence is not meaningful (e.g., two documents not containing a rare word doesn't make them similar).

8.3 Example: Website User Behavior

Consider tracking user behavior on a website with binary features:

```
User | Profile | FAQ | HelpForum | Newsletter | Liked
-----|---------|-----|-----------|------------|------
  1  |  true   |true |   true    |   false    | true
  2  |  true   |false|   false   |   false    | false
```

New user X: Profile=true, FAQ=false, HelpForum=true, Newsletter=false, Liked=false
Comparing X to User 1:

- CP = 2 (Profile, HelpForum both true)
- CA = 1 (Newsletter both false)
- PA = 0 (X true, User1 false)
- AP = 2 (FAQ: X false, User1 true; Liked: X false, User1 true)

Jaccard(X, User1) = 2 / (2 + 0 + 2) = 0.5
Sokal-Michener(X, User1) = (2 + 1) / 5 = 0.6

8.4 Cosine Similarity (for Continuous Features)

Sometimes we want to compare instances based on their DIRECTION (pattern of values) rather than their magnitude. For example, two customers who buy products in similar proportions should be considered similar, even if one buys 10x more overall.
Cosine Similarity Formula:
$Sim^{cosine}(a, b) = (a \cdot b) / (||a|| \times ||b||) = (\sum a_i b_i) / (\sqrt{\sum a_i^2} \times \sqrt{\sum b_i^2})$
Key properties:

- Measures the cosine of the angle between two vectors
- Range: -1 to 1 (or 0 to 1 if features are non-negative)
- Value of 1 = vectors point in same direction (most similar)
- Value of 0 = vectors are perpendicular (no similarity)
- IGNORES magnitude — only cares about direction!

Common use case: Text similarity! Documents are represented as vectors of word counts (or TF-IDF). Cosine similarity finds documents with similar word distributions regardless of document length.
Example:
A = (3, 4), B = (6, 8)

Euclidean distance = $\sqrt{((3-6)^2 + (4-8)^2)} = 5$ (seems different!)
Cosine similarity = $(3\times6 + 4\times8) / (5 \times 10) = 50/50 = 1.0$ (identical direction!)

8.5 Mahalanobis Distance

The problem with Euclidean distance: It treats all directions equally and doesn't account for how the data is actually distributed. Two points that are the same Euclidean distance from the center might have very different "statistical" distances if the data is correlated.
Mahalanobis Distance uses the covariance structure of the data to account for:

- **Feature correlations:** If X and Y are correlated, movement along the correlation direction is "cheaper"
- **Different variances:** Automatically scales features by their variance

Intuition with correlated data:
Imagine data points forming an elongated ellipse (X and Y are correlated). Two points might have the SAME Euclidean distance from the center, but:

- Point 1 lies along the major axis (direction of correlation) → statistically "normal"
- Point 2 lies perpendicular to the correlation → statistically "unusual"

Mahalanobis distance captures this: Point 1 has LOWER Mahalanobis distance than Point 2, even though their Euclidean distances are equal.
Formula:
$D_m(x) = \sqrt{((x - \mu)^T S^{-1} (x - \mu))}$
Where S is the covariance matrix of the data and μ is the mean.
When to use: When features are correlated and you want distance to respect the data's natural structure. Common in anomaly detection and multivariate statistics.

8.6 Summary: Choosing the Right Similarity Measure

Data Type	Measure	When to Use
Continuous	Euclidean	General purpose, scaled features
Continuous	Manhattan	Robust to outliers
Continuous	Cosine	Direction matters, not magnitude
Continuous	Mahalanobis	Correlated features
Binary	Jaccard	Absence not meaningful
Binary	Sokal-Michener	Both presence and absence matter
Binary	Russel-Rao	Only presence matters
Text	Cosine (TF-IDF)	Document similarity

Lecture 8: Neural Networks

Deep Intuition: Learning Representations
The Limitation of Linear Models
Linear models only learn lines. XOR cannot be solved linearly. We need NON-LINEAR capabilities.
The Neural Network Insight
Stack layers, each transforms data. Hidden layers learn useful REPRESENTATIONS making final prediction easy.
Why Activation Functions?
Without them, stacked linear = still linear. Activation introduces non-linearity.
Backpropagation
Solves credit assignment - how much did each weight contribute to error?
One Sentence Mastery: "Neural networks stack nonlinear layers, using backprop to learn representations."
[Supervised Learning — Classification/Regression]

1. Why Neural Networks?

Neural networks are inspired by biological neurons in the brain. They're powerful because they can:

- **Learn automatically** from data without explicit programming
- **Handle non-linear relationships** that linear models can't capture
- **Universal approximators:** Can approximate ANY function to arbitrary precision (with enough neurons)

Universal Approximation Theorem (Hartman, Keeler, Kowalski, 1990):
"Only one level of hidden neurons is sufficient to approximate any function (with a finite number of discontinuities) with arbitrary precision, provided that the activation functions of the hidden neurons are non-linear."
This is a remarkable theoretical result! It means neural networks are incredibly flexible function approximators.

2. From Biology to Math: The Artificial Neuron

2.1 Biological Neurons (The Inspiration)

Real neurons in your brain have these key parts:

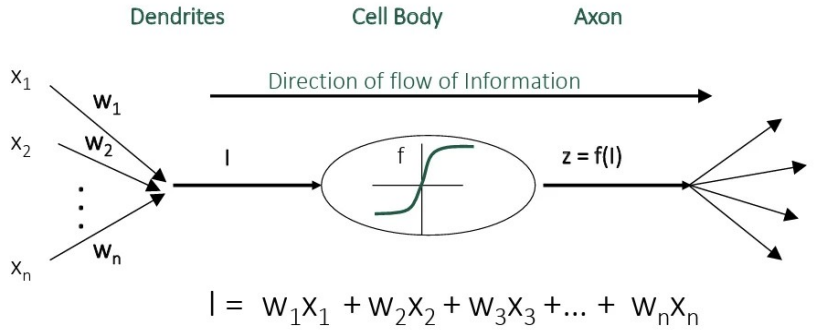
- **Dendrites:** Receive signals from other neurons (INPUTS)
- **Cell Body:** Processes the signals — if total stimulation exceeds threshold, neuron "fires" (PROCESSING)

- **Axon:** Carries output signal to other neurons (OUTPUT)
- **Synapse:** Connection between neurons — can be strong or weak (CONNECTION STRENGTH)

2.2 The Artificial Neuron (Perceptron)

We translate this biological concept into math:

Artificial neuron



- Receives Inputs x_1 x_2 ... x_p from other neurons or environment
- Inputs fed-in through connections with 'weights'
- Total Input = Weighted sum of inputs from all sources
- Transfer function (Activation function) converts the input to output
- Output goes to other neurons or environment

Figure 42: The Perceptron Model — An Artificial Neuron

How it works (step by step):

Step 1: Receive inputs

The neuron receives inputs x_1, x_2, ..., x_n (like dendrites receiving signals)

Step 2: Weight the inputs

Each input has a weight w_1, w_2, ..., w_n (like synapse strengths). Weights can be positive (excitatory) or negative (inhibitory).

Step 3: Sum the weighted inputs

$z = w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n = w_0 + \sum w_ix_i$

Where w_0 is the "bias" term (like a baseline threshold).

Step 4: Apply activation function

$y = a(z)$

The activation function transforms z into the final output. This is the "decision" of whether the neuron fires.

3. Activation Functions: The Heart of Neural Networks

The activation function is CRUCIAL. It's what makes neural networks able to learn non-linear patterns. Without it, a neural network would just be a fancy linear regression!

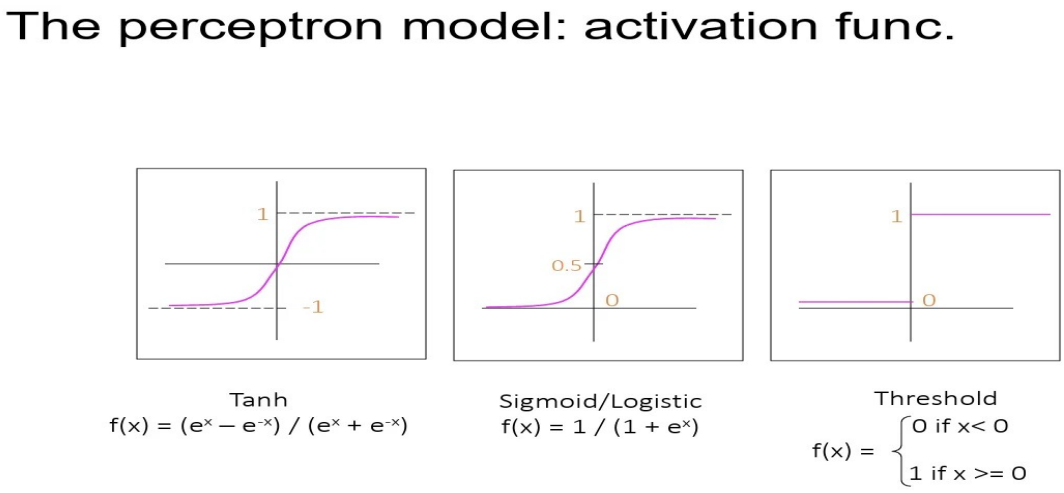


Figure 43: Common Activation Functions

3.1 The Sigmoid Function (Most Important to Understand!)

Formula: $\sigma(x) = 1 / (1 + e^{-x})$
What does this actually do? Let me explain step by step:

105. When x is a very large positive number (like +10): $e^{-10} \approx 0$, so $\sigma(10) = 1/(1+0) = 1$
106. When x is a very large negative number (like -10): $e^{10} \approx 22026$, so $\sigma(-10) = 1/(1+22026) \approx 0$
107. When x = 0: $e^{0} = 1$, so $\sigma(0) = 1/(1+1) = 0.5$

Key properties of the sigmoid:

- **Output range:** Always between 0 and 1 (perfect for probabilities!)
- **S-shaped curve:** Smooth transition from 0 to 1
- **Differentiable:** We can calculate its slope at any point (crucial for learning!)
- **“Squashes” any input:** No matter how big or small x is, output is always in [0,1]

Beautiful mathematical property: The derivative of sigmoid is: $\sigma'(x) = \sigma(x) \times (1 - \sigma(x))$
This makes calculations very efficient — once you compute σ(x), you can easily get its derivative!

3.2 Other Activation Functions

Tanh (Hyperbolic Tangent): $f(x) = (e^{x} - e^{-x}) / (e^{x} + e^{-x})$

- Output range: -1 to 1 (centered at zero)
- Often works better than sigmoid for hidden layers

Threshold (Step) Function: f(x) = 0 if x < 0, else 1

- Binary output: 0 or 1
- Problem: not differentiable at 0, so hard to use with gradient descent

ReLU (Rectified Linear Unit): f(x) = max(0, x)

- Most popular today for deep learning
- Output: 0 for negative inputs, x for positive inputs
- Faster to compute, avoids “vanishing gradient” problem

4. What Can a Single Perceptron Do?

A single perceptron can learn to compute simple logical functions. Let’s see how:

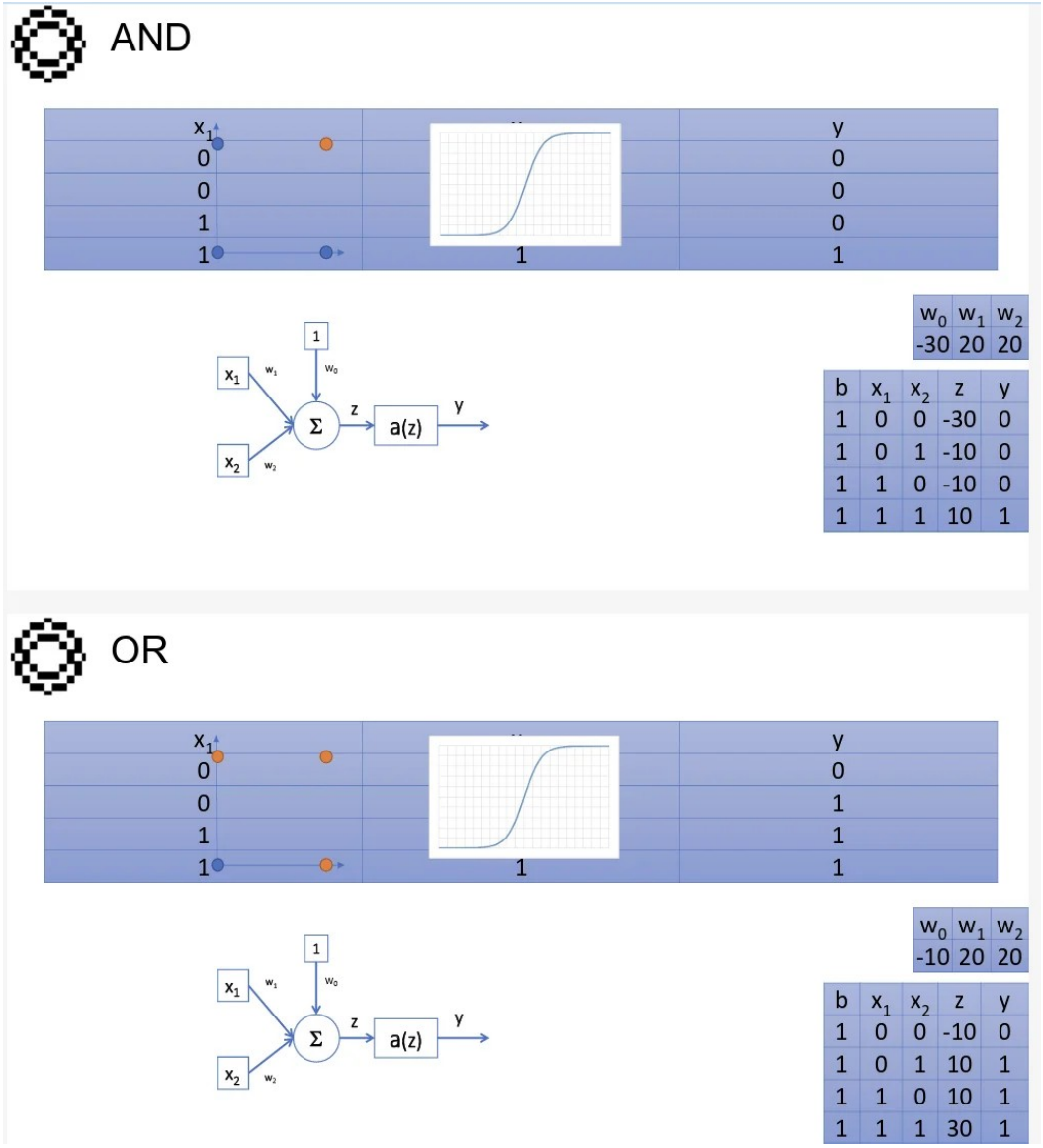


Figure 44: Single Perceptron Learning AND and OR Functions

4.1 AND Gate Example

AND returns 1 only when BOTH inputs are 1:

x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

Solution weights: $w_0 = -30$, $w_1 = 20$, $w_2 = 20$

Let's verify: $z = -30 + 20x_1 + 20x_2$

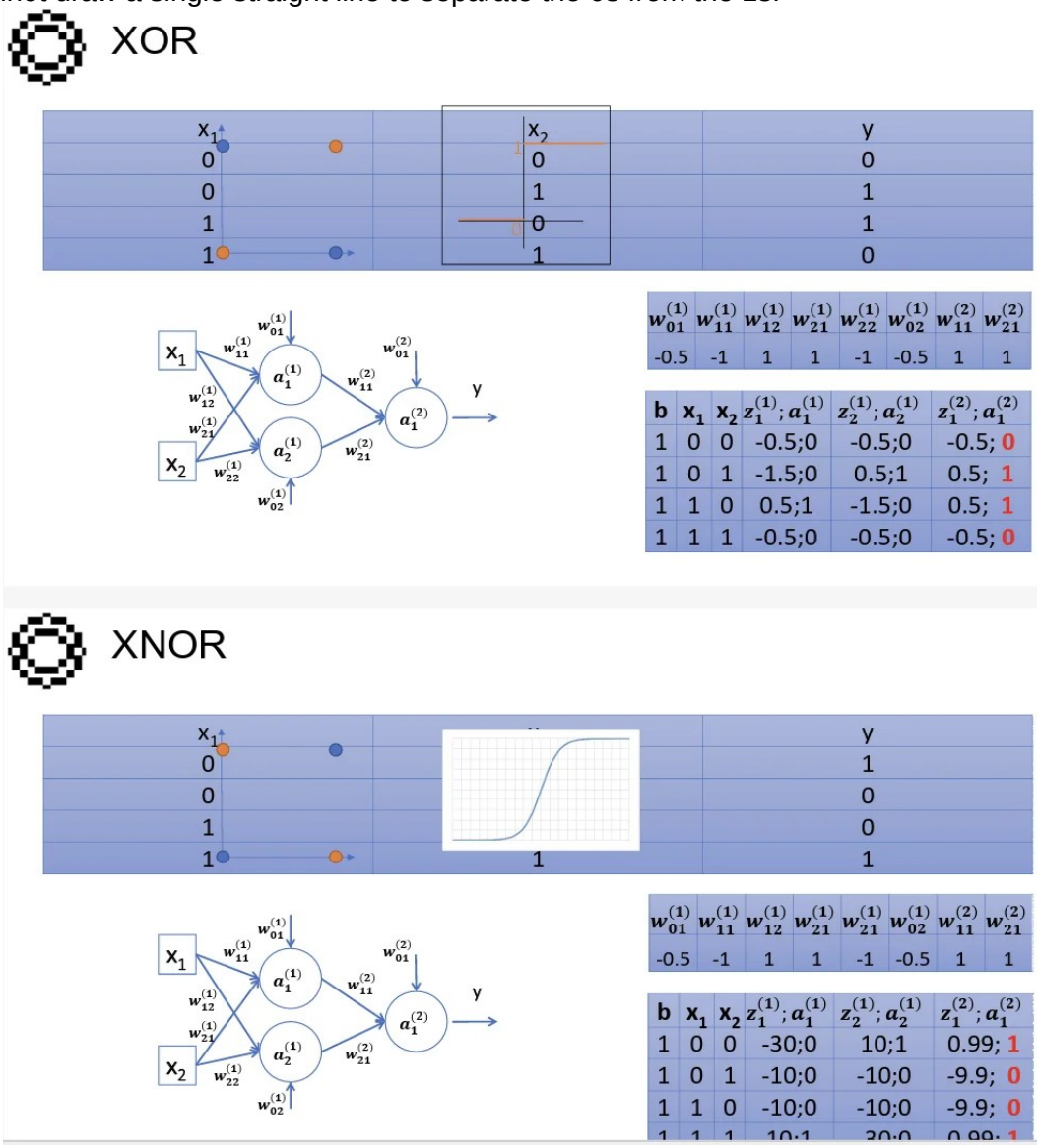
- (0,0): $z = -30 + 0 + 0 = -30 \rightarrow \sigma(-30) \approx 0$ ✓
- (0,1): $z = -30 + 0 + 20 = -10 \rightarrow \sigma(-10) \approx 0$ ✓
- (1,0): $z = -30 + 20 + 0 = -10 \rightarrow \sigma(-10) \approx 0$ ✓
- (1,1): $z = -30 + 20 + 20 = 10 \rightarrow \sigma(10) \approx 1$ ✓

4.2 The XOR Problem: Why We Need Multiple Layers

XOR (exclusive or) returns 1 when inputs are DIFFERENT:

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

A single perceptron CANNOT learn XOR! Why? Because XOR is not linearly separable — you cannot draw a single straight line to separate the 0s from the 1s.



$w_{01}^{(1)}$	$w_{11}^{(1)}$	$w_{12}^{(1)}$	$w_{21}^{(1)}$	$w_{22}^{(1)}$	$w_{02}^{(1)}$	$w_{11}^{(2)}$	$w_{21}^{(2)}$
-0.5	-1	1	1	-1	-0.5	1	1

b	x_1	x_2	$z_1^{(1)}; a_1^{(1)}$	$z_2^{(1)}; a_2^{(1)}$	$z_1^{(2)}; a_1^{(2)}$
1	0	0	-0.5;0	-0.5;0	-0.5; 0
1	0	1	-1.5;0	0.5;1	0.5; 1
1	1	0	0.5;1	-1.5;0	0.5; 1
1	1	1	-0.5;0	-0.5;0	-0.5; 0

$w_{01}^{(1)}$	$w_{11}^{(1)}$	$w_{12}^{(1)}$	$w_{21}^{(1)}$	$w_{21}^{(1)}$	$w_{02}^{(1)}$	$w_{11}^{(2)}$	$w_{21}^{(2)}$
-0.5	-1	1	1	-1	-0.5	1	1

b	x_1	x_2	$z_1^{(1)}; a_1^{(1)}$	$z_2^{(1)}; a_2^{(1)}$	$z_1^{(2)}; a_1^{(2)}$
1	0	0	-30;0	10;1	0.99; 1
1	0	1	-10;0	-10;0	-9.9; 0
1	1	0	-10;0	-10;0	-9.9; 0
1	1	1	10;1	30;0	0.99; 1

Figure 45: XOR Requires a Hidden Layer

Solution: Add a HIDDEN LAYER! With one hidden layer, we can solve XOR by combining simpler functions.

5. Multi-Layer Perceptron (MLP): The Full Neural Network

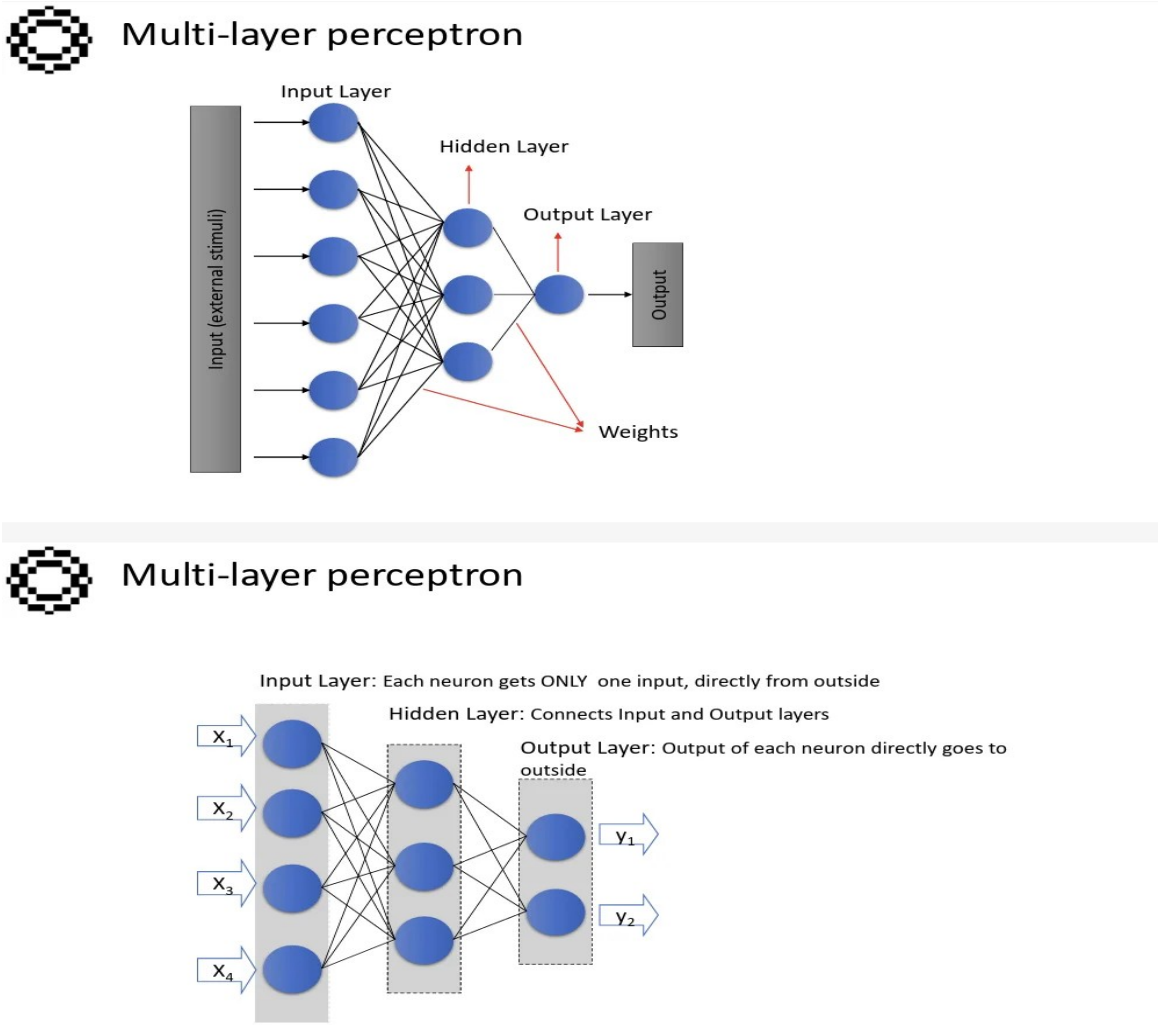


Figure 46: Multi-Layer Perceptron Architecture

5.1 The Three Types of Layers

Input Layer: One neuron per input feature. These just pass the input values through (no computation).
Hidden Layer(s): Where the “magic” happens! These learn intermediate representations. Each neuron connects to ALL neurons in the previous layer.
Output Layer: Produces the final prediction. For classification: one neuron per class.

5.2 More Layers = More Complex Decision Boundaries

Layers

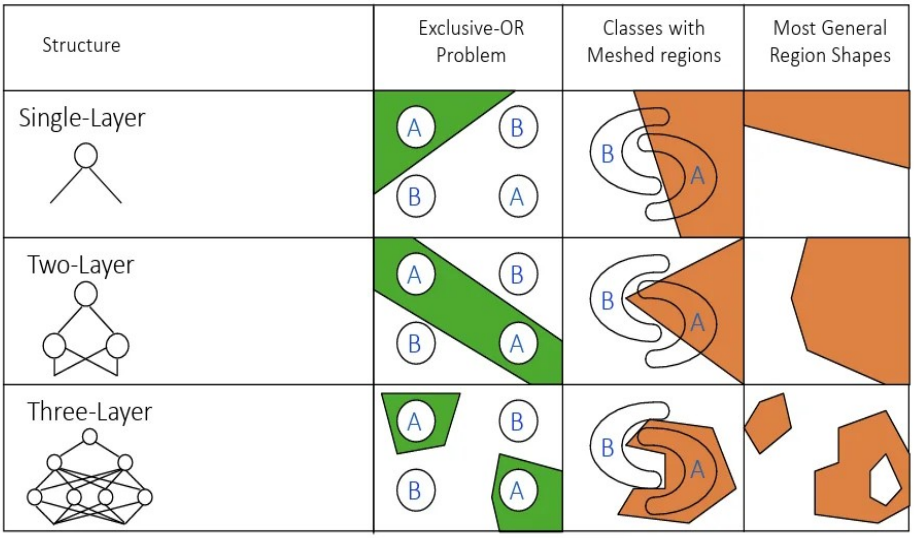


Figure 47: How Layers Affect Decision Boundary Complexity

- **Single layer:** Can only learn linear boundaries (like logistic regression)
- **Two layers:** Can learn convex regions (shapes without “dents”)
- **Three+ layers:** Can learn arbitrary complex shapes!

6. How Neural Networks Learn: Backpropagation

The key question: How do we find the right weights?

6.1 The Learning Principle

General idea:

- Weights that help achieve good results should be REINFORCED (made stronger)
- Weights that lead to bad results should be WEAKENED

6.2 Forward Propagation (Making a Prediction)

108. Input values enter at the input layer
109. Each neuron computes: $z = \sum w_i x_i + b$, then $a = \sigma(z)$
110. Outputs flow forward through the network layer by layer
111. Final output layer produces prediction

6.3 Backpropagation (Learning from Mistakes)

After making a prediction, we compare it to the true answer and calculate the ERROR. Then we “backpropagate” this error through the network to update weights.

The weight update rule:

$new\_w_i^{\check{}j} = old\_w_i^{\check{}j} + \alpha \times (target - output) \times x_i$

Where α (alpha) is the learning rate — how big of a step we take.

Breaking this down:

- **(target - output):** The error. If positive, we under-predicted. If negative, we over-predicted.
- **× x_i:** Weight change is proportional to the input. Bigger inputs get bigger weight changes.
- **× α:** Learning rate controls step size. Too big = overshoot. Too small = slow learning.

6.4 The Delta Rule (For Hidden Layers)

For output neurons, the error is simple: (target - output). But what about hidden neurons? They don't have a "target"!

For output layer unit j:

$Err_j = o_j \times (1 - o_j) \times (t_j - o_j)$

For hidden layer unit j:

$Err_j = o_j \times (1 - o_j) \times \sum(Err_k \times w_{jk})$

Hidden layer errors are computed by "backpropagating" the output errors through the weights!

Weight update: $\Delta w_{ij} = \alpha \times Err_j \times a_i$

6.5 The Error Surface and Local Minima

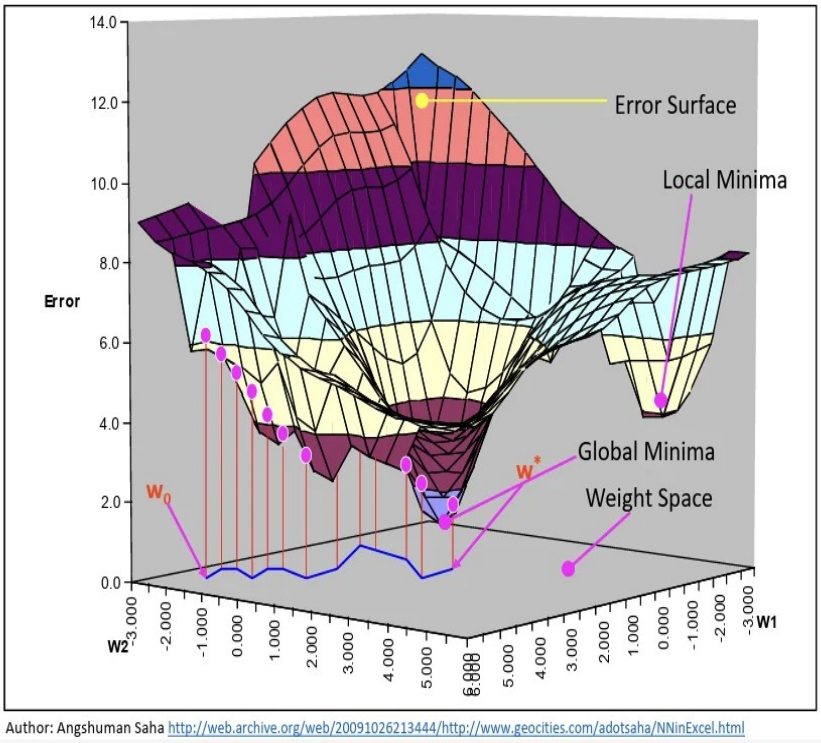


Figure 48: Error Surface with Local and Global Minima

Imagine the error as a mountainous landscape where height = error. Gradient descent is like a ball rolling downhill, trying to find the lowest point.

Global minimum: The absolute lowest error — the best possible solution.

Local minimum: A valley that's lower than its neighbors but not the absolute lowest. Gradient descent can get "stuck" here!

Strategies to avoid local minima:

- Multiple random initializations (try different starting points)

- Momentum (ball keeps rolling through small bumps)
- Learning rate schedules (start fast, slow down)

7. Key Takeaways for Lecture 8

- **Artificial neuron:** $z = \sum w_i x_i + b$, then $y = \text{activation}(z)$
- **Sigmoid function:** $\sigma(x) = 1/(1+e^{-x})$, squashes any input to [0,1]
- **Single perceptron:** Can only learn linearly separable functions (AND, OR, but NOT XOR)
- **Hidden layers:** Enable learning non-linear, complex decision boundaries
- **Universal approximation:** One hidden layer can approximate any function (in theory)
- **Forward propagation:** Input → hidden → output (making predictions)
- **Backpropagation:** Output → hidden → input (learning from errors)
- **Weight update:** new_w = old_w + α × error × input
- **Local minima:** Gradient descent can get stuck; use multiple initializations

8. Deep Dive: Understanding the Concepts Visually

This section provides additional visual explanations for the concepts that are often confusing.

8.1 Why Can’t a Single Line Separate XOR? (Linear Separability)

The term “linearly separable” means: Can you draw ONE straight line (or hyperplane in higher dimensions) that perfectly separates the two classes?

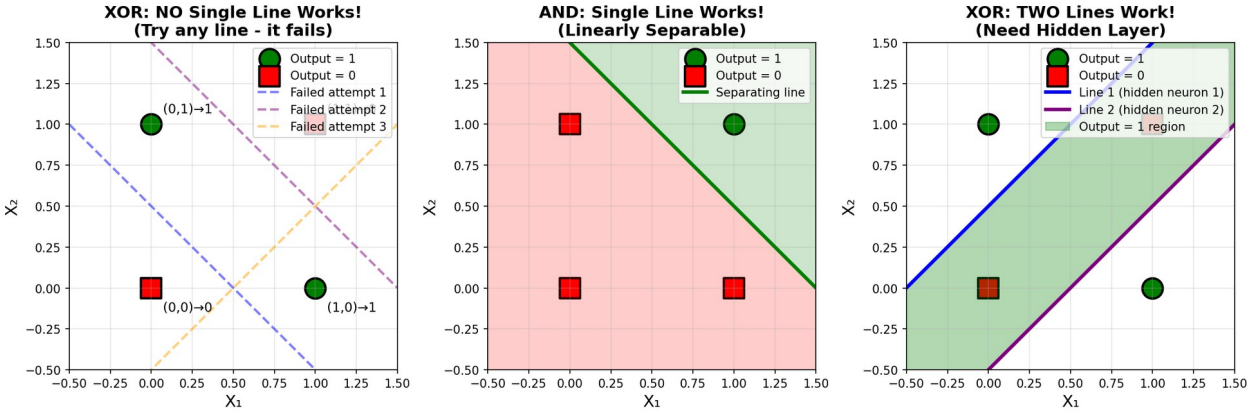


Figure 49: Visual Proof That XOR Cannot Be Solved With One Line

Look at the left plot: The green circles (output=1) are at positions (0,1) and (1,0). The red squares (output=0) are at (0,0) and (1,1). They form a diagonal pattern! No matter how you draw a single straight line, you will ALWAYS have at least one point on the wrong side.

Compare to AND (middle plot): For AND, the single green circle (1,1) is in the corner, and all red squares are clustered on the other side. ONE line easily separates them!

The solution (right plot): Use TWO lines! Together they create a “stripe” region. Points inside the stripe are class 1, points outside are class 0. Each line is computed by one hidden neuron!

8.2 How Exactly Does a Hidden Layer Solve XOR?

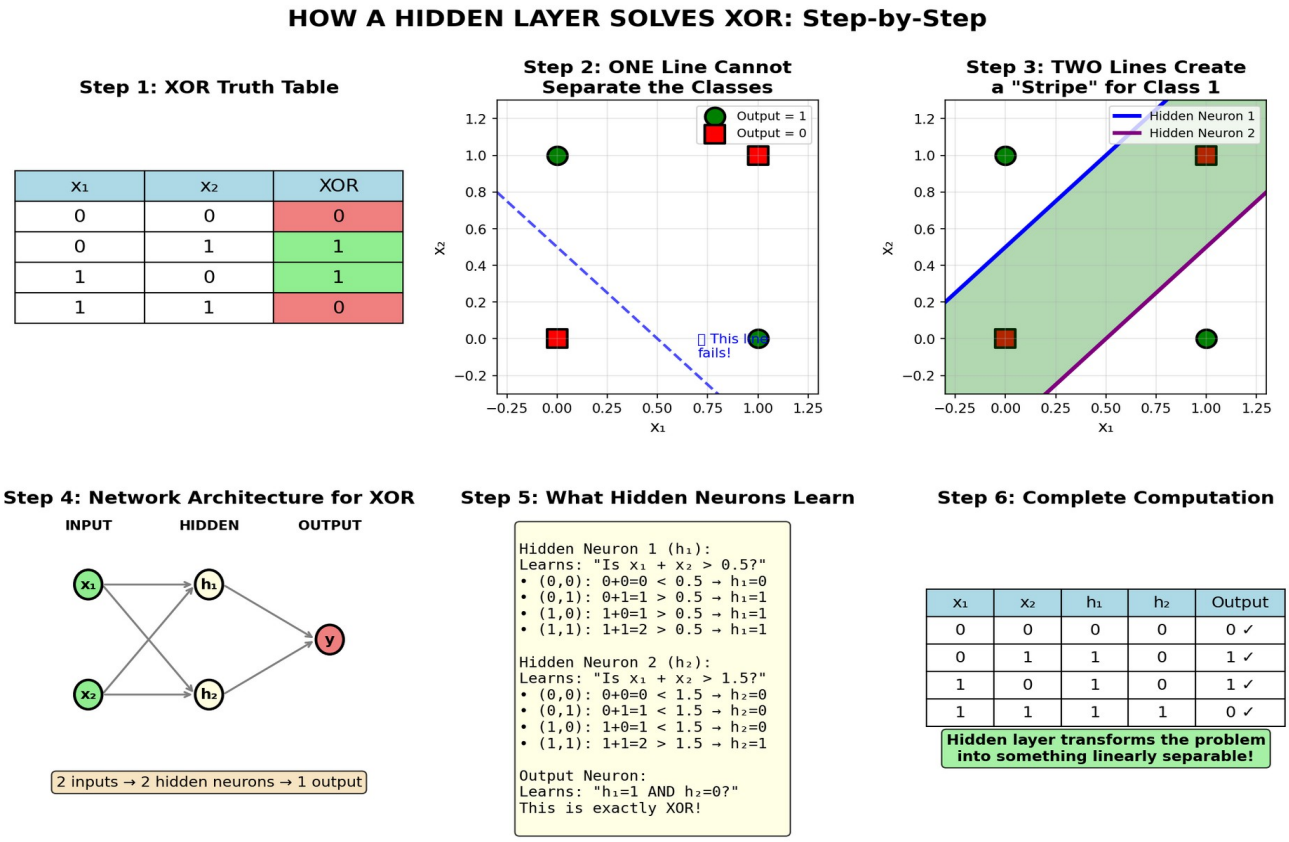


Figure 50: Step-by-Step: How Hidden Neurons Solve XOR

The key insight: The hidden layer TRANSFORMS the input space into a new space where the problem BECOMES linearly separable!

Hidden Neuron 1 learns to ask: "Is $x_1 + x_2$ greater than 0.5?" This fires (outputs 1) when at least one input is 1.

Hidden Neuron 2 learns to ask: "Is $x_1 + x_2$ greater than 1.5?" This fires only when BOTH inputs are 1.

Output Neuron combines these: Output 1 when h_1=1 AND h_2=0 (at least one input, but not both). This IS the XOR logic!

8.3 Is Multi-Layer Perceptron ONLY for XOR? NO!

XOR is just the SIMPLEST example to explain why hidden layers matter. In reality, MLPs are used for much more complex problems!

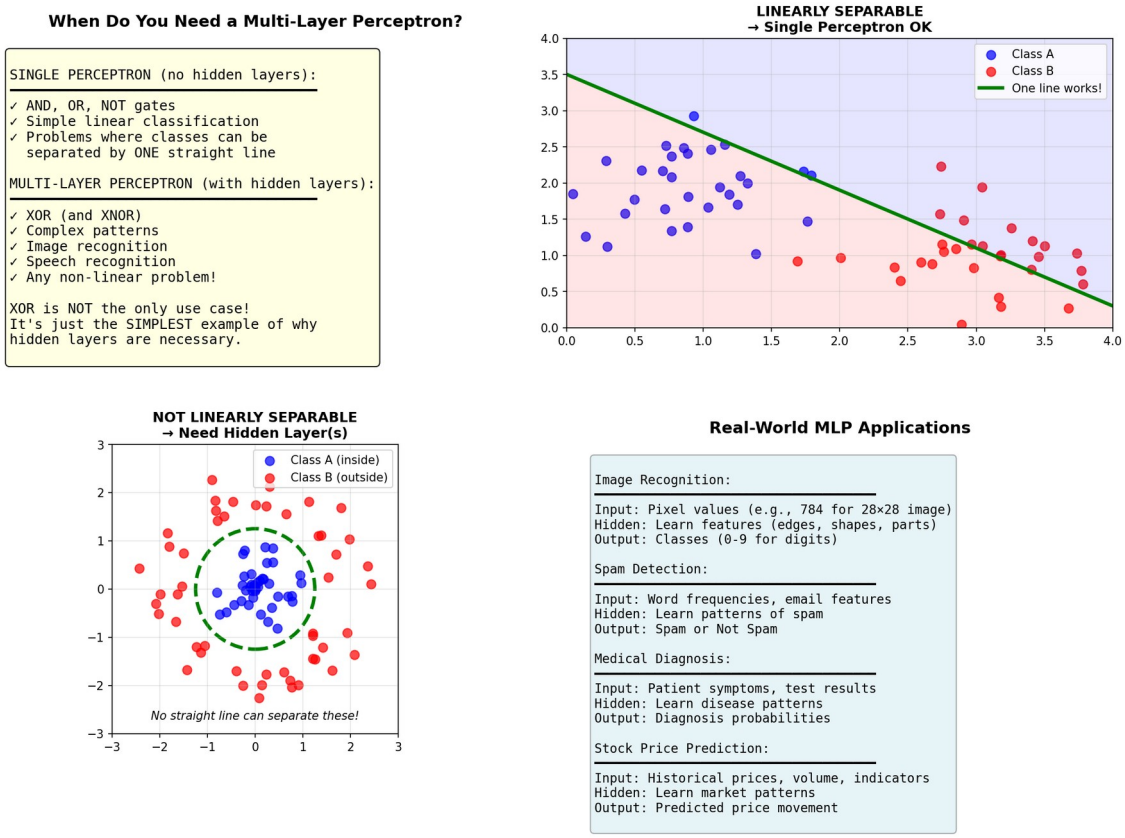


Figure 51: When Do You Need Hidden Layers? Real-World Applications

The rule: If your data has complex, non-linear patterns (like circles, spirals, or any shape you can't separate with a straight line), you need hidden layers!

8.4 Sigmoid: A Deeper Visual Explanation

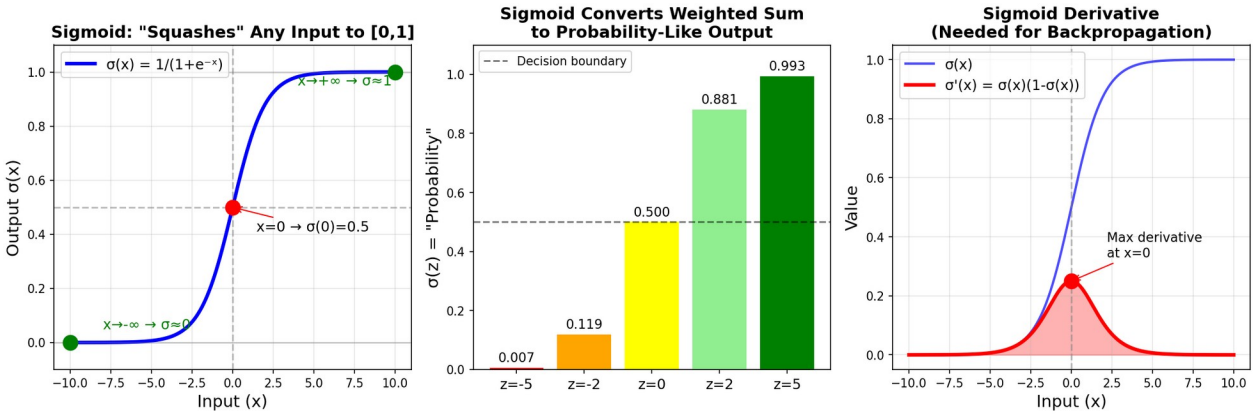


Figure 52: Understanding the Sigmoid Function Visually

Why does sigmoid matter?

- **Bounded output:** The weighted sum z can be ANY number (-1000, +50000, anything!). Sigmoid squashes it to always be between 0 and 1.
- **Probability interpretation:** Output can be interpreted as "probability of class 1". If output is 0.8, the network is 80% confident it's class 1.
- **Non-linearity:** Without sigmoid, stacking layers would just be multiplying matrices — still linear! Sigmoid adds the non-linearity that allows learning complex patterns.

- **Differentiable:** We can compute its derivative (σ’ = σ(1-σ)), which is essential for backpropagation.

8.5 Forward vs Backward Propagation Visualized

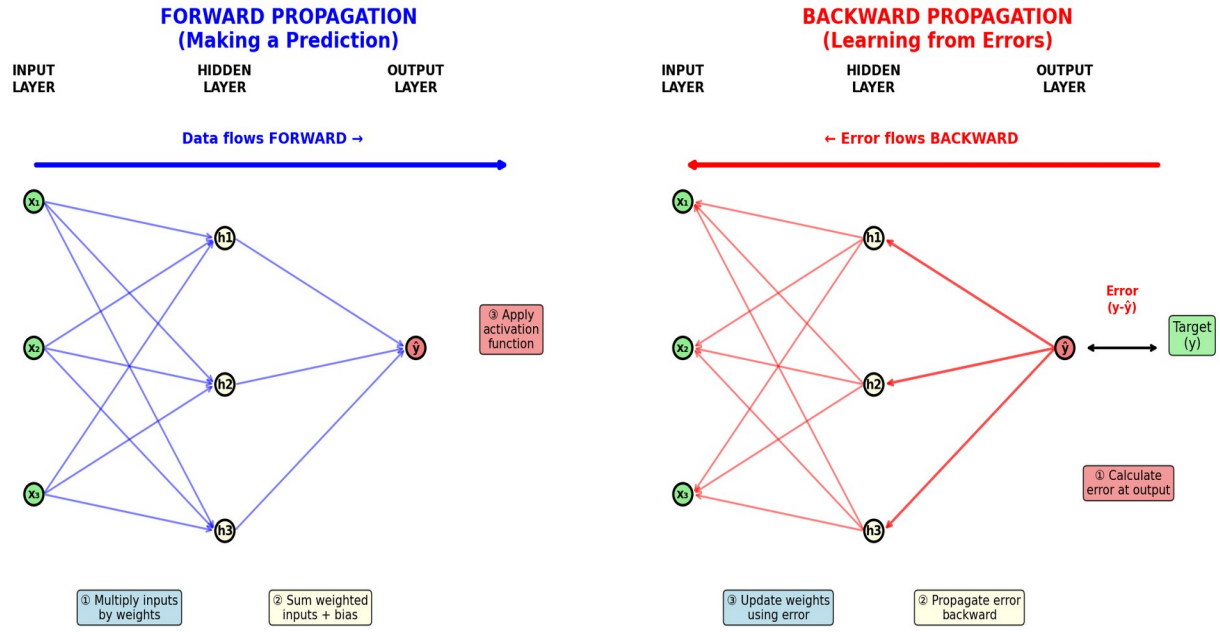


Figure 53: Forward Propagation (Prediction) vs Backward Propagation (Learning)

Forward Propagation is like taking an exam: You have inputs (questions), you process them through the network, and produce an output (answers).

Backward Propagation is like getting your exam back graded: You see the error (how wrong you were), and you trace back through your work to figure out where you went wrong and how to improve.

8.6 Error Formulas Explained Simply

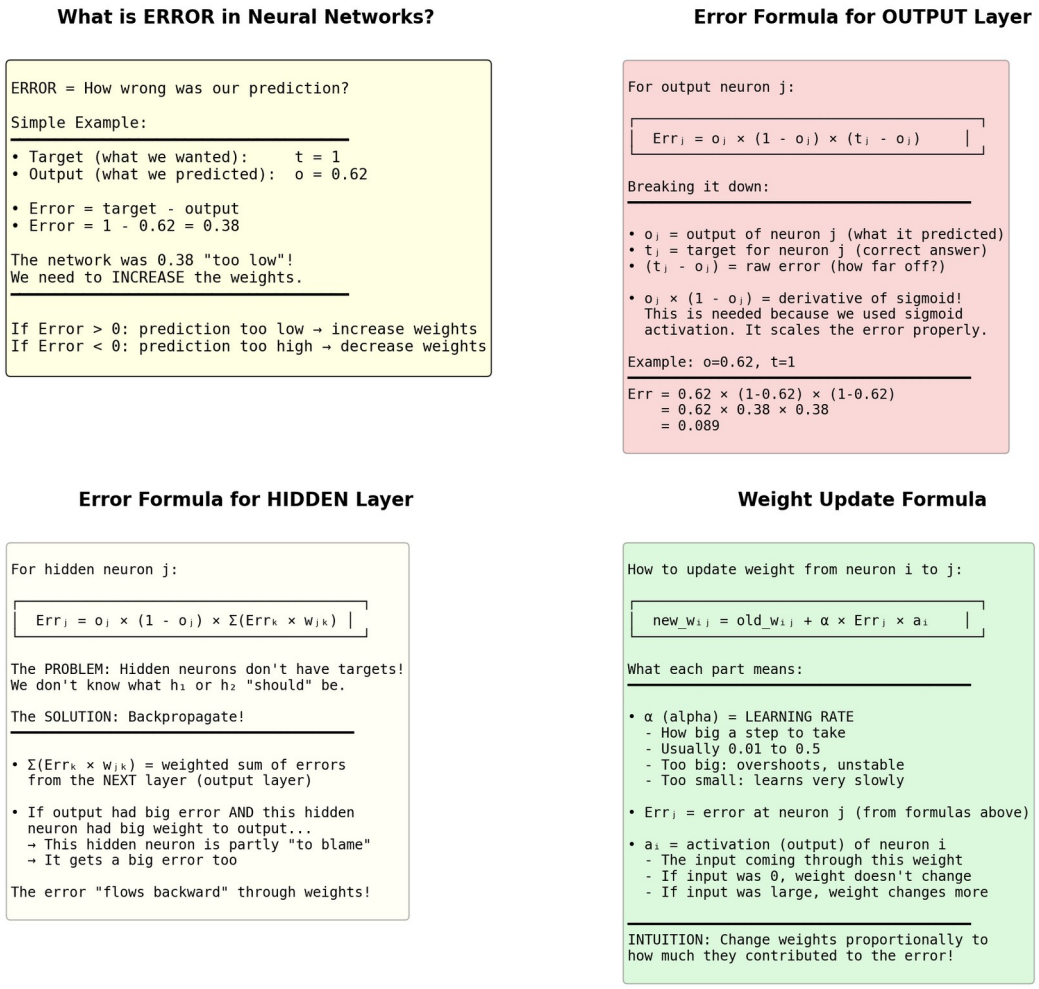


Figure 54: The Error Formulas Explained in Plain Language

The core idea:

112. **Error** = How wrong was the prediction? (target - output)
113. **× sigmoid derivative** = Scale by how “certain” the neuron was (neurons at extremes change less)
114. **Hidden layer error** = Weighted sum of errors from the next layer (blame is distributed proportionally)
115. **Weight update** = old weight + (learning rate × error × input)

Intuition: Weights that contributed more to the error get changed more. Weights with small inputs (near zero) barely change. The learning rate controls how big each step is.

8.7 Summary: The Big Picture

```
NEURAL NETWORK LEARNING CYCLE:
==========================================
1. Initialize weights randomly
2. FORWARD: Input → Hidden → Output (make prediction)
3. Calculate error (target - output)
4. BACKWARD: Propagate error back through network
5. Update weights using error and learning rate
6. Repeat steps 2-5 for many epochs until error is small
```

Lecture 9: Decision Trees

Deep Intuition: Learning to Ask the Right Questions
The Core Idea
Like "20 Questions" - find splits that separate classes well. Inherently interpretable.
Good Splits
Reduce uncertainty. GINI and Information Gain measure class-mixing reduction.
Overfitting Danger
Can split until one example per leaf - memorizes noise. Solution: limit depth, prune.
One Sentence Mastery: "Decision trees ask yes/no questions maximizing separation - must prune to avoid noise."
[Supervised Learning — Classification/Regression]

1. What is a Decision Tree?

A decision tree is like a flowchart for making decisions. It asks a series of questions about your data, and based on the answers, it leads you to a prediction. Think of it like the game “20 Questions” — each question narrows down the possibilities until you reach an answer.
Why are decision trees special?

- **Interpretable:** You can explain WHY a decision was made in plain English
- **Visual:** Can be drawn as a diagram that anyone can understand
- **Convertible to rules:** Can be expressed as IF-THEN statements or SQL queries

Decision Trees

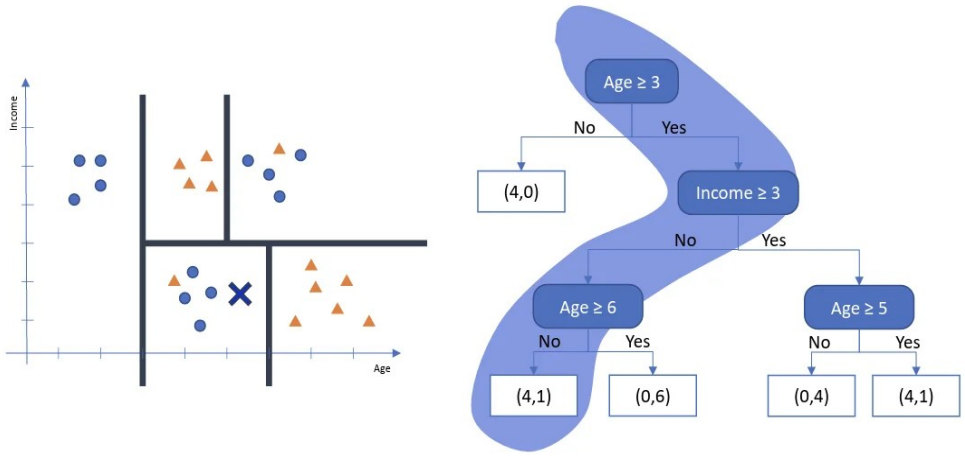


Figure 55: Decision Tree Example — Predicting Based on Age and Income

1.1 Anatomy of a Decision Tree

Root Node: The topmost node — the first question asked. Contains ALL training examples.
Internal Nodes: Decision points that test a feature (e.g., "Age ≥ 3?")
Branches: The outcomes of a test (Yes/No, or specific values)
Leaf Nodes: Terminal nodes that give the final prediction. Often shown as (count of class A, count of class B).
Reading the figure: The left plot shows data points (blue circles vs orange triangles) in Age-Income space. The tree on the right shows how the algorithm split this space using questions. The notation (4,1) means "4 examples of class 1, 1 example of class 2" in that leaf.

2. How Decision Trees Create Rules

Each path from root to leaf becomes an IF-THEN rule:

```
Rule 1: IF Age < 3 THEN Class = Blue (with 4 blue, 0 orange)
Rule 2: IF Age ≥ 3 AND Income < 3 AND Age < 6 THEN Class = Blue (4 blue, 1
orange)
Rule 3: IF Age ≥ 3 AND Income < 3 AND Age ≥ 6 THEN Class = Orange (0 blue, 6
orange)
Rule 4: IF Age ≥ 3 AND Income ≥ 3 AND Age < 5 THEN Class = Orange (0 blue, 4
orange)
Rule 5: IF Age ≥ 3 AND Income ≥ 3 AND Age ≥ 5 THEN Class = Blue (4 blue, 1
orange)
```

This is incredibly valuable for understanding WHY the model makes certain predictions!

3. Advantages and Disadvantages

3.1 Advantages

- **Interpretability:** Easy to understand and explain to non-technical stakeholders
- **Handles mixed data types:** Works with categorical, ordinal, and continuous features together
- **No scaling needed:** Unlike KNN or neural networks, features don't need normalization
- **Automatic feature selection:** The most important features appear near the top of the tree
- **Works for regression too:** Leaves can contain average values instead of class labels
- **Nonparametric:** Makes no assumptions about data distribution

3.2 Disadvantages

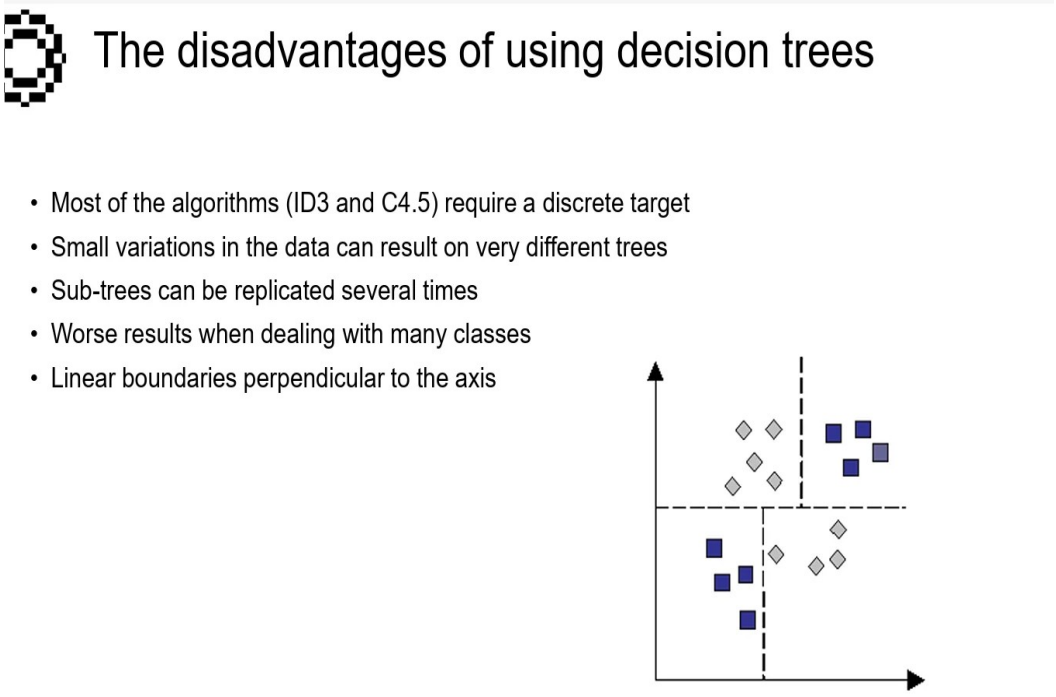


Figure 56: Disadvantage — Decision Trees Create Axis-Perpendicular Boundaries

- **Axis-perpendicular splits only:** Can only split parallel to axes (horizontal/vertical lines), not diagonal. This is a BIG limitation for diagonally-oriented data!
- **Unstable:** Small changes in data can result in completely different trees
- **Prone to overfitting:** Deep trees can memorize training data
- **Struggles with many classes:** Performance degrades with many target classes
- **Sub-tree replication:** Same patterns may be learned in multiple branches

4. Building a Decision Tree: The Key Questions

When building a tree, the algorithm must answer:

116. **Which feature to split on?** We want the feature that best separates the classes.
117. **Where to split?** For continuous features, what threshold value?
118. **When to stop?** When is the tree "good enough"?

4.1 The Greedy Algorithm

Decision trees are built using a GREEDY, TOP-DOWN, RECURSIVE approach:

```
1. Start with ALL examples at the root
2. Select the BEST feature to split on (using a metric)
3. Create child nodes for each outcome of the split
4. RECURSIVELY repeat steps 2-3 for each child node
5. STOP when a stopping condition is met
```

Stopping conditions:

- All examples in a node belong to the same class (pure node)
- No more features to split on (use majority voting)
- No examples left in a node
- Maximum depth reached (hyperparameter)
- Minimum samples per leaf reached (hyperparameter)

5. Measuring Split Quality: Purity Metrics

How do we know which split is “best”? We want splits that create PURE nodes (nodes where most examples belong to one class). Several metrics measure this:

5.1 Classification Accuracy (Simple Approach)

$f(a) = (1/n) \times \sum C_i$
Where C_i is the count of the most frequent class in each child node.
Example: If a split creates two nodes: Node A has (8 yes, 2 no) and Node B has (1 yes, 9 no):
Accuracy = (8 + 9) / 20 = 17/20 = 85%
This measures “dominance” or “purity” — how well the majority class dominates each node.

5.2 Entropy and Information Gain (Used by ID3, C4.5)

Entropy measures the “disorder” or “uncertainty” in a node:
$Entropy(S) = -\sum p_i \times \log_2(p_i)$
Where p_i is the proportion of class i in the node.
Entropy intuition:

- Entropy = 0: Perfect purity (all examples same class)
- Entropy = 1: Maximum uncertainty (50-50 split for binary)

Example calculation:
Node with 9 yes, 5 no (total 14):
p(yes) = 9/14, p(no) = 5/14
Entropy = $-(9/14)\log_2(9/14) - (5/14)\log_2(5/14) = 0.940$
Information Gain = reduction in entropy after a split:
$Gain(S, A) = Entropy(S) - \sum (|S_v|/|S|) \times Entropy(S_v)$
Choose the feature with the HIGHEST information gain!

5.3 Gini Impurity (Used by CART)

$Gini(S) = 1 - \sum p_i^2$
Gini intuition: Probability of misclassifying a randomly chosen element if labeled according to the class distribution.

- Gini = 0: Perfect purity
- Gini = 0.5: Maximum impurity (50-50 for binary)

Example: Node with 9 yes, 5 no:
Gini = $1 - (9/14)^2 - (5/14)^2 = 1 - 0.413 - 0.128 = 0.459$

6. Popular Decision Tree Algorithms

6.1 ID3 (Iterative Dichotomiser 3)

Created by: Quinlan, 1986
Split criterion: Information Gain (entropy-based)
Features:

- Only handles CATEGORICAL features
- Creates multi-way splits (one branch per category value)
- Does not handle missing values
- No pruning (tends to overfit)

Limitation: Biased towards features with many values (e.g., ID number would be selected first because each ID perfectly predicts the class!)

6.2 C4.5 (and C5.0)

Created by: Quinlan, 1993 (successor to ID3)
Split criterion: Gain Ratio (normalized information gain)
Gain Ratio = Information Gain / Split Information
This penalizes features with many values, solving ID3's bias problem.
Improvements over ID3:

- Handles CONTINUOUS features (finds best threshold automatically)
- Handles MISSING VALUES (distributes examples probabilistically)
- Includes PRUNING to prevent overfitting
- Can convert tree to IF-THEN rules

6.3 CART (Classification And Regression Trees)

Created by: Breiman et al., 1984
Split criterion: Gini Impurity (for classification) or MSE (for regression)
Key features:

- Always creates BINARY splits (yes/no questions only)
- Works for both CLASSIFICATION and REGRESSION
- Uses cost-complexity pruning
- Default algorithm in scikit-learn's DecisionTreeClassifier

For regression: Leaves contain the AVERAGE target value of examples that reach that leaf.

6.4 CHAID (Chi-Squared Automatic Interaction Detection)

Created by: Hartigan, 1975
Split criterion: Chi-squared statistical test
Key features:

- Uses statistical significance to determine splits
- Can create multi-way splits
- Stops splitting when chi-squared test is not significant
- Popular in market research and survey analysis

6.5 Algorithm Comparison

Algorithm	Split Type	Criterion	Continuous?	Missing?
ID3	Multi-way	Information Gain	No	No
C4.5	Multi-way	Gain Ratio	Yes	Yes
CART	Binary	Gini Impurity	Yes	Yes
CHAID	Multi-way	Chi-squared	Yes	Yes

7. Worked Example: Building a Decision Tree

Let's build a tree to predict whether someone will play tennis based on weather conditions.

Dataset:

Day	Outlook	Temp	Humidity	Wind	Play?
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
3	Overcast	Hot	High	Weak	Yes
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Strong	Yes
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

Step 1: Calculate overall entropy

Total: 14 examples (9 Yes, 5 No)

Entropy(S) = $-(9/14)\log_2(9/14) - (5/14)\log_2(5/14) = 0.940$

Step 2: Calculate information gain for "Outlook"

Sunny: 5 examples (2 Yes, 3 No) → Entropy = 0.971

Overcast: 4 examples (4 Yes, 0 No) → Entropy = 0 (pure!)

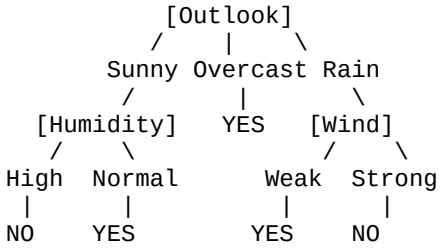
Rain: 5 examples (3 Yes, 2 No) → Entropy = 0.971

Weighted average: (5/14)×0.971 + (4/14)×0 + (5/14)×0.971 = 0.693

Gain(Outlook) = 0.940 - 0.693 = 0.247

After calculating for all features, Outlook has the highest gain, so it becomes the root!

Step 3: Result



8. Key Takeaways for Lecture 9

- **Decision trees:** Flowchart-like classifiers that ask questions about features
- **Main advantage:** Interpretable! Can explain predictions in plain English
- **Main limitation:** Only axis-perpendicular splits (no diagonal boundaries)
- **Entropy:** Measures disorder, 0 = pure, 1 = maximum uncertainty
- **Information Gain:** Entropy reduction after a split
- **Gini Impurity:** Alternative to entropy, used by CART
- **ID3:** Original algorithm, categorical only, uses Information Gain
- **C4.5:** Improved ID3 with continuous features, missing values, pruning
- **CART:** Binary splits, Gini impurity, works for regression too
- **CHAID:** Uses chi-squared test, popular in market research

9. Pruning: Preventing Overfitting

A fully grown tree can memorize noise in the training data (overfitting). Pruning helps create a more generalizable model.

9.1 Pre-pruning (Early Stopping)

Stop growing the tree early based on criteria:
• Maximum depth reached
• Minimum samples per node
• Information gain below threshold
Risk: May stop too early and miss important patterns.

9.2 Post-pruning

Grow a full tree first, then remove weak branches:
1. Build the complete tree
2. Evaluate each subtree on a validation set
3. Replace subtrees that don't improve validation accuracy with leaves
4. Stop when pruning no longer helps

9.3 Cost-Complexity Pruning (Weakest Link Pruning)

This is the standard pruning method used in CART (and scikit-learn). The key insight: we need to balance tree accuracy against tree complexity.

The Tree Score Formula

Tree Score = MSE + αT
Where:
• MSE = Mean Squared Error of the tree (lower = more accurate)
• T = Number of terminal nodes (leaves) in the tree
• α = Complexity penalty parameter (tuning parameter)
• αT = Tree Complexity Penalty (compensates for number of leaves)

Worked Example: Which Tree to Use?

Consider 5 trees of decreasing complexity:
• Full Tree: MSE = 0, T = 6 leaves (perfectly fits training data)
• Tree: MSE = 36.67, T = 5 leaves
• Sub-Tree 1: MSE = 77, T = 4 leaves
• Sub-Tree 2: MSE = 1013.34, T = 3 leaves
• Sub-Tree 3: MSE = 5368.89, T = 2 leaves
• Sub-Tree 4: MSE = 9171.68, T = 1 leaf (just predicts mean)
Notice: As we remove leaves, MSE increases (less accurate) but tree gets simpler.

Calculating Tree Scores (with α = 1000)

• Tree: 36.67 + 1000 × 5 = 5036.6
• Sub-Tree 1: 77 + 1000 × 4 = 4077
• **Sub-Tree 2: 1013.34 + 1000 × 3 = 4013.34 ← LOWEST (best!)**
• Sub-Tree 3: 5368.89 + 1000 × 2 = 7368.89
• Sub-Tree 4: 9171.68 + 1000 × 1 = 10171.68
We pick Sub-Tree 2 because it has the lowest Tree Score!

How to Find the Optimal α (Using Cross-Validation)

Step 1: Compute the Cost Complexity Pruning Path
• Train a fully grown decision tree on the training data
• Calculate the α value for each subtree that minimizes Tree Score
• This gives you a sequence of α values and corresponding trees
Step 2: Split Data for Cross-Validation
• Divide dataset into k folds (e.g., k=10)
• For each fold: use k-1 folds for training, 1 fold for validation
Step 3: Prune the Tree for Each α
• For each α value in your sequence:
  - For each fold: train tree, prune with this α, evaluate on validation set
  - Record the validation performance
Step 4: Average the Validation Scores
• For each α, compute the average validation score across all k folds
Step 5: Select the Optimal α
• Choose the α that maximizes average validation score (or minimizes validation loss)
Step 6: Train the Final Model
• Train decision tree on the FULL training dataset
• Prune with the optimal α to get your final model

Effect of α on Tree Size

• α = 0: Full tree (no penalty for complexity) → overfits
• α = small: Larger trees allowed
• α = large: Smaller trees preferred (heavy penalty for leaves)
• α = ∞: Single node (just predict mean/majority)

9.3 The Overfitting Pattern

As tree complexity increases:
• Training error decreases steadily (the tree memorizes)
• Validation error decreases then INCREASES (overfitting!)
The best tree is at minimum validation error, not minimum training error.

10. Handling Different Feature Types

10.1 Categorical Features

For a feature with values {Red, Blue, Green}:
Multiway split (ID3/C4.5): One branch per category value
Binary split (CART): Split into two groups, e.g., {Red, Blue} vs {Green}

10.2 Numeric Features

Numeric features use threshold splits: A ≤ t?
How to find the best threshold:
1. Sort all distinct values of the feature
2. Consider midpoints between adjacent values as candidates
3. Calculate information gain for each candidate threshold
4. Choose the threshold with highest gain
Example: For ages [18, 20, 22, 26], candidate thresholds are 19, 21, 24

11. Regression Trees

Decision trees can also predict continuous values (regression):
• Instead of predicting the majority class, predict the MEAN value in each leaf
• Instead of using entropy/Gini, minimize Mean Squared Error (MSE)
• Split criterion: Choose the split that minimizes the weighted MSE of child nodes
$MSE = (1/n) \Sigma(y_i - \bar{y})^2$
where $\bar{y}$ is the mean of the target values in the node.

Lecture 10: Ensemble Methods

Deep Intuition: The Wisdom of Crowds
Why Combining Helps
Uncorrelated errors cancel when averaged. Diverse opinions beat single experts.
Three Strategies:
BAGGING: Random samples, reduces variance (Random Forest)
BOOSTING: Sequential, focus on mistakes, reduces bias (XGBoost)
STACKING: Different models + meta-model combines them
One Sentence Mastery: "Ensembles combine diverse models so errors cancel - wisdom of crowds."
[Supervised Learning — Ensemble Classification/Regression]

1. What are Ensemble Methods?

An ensemble combines multiple models to produce better predictions than any single model alone. Theoretical and empirical studies have demonstrated that an ensemble of classifiers is typically more accurate than a single classifier.
Think of it like "The Wisdom of Crowds" - aggregating many opinions often beats relying on a single expert.

2. Three Types of Ensemble Learning

2.1 Bagging (Bootstrap Aggregation)

Train multiple models of the SAME TYPE on different subsets of training data, then combine predictions.
Key characteristics:
• Bootstrapping: Create diverse samples by randomly selecting data WITH replacement
• Parallel training: Train each model independently (can run in parallel)
• Aggregation: Combine via majority voting (classification) or averaging (regression)
Example: Random Forest - Many decision trees trained on bootstrap samples
Effect: Reduces VARIANCE

2.2 Boosting

Train weak models SEQUENTIALLY, with each new model focusing on the mistakes of previous models.
Key characteristics:
• Sequential training: Each model learns from errors of previous models
• Higher weights: Misclassified samples get higher weight in next iteration
• Uses weak learners: Simple models (e.g., decision stumps) combined into strong learner
Examples: AdaBoost, Gradient Boosting (GBoost), XGBoost
Effect: Reduces BIAS

2.3 Stacking

Train multiple DIFFERENT models, then use a meta-learner to combine their outputs.

Key characteristics:
• Diverse base learners: Use different algorithms (e.g., tree + neural net + SVM)
• Two-level learning: Level 1 trains base models, Level 2 learns how to combine them
• Meta-classifier: A model that takes base model outputs as inputs
Effect: Reduces BOTH bias and variance

3. Why Do Ensembles Work?

3.1 The Bias-Variance Decomposition

Expected Prediction Error = Bias² + Variance + Irreducible Error
• Bias: How far off predictions are on average (systematic error)
• Variance: How much predictions vary across different training sets
• Irreducible Error: Noise in the data that cannot be eliminated
How ensembles help:
• Bagging reduces variance (averaging smooths out fluctuations)
• Boosting reduces bias (iteratively corrects errors)
• Stacking reduces both (combines strengths of different models)

3.2 Dietterich's Three Reasons (2000)

1. Statistical Reason:
With limited data, many hypotheses may fit equally well. Averaging across multiple good classifiers gives a better approximation to the optimal classifier.
2. Computational Reason:
• Imperfect training: Algorithms may get stuck in local optima
• Too much data: Train on subsets and aggregate
• Too little data: Resampling creates more training sets
• Divide and conquer: Split complex problems into simpler ones
• Data fusion: Combine classifiers trained on different feature sets
3. Representational Reason:
The true decision boundary may not be representable by any single model class. Combining models can approximate boundaries that individual models cannot.

4. How to Set Up an Ensemble

Key decisions at four levels:

4.1 Combiner Level

How are individual outputs combined?
Non-trainable combiners: Simple majority voting, averaging
Trainable combiners: Weighted voting, Naive Bayes combiner
Meta-classifier: A learned model that takes base model outputs as features (stacked generalization)

4.2 Classifier Level

• Same or different classifiers?
• Which base classifier? (Decision trees are most common)
• How many classifiers needed?
• Independent or sequential training?

4.3 Feature Level

• Use all features or subsets?
• How to select/extract feature subsets?
Random Forest uses random feature subsets at each split.

4.4 Data Level

• Horizontal partitioning: Different subsets of rows for each classifier
• Vertical partitioning: Different subsets of features for each classifier
• Bootstrap sampling: Random sampling with replacement

5. Creating Diversity

Diversity is essential! If all models make the same predictions, combining them adds no value.
Ways to create diversity:
a. Different training approaches/parameters (e.g., different random initializations)
b. Manipulate training samples (bootstrap sampling, inducing label noise)
c. Different label targets (each classifier solves a different subtask)
d. Partition training data (horizontal: rows; vertical: features)
e. Use different classifier models (hybrid ensembles)

6. Overall Strategies

Classifier Fusion:
• Each ensemble member knows the WHOLE feature space
• All members contribute to every prediction
• Works with: Large ensemble size (L=100+), same classifier model, non-trained combiners
Classifier Selection:
• Each ensemble member specializes in PART of the feature space
• Only relevant members contribute to each prediction
• Works with: Small ensemble size (L=3-8), different classifier models, trained combiners

7. Types of Classifier Outputs

How much information does each classifier provide?
Type I (Abstract level): Just the predicted class label. No confidence information.
Type II (Ranked level): Ranked list of possible classes. Good for many-class problems.
Type III (Measurement level): Confidence scores/probabilities for each class. Most informative - enables sophisticated combination methods.

8. Seven Benefits of Ensemble Learning

1. Improved Accuracy: Typically better than any single model
2. Reduced Overfitting: Bagging reduces variance through averaging
3. Increased Robustness: If one model fails, others compensate
4. Versatility: Works across different algorithms and domains
5. Handles Complex Problems: Useful when no single model is optimal
6. Reduced Model Bias: Boosting corrects systematic errors
7. Imbalanced Data: Boosting focuses on misclassified (often minority class) instances

9. Bagging in Detail

Bootstrap Aggregation Process:
1. Create B bootstrap samples (random sampling WITH replacement)
2. Train a base model on each bootstrap sample (in parallel)
3. For prediction: aggregate all models
   - Classification: majority vote
   - Regression: average of predictions

Note: In bootstrap sampling, a value/instance CAN be repeated in a sample. On average, about 63% of original data appears in each bootstrap sample.

9.1 Random Forest

The most popular bagging algorithm. Uses decision trees with two sources of randomness:
1. Bootstrap sampling of training data
2. Random subset of features considered at each split

This double randomness creates diverse trees that make different errors, improving ensemble accuracy.

10. Key Takeaways for Lecture 10

Ensemble methods: Combine multiple models for better predictions
Three types: Bagging (parallel, reduces variance), Boosting (sequential, reduces bias), Stacking (meta-learner, reduces both)
Bias-Variance: Error = Bias² + Variance + Irreducible Error
Diversity is key: Identical models add no value when combined
Wisdom of crowds: Many simple models can beat one expert model
Random Forest: Bootstrap samples + random feature subsets with decision trees
Combiner types: Non-trainable (voting), Trainable (weighted), Meta-classifier (stacking)
==========

11. Voting Methods in Detail

11.1 Types of Voting

Unanimity: ALL classifiers must agree. Very strict - rarely achieved.
Simple Majority: More than half must agree.
Plurality: The class with the most votes wins. Ties resolved arbitrarily.

11.2 Majority Voting Accuracy (The Math!)

Assume L classifiers (L is odd), each with probability p of being correct, and classifiers are INDEPENDENT.
The ensemble is correct if at least floor(L/2) + 1 classifiers are correct.
Worked Example with L=3 classifiers:
Majority vote is correct if 2 or 3 classifiers are correct:
P(all 3 correct) = p x p x p = p cubed
P(exactly 2 correct) = 3 x p squared x (1-p)
P(majority correct) = p cubed + 3 x p squared x (1-p)
Example: If p = 0.6, then P(majority) = 0.216 + 3(0.36)(0.4) = 0.216 + 0.432 = 0.648, which exceeds 0.6!
Key insight: If p exceeds 0.5 (better than random), majority voting IMPROVES accuracy!

11.3 Weighted Majority Vote

If classifiers have different accuracies, give more weight to better classifiers.
Predict the class with highest weighted support, where weight b_i reflects classifier i's reliability.

11.4 Averaging for Regression

For regression: simply average the outputs: y_hat = (1/L) x sum of all predictions

12. Bootstrapping Explained

Create samples of size B from dataset Z of size N by randomly drawing WITH REPLACEMENT.
Key properties:
Same instance CAN appear multiple times in one sample
Some instances will not appear at all in a given sample
About 63% of original data appears in each bootstrap sample
Why 63%? The probability of NOT being selected in N draws is (1-1/N)^N, which approaches 1/e, which is about 0.37

13. Why Does Bagging Work?

Theorem (Lam and Suen, 1997):
If classifier outputs are INDEPENDENT and have the same individual accuracy p that exceeds 0.5, then majority voting is GUARANTEED to improve on individual performance.

13.1 Stable vs Unstable Learners

Unstable learners (like Decision Trees): Small changes in training data lead to big changes in model. Bagging helps a LOT.
Stable learners (like KNN): Small changes in training data lead to small changes in model. Bagging helps LESS.

This is why Decision Tree bagging ensembles (Random Forest) outperform KNN bagging ensembles!

14. Random Forest in Depth

Random Forest = Bagging + Random Feature Selection at each split

14.1 Two Sources of Randomness

1. Bootstrap sampling: Each tree trained on a different bootstrap sample
2. Random feature subsets: At each node split, only consider a random subset of features

14.2 Feature Subset Size

If there are X total features, at each split consider only x features where x is much smaller than X.
Common choices:
Classification: x = sqrt(X) (square root of total features)
Regression: x = X/3 (one-third of total features)

14.3 Random Forest Algorithm

For each tree: (a) Create bootstrap sample, (b) Grow tree - at each node randomly select K features, find best split among only those K, (c) NO PRUNING - grow tree fully
To predict: aggregate all trees (majority vote or average)

14.4 Why No Pruning?

High variance trees make different errors on different samples. Averaging cancels out these random errors. Pruning would make trees more similar, reducing diversity!

14.5 Ensemble Size Guidelines

Test accuracy improves until about 10-50 base estimators, then diminishing returns. Common defaults: 100-500 trees.

15. Updated Key Takeaways

Voting types: Unanimity, Simple Majority, Plurality
Majority voting math: If p exceeds 0.5 and classifiers are independent, ensemble beats individuals
Bootstrapping: Sample WITH replacement, about 63% of data appears in each sample
Unstable learners: Decision trees benefit most from bagging
Random Forest: Bootstrap + random feature subsets (usually sqrt(X) features per split)
No pruning in RF: Fully grown trees = high variance = more diversity = better ensemble

16. Boosting: From Weak to Strong

Core Idea: A set of "weak" learners (just slightly better than random guessing) can be "boosted" into a "strong" learner.

16.1 Key Characteristics of Boosting

Reduces BIAS (unlike bagging which reduces variance)
Base models are LOW VARIANCE but HIGH BIAS (simple models like decision stumps)

Sequential/Iterative: Each new model is influenced by performance of previous ones
New models become "experts" for instances misclassified by earlier models
Models complement each other - each fixes mistakes of previous ones

16.2 General Boosting Pseudocode

Input: Data distribution D, Base learner L, Number of iterations T
1. D_1 = D (initialize distribution - equal weights)
2. For t = 1 to T:
   3. h_t = L(D_t) - train weak learner from distribution D_t
   4. epsilon_t = Error of h_t - evaluate the error
   5. D_{t+1} = AdjustDistribution(D_t, epsilon_t) - update weights
6. Output: H(x) = CombineLearners() - weighted combination

17. AdaBoost (Adaptive Boosting)

Freund and Schapire (1997) - the most famous boosting algorithm

17.1 AdaBoost vs Random Forest

Random Forest:
- Creates FULL SIZE trees
- Each tree vote worth the SAME
- Order trees are built is IGNORED

AdaBoost:
- Creates STUMPS (trees with one node, two leaves)
- Stumps have DIFFERENT importance in final classification
- Each stump CONSIDERS previous stump mistakes

17.2 AdaBoost Step-by-Step (Worked Example)

Step 1: Initialize observation weights
Start with equal weights: w_i = 1/n for all n observations
Example: 10 observations, each gets weight 1/10 = 0.1
Step 2: Create a stump
Build a decision stump (one split) for each feature
Step 3: Calculate Gini index for each stump
Pick the stump with the LOWEST Gini (best split)
Step 4: Calculate the weight of this classifier (alpha)
First, calculate Total Error = sum of weights of misclassified samples
Example: 2 misclassified samples with weight 1/10 each: Total Error = 2/10 = 0.2
Alpha Formula:
alpha = (1/2) x ln((1 - TotalError) / TotalError)
Example: alpha = 0.5 x ln((1-0.2)/0.2) = 0.5 x ln(4) = 0.5 x 1.386 = 0.693

17.3 Understanding the Alpha Formula

The alpha value determines how much "say" this stump gets in the final vote:
- If TotalError is SMALL (good classifier): alpha is LARGE (more say)
- If TotalError is LARGE (bad classifier): alpha is SMALL or NEGATIVE (less say)
- If TotalError = 0.5 (random guessing): alpha = 0 (no say)
- If TotalError exceeds 0.5 (worse than random): alpha is NEGATIVE (reverse predictions!)

Step 5: Update observation weights

For MISCLASSIFIED samples (increase weight - focus on them next!):
new_weight = old_weight x e^(alpha)
Example: 0.1 x e^0.693 = 0.1 x 2.0 = 0.2
For CORRECTLY CLASSIFIED samples (decrease weight):
new_weight = old_weight x e^(-alpha)
Example: 0.1 x e^(-0.693) = 0.1 x 0.5 = 0.05
Then normalize all weights to sum to 1.
Step 6: Repeat for T iterations
Build next stump using updated weights (misclassified samples now matter more!)
Step 7: Final Prediction
H(x) = sign(sum of alpha_t x h_t(x) for all t)
Each stump votes, weighted by its alpha. The sign of the sum determines the class.

18. Gradient Boosting

Friedman (2001) - A more general and powerful boosting framework

18.1 Key Differences from AdaBoost

- Uses GRADIENTS of the loss function directly (not just weights)
- Each new model approximates the RESIDUAL (error) from previous iteration
- Flexible with ANY differentiable loss function
- Works for both classification AND regression

18.2 The Core Intuition

Loss Function (for regression): L = (1/2)(Observed - Predicted)^2
Gradient (derivative): dL/dPredicted = -(Observed - Predicted) = Predicted - Observed
Negative Gradient = Residual: Observed - Predicted
So gradient boosting for squared error is literally fitting to the residuals!

18.3 Gradient Boosting Algorithm

Input: Data, differentiable Loss function L
Step 1: Initialize with constant value
F_0(x) = value that minimizes sum of L(y_i, gamma)
For squared error loss, this is just the MEAN of all y values!
Step 2: For t = 1 to T iterations:
  a. Compute pseudo-residuals: r_it = -[dL/dF(x_i)] at F=F_{t-1}
  b. Fit a tree to these pseudo-residuals
  c. Calculate average residual for each leaf
  d. Update: F_t(x) = F_{t-1}(x) + learning_rate x (tree prediction)
Output: F_T(x) - sum of all trees

18.4 Gradient Boosting Worked Example (Regression)

Iteration 1 (t=1): Initialize
- Suppose mean of all target values = 73.3
- F_0(x) = 73.3 for all samples
- Calculate pseudo-residuals: r_i = y_i - 73.3
Example: If actual=64, residual = 64 - 73.3 = -9.3
Iteration 2 (t=2): First Tree
- Build a tree to predict the RESIDUALS (not original target!)

- For each leaf, calculate average of residuals in that leaf
- Update predictions: New_Pred = Old_Pred + learning_rate x Leaf_Value
Example with learning_rate=0.1: New_Pred = 73.3 + 0.1 x (-12.05) = 72.10
Iteration 3+ (t=3,...): Keep Adding Trees
- Recalculate residuals based on NEW predictions
- Build another tree on these new residuals
- Update predictions again
Each iteration, residuals get smaller - predictions get better!

18.5 Why Use a Learning Rate?

Taking SMALL steps (learning_rate = 0.1) instead of big steps:
- Results in better predictions on TEST data
- Lowers variance (less overfitting)
- Requires more iterations, but more robust
Trade-off: smaller learning rate needs more trees

18.6 Popular Gradient Boosting Implementations

XGBoost (Chen and Guestrin, 2016): L1 and L2 regularization, handles missing values
LightGBM (Ke et al., 2017): Leaf-wise (best-first) growth, very fast
CatBoost (Prokhorenkova et al., 2018): Handles categorical features well

19. Stacking (Stacked Generalization)

Uses a META-LEARNER to combine diverse base models optimally.

19.1 Key Characteristics

- Uses DIVERSE set of learning algorithms as base learners (unlike bagging/boosting)
- Combiner is a TRAINED model (meta-learner), not just voting or averaging
- Two levels: Level 1 (base learners) and Level 2 (meta-learner)

19.2 How Stacking Works (Two Levels)

Level 1 - Train Base Learners:
Train multiple DIFFERENT models on original training data
Example: Random Forest, SVM, Neural Network, KNN
Level 2 - Train Meta-Learner:
Use base learner PREDICTIONS as new features
Keep original labels as target
Train a meta-classifier to learn how to combine base predictions

19.3 The Overfitting Problem

Problem: If we use the SAME data to train base learners AND create the meta-learner training set, we risk overfitting!
The base learners have already "seen" the training data, so their predictions on it are overly optimistic.

19.4 Solution: Stacking with Cross-Validation

1. Split training data into K folds
2. For each fold:
   - Train base learners on K-1 folds

- Get predictions on the held-out fold
3. These out-of-fold predictions become training data for meta-learner
4. Train meta-learner on these predictions + true labels
5. Retrain base learners on ALL data for final model
This way, meta-learner sees predictions on data the base learners did NOT train on!

19.5 Stacking Benefits

- Leverages diverse base models (each captures different patterns)
- Meta-learner learns OPTIMAL way to combine them
- Often achieves superior performance to any single model
- Can reduce both bias AND variance

20. Challenges and Open Problems in Ensembles

Quantifying Diversity: How do we measure if our models are diverse enough?
Generating Diverse Models: Computationally expensive, especially with deep learning
Adaptive Ensembling: Dynamic adaptation to changing data or model performance
Interpretability: Ensembles are harder to explain than single models

21. Complete Ensemble Summary

BAGGING (Random Forest):
- Parallel training, bootstrap samples, reduces VARIANCE
- Best with unstable learners (decision trees)
BOOSTING (AdaBoost, Gradient Boosting):
- Sequential training, focus on mistakes, reduces BIAS
- AdaBoost: weight updates, alpha formula
- Gradient Boosting: fits to residuals, uses gradients
STACKING:
- Different algorithms, meta-learner combines them
- Use cross-validation to avoid overfitting
- Reduces both bias AND variance

Lecture 11: Support Vector Machines (SVM)

Deep Intuition: Maximum Margin Classification
Why Maximum Margin?
Barely-separating line is fragile. Wide margin = robust.
The Math
Margin = 2/||w||. Max margin = min ||w||. SVM is inherently regularized.
Support Vectors
Only margin points matter. Most points irrelevant. Robust to noise elsewhere.
Kernel Trick
Non-separable? Kernels implicitly map to higher dimensions where it becomes separable.
One Sentence Mastery: "SVM finds maximum-margin boundary using support vectors, kernels for non-linear data."
[Supervised Learning — Classification]

1. What is an SVM?

Support Vector Machine (SVM) is a supervised machine learning algorithm used for both classification and regression.
Key idea: Find the hyperplane that best separates the classes with the MAXIMUM MARGIN.
Each data item is a point in n-dimensional space (one dimension per feature).
Classification is performed by finding the hyperplane that differentiates the classes.

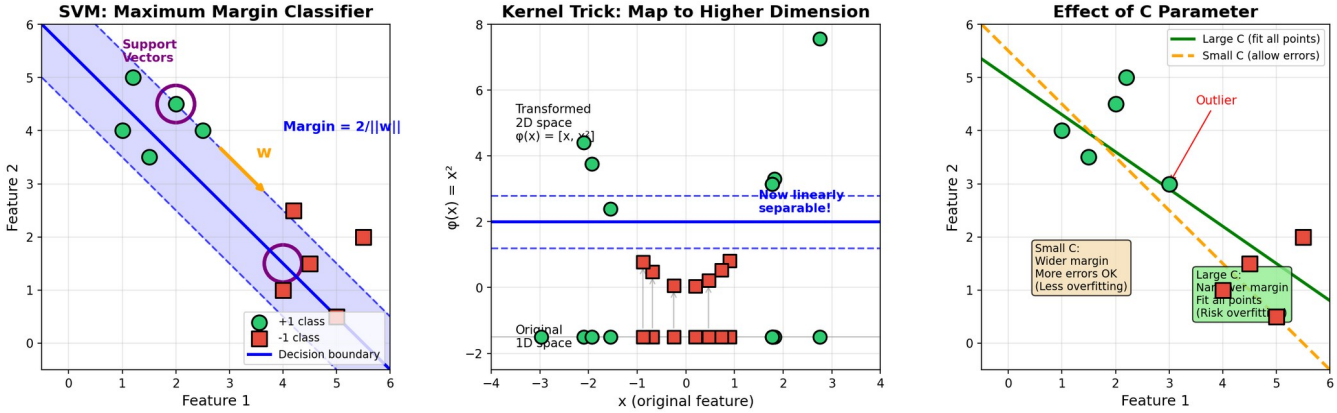


Figure: SVM Concepts - Maximum Margin, Kernel Trick, and C Parameter Effect

2. The Learning Problem

Given:
Training data: inputs x_i in R^n, labels y_i in {-1, +1}
Goal: Learn a classifier that generalizes well to new data.
Linear Decision Function:
f(x) = w dot x + b
Prediction Rule:
If f(x) is at least 0, predict class +1
If f(x) is less than 0, predict class -1

3. Why Not Just Any Separating Line?

Many hyperplanes can separate the data. Different classifiers (KNN, Decision Tree, Neural Network) find different boundaries.
SVM chooses the one that MAXIMIZES THE MARGIN - the distance to the closest points from either class.
Why maximize margin? It improves generalization and reduces overfitting.

4. The Geometric Intuition

Think of the separating hyperplane as a "street" with the decision boundary as the median line.
The weight vector w:
- Is PERPENDICULAR to the decision boundary (median line)
- Points toward the positive class side
To classify a new point u:
- Project u onto w (compute dot product w dot u)
- Check if w dot u + b is at least 0 (positive side) or less than 0 (negative side)

5. Encoding Correct Classification

For correct classification, each point must satisfy:
y_i (w dot x_i + b) is greater than 0
Problem: Multiplying w and b by any constant gives the same classifier but different values.
Solution: To remove scale ambiguity, we impose a constraint:
For positive samples: w dot x_+ + b is at least 1
For negative samples: w dot x_- + b is at most -1
Combined (using y_i):
y_i (w dot x_i + b) is at least 1 for all i
Points that satisfy equality (y_i (w dot x_i + b) = 1) are the SUPPORT VECTORS - they lie on the margin boundary.

6. Calculating the Margin Width

Step 1: Distance from a point to the hyperplane
distance = |w dot x + b| / ||w||
Step 2: For support vectors on the margin
|w dot x + b| = 1 (by our constraint)
So distance to each margin = 1 / ||w||
Step 3: Total margin width (street width)
Taking the difference between a positive and negative support vector:
width = (x_+ - x_-) dot (w / ||w||)
Since (w dot x_+) = 1 - b and (w dot x_-) = -1 - b:
width = [(1-b) - (-1-b)] / ||w|| = (1-b+1+b) / ||w|| = 2 / ||w||
KEY RESULT: Margin width = 2 / ||w||

7. The Optimization Problem

Goal: Maximize margin = 2/||w||
Equivalent to: Minimize ||w||
For mathematical convenience: Minimize (1/2)||w||^2
Hard-Margin SVM Primal Formulation:
Minimize: (1/2)||w||^2
Subject to: y_i (w dot x_i + b) is at least 1 for all i
This is a CONVEX optimization problem - no local minima, guaranteed global solution!

8. Solving with Lagrange Multipliers

To solve constrained optimization, we build the Lagrangian:
L(w, b, alpha) = (1/2)||w||^2 - Sum_i alpha_i [y_i(w dot x_i + b) - 1]
Where alpha_i is at least 0 are Lagrange multipliers (one per constraint)
Step 1: Differentiate with respect to w
dL/dw = w - Sum_i alpha_i y_i x_i = 0
Therefore: w = Sum_i alpha_i y_i x_i
Step 2: Differentiate with respect to b
dL/db = -Sum_i alpha_i y_i = 0
Therefore: Sum_i alpha_i y_i = 0

9. Key Insights from Optimization

Insight 1: w is a LINEAR COMBINATION of training points:
w = Sum_i alpha_i y_i x_i
Insight 2: Points with alpha_i greater than 0 are SUPPORT VECTORS
All other points (alpha_i = 0) are irrelevant to the final boundary!
Insight 3: The classifier depends only on DOT PRODUCTS:
f(x) = Sum_i alpha_i y_i (x_i dot x) + b
This is the foundation of the KERNEL TRICK!

10. The Dual Formulation

Substituting w = Sum_i alpha_i y_i x_i back into L:
Dual Problem:
Maximize: L = Sum_i alpha_i - (1/2) Sum_i Sum_j alpha_i alpha_j y_i y_j (x_i dot x_j)
Subject to: alpha_i is at least 0 and Sum_i alpha_i y_i = 0
Notice: The optimization depends ONLY on dot products of pairs of samples (x_i dot x_j)

11. The Kernel Trick

Since the algorithm only uses dot products, we can replace them with a KERNEL FUNCTION:
K(x_i, x_j) = phi(x_i) dot phi(x_j)
where phi(x) maps x to a higher-dimensional space.
Magic: We NEVER explicitly compute phi(x)! The kernel computes the dot product directly.
The classifier becomes: f(x) = Sum_i alpha_i y_i K(x_i, x) + b
Common Kernels:
Linear: K(x,z) = x dot z
Polynomial: K(x,z) = (x dot z + c)^d
RBF (Gaussian): K(x,z) = exp(-gamma ||x-z||^2)
RBF kernel can map to INFINITE dimensional space!

12. Soft Margin SVM and the C Parameter

Real data is rarely perfectly separable. We introduce SLACK VARIABLES xi_i.
Soft-Margin Formulation:
Minimize: (1/2)||w||^2 + C Sum_i xi_i
Subject to: y_i(w dot x_i + b) is at least 1 - xi_i, and xi_i is at least 0
Interpretation of xi_i:
xi_i = 0: Point is correctly classified outside margin
0 less than xi_i less than 1: Point is inside margin but correctly classified
xi_i greater than 1: Point is misclassified
The C Parameter:
C controls the trade-off between:
- Large margin (small ||w||)
- Few classification errors (small sum of xi_i)
Large C: Penalize errors heavily, smaller margin, risk of OVERFITTING (low bias, high variance)
Small C: Allow more errors, larger margin, risk of UNDERFITTING (high bias, low variance)

13. The Gamma Parameter (RBF Kernel)

For RBF kernel K(x,z) = exp(-gamma ||x-z||^2), gamma controls the influence reach of each support vector.
Small gamma: Smooth, global influence, simpler decision boundary
Large gamma: Very local influence, complex decision boundary
Typical combinations:
High gamma + high C: Strong overfitting risk
Low gamma + low C: Strong underfitting risk

14. Worked Example

Data: x = {(3,4), (4,1), (6,6)}, y = {+1, +1, -1}
Primal Problem:
Minimize (1/2)(w1^2 + w2^2)
Subject to:
+1 x (w dot [3,4] + b) is at least 1
+1 x (w dot [4,1] + b) is at least 1
-1 x (w dot [6,6] + b) is at least 1
Expanding: 3w1 + 4w2 + b is at least 1, 4w1 + w2 + b is at least 1, -6w1 - 6w2 - b is at least 1
This can be solved using quadratic programming to find optimal w and b.

15. SVM Summary - The Complete Picture

1. Decision Function: f(x) = w dot x + b
2. Constraint: y_i(w dot x_i + b) is at least 1 (to fix scale)
3. Margin: width = 2/||w||
4. Objective: Minimize (1/2)||w||^2 (maximize margin)
5. Lagrangian: Reveals w = Sum alpha_i y_i x_i
6. Support Vectors: Points with alpha_i greater than 0 (on the margin)
7. Dot Products: Classifier depends only on dot products, enabling kernel trick
8. Kernels: K(x,z) = phi(x) dot phi(z) - implicit high-dimensional mapping
9. Soft Margin: C parameter controls margin/error trade-off
10. RBF gamma: Controls locality of influence

16. What You Should Be Able to Explain

"I define a linear classifier f(x) = w dot x + b. I encode correct classification as y_i(w dot x_i + b) is at least 1 to fix scale. The geometric margin equals 2/||w||, so maximizing margin means minimizing (1/2)||w||^2. This leads to a convex constrained optimization problem. Using Lagrange multipliers, I derive the dual and obtain w as a weighted sum of support vectors. The classifier depends only on dot products, enabling the kernel trick. The C parameter controls regularization and margin violations. Kernel parameters like gamma control decision boundary complexity."

17. Complete Worked Example (Solving the System)

Data: x = {(3,4), (4,1), (6,6)}, y = {+1, +1, -1}
Step 1: Set up constraints
From y_i(w dot x_i + b) at least 1:
Point (3,4), y=+1: 3w1 + 4w2 + b at least 1

Point (4,1), y=+1: 4w1 + w2 + b at least 1
Point (6,6), y=-1: -6w1 - 6w2 - b at least 1
Step 2: Change signs for standard form (less than or equal)
-3w1 - 4w2 + b at most -1
-4w1 - w2 + b at most -1
6w1 + 6w2 - b at most -1
Step 3: Build and solve the Lagrangian
L(w,b,alpha) = (1/2)(w1^2 + w2^2) + alpha1(-3w1 - 4w2 + b + 1) + alpha2(-4w1 - w2 + b + 1) + alpha3(6w1 + 6w2 - b + 1)
Step 4: Calculate gradient and set to zero
dL/dw1 = w1 + alpha1(-3) + alpha2(-4) + alpha3(6) = 0
dL/dw2 = w2 + alpha1(-4) + alpha2(-1) + alpha3(6) = 0
dL/db = alpha1 + alpha2 + alpha3 = 0
Plus KKT conditions for each constraint
Step 5: Solve the system (matrix form)
Setting alpha2 = 0 (point (4,1) is not a support vector), solve:
Solution:
w1 = -0.4615
w2 = -0.3076
b = -3.6153
alpha1 = 0.1538
alpha2 = 0 (not a support vector)
alpha3 = 0.1538
Step 6: Final hyperplanes
Decision boundary: -0.4615x1 - 0.3076x2 = -3.6153
Positive margin: -0.4615x1 - 0.3076x2 = -3.6153 + 1
Negative margin: -0.4615x1 - 0.3076x2 = -3.6153 - 1
Notice: Only points (3,4) and (6,6) are support vectors (alpha greater than 0). Point (4,1) has alpha=0.

18. Soft Margin in Depth

The Problem: Real data is rarely perfectly linearly separable.
The Solution: Introduce slack variables xi_i that allow some points to violate the margin.
Soft Margin Objective:
Minimize: (1/2)||w||^2 + C x (number of mistakes)
More precisely: (1/2)||w||^2 + C x Sum_i xi_i
What xi_i measures:
xi_i = 0: Point correctly classified, outside margin
0 less than xi_i less than 1: Point inside margin but on correct side
xi_i = 1: Point exactly on decision boundary
xi_i greater than 1: Point misclassified (on wrong side)
The C Parameter (Capacity Parameter):
C controls the trade-off between error penalty and margin stability.
C is adjusted by the user - it is DATA DEPENDENT.
Good starting value: C = 100
Effect of C (Visual Understanding):
C = 1000 (very large): Almost hard margin, fits training data tightly, complex boundary
C = 10 (medium): Balanced trade-off
C = 0.1 (small): Allows many errors, smoother boundary, wider margin
As C decreases: Points are allowed to move inside the margin (more tolerance for errors)

19. Kernel Functions in Detail

Why Kernels?
When data is not linearly separable, we project it to a higher-dimensional space where it becomes separable.
The mapping x maps to (x, x^2) can make inseparable data separable!
Example: 1D to 2D mapping
Original 1D: Points clustered in center vs points on edges - NOT separable by a single point
After mapping x maps to (x, x^2): Inner points stay low (small x^2), outer points go high (large x^2)
Now a horizontal line can separate them!
Common Kernel Functions:
1. Polynomial Kernel with degree d:
K(x,y) = (x^T y + 1)^d
Maps to polynomial feature space of degree d
2. Radial Basis Function (RBF) / Gaussian Kernel:
K(x,y) = exp(-||x - y||^2 / (2 sigma^2))
Or equivalently: K(x,y) = exp(-gamma ||x - y||^2) where gamma = 1/(2 sigma^2)
Maps to INFINITE dimensional space! Closely related to radial basis function neural networks.
3. Sigmoid Kernel:
K(x,y) = tanh(kappa x^T y + theta)
Related to neural networks with sigmoid activation

20. The Gamma Parameter (RBF Kernel)

Gamma is the free parameter of the Gaussian RBF kernel:
K(x_i, x_j) = exp(-gamma ||x_i - x_j||^2), where gamma greater than 0
Intuition: Gamma controls how far the influence of each support vector reaches.
Small gamma (large variance):
- Support vector has influence over LARGE area
- Even distant points are influenced by this support vector
- Results in SMOOTHER, simpler decision boundary
- Risk of UNDERFITTING

Large gamma (small variance):
- Support vector has influence over SMALL area only
- Only very close points are influenced
- Results in MORE COMPLEX, wiggly decision boundary
- Risk of OVERFITTING

Parameter Tuning Summary:
High C + High gamma: Complex boundary, fits training data tightly, OVERFITTING risk
Low C + Low gamma: Simple boundary, allows errors, UNDERFITTING risk
Use cross-validation to find the best C and gamma for your data!

21. Final SVM Checklist

Can you explain these concepts?
1. Why we maximize margin (generalization)
2. The constraint y_i(w dot x_i + b) at least 1 (scale fixing)
3. Margin width = 2/||w|| (geometry)
4. Minimize (1/2)||w||^2 subject to constraints (optimization)
5. Lagrangian gives w = Sum alpha_i y_i x_i (support vectors)

6. Dual depends only on dot products (kernel trick foundation)
7. Kernel K(x,y) = phi(x) dot phi(y) without computing phi (the trick!)
8. Soft margin with slack xi_i (handling non-separable data)
9. C controls error vs margin trade-off
10. Gamma controls support vector influence reach (RBF kernel)
If you can explain all 10, you understand SVM at a masters level!