Machine Learning Notes

Alexander Rom
University of Southern California

Abstract

These notes cover the main topics of a machine learning course, from linear regression and classification through model selection, tree-based methods, and support vector machines to unsupervised learning, semi-supervised and active learning, and neural networks. Each topic is set out with its key definitions, formulas, and figures, together with short explanations of the ideas behind them.

Contents

Chapter 1

Intro and Linear Regression

1.1 Intro

⋆ **Types of Learning**

- **Supervised Learning**
  Goal: Prediction.
  You are given examples of the correct answer (labels), and you learn a rule that maps inputs to outputs. The point is to make accurate predictions on new data, not just memorize the training set.
- **Semi-Supervised Learning**
  Goal: Better prediction using few labels and many unlabeled examples.
  Labels are limited, but unlabeled data still reveals the structure of the input space (like clusters or smooth regions). A small number of labels can guide the model to place decision boundaries in sensible places.
- **Unsupervised Learning**
  Goals: Discover similarity/dissimilarity; dimensionality reduction.
  There are no "right answers" provided, so the task is to uncover hidden structure in the data. This often means grouping similar points or compressing data into fewer dimensions while keeping the important patterns.
- **Reinforcement Learning**
  Goal: Choose actions to maximize long-run reward.
  Learning happens by interacting with an environment and receiving feedback that can be delayed and noisy. The core challenge is making sequences of decisions that pay off over time, balancing exploration (trying new actions) and exploitation (using what works).

1.1.1 Supervised Learning

Starting point (data). We observe an **outcome** Y (also called dependent variable, response, target) and a vector of p **predictors** $X = (X_1, \ldots, X_p)$ (also called inputs, regressors, covariates, features, independent variables).

Supervised learning is about learning a rule that maps inputs X to an output Y. The observed outcome provides the reference that guides learning, so the objective is a mapping that performs well on unseen data.

Training data. We are given N labeled examples (the **training data**):

$$(x_1, y_1), \ldots, (x_N, y_N),$$

where each (x_i, y_i) is an observed **instance** of (X, Y).

The training data are a finite sample from the underlying data-generating process. Effective learning captures structure that persists beyond these observations rather than noise specific to the sample.

Two main supervised problem types.

- **Regression:** Y is quantitative (numerical), e.g. price, blood pressure.
- **Classification:** Y takes values in a finite, unordered set, e.g. $\{\text{survived}, \text{died}\}$, digits 0–9, cancer class, it is qualitative

The distinction between regression and classification is determined entirely by the type of the outcome variable. The same input features can lead to either problem depending on how Y is defined.

Objectives.

- **Generalization:** accurately predict unseen test cases.
- **Interpretation:** understand how predictors affect the outcome.
- **Assessment:** evaluate the quality and reliability of predictions.

Generalization is the central goal of supervised learning. Interpretation and assessment become essential when models are used for scientific insight or decision-making, not just raw prediction.

⋆ **Classification.** A **classification algorithm**, called a **classifier**, learns from labeled examples where inputs are paired with outputs (the **training data**) in order to map inputs to discrete output classes. After training, the classifier can assign a predicted class to new inputs for which the true output is unknown.

Classification focuses on learning decision rules that separate inputs into categories rather than predicting numerical values. The quality of a classifier depends on how well these learned rules generalize to new data, not how perfectly they fit the training examples.

⋆ **Regression.** A supervised learning problem in which the outcome variable Y is **quantitative** (numerical), and the goal is to learn a function that maps input variables X to a continuous-valued prediction of Y.

Regression models estimate how changes in the inputs are associated with changes in the outcome. Rather than assigning categories, the model produces a numerical prediction, often interpreted as an average or expected value given the inputs.

1.1.2 Unsupervised Learning

⋆ **Unsupervised Learning.** A learning setting in which there is no observed **outcome** variable. Instead, we observe only a set of **predictors** (features) measured on a collection of samples, and the goal is to uncover structure in the data without labeled outputs.

Unsupervised learning treats the data itself as the object of study. Since no labels define what is "correct," the task is exploratory: understand how observations or features relate to each other.

Data and objective. We observe inputs $X_1, \ldots, X_N$ with no corresponding responses Y. The objective is typically vague and problem-dependent, such as:

- finding groups of samples that behave similarly,
- finding features that vary or behave together,
- finding low-dimensional representations that capture most of the variation.

Because there is no single target to predict, success is harder to define than in supervised learning. Different objectives reflect different notions of what "structure" in the data means.

Evaluation and role. Performance is often difficult to assess directly, since there is no ground truth. Unsupervised learning is frequently used as a **pre-processing** or exploratory step before supervised learning.

Without labels, there is no obvious error metric. As a result, unsupervised methods are commonly used to organize, summarize, or simplify data before applying predictive models.

Core problem types.

- **Clustering:** partition samples into subsets that share common characteristics.
- **Dimensionality reduction:** construct new features from the original inputs that retain important information in fewer dimensions.

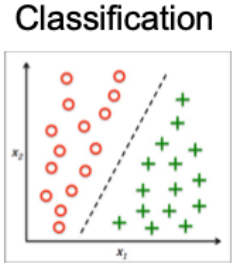
Clustering focuses on similarity between observations, while dimensionality reduction focuses on representing the data efficiently. Both aim to reveal latent structure hidden in high-dimensional inputs.

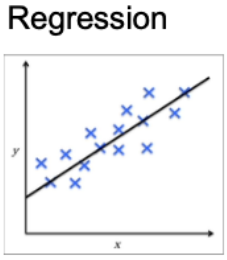
Examples.

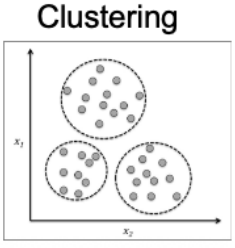
- **Anomaly detection:** identifying unusual or rare observations (e.g., fraud) when the form of abnormal behavior is unknown.
- **Recommendation systems:** grouping users by similarity in preferences (e.g., movie tastes) to infer structure without explicit labels.

These applications rely on the assumption that meaningful patterns exist in the data even when no explicit outcomes are observed. The model's usefulness comes from revealing those patterns, not predicting a predefined label.

Common methods. Representative techniques include **K-means** clustering and dimensionality-reduction methods such as **PCA** and **ICA**.







1.1.3 Semi-Supervised Learning

⋆ **Semi-supervised Learning.** A class of supervised learning methods that use a small amount of labeled data together with a large amount of unlabeled data to improve learning performance.

Semi-supervised learning sits between supervised and unsupervised learning. The labeled data provide guidance about the task, while the unlabeled data reveal the structure of the input space that helps shape better decision boundaries.

Key characteristics.

- Combines labeled and unlabeled data during training.
- Typically assumes labeled data are scarce or expensive to obtain.
- Uses unlabeled data to exploit geometric or cluster structure in the inputs.

By leveraging many unlabeled examples, the model can learn how data are distributed even when labels are limited. This often leads to better generalization than using the labeled data alone.

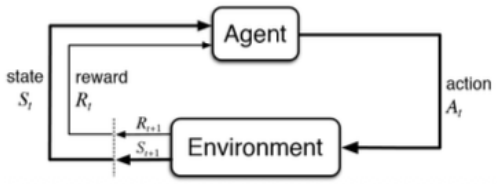
1.1.4 Reinforcement Learning

⋆ **Reinforcement Learning.** A learning framework in which an **agent** interacts with an **environment** by taking **actions** that move the system between **states**, receiving **rewards** as feedback.

Reinforcement learning is not about predicting a fixed output from an input. It is about learning how to behave over time through interaction, where actions influence future states and rewards.

Setting. At each time step t:

- the agent observes the current **state** S_t,
- selects an **action** A_t,
- the environment transitions to a new state S_{t+1},
- a **reward** R_t may be received.



Actions affect not only immediate rewards but also future opportunities. This feedback loop makes reinforcement learning fundamentally different from standard input–output learning.

Learning objective. The goal is to learn a behavior (policy) that maximizes cumulative reward over time.

Because rewards can be delayed, the agent must reason about long-term consequences. Good short-term actions are not always good long-term actions.

Characteristics.

- No labeled input–output pairs.
- Feedback is evaluative (reward-based), not instructive.
- Learning occurs through interaction with the environment.

The agent is never told the correct action explicitly. It must discover effective behavior through trial, error, and feedback.

1.1.5 Terminology

Instance. A single data point (one "thing" being classified, regressed, or clustered). Also called an **observation**. In tabular data, each row corresponds to one instance.

Think "one row of data."
Example: i-th instance $\equiv (x_i, y_i)$ (supervised).

Feature. An input variable describing an instance. Also called an attribute, **covariate**, or **predictor**; often referred to as an **independent variable**.

Features are the inputs available to the model.
Example: $X = (X_1, X_2, \ldots, X_p)$.

Feature vector. The list of feature values for a single instance.

This is the numeric object the model operates on.
Example: $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip}) \in \mathbb{R}^p$.

Label. The value of the output variable being predicted. Also called the **dependent variable**.

Labels define the prediction target.
Example: $Y = y_i$.

Class. In classification, the set of possible values a label can take.

Classes are the categories assigned to instances.
Example: $y_i \in \{1, 2, \ldots, K\}$.

Training instance. A **feature vector** paired with a **label**.

A labeled example used for learning.
Example: (x_i, y_i).

Training data. The collection of all training instances; also called **labeled data**.

This is the dataset used to fit the model.
Example: $\{(x_i, y_i)\}_{i=1}^N$.

Unlabeled data. A dataset consisting only of **feature vectors**, with no associated labels.

Common in unsupervised and semi-supervised learning.
Example: $\{x_i\}_{i=1}^N$.

Prediction. The task of predicting the output Y given inputs X using a learned model $\hat{f}$ as an estimator of the true function f.

- **Reducible error**: error due to imperfect estimation of f.
- **Irreducible error**: noise ε not explained by the inputs.

Prediction focuses on accuracy. Even with the optimal model, irreducible error sets a lower bound on how well Y can be predicted from X.
Example: $\hat{Y} = \hat{f}(X)$.

Inference. The task of understanding the relationship between the output Y and individual predictors X_i.

Inference prioritizes interpretability over accuracy. Models used for inference should reveal how predictors influence the outcome rather than act as black boxes.
Example: effect of changing X_i on Y.

1.1.6 The Regression Function $f(x)$

⋆ **Regression function $f(x)$.**

$$f(x) = \mathbb{E}(Y \mid X = x),$$

where $\mathbb{E}(\cdot)$ denotes the **expected value** (conditional mean, or average) of Y given $X = x$. If X is a vector $X = (X_1, \ldots, X_p)$, then

$$\boxed{f(x_1, x_2, x_3) = \mathbb{E}(Y \mid X_1 = x_1,\ X_2 = x_2,\ X_3 = x_3).}$$

The regression function gives the average value of Y among all observations with the same inputs $X = x$. It smooths out random variation and represents the systematic part of the relationship between X and Y.

Optimality under mean-squared error. For any predictor $g(X)$,

$$f(x) = \mathbb{E}(Y \mid X = x) \quad \text{minimizes} \quad \mathbb{E}\Big[(Y - g(X))^2 \mid X = x\Big]$$

over all functions g, at every point $X = x$.

Under squared-error loss, predicting the conditional average is optimal. No other function can achieve a lower expected prediction error at the same $X = x$.

Irreducible error. Define

$$\varepsilon = Y - f(X).$$

At a fixed $X = x$, the outcome Y typically varies around its average value $f(x)$. This intrinsic variability cannot be explained by X and leads to irreducible error.

Error decomposition. For any estimate $\hat{f}$,

$$\mathbb{E}\Big[(Y - \hat{f}(X))^2 \mid X = x\Big] = \Big[f(x) - \hat{f}(x)\Big]^2 + \mathrm{Var}(\varepsilon \mid X = x).$$

$$\boxed{\underbrace{\Big[f(x) - \hat{f}(x)\Big]^2}_{\textbf{Reducible error}} + \underbrace{\mathrm{Var}(\varepsilon \mid X = x)}_{\textbf{Irreducible error}}.}$$

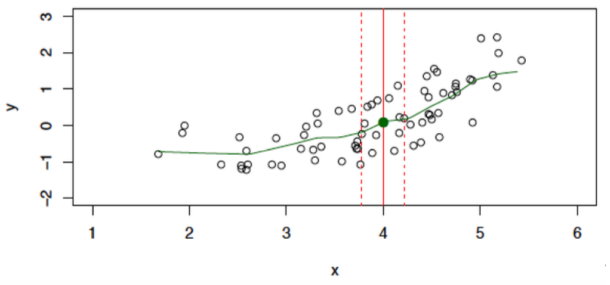
The first term reflects how well the model approximates the true regression function. The second term is noise inherent to the data-generating process and cannot be reduced by improving the model.

Course focus. Methods studied aim to reduce the **reducible error** by constructing better approximations $\hat{f}(x)$ to $f(x)$.

Learning algorithms compete on how efficiently they approximate the conditional mean without fitting noise.

1.1.7 Nearest-Neighbor Regression

Problem: estimating $f(x) = \mathbb{E}(Y \mid X = x)$ from data. In practice, we typically have few (if any) observations with $X = x$ exactly, so we cannot directly compute the conditional **expected value** $\mathbb{E}(Y \mid X = x)$ from the sample.



Conditioning on $X = x$ is a "zero-probability event" for continuous predictors, so exact matches are rare. We need to approximate the conditional mean using nearby points.

Solution: local averaging using a neighborhood.
Relax the definition by averaging outcomes for observations whose predictors fall in a **neighborhood** of x:

$$\boxed{\hat{f}(x) = \text{Ave}\Big(Y \mid X \in N(x)\Big),}$$

where $N(x)$ is a neighborhood around x.

This replaces "exactly at x" with "close to x." Choosing $N(x)$ controls the bias–variance tradeoff: larger neighborhoods reduce variance but increase bias.

Defining a neighborhood $N(x^*)$. Two common ways to define the neighborhood around a target point x^*:

- **Radius-based neighborhood:**

$$N(x^*) = \{x : d(x, x^*) \leq r\},$$

  where $d(\cdot, \cdot)$ is a distance metric and $r > 0$ is a fixed radius.
- **k-nearest neighbors (KNN):**

$$N(x^*) = \{\text{the } k \text{ data points closest to } x^*\},$$

  where k is fixed and the radius adapts to include exactly k points.

Radius-based neighborhoods fix distance and allow the number of points to vary, while KNN fixes the number of points and lets the effective radius vary. Both define locality for estimating $\mathbb{E}(Y \mid X = x^*)$.

Nearest-neighbor regression. A **non-parametric** method for estimating the regression function $f(x) = \mathbb{E}(Y \mid X = x)$ by averaging the responses of observations whose predictors are closest to x.

$$\hat{f}(x) = \frac{1}{|N_k(x)|} \sum_{i:\, x_i \in N_k(x)} y_i,$$

where $N_k(x)$ denotes the set of the k nearest neighbors of x in predictor space.

Nearest-neighbor regression estimates the conditional mean by local averaging rather than by fitting a global functional form. It relies on the idea that nearby points have similar outcomes.

When nearest-neighbor regression works well.

- Small predictor dimension p (e.g. $p \leq 4$).
- Large sample size N.

In low dimensions, neighborhoods can remain local while still containing enough points to reduce variance.

Bias–variance behavior.

- Larger neighborhoods reduce variance but increase bias.
- Smaller neighborhoods reduce bias but increase variance.

Choosing the neighborhood size is a bias–variance tradeoff rather than a parameter estimation problem.

Comparison with parametric regression.

- **Nearest-neighbor regression:**

$$\hat{f}(x) = \text{Ave}(Y \mid X \in N_k(x)),$$

  no assumed functional form for f.
- **Linear regression:**

$$\hat{f}(x) = x^\top \hat{\beta},$$

  where parameters $\hat{\beta}$ are estimated from the full dataset.

Parametric regression imposes structure to stabilize estimation in high dimensions, while nearest-neighbor regression trades structure for flexibility and locality.

Curse of dimensionality. A phenomenon where many learning methods degrade rapidly as the number of predictors p increases, because data become sparse in high-dimensional spaces.

As dimension grows, points that seem "close" in low dimensions become far apart. Local neighborhoods lose their meaning, making local averaging and distance-based methods unreliable.

Effect on local methods. To include a fixed fraction of the data (e.g. 10%) in a neighborhood $N(x)$, the neighborhood must grow very large when p is large.

In high dimensions, a neighborhood large enough to reduce variance is no longer local, so estimates of $\mathbb{E}(Y \mid X = x)$ become biased and noisy.

Implications.

- Nearest-neighbor methods perform well only for small p.
- Distance-based similarity becomes less informative as p increases.
- Structured or parametric models are often preferred in high dimensions.

The curse of dimensionality motivates imposing structure, reducing dimension, or regularizing models rather than relying purely on locality.

1.1.8 Parametric and Structured Models

Structured (parametric) models. Structured models estimate a finite set of parameters by fitting a specified functional form to training data. The model structure is fixed in advance, and learning reduces to estimating its parameters.

By restricting the class of functions, parametric models trade flexibility for stability, especially in moderate or high dimensions.

Linear model. A central example of a parametric model is the **linear model**:

$$f_L(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p,$$

which is fully specified by $p + 1$ parameters $\beta_0, \beta_1, \ldots, \beta_p$.

Despite its simplicity, the linear model often captures the dominant structure in the data and provides a strong baseline for prediction and inference.

Parameter estimation. The parameters $\beta_0, \ldots, \beta_p$ are estimated by fitting the model to the training data, typically by minimizing a loss function such as squared error.

Although the linear form is almost never exactly correct, its low variance and interpretability make it a powerful approximation to the unknown regression function $f(x)$.

Comparison with non-parametric methods.

- Parametric models impose structure and scale well with dimension.
- Non-parametric methods adapt flexibly to the data but suffer in high dimensions.

Statistical learning is largely about choosing how much structure to impose in order to balance bias, variance, and interpretability.

1.1.9 Trade-offs: Model Accuracy and Bias-Variance

Assessing model accuracy. Suppose a model $\hat{f}(x)$ is fit on training data

$$\mathrm{Tr} = \{(x_i, y_i)\}_{i=1}^{N}.$$

A natural measure of performance is the mean squared error (MSE).

Training error.

$$\boxed{\mathrm{MSE}_{\mathrm{Tr}} = \frac{1}{N} \sum_{i \in \mathrm{Tr}} \left[y_i - \hat{f}(x_i) \right]^2.}$$

Training error measures how well the model fits the data it was trained on, but it is typically optimistic and favors overly flexible models.

Test error. Using fresh test data

$$\mathrm{Te} = \{(x_i, y_i)\}_{i=1}^{M},$$

the test MSE is

$$\boxed{\mathrm{MSE}_{\mathrm{Te}} = \frac{1}{M} \sum_{i \in \mathrm{Te}} \left[y_i - \hat{f}(x_i) \right]^2.}$$

Test error estimates how well the model generalizes to new data and is the primary criterion for model comparison.

Effect of model flexibility. As model **flexibility** increases:

- $\mathrm{MSE}_{\mathrm{Tr}}$ typically decreases monotonically.
- $\mathrm{MSE}_{\mathrm{Te}}$ often follows a U-shaped curve.

Too little flexibility leads to underfitting, while too much flexibility leads to overfitting. The optimal model minimizes test error.

Bias–variance trade-off. Assume the data-generating process

$$Y = f(X) + \varepsilon, \quad \text{with } f(x) = \mathbb{E}(Y \mid X = x).$$

For a test point (x_0, y_0),

$$\boxed{\mathbb{E}\left[(y_0 - \hat{f}(x_0))^2 \right] = \mathrm{Var}(\hat{f}(x_0)) + \left[\mathrm{Bias}(\hat{f}(x_0)) \right]^2 + \mathrm{Var}(\varepsilon).}$$

This decomposition explains why prediction error does not vanish even with large data. The first term reflects instability of the learning algorithm across different training samples, the second term captures

systematic error from model misspecification, and the final term is irreducible noise inherent in the data-generating process. Model selection is therefore about controlling variance and bias, since $\mathrm{Var}(\varepsilon)$ *cannot be reduced.*

Bias.

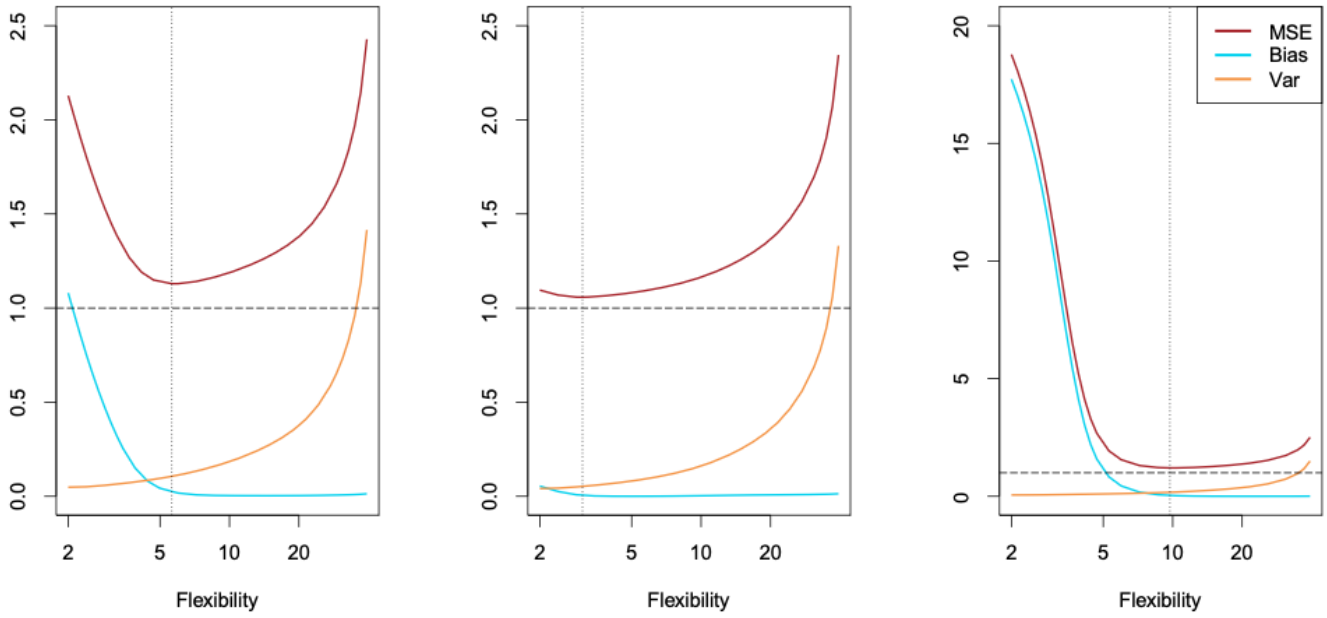
$$\boxed{\mathrm{Bias}(\hat{f}(x_0)) = \mathbb{E}[\hat{f}(x_0)] - f(x_0).}$$

The expectation averages over both the randomness in the training data and the noise in the response. ***Bias arises from training error*** *because a restrictive model cannot represent the true regression function* $f(x)$*, even when fit optimally on the training data. When the model class is too simple, minimizing training error still leaves systematic discrepancies between* $\hat{f}(x)$ *and* $f(x)$*, which appear as bias rather than randomness.*

Interpretation.

- Increasing flexibility typically reduces bias.
- Increasing flexibility typically increases variance.

Choosing model flexibility to minimize average test error amounts to balancing bias and variance. This is the central trade-off in statistical learning.



Bias and variance are inversely related because increasing model flexibility changes how the learning process responds to the training data. Simple models restrict the set of functions that can be learned, forcing systematic error (high bias) but producing stable estimates across samples (low variance), while highly flexible models can closely fit the training data, reducing bias but making the fitted function sensitive to noise and sample fluctuations, increasing variance. This trade-off is a consequence of learning from finite data under noise: the more freedom a model has to adapt, the more it reacts to random variation rather than true signal.

1.1.10 Classification

⋆ **Classification.** A supervised learning problem in which the response variable Y is **qualitative**, taking values in a finite set of classes

$$\mathcal{C} = \{1, 2, \ldots, K\}.$$

Examples include spam vs. ham email classification or digit recognition with $\mathcal{C} = \{0, 1, \ldots, 9\}$.

Unlike regression, classification predicts categories rather than numerical values. The goal is to assign each observation to the most plausible class.

Classification goal. Given predictors $X = (X_1, \ldots, X_p)$, construct a classifier

$$C(X) \in \mathcal{C}$$

that assigns a class label to a future unlabeled observation.

- Accurately predict the class label.
- Quantify uncertainty in each classification.
- Understand the roles of individual predictors X_j.

Conditional class probabilities. For classes indexed by $k = 1, \ldots, K$, define

$$p_k(x) = \Pr(Y = k \mid X = x).$$

These probabilities describe how likely each class is at a given input x. Classification is fundamentally about comparing these quantities.

Bayes optimal classifier. The **Bayes classifier** assigns x to the class with the highest conditional probability:

$$C(x) = j \quad \text{if} \quad p_j(x) = \max_{k=1,\ldots,K} p_k(x).$$

This rule minimizes the probability of misclassification at each x. No classifier can do better on average if the true $p_k(x)$ are known.

Misclassification error rate. For a fitted classifier $\hat{C}(x)$, performance is measured by

$$\mathrm{Err}_{\mathrm{Te}} = \frac{1}{|\mathrm{Te}|} \sum_{i \in \mathrm{Te}} \mathbf{1}\{y_i \neq \hat{C}(x_i)\}.$$

This is the fraction of test observations that are misclassified. It directly measures predictive accuracy.

Optimality of the Bayes classifier. The Bayes classifier (using the true $p_k(x)$) achieves the smallest possible misclassification error in the population.

All practical classifiers aim to approximate this ideal rule using finite data.

Practical considerations.

- Nearest-neighbor methods can estimate $p_k(x)$, but degrade as dimension grows.
- Estimating the classifier $C(x)$ is often easier than accurately estimating all $p_k(x)$.

Structured classification models.

- Directly model the classifier $C(x)$: e.g. **support vector machines**.
- Model the probabilities $p_k(x)$: e.g. **logistic regression**, **generalized additive models**.

Different methods trade off interpretability, flexibility, and statistical efficiency, but all target the same underlying Bayes decision rule.

1.1.11 Bias-Variance Trade-off Proof

⋆ **Bias-Variance Trade-off Proof Bias–Variance Trade-Off.** Assume the data-generating process

$$Y = f(X) + \varepsilon, \quad \text{with } f(x) = \mathbb{E}(Y \mid X = x),$$

where ε is noise with $\mathbb{E}(\varepsilon) = 0$ and variance $\mathrm{Var}(\varepsilon)$.

For a test point x_0, the expected prediction error decomposes as

$$\boxed{\mathbb{E}\Big[(y_0 - \hat{f}(x_0))^2\Big] = \mathrm{Var}(\hat{f}(x_0)) + \Big[\mathrm{Bias}(\hat{f}(x_0))\Big]^2 + \mathrm{Var}(\varepsilon).}$$

- **Bias** measures systematic error: $\mathrm{Bias}(\hat{f}(x_0)) = \mathbb{E}[\hat{f}(x_0)] - f(x_0)$.
- **Variance** measures sensitivity to the training sample: $\mathrm{Var}\Big(\hat{f}(x_0)\Big) = \mathbb{E}\Big[\Big(\hat{f}(x_0) - \mathbb{E}[\hat{f}(x_0)]\Big)^2\Big]$.
- **Irreducible error** $\mathrm{Var}(\varepsilon)$ cannot be eliminated by any model: $\mathrm{Var}(\varepsilon) = \mathbb{E}[(\varepsilon - \mathbb{E}[\varepsilon])^2]$.

  Under the standard assumption $\mathbb{E}[\varepsilon] = 0 \to \mathrm{Var}(\varepsilon) = \mathbb{E}(\varepsilon^2) = \sigma^2$.

Increasing model flexibility reduces bias but increases variance, since the model fits the training data more closely and becomes sensitive to sampling noise. Simpler models have higher bias but lower variance. Choosing a model is about balancing these opposing forces to minimize test error.

Proof (bias–variance decomposition, shorter). Assume

$$Y = f(X) + \varepsilon, \quad f(x) = \mathbb{E}(Y \mid X = x), \quad \mathbb{E}(\varepsilon \mid X) = 0, \quad \mathrm{Var}(\varepsilon \mid X) = \sigma^2.$$

Fix a test input x_0 and write $y_0 = f(x_0) + \varepsilon_0$. Then

$$\mathbb{E}\Big[(y_0 - \hat{f}(x_0))^2 \mid X = x_0\Big] = \mathbb{E}\Big[(f(x_0) + \varepsilon_0 - \hat{f}(x_0))^2 \mid X = x_0\Big].$$

Add and subtract $\mathbb{E}[\hat{f}(x_0) \mid X = x_0]$:

$$f(x_0) + \varepsilon_0 - \hat{f}(x_0) = \Big(f(x_0) - \mathbb{E}[\hat{f}(x_0) \mid X = x_0]\Big) + \Big(\mathbb{E}[\hat{f}(x_0) \mid X = x_0] - \hat{f}(x_0)\Big) + \varepsilon_0.$$

Square and take conditional expectation. Using $\mathbb{E}(\varepsilon_0 \mid X = x_0) = 0$ and independence of ε_0 from the training sample (hence from $\hat{f}(x_0)$), the cross-terms vanish, so

$$\mathbb{E}\Big[(y_0 - \hat{f}(x_0))^2 \mid X = x_0\Big] = \Big(f(x_0) - \mathbb{E}[\hat{f}(x_0) \mid X = x_0]\Big)^2 + \mathbb{E}\Big[\Big(\hat{f}(x_0) - \mathbb{E}[\hat{f}(x_0) \mid X = x_0]\Big)^2 \mid X = x_0\Big] + \mathbb{E}(\varepsilon_0^2 \mid X$$

Recognize the terms:

$$\Big(f(x_0) - \mathbb{E}[\hat{f}(x_0) \mid X = x_0]\Big)^2 = \Big[\mathrm{Bias}(\hat{f}(x_0) \mid X = x_0)\Big]^2,$$

$$\mathbb{E}\Big[\Big(\hat{f}(x_0) - \mathbb{E}[\hat{f}(x_0) \mid X = x_0]\Big)^2 \mid X = x_0\Big] = \mathrm{Var}(\hat{f}(x_0) \mid X = x_0),$$

$$\mathbb{E}(\varepsilon_0^2 \mid X = x_0) = \mathrm{Var}(\varepsilon \mid X = x_0) = \sigma^2.$$

Therefore,

$$\boxed{\mathbb{E}\Big[(y_0 - \hat{f}(x_0))^2 \mid X = x_0\Big] = \mathrm{Var}(\hat{f}(x_0) \mid X = x_0) + \Big[\mathrm{Bias}(\hat{f}(x_0) \mid X = x_0)\Big]^2 + \mathrm{Var}(\varepsilon \mid X = x_0).} \quad \square$$

Bias and variance sources.

- **Bias** arises from **underfitting**: the model is too simple to capture the true structure of $f(x)$, leading to systematic error.
- **Variance** arises from **overfitting**: the model is too flexible and fits noise in the training data, leading to instability across samples.

Underfitting produces consistent but wrong predictions, while overfitting produces accurate training fits that fail to generalize.

Proof: Irreducible vs Reducible error . Assume

$$Y = f(X) + \varepsilon, \qquad f(x) = \mathbb{E}(Y \mid X = x), \qquad \mathbb{E}(\varepsilon \mid X) = 0.$$

Let $\hat{f}$ be any estimator built from training data $\mathcal{T}$. For a fixed test point x, condition on $(X = x, \mathcal{T})$ so $\hat{f}(x)$ is a constant.

$$\mathbb{E}\Big[(Y - \hat{f}(X))^2 \mid X = x, \mathcal{T}\Big] = \mathbb{E}\Big[(f(x) + \varepsilon - \hat{f}(x))^2 \mid X = x, \mathcal{T}\Big] =$$
$$= (f(x) - \hat{f}(x))^2 + \underbrace{2(f(x) - \hat{f}(x))\,\mathbb{E}(\varepsilon \mid X = x, \mathcal{T})}_{=0} + \mathbb{E}(\varepsilon^2 \mid X = x, \mathcal{T}).$$

Using $\mathbb{E}(\varepsilon \mid X) = 0$,

$$\mathbb{E}(\varepsilon \mid X = x, \mathcal{T}) = 0, \qquad \mathbb{E}(\varepsilon^2 \mid X = x, \mathcal{T}) = \mathrm{Var}(\varepsilon \mid X = x).$$

So

$$\boxed{\mathbb{E}\Big[(Y - \hat{f}(X))^2 \mid X = x, \mathcal{T}\Big] = \underbrace{(f(x) - \hat{f}(x))^2}_{\text{reducible error}} + \underbrace{\mathrm{Var}(\varepsilon \mid X = x)}_{\text{irreducible error}}.}$$

The reducible part is how far your fitted rule $\hat{f}(x)$ is from the true conditional mean $f(x)$. The irreducible part is the inherent spread of $Y \mid X = x$ (noise, omitted factors), which no method using only X can remove.

1.2 Linear Regression

1.2.1 OLS

OLS selects the line that makes the observed data as close as possible to the model in squared-error terms. Squaring residuals penalizes large deviations more heavily and yields a tractable optimization problem with a unique solution.

Residual. The ith **residual** is

$$\boxed{e_i = y_i - \hat{y}_i.}$$

A residual measures the vertical distance between the observed outcome and the fitted line. It captures the part of Y not explained by the linear model at x_i.

⋆ **Residual sum of squares (RSS).** The **residual sum of squares** is defined as

$$\text{RSS} = e_1^2 + e_2^2 + \cdots + e_n^2 \qquad \Rightarrow \qquad \boxed{\text{RSS} = \sum_{i=1}^{n} \big(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i\big)^2.}$$

RSS aggregates the squared prediction errors across all observations. Least squares estimation chooses coefficients that make the overall discrepancy between data and model as small as possible.

⋆ **Normality of the error term.** Assume the error satisfies with mean zero and standard deviation

$$\boxed{\varepsilon \sim \mathcal{N}(0, \sigma^2)}$$

Normality of the error is not required to estimate the regression line, but it is crucial for statistical inference. Confidence intervals and hypothesis tests rely on this assumption to obtain exact finite-sample distributions.

⋆ **Normality approximation for coefficient estimates.** Under repeated sampling,

$$\frac{\hat{\beta}_1 - \beta_1}{\text{SE}(\hat{\beta}_1)} \;\approx\; \mathcal{N}(0, 1).$$

After centering by the true parameter and scaling by its standard error, the slope estimator behaves like a standard normal random variable. This approximation justifies confidence intervals and hypothesis tests based on normal or t-critical values.

Error term vs. residual. Let the true data-generating process be $Y_i = f(x_i) + \varepsilon_i$, and let the fitted value be $\hat{y}_i = \hat{f}(x_i)$. The observed **residual** satisfies

$$\boxed{e_i = y_i - \hat{y}_i \;=\; \varepsilon_i - \big(\hat{f}(x_i) - f(x_i)\big).}$$

The error term ε_i is unobserved and belongs to the true model, while the residual e_i depends on estimated coefficients. Residuals approximate errors only when the fitted model is close to the true regression function.

⋆ **Sampling distribution of an estimator.** An estimator $\hat{\beta}$ is a **random variable**: it has a probability distribution, with a mean and variance.

$$\boxed{\hat{\beta} \sim \mathcal{N}\big(\beta,\ \text{Var}(\hat{\beta})\big)}$$

Because $\hat{\beta}$ depends on the random training sample, its value varies across samples.

Unbiasedness. An estimator is **unbiased** if

$$\mathbb{E}(\hat{\beta}) = \beta, \quad \text{and} \quad \mathbb{E}(\hat{\beta} - \beta) = 0.$$

Unbiasedness means that, on average over repeated samples, the estimator recovers the true parameter.

⋆ **OLS Least squares estimation** Fit a line to data $\{(x_i, y_i)\}_{i=1}^n$ by predicting

Least squares criterion. Choose $\hat{\beta}_0, \hat{\beta}_1$ to minimize the **residual sum of squares**:

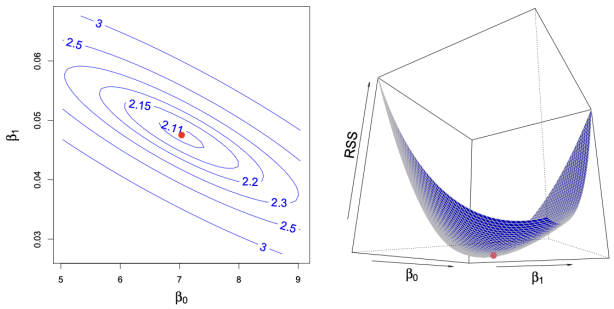
$$\text{RSS}(\beta_0, \beta_1) = \sum_{i=1}^{n} \big(y_i - \beta_0 - \beta_1 x_i\big)^2 \quad \Rightarrow \quad \boxed{(\hat{\beta}_0, \hat{\beta}_1) = \arg\min_{\beta_0, \beta_1} \text{RSS}(\beta_0, \beta_1)}$$

OLS slope:

$$\boxed{\hat{\beta}_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} = \frac{S_{XY}}{S_X^2}.}$$

OLS intercept:

$$\boxed{\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}}$$



Sample means: $\bar{y} = \frac{1}{n}\sum_{i=1}^n y_i, \qquad \bar{x} = \frac{1}{n}\sum_{i=1}^n x_i.$

Least squares chooses the line that best fits the cloud of points in the sense of minimizing squared vertical errors. The slope is "covariance over variance," so stronger co-movement between X and Y increases $\hat{\beta}_1$, while more spread in X (larger $\sum(x_i - \bar{x})^2$) stabilizes the estimate.

⋆ **Accuracy of coefficient estimates.** The variability of an estimator across repeated samples is measured by its **standard error**. It quantifies how much the **estimate would vary if we repeatedly sampled new datasets** from the same population.

$$\boxed{\text{SE}(\hat{\beta}_1)^2 = \frac{\sigma^2}{\sum_{i=1}^n (x_i - \bar{x})^2}} \qquad \text{and} \qquad \boxed{\text{SE}(\hat{\beta}_0)^2 = \sigma^2 \left[\frac{1}{n} + \frac{\bar{x}^2}{\sum_{i=1}^n (x_i - \bar{x})^2}\right]}$$

Error variance. The error variance measures the **variability of the unobserved error term** around the true regression function and is defined as $\boxed{\sigma^2 = \text{Var}(\varepsilon)}$ (noise you can't control). The error term ε is not observed, so σ^2 cannot be known directly. In practice, it is estimated from the data using the residuals, typically by averaging squared residuals after fitting the model.

Standard errors describe sampling uncertainty, not model fit. *The slope estimate becomes more precise when the error variance is small and when the predictor values are widely spread, while the intercept is harder to estimate accurately when the mean of X is far from zero.*

SD Standard deviation measures how spread out the data are in the underlying population or process. **SE Standard error** measures how uncertain our estimate is, reflecting how much it would vary across repeated samples.

Higher variability in the predictor X improves estimation precision. $\text{SE}(\hat{\beta}_1)^2 = \frac{\sigma^2}{\sum_{i=1}^n (x_i - \bar{x})^2}$, a larger **variance of X** (i.e. larger $\sum(x_i - \bar{x})^2$) increases the denominator and therefore **reduces the standard error**.

Intuitively, more spread in the independent variable X provides more information to identify the slope accurately.

1.2.2 Confidence Intervals

⋆ **Confidence intervals for coefficient estimates.** These standard errors can be used to compute **confidence intervals**. A 95% confidence interval is defined as a range of values such that 95% of times, the range will contain the true unknown value of the parameter.

Normal approximate 95% confidence interval for β_1.

$$\hat{\beta}_1 \pm 1.96 \cdot \mathrm{SE}(\hat{\beta}_1) \quad \Longleftrightarrow \quad \left[\hat{\beta}_1 - 1.96 \cdot \mathrm{SE}(\hat{\beta}_1),\ \hat{\beta}_1 + 1.96 \cdot \mathrm{SE}(\hat{\beta}_1)\right].$$

Interpreted in repeated sampling, the interval procedure captures the true β_1 about 95% of the time. The number 2 is a rule-of-thumb from the normal approximation.

t confidence interval for β_1. An interval that will contain the **true unknown value of the parameter β_1 in $1-\alpha$ percent of times** is

$$\boxed{\left[\hat{\beta}_1 - t_{n-2,\alpha/2} \cdot \mathrm{SE}(\hat{\beta}_1),\ \hat{\beta}_1 + t_{n-2,\alpha/2} \cdot \mathrm{SE}(\hat{\beta}_1)\right]}$$

Using the t-critical value accounts for estimating σ and is especially important when n is not large. The $n-2$ degrees of freedom reflect that two parameters (β_0, β_1) are estimated in simple linear regression.

Why t appears in confidence intervals. In practice, the **variance of the error term ε is unknown**, i.e. $\sigma^2 = \mathrm{Var}(\varepsilon)$, so $\mathrm{SE}(\hat{\beta}_1)$ **uses an estimate of σ** computed from residuals. This makes the denominator random, and

$$\boxed{\frac{\hat{\beta}_1 - \beta_1}{\widehat{\mathrm{SE}}(\hat{\beta}_1)} \sim t_{n-2}}$$

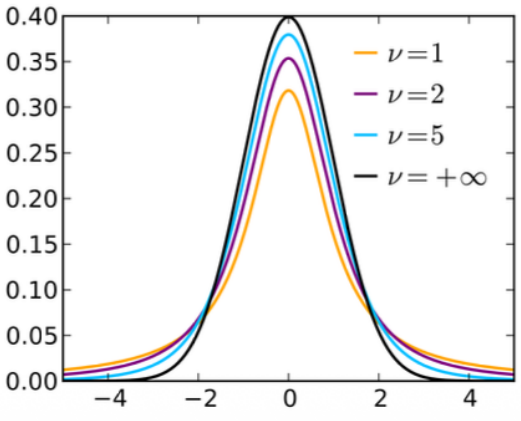
Estimating the error variance $[\mathrm{Var}(\varepsilon)]$ $\boxed{\hat{\sigma}^2 = \dfrac{1}{n-2}\sum_{i=1}^{n} e_i^2 = \dfrac{1}{n-2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2.}$

Standard error coef (unknown variance). $\boxed{\widehat{\mathrm{SE}}(\hat{\beta}_1) = \sqrt{\dfrac{\hat{\sigma}^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}}}$

Estimating σ adds extra uncertainty, so the standardized statistic has heavier tails than a normal. The $n-2$ degrees of freedom reflect estimating two parameters (β_0, β_1) in simple linear regression.

⋆ **Degrees of freedom ν.** The parameter ν in the Student **t-distribution** represents the number of **degrees of freedom**, i.e. the amount of independent information available after estimating model parameters. Each estimated parameter consumes one degree of freedom. With fewer remaining degrees of freedom, uncertainty about the error variance increases.

Degrees of freedom measure how much data remain to estimate variability once constraints are imposed. Fewer degrees of freedom mean more uncertainty in variance estimation, which produces heavier tails.



Limiting behavior $\nu \to \infty$ (CLT) As the degrees of freedom increase, $\nu \to \infty \quad \Rightarrow \quad t_\nu \to Z \sim \mathcal{N}(0,1)$

With many degrees of freedom, variance estimation becomes precise and the t-distribution approaches the standard normal distribution.

1.2.3 Hypothesis Testing

⋆ **Hypothesis testing for regression coefficients.** Standard errors can be used to perform **hypothesis tests** on regression coefficients. The most common test examines whether there is a relationship between X and Y.

Null and alternative hypotheses (conceptual).

$$\begin{array}{|ll|} \hline H_0 : \text{There is no relationship between } X \text{ and } Y & \Rightarrow H_0 : \ \beta_1 = 0 \\ H_A : \text{There is some relationship between } X \text{ and } Y & \Rightarrow H_A : \ \beta_1 \neq 0. \\ \hline \end{array}$$

Hypothesis testing formalizes the question of whether the predictor provides meaningful information about the response beyond random noise.
If $\beta_1 = 0$, the model reduces to $Y = \beta_0 + \varepsilon$, meaning changes in X do not affect the expected value of Y. Rejecting H_0 provides evidence that X is associated with Y.

LOGIC Hypothesis testing A formal procedure for deciding whether the data provide sufficient evidence to reject a default assumption (the null hypothesis).

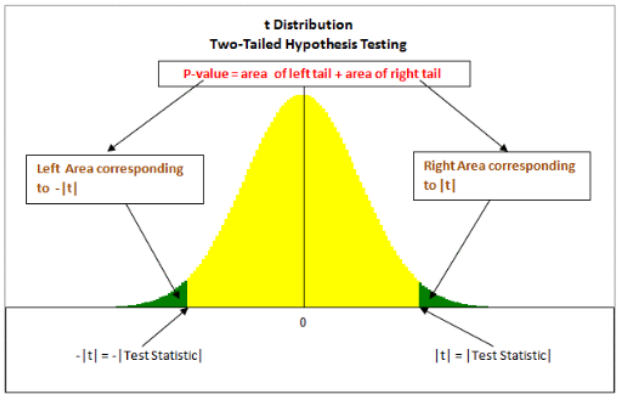
- **Step 1: State the null hypothesis.** Assume the **null hypothesis** H_0 is true.
  This follows the principle "innocent until proven guilty."
- **Step 2: Fix the significance level.** Choose a probability α of rejecting a true null hypothesis:

  $$\alpha = \Pr(\text{reject } H_0 \mid H_0 \text{ is true}).$$

  This is the **Type I error rate**. α must be chosen in advance and represents the standard of proof (e.g. $\alpha = 0.05$).
- **Step 3: Compute the evidence against H_0.** Using the observed data, compute a **test statistic** and the corresponding ***p*-value**, which measures how extreme the data are under H_0.
- **Step 4: Decision rule.** $\boxed{\textbf{Reject} \quad H_0 \text{ if } p\text{-value} < \alpha}$ otherwise **Fail to reject** H_0.
  Failing to reject does not mean H_0 is true; it means the evidence is insufficient.

Hypothesis testing controls the probability of false conviction (Type I error) by fixing α before seeing the data.

⋆ **Test statistic for hypothesis testing.** To test the **null hypothesis** $\beta_1 = 0$ (slope estimate is centered at zero) compute the ***t*-statistic** that will measure how many standard errors the estimated slope is away from zero $\boxed{t = \dfrac{\hat{\beta}_1 - 0}{\text{SE}(\hat{\beta}_1)}}$



Sampling distribution under the Null. Assuming

$$H_0 : \beta_1 = 0 \quad \Rightarrow \quad t \sim t_{n-2},$$

a Student t-distribution with $n - 2$ degrees of freedom.

⋆ **Significance level α.** If the **null hypothesis** is true, the probability of observing

$$t > t_{n-2,\alpha/2} \quad \text{or} \quad t < -t_{n-2,\alpha/2} \quad \text{is } \alpha$$

The value α is the probability of rejecting a true null hypothesis, i.e. a **probability of Type-I error**, and must be chosen ahead of time.

Under H_0, the test statistic satisfies $t \sim t_{n-2}$. Choosing a significance level α fixes the total probability of rejection when the null hypothesis is true. For a two-sided test, this probability is split evenly across the two tails of the distribution, so the rejection region is defined by

$$P\Big(t > t_{n-2,\alpha/2} \text{ or } t < -t_{n-2,\alpha/2} \,\Big|\, H_0\Big) = P\Big(t > t_{n-2,\alpha/2}\Big) + P\Big(t < -t_{n-2,\alpha/2}\Big) = \alpha.$$

$$\Rightarrow \boxed{P\Big(|t| > t_{n-2,\alpha/2} \,\Big|\, H_0\Big) = \alpha.}$$

Compute the t-statistic and reject H_0 if $|t| > t_{n-2,\alpha/2}$; values of t falling beyond these cutoffs are sufficiently unlikely under H_0 and therefore provide evidence against it and suppurts rejection.

⋆ **p-value.** is the **probability, under H_0, of observing a test statistic (t-value)** with magnitude at least as large as the one observed. AKA *The p-value measures how likely it is to observe a test statistic at least as extreme as the one obtained if the null hypothesis were true.*

$$\boxed{p = P\Big(|T| \geq |t_{\text{obs}}| \,\Big|\, H_0\Big), \qquad T \sim t_{n-2}.}$$

- T is the **random test statistic** under the null hypothesis, representing the value the statistic would take across repeated samples.
- t_{obs} is the **observed test statistic** computed from the data.
- $|\cdot|$ reflects a **two-sided test**, counting extreme outcomes in both directions.
- $P(\cdot \mid H_0)$ probability computed assuming the null hypothesis is true.

The p-value converts the observed t-statistic into a probability using the null distribution. It directly relates to the significance level α: rejecting H_0 at level α is equivalent to having $p \leq \alpha$. Larger $|t_{obs}|$ correspond to smaller p-values, indicating stronger evidence against the null.

Decision rule. If the p-value is very small, the probability of observing such an extreme t-statistic under $H_0 : \beta_1 = 0$ is very small, so we reject the null hypothesis.

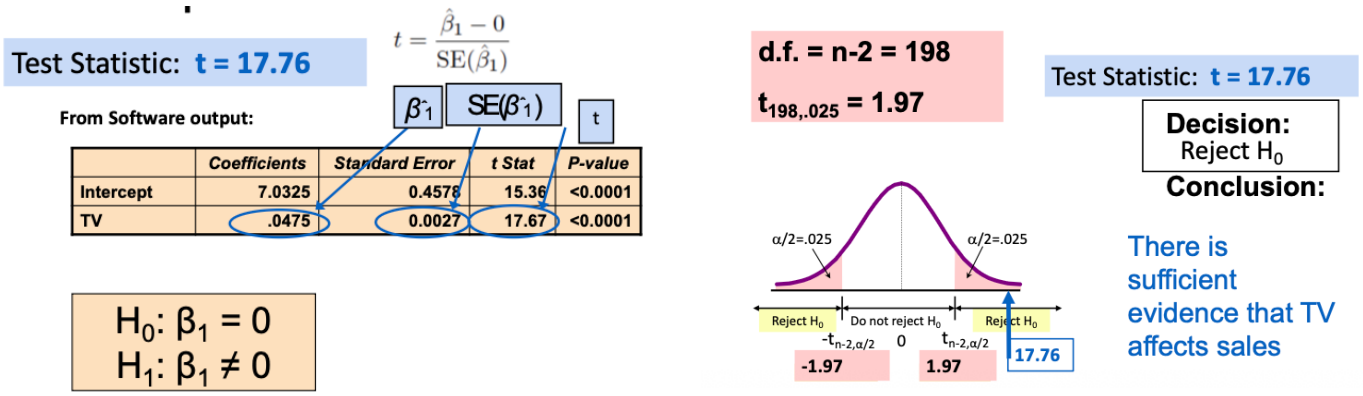
Summury: A **t-statistic** measures how many standard errors a sample mean (or coefficient estimate) is away from a hypothesized population value under the null hypothesis. The **p-value** is the probability of observing a t-statistic at least as extreme as the one obtained if the null hypothesis (no real difference) were true; a small p-value (typically < 0.05) indicates that such an extreme value is unlikely under H_0 and leads to rejection of the null. The two are directly related: larger absolute values of the t-statistic correspond to smaller p-values, providing stronger evidence against the null hypothesis.

Rejection region logic (critical value and p-value). Fix a significance level α and compute the test statistic t.

Critical t=value approach: Reject H_0 if $|t| > t_{n-2,\alpha/2}$,

p-value approach: Reject H_0 if $p \leq \alpha$, $\quad p = P(|T| \geq |t| \mid H_0)\,,\ T \sim t_{n-2}$.

Both approaches use the same t_{n-2} distribution under the null hypothesis. The critical value method compares the statistic to a cutoff, while the p-value method compares the tail probability to α; they are equivalent decision rules.



1.2.4 Model Fit: RSE, TSS, and R^2

⋆ **RSE: Residual Standard Error** Measures the typical size of the residuals (p is number of coef)

$$\text{RSE} = \sqrt{\frac{1}{n-p-1}\text{RSS}} = \sqrt{\frac{1}{n-p-1}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} = \hat{\sigma}^2$$

RSE estimates the standard deviation of the noise term $\sigma^2 = \text{Var}(\varepsilon)$. It measure, in the units of Y, how much on average the response deviated from the regression line.

⋆ **RSS: Residual Sum of Squares** is $\text{RSS} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

RSS measures the total squared deviation between the observed responses and the fitted values from the regression model. It quantifies the overall amount of variation in the data that is not explained by the model. *The RSE estimates the standard deviation of the noise term ε. It summarizes, in the units of Y, how far observed responses typically deviate from the fitted regression line.*

Interpretation of RSE. A smaller RSE indicates a model that fits the data more closely, while a larger RSE indicates greater unexplained variability. RSE provides an overall measure of model accuracy, complementing coefficient-level uncertainty measures such as standard errors.

⋆ **TSS: Total variation decomposition** The total variation in the response variable can be decomposed as

$$TSS = Regression \quad SS + RSS$$

$$\textbf{Total Sum of Squares} = \textbf{Regression Sum of Squares} + \textbf{Residual Sum of Squares}$$

$$\sum_{i=1}^{n}(y_i - \bar{y})^2 \quad = \quad \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2 \quad + \quad \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

This identity splits total variation in Y into the part explained by the regression model and the part left unexplained.

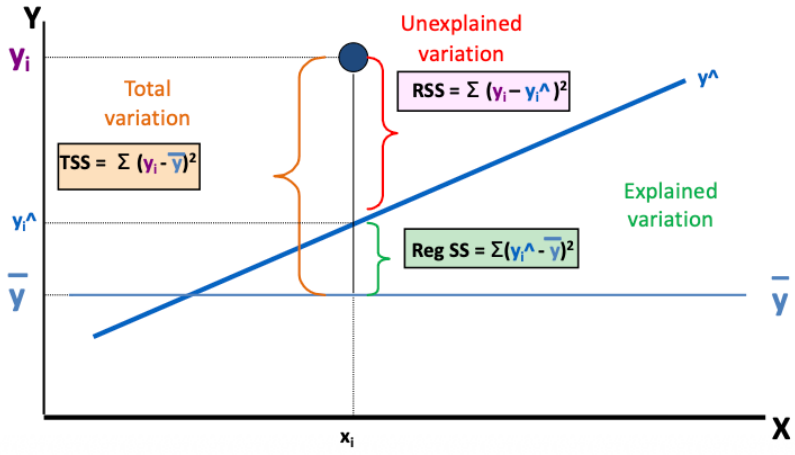
TSS: Total sum of squares. Measures the total variation of the observed responses around their mean $\bar{y}$. TSS captures how spread out the data are before using any model. $\text{TSS} = \sum_{i=1}^{n}(y_i - \bar{y})^2 = (n-1)\, s_Y^2$,

where s_Y^2 **is the sample variance of** Y.

The total sum of squares equals the sample variance multiplied by its degrees of freedom, representing total (unnormalized) variability in the response variable.

RegSS or ESS : Regression sum of squares [Explained Sum of Squares]. Measures the variation in Y explained by the linear relationship between X and Y. RegSS reflects how much of the total variation is captured by the fitted regression line.

RSS: Residual sum of squares. Measures the variation in Y not explained by the regression model. RSS represents unexplained variation due to noise, omitted variables, or model misspecification.



⋆ R^2 **(coefficient of determination).** The R^2 statistic measures the fraction of total variation in the response variable explained by the regression model:

$$\boxed{R^2 = \frac{\text{TSS} - \text{RSS}}{\text{TSS}} = 1 - \frac{\text{RSS}}{\text{TSS}} = \frac{\text{RegSS}}{\text{TSS}}}$$

R^2 compares unexplained variation to total variation, providing a unit-free measure of overall model fit between 0 and 1.

Interpretation: An $R^2 = 0.75$ means that 75% of the total variation in the response variable Y is explained by the regression model, while the remaining 25% is due to unexplained variation and noise.

R^2 **Relationship with correlation** In simple linear regression with one predictor,

$$R^2 = r^2,$$

where r is the sample correlation between X and Y:

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2}\sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}} = \frac{S_{XY}}{S_X S_Y}.$$

In the simple regression case, the explanatory power of the model is entirely determined by the strength of linear association between X and Y.

1.2.5 Correlations Among Predictors (Multicollinearity)

⋆ **Correlations among predictors (multicollinearity).** When predictors are highly correlated, several problems arise: 1) The **variance of coefficient estimates** increases, sometimes dramatically. 2) Interpretation becomes hazardous: changing one predictor X_j typically changes other predictors as well.

Correlation among predictors reduces the amount of independent information available to estimate each coefficient.

Why coefficient variance increases. In multiple linear regression, the variance of the OLS estimator satisfies

$$\text{Var}(\hat{\beta}) = \sigma^2 (X^\top X)^{-1}.$$

When predictors are highly correlated, the matrix $X^\top X$ becomes nearly singular, causing $(X^\top X)^{-1}$ to have large entries. This inflates the variances of the coefficient estimates, making them unstable and sensitive to small changes in the data.

1.2.6 Multiple Regression

⋆ **Multiple least squares regression (OLS).** In multiple regression, the fitted value for observation i is

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \hat{\beta}_2 x_{i2} + \cdots + \hat{\beta}_p x_{ip}.$$

The **residual sum of squares** is

$$\text{RSS} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} \Big(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_{i1} - \hat{\beta}_2 x_{i2} - \cdots - \hat{\beta}_p x_{ip} \Big)^2.$$

Multiple regression extends simple regression by allowing p predictors. OLS chooses coefficients that make the overall squared residuals as small as possible.

OLS coefficient estimates. The values $\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_p$ that minimize RSS are the multiple least squares regression coefficient estimates:

$$(\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_p) = \arg \min_{\beta_0, \beta_1, \ldots, \beta_p} \sum_{i=1}^{n} \Big(y_i - \beta_0 - \beta_1 x_{i1} - \cdots - \beta_p x_{ip} \Big)^2.$$

Each coefficient is chosen jointly, because changing one predictor's coefficient changes the fitted values for all observations.

Matrix form (multiple regression). Let

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \qquad X = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{bmatrix}, \qquad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \qquad \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_n \end{bmatrix}.$$

Then the model is

$$y = X\beta + \varepsilon, \qquad \hat{y} = X\hat{\beta}, \qquad e = y - \hat{y} = y - X\hat{\beta}.$$

This notation treats the dataset as one vector equation. Each row of X corresponds to one observation, and the intercept is included as the first column of ones.

OLS in matrix form. OLS chooses $\hat{\beta}$ to minimize the residual sum of squares

$$\text{RSS}(\beta) = \|y - X\beta\|^2 = (y - X\beta)^\top (y - X\beta),$$

so

$$\hat{\beta} = \arg\min_{\beta} \ \|y - X\beta\|^2.$$

Minimizing $\|y - X\beta\|^2$ is the same as minimizing the sum of squared residuals across all observations.

Squared norm of the residual vector. Let $r = y - X\hat{\beta}$ denote the vector of residuals. Then

$$\|r\|^2 = r^\top r = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \text{RSS}.$$

The squared Euclidean norm of the residual vector equals the residual sum of squares. Minimizing $\|r\|^2$ is therefore equivalent to minimizing the total squared prediction error across all observations.

⋆ **Standard error (SE) of $\hat{\beta}_j$ in multiple regression.** In multiple regression, the standard error of a coefficient estimate is

$$\mathrm{SE}(\hat{\beta}_j) = \sqrt{\hat{\sigma}^2 \, [(X^\top X)^{-1}]_{jj}},$$

where $\hat{\sigma}^2 = \mathrm{RSS}/(n-p-1)$ and $[(X^\top X)^{-1}]_{jj}$ is the j-th diagonal element of $(X^\top X)^{-1}$.

Unlike simple regression, the standard error depends not only on the variability of X_j but also on its correlation with other predictors through $(X^\top X)^{-1}$.

⋆ **Confidence interval for β_i in multiple regression.** An interval that will contain the true unknown value of the parameter β_i in $1-\alpha$ percent of times is

$$\left[\hat{\beta}_i - t_{n-p-1,\alpha/2}\,\mathrm{SE}(\hat{\beta}_i),\ \hat{\beta}_i + t_{n-p-1,\alpha/2}\,\mathrm{SE}(\hat{\beta}_i)\right].$$

The degrees of freedom are $n-p-1$ because we estimate p slopes plus an intercept. The t-critical value accounts for estimating the error variance from the residuals.

⋆ **Hypothesis testing for coefficients in multiple regression.** For each coefficient β_j in a multiple regression model with p predictors, hypothesis testing is conducted separately.

***t*-statistic.** The test statistic for coefficient β_j is

$$t_j = \frac{\hat{\beta}_j - 0}{\mathrm{SE}(\hat{\beta}_j)},$$

where $\mathrm{SE}(\hat{\beta}_j)$ is the standard error of $\hat{\beta}_j$, accounting for error variance and correlations among predictors. The form of the t-statistic is unchanged from simple regression; only the standard error differs because coefficients are estimated jointly.

Sampling distribution. Under the null hypothesis $H_0 : \beta_j = 0 \Rightarrow t_j \sim t_{n-p-1}$.
The degrees of freedom decrease to $n-p-1$ because p slope coefficients and one intercept are estimated.

***p*-value.** The two-sided p-value for testing $H_0 : \beta_j = 0$ is $p_j = P(|T| \geq |t_j| \,|\, H_0)\,, \qquad T \sim t_{n-p-1}$.
The p-value measures how extreme the observed coefficient estimate is relative to the same t-distribution used for critical-value testing.

Interpretation. A small p_j (or large $|t_j|$) provides evidence that predictor X_j has a nonzero partial effect on Y, holding all other predictors fixed.

1.2.7 F-Test for Overall Significance

⋆ **F-Test for Overall Significance.** The **F-test** determines if there is a linear relationship between **all** of the **predictors** $(X_1, \ldots, X_p)$ considered together and the **outcome** (Y).

Hypotheses.

$$\boxed{\begin{aligned} &H_0 : \beta_1 = \beta_2 = \cdots = \beta_p = 0 \quad \text{(no linear relationship)} \\ &H_1 : \text{at least one } \beta_i \neq 0 \quad \text{(at least one predictor is useful)} \end{aligned}}$$

This test evaluates the global utility of the model. It checks whether the group of independent variables provides more explanatory power than a model with only an intercept.

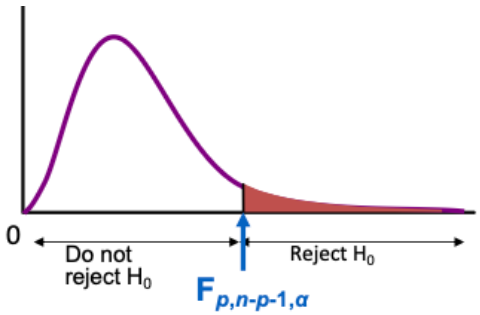
⋆ **F-statistic.** The test uses the ratio of explained variance to unexplained variance to determine if at least one predictor is useful:

$$\boxed{F = \frac{(\text{TSS} - \text{RSS})/p}{\text{RSS}/(n-p-1)} \sim F_{p,n-p-1}}$$

- **Numerator degrees of freedom**: p (number of predictors)
- **Denominator degrees of freedom**: $n - p - 1$.

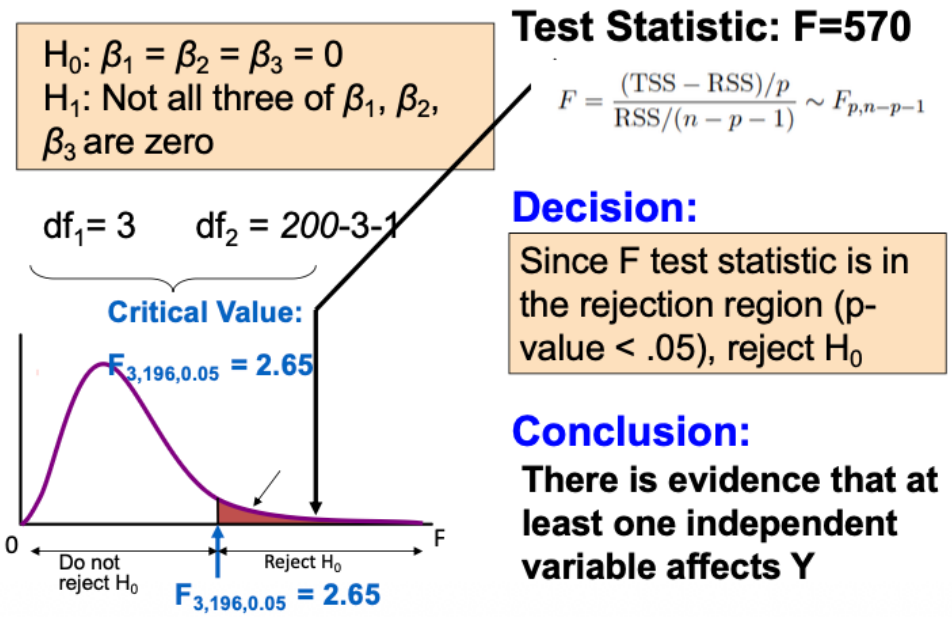
Decision Rule. Reject H_0 if the observed F is sufficiently large:

$$\boxed{F > F_{p,n-p-1,\alpha}}$$



If the F-statistic exceeds the critical value, we conclude that the model explains a significant portion of the response variation. The F-distribution is non-negative and right-skewed, meaning we only reject in the upper tail.

Example:



Test Statistic: F=570

$$F = \frac{(\text{TSS} - \text{RSS})/p}{\text{RSS}/(n-p-1)} \sim F_{p,n-p-1}$$

Decision:

Since F test statistic is in the rejection region (p-value < .05), reject H_0

Conclusion:

There is evidence that at least one independent variable affects Y

1.2.8 Deciding on the Important Variables

⋆ **Deciding on the important variables.** Often we cannot examine all possible models, since there are 2^p of them; for example, when $p = 40$ there are over a trillion models. Instead, we need an **automated approach** that searches through a subset.

Forward selection.

- Begin with the **null model** (contains an intercept but no predictors).
- Fit p simple linear regressions and add to the null model the variable that results in the **lowest RSS**.
- Add to that model the variable that results in the lowest RSS among all two-variable models.
- Continue until some **stopping rule** is satisfied, for example when all remaining variables have a p-value above some threshold.

Forward selection is a constructive method starting from nothing. It is efficient but can be suboptimal if the inclusion of an early variable prevents a better combination of variables later.

Backward .

- Start with **all variables** in the model.
- Remove the variable with the **largest p-value** (the least statistically significant).
- The new $(p-1)$-variable model is fit, and the variable with the largest p-value is removed.
- Continue until a **stopping rule** is reached, such as when all remaining variables have a significant p-value.

Backward selection begins with the full model and prunes it down. This method requires that the number of observations n is larger than the number of predictors p to allow the full model to be estimated.

1.2.9 Qualitative Predictors

⋆ **Qualitative Predictors.** Some **predictors** are not quantitative but are **qualitative**, taking a discrete set of values. These are also called **categorical predictors** or **factor variables**.

Dummy variables (Binary). For a variable with two levels (e.g., gender), we create a numerical **indicator**:

$$x_i = \begin{cases} 1 & \text{if } i\text{th person is female} \\ 0 & \text{if } i\text{th person is male} \end{cases} \implies y_i = \beta_0 + \beta_1 x_i + \epsilon_i = \begin{cases} \beta_0 + \beta_1 + \epsilon_i & \text{if female} \\ \beta_0 + \epsilon_i & \text{if male} \end{cases}$$

In this binary model, β_0 represents the average response for the baseline group (male), while β_1 represents the average difference between females and males.

⋆ **Qualitative predictors with > 2 levels.** With more than two levels, we create **additional dummy variables**. There will always be **one fewer dummy variable** than the total number of levels.

Baseline level. The level with no assigned dummy variable is the **baseline**. All other coefficients are interpreted as **offsets** from this baseline.

Multi-level Example (Ethnicity). For levels {Asian, Caucasian, African American}, we define:

$$x_{i1} = \begin{cases} 1 & \text{Asian} \\ 0 & \text{not Asian} \end{cases}, \quad x_{i2} = \begin{cases} 1 & \text{Caucasian} \\ 0 & \text{not Caucasian} \end{cases}$$

The resulting regression model becomes:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \epsilon_i = \begin{cases} \beta_0 + \beta_1 + \epsilon_i & \text{if Asian} \\ \beta_0 + \beta_2 + \epsilon_i & \text{if Caucasian} \\ \beta_0 + \epsilon_i & \text{if African Am. (Baseline)} \end{cases}$$

Using $k-1$ dummies avoids the "dummy variable trap" (perfect collinearity). The intercept β_0 captures the mean of the baseline category, and the t-tests for β_1, β_2 check for significant differences against that baseline.

1.2.10 Interaction Effects and Non-linear Relationships

⋆ **Interaction Effects (Synergy).** The **additive assumption** in linear models states that the effect of a predictor on the response is independent of other predictors. **Interactions** relax this by allowing the impact of one variable to depend on the level of another.

Interaction Model Equation. For two predictors X_1 and X_2, the model includes a third term $(X_1 \times X_2)$:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 (X_1 \times X_2) + \epsilon$$

By factoring, we see the **adjusted slope** for X_1 is $(\beta_1 + \beta_3 X_2)$.

Interactions capture synergy, where the joint effect of predictors is greater (or smaller) than the sum of their individual effects. In practice, if an interaction term is significant, we must include the main effects even if their individual p-values are not significant.

⋆ **Hierarchical Principle.** If we include an **interaction** in a model, we should also include the **main effects**, even if the p-values associated with their coefficients are not statistically significant.

Hypothesis Testing for Interactions. We test the null hypothesis $H_0 : \beta_3 = 0$ using a **t-statistic** or **F-test**. A low p-value indicates strong evidence that the interaction effect exists.

⋆ **Model Comparison and R-squared.** Adding interaction terms typically increases the R^2, as the model becomes more flexible and can explain a higher percentage of the **variability** in the response.

To evaluate if the added complexity is justified, we look at the **proportion of remaining variability explained by interaction.**

$$\text{Explained Residual Variation} = \frac{R^2_{\text{interaction}} - R^2_{\text{additive}}}{100 - R^2_{\text{additive}}}$$

This formula quantifies how much of the variation left unexplained by the additive model is captured by adding an interaction term. It expresses the improvement in model fit relative to the remaining "error" in the simpler model.

⋆ **Non-linear Relationships.** The linear model can be extended to accommodate **non-linear relationships** between the predictors and the response by using **polynomial regression**.

⋆ **Interaction between Quantitative and Qualitative Variables.** In a general case, we can allow the relationship between a **quantitative predictor** (X) and the **outcome** (Y) to vary depending on the level of a **qualitative variable** (D). This effectively creates different regression lines for different categories.

The Interaction Model Equation. For a quantitative variable X and a binary dummy variable $D \in \{0, 1\}$, the model is:

$$\boxed{Y = \beta_0 + \beta_1 X + \beta_2 D + \beta_3 (X \times D) + \epsilon}$$

By rearranging terms based on the value of D, we obtain two distinct models:

- **Baseline Category ($D = 0$):** $Y = \beta_0 + \beta_1 X + \epsilon$
- **Comparison Category ($D = 1$):** $Y = (\beta_0 + \beta_2) + (\beta_1 + \beta_3) X + \epsilon$

This model allows for both a different intercept and a different slope for each group. β_2 represents the change in the intercept, while β_3 represents the change in the slope (the interaction effect) between the two categories. **Comparison of Model Types.**

Model Type	Equation	Geometric Result
Additive	$Y = \beta_0 + \beta_1 X + \beta_2 D$	Two parallel lines
Interaction	$Y = \beta_0 + \beta_1 X + \beta_2 D + \beta_3 (XD)$	Two lines with different slopes

⋆ **Interpretation of Coefficients.**

- β_1: The slope of the quantitative predictor for the **baseline group**.
- β_2: The **difference in the intercept** between the comparison group and the baseline group.
- β_3: The **difference in the slope** (rate of change) between the comparison group and the baseline group.

If β_3 is statistically significant, it implies that the effect of the quantitative variable X on the response Y depends on the category D. Ignoring this interaction when it exists would lead to a model that assumes a uniform effect across all groups, which is a significant oversimplification.

Polynomial Regression. To account for a non-linear "curve" in the data, we can include **squared** or **higher-order** terms as independent predictors:

$$\boxed{Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_d X^d + \epsilon}$$

While this is a non-linear function of the predictor X, it is still a linear model because it is linear in the coefficients β. This allows us to use standard least squares to fit complex, curved patterns in the data.

1.3 Extra

1.3.1 Confidence Intervals and Sample Size

⋆ **Confidence Intervals and Sample Size.** The central limit theorem provides an approximate normal law for the standardized sample mean. Confidence intervals invert this approximation: rather than studying the distribution of $\bar{X}_n$ for a fixed mean μ_X, one constructs a random interval centered at $\bar{X}_n$ that contains the unknown parameter μ_X with a prescribed long–run frequency.

Setup. Assume $X_1, \dots, X_n$ are i.i.d. with mean μ_X and known variance σ_X^2. For sufficiently large n, the CLT yields

$$Z_n = \frac{\bar{X}_n - \mu_X}{\sigma_X/\sqrt{n}} \xrightarrow{d} Z \sim \mathcal{N}(0,1),$$

so the distribution of Z_n is well approximated by the standard normal law.

Confidence Interval for the Mean (Known Variance) Let $1-\alpha$ be a desired confidence level and let $z_{\alpha/2}$ denote the $(1-\alpha/2)$–quantile of the standard normal distribution, defined by

$$P(-z_{\alpha/2} \le Z \le z_{\alpha/2}) = 1-\alpha, \qquad Z \sim \mathcal{N}(0,1).$$

Using the CLT approximation for Z_n,

$$P\left(-z_{\alpha/2} \le \frac{\bar{X}_n - \mu_X}{\sigma_X/\sqrt{n}} \le z_{\alpha/2}\right) \approx 1-\alpha.$$

Rearranging the inequalities yields

$$\boxed{P\left(\bar{X}_n - z_{\alpha/2}\frac{\sigma_X}{\sqrt{n}} \le \mu_X \le \bar{X}_n + z_{\alpha/2}\frac{\sigma_X}{\sqrt{n}}\right) \approx 1-\alpha}$$

The confidence level $1-\alpha$ is a *coverage probability*: if the sampling experiment were repeated indefinitely and a new interval constructed each time, the proportion of intervals containing the fixed parameter μ_X would approach $1-\alpha$. The randomness lies in the interval, not in μ_X.

⋆ **Confidence Interval for the Mean (Unknown Variance)**

Setup. Assume $X_1, \dots, X_n$ are i.i.d. with mean μ_X and unknown variance σ_X^2. Let

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i, \qquad S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X}_n)^2$$

Studentized mean. When σ_X^2 is unknown, replace it by S^2 to form $T_n = \frac{\bar{X}_n - \mu_X}{S/\sqrt{n}} \sim t_{n-1}$, where t_{n-1} denotes the Student t-distribution with $n-1$ degrees of freedom.

Estimating the variance introduces extra uncertainty, which is captured by the heavier tails of the t-distribution compared to the normal distribution.

Let $1-\alpha$ **be the desired confidence level** and let $t_{\alpha/2,n-1}$ denote the $(1-\alpha/2)$-quantile of the t_{n-1} distribution. Then

$$\boxed{P\left(\bar{X}_n - t_{\alpha/2,n-1}\frac{S}{\sqrt{n}} \le \mu_X \le \bar{X}_n + t_{\alpha/2,n-1}\frac{S}{\sqrt{n}}\right) = 1-\alpha}$$

When the variance is unknown, confidence intervals are wider to reflect additional estimation uncertainty. As n increases, the t-distribution approaches the standard normal and the interval converges to the known-variance case.

Sample Size Determination. Suppose a maximal half–width $\varepsilon > 0$ is prescribed, so that

$$z_{\alpha/2}\frac{\sigma_X}{\sqrt{n}} \le \varepsilon.$$

Solving for n yields the sample size requirement

$$n \ge \left(\frac{z_{\alpha/2}\sigma_X}{\varepsilon}\right)^2.$$

Higher confidence levels (smaller α) and tighter accuracy requirements (smaller ε) both increase the required sample size, expressing the fundamental trade–off between reliability and precision.

Two-predictor case: correlation inflates variance. Consider the multiple linear regression with two predictors:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon, \qquad \mathbb{E}(\varepsilon \mid X) = 0, \qquad \mathrm{Var}(\varepsilon \mid X) = \sigma^2.$$

With two predictors, we can see explicitly how correlation between x_1 and x_2 increases the variance of the OLS slope estimates.

OLS variance formula. Let $X = [\mathbf{1}, x_1, x_2]$. Then

$$\mathrm{Var}(\hat{\beta} \mid X) = \sigma^2 (X^\top X)^{-1}.$$

1.3.2 Multicollinearity: Two-Predictor Proof

Proof (center predictors to isolate the slopes). Let $\tilde{x}_j = x_j - \bar{x}_j$ be centered predictors and $\tilde{X} = [\tilde{x}_1, \tilde{x}_2]$. (Centering does not change $\hat{\beta}_1, \hat{\beta}_2$.) Then

$$\mathrm{Var}\left(\begin{bmatrix}\hat{\beta}_1 \\ \hat{\beta}_2\end{bmatrix} \,\middle|\, \tilde{X}\right) = \sigma^2 (\tilde{X}^\top \tilde{X})^{-1}.$$

Compute

$$\tilde{X}^\top \tilde{X} = \begin{bmatrix} \sum_{i=1}^n \tilde{x}_{1i}^2 & \sum_{i=1}^n \tilde{x}_{1i}\tilde{x}_{2i} \\ \sum_{i=1}^n \tilde{x}_{1i}\tilde{x}_{2i} & \sum_{i=1}^n \tilde{x}_{2i}^2 \end{bmatrix} = \begin{bmatrix} S_{11} & S_{12} \\ S_{12} & S_{22} \end{bmatrix}.$$

Using the 2×2 inverse formula,

$$(\tilde{X}^\top \tilde{X})^{-1} = \frac{1}{S_{11}S_{22} - S_{12}^2} \begin{bmatrix} S_{22} & -S_{12} \\ -S_{12} & S_{11} \end{bmatrix}.$$

Therefore,

$$\mathrm{Var}(\hat{\beta}_1 \mid X) = \sigma^2 \frac{S_{22}}{S_{11}S_{22} - S_{12}^2}, \qquad \mathrm{Var}(\hat{\beta}_2 \mid X) = \sigma^2 \frac{S_{11}}{S_{11}S_{22} - S_{12}^2}.$$

Now write S_{12} in terms of the sample correlation r_{12}:

$$r_{12} = \frac{S_{12}}{\sqrt{S_{11}S_{22}}} \quad \Rightarrow \quad S_{11}S_{22} - S_{12}^2 = S_{11}S_{22}(1 - r_{12}^2).$$

Plugging in yields

$$\boxed{\mathrm{Var}(\hat{\beta}_1 \mid X) = \frac{\sigma^2}{S_{11}(1 - r_{12}^2)}, \qquad \mathrm{Var}(\hat{\beta}_2 \mid X) = \frac{\sigma^2}{S_{22}(1 - r_{12}^2)}.}$$

The inflation factor is $\frac{1}{1-r_{12}^2}$. As $|r_{12}| \to 1$, we have $1 - r_{12}^2 \to 0$, so the variances blow up: coefficient estimates become unstable and standard errors become large.

Chapter 2

Classification

2.1 Classification Problems and Concepts

⋆ **Classification.** In classification, **qualitative variables** take values in an unordered set C.

- Examples: eye color $\in$ {brown, blue, green}, email $\in$ {spam, ham}.
- Digit recognition: digit $\in \{0, 1, \ldots, 9\}$ (even though they are numbers, **order does not matter** for this task).

Task of classification Given a feature vector (X) and a qualitative response (Y) taking values in C, the goal is to build a classification function $C(X)$ that predicts the value of Y: $\boxed{C(X) \in C}$

⋆ **Class Probabilities.** Rather than just assigning a hard label, we are often more interested in estimating the **probabilities** *(measure of confidence for our predictions)* that an observation X belongs to each specific category k in C. $\boxed{p_k(x) = \Pr(Y = k \mid X = x)}$

⋆ **IQR Outlier Detection Rules.** *(Boxplots)* The **Interquartile Range (IQR)** is used to identify outliers: $\text{IQR} = Q_3 - Q_1$
Data points are considered outliers if they fall outside the bounds:

$$\boxed{X > Q_3 + 1.5 \cdot \text{IQR} \quad \text{or} \quad X < Q_1 - 1.5 \cdot \text{IQR}}$$

⋆ **Multiclass classification.** A classification task with **more than two classes**. It assumes that each sample is **assigned to one and only one label** (mutually exclusive). *EX. Classifying an image as a horse, a bird, or a fish. An animal can be a horse or a bird, but not both simultaneously.*

⋆ **Multilabel classification.** A classification task that assigns a **set of target labels** to each sample. These labels predict properties of a data point that are **not mutually exclusive**.

- Example: Document topic tagging. A single text might be about religion, politics, and finance at the same time, or none of these.

Multilabel models evaluate the presence or absence of multiple independent traits concurrently, allowing a single observation to effectively belong to multiple categories at once.

⋆ **Binary classification.** A classification task that assigns **only one of two possible classes** to each observation. **Complex multi-class and multi-label tasks** can frequently be **broken down and solved using standard binary classification** techniques (e.g., one-vs-all), **binary classification forms the core foundational focus of many classification methods.**

2.1.1 Odds, Likelihood, and Cross-Entropy

⋆ **Odds.** The probability of an event divided by the probability of its complement is called its **odds**.
Ex. The probability of winning in a casino is 1%.

Odds of winning in that casino: $O(W) = \frac{\Pr(W)}{1-\Pr(W)} = \frac{0.01}{0.99} = \frac{1}{99}$

⋆ **Bernoulli Random Variable.** A **Bernoulli random variable** ($Y \in \{0, 1\}$) represents a single trial with exactly two possible, mutually exclusive outcomes: typically called "success" ($Y = 1$) and "failure" ($Y = 0$).

PMF: Let p be the probability of success. The probabilities of the two outcomes are:
- $\Pr(Y = 1) = p$ and $\Pr(Y = 0) = 1 - p$

This can be written in a single, compact equation for $y \in \{0, 1\}$:

$$\boxed{\Pr(Y = y) = p^y(1 - p)^{1-y}} \quad \mathbb{E}(Y) = p \quad \text{and} \quad \text{Var}(Y) = p(1 - p)$$

⋆ **Joint Probability vs. Likelihood.** Mathematically, are the exact same formula. The difference is entirely in **perspective**, which values are considered fixed and which are allowed to vary.

Joint Probability (function of the data). Treats the parameters (β_0, β_1) as **fixed and known**, and treats the data (Y) as the variable.
Question: "Given these specific parameters, what is the probability of generating this exact dataset?"

⋆ **Likelihood Function (function of the parameters).** Treats the observed data (Y) as **fixed and known**, and treats the parameters (β_0, β_1) as the variables.
Question: "Given the data we already observed, how likely is it that different values of β_0 and β_1 generated it?"

Idea: Before you collect data, you use the joint probability to think about what might happen. After you collect data, the data is locked in (constant), so you use the likelihood function to slide the parameter values around until you find the ones that best explain the data you actually got.

⋆ **Bernoulli Joint probability mass function.** For a dataset of N independent observations $\{(x_i, y_i)\}_{i=1}^N$, the **joint probability mass function** is the product of the individual Bernoulli probabilities:

$$\boxed{\Pr(Y_1 = y_1, \ldots, Y_N = y_N \mid X) = \Pr(Y_1 = y_1 \mid x_1) \times \cdots \times \Pr(Y_N = y_N \mid x_N) =}$$

$$\boxed{= \prod_{i=1}^{N} \Pr(Y_i = y_i \mid x_i) = \prod_{i=1}^{N} p(x_i)^{y_i} \Big(1 - p(x_i)\Big)^{1-y_i}}$$

Hint let $p(x_i) = \frac{e^{\beta_0+\beta_1 x_i}}{1+e^{\beta_0+\beta_1 x_i}}$

⋆ **The Likelihood Function.** The likelihood of observing this specific sequence of k heads and $N-k$ tails, as a function of p, is: $\boxed{L(p) = \prod_{i=1}^{N} p^{y_i}(1 - p)^{1-y_i} = p^k(1 - p)^{N-k}}$

⋆ **Maximum Likelihood:** Consider a simple experiment: flipping a biased coin N times with an unknown probability of heads p. We observe k heads ($Y = 1$) and $N - k$ tails ($Y = 0$). We want to find the best estimate for p. With **Likelihood Function** $\boxed{L(p) = p^{y=k}(1 - p)^{1-y=N-k}}$

For example: Suppose we flip a coin $N = 10$ times and observe exactly $k = 7$ heads. The likelihood function for this specific sequence is $L(p) = p^7(1-p)^3$. If we test a specific hypothesis for p, say $p = 0.5$ (a fair coin), we get a specific numerical **likelihood value**: $L(0.5) = (0.5)^7(1 - 0.5)^3 \approx 0.00097$

This number is **not the probability that the coin is fair**. Instead, it tells us: " **Probability of getting exactly this sequence of 7 heads and 3 tails is 0.00097, if the coin were perfectly fair,"** It measures how well the parameter $p = 0.5$ explains our fixed data.

The Likelihood Ratio as a value. Now suppose we want to compare two competing hypotheses: our fair coin guess ($p = 0.5$) versus a biased coin guess ($p = 0.7$). First, we calculate the likelihood value for the biased guess: $L(0.7) = (0.7)^7(1-0.7)^3 \approx 0.00222$

The **likelihood ratio** directly compares these two competing models by dividing their likelihood values: $\boxed{\text{Ratio} = \frac{L(0.7)}{L(0.5)} = \frac{0.00222}{0.00097} \approx 2.29}$

Likelihood ratio of 2.29 means that our **exact observed dataset is about 2.29 times more mathematically probable to occur if the coin's true bias is $p = 0.7$ compared to if it were perfectly fair at $p = 0.5$.** The ratio gives us concrete, relative evidence of which parameter is a "better fit" for the reality we observed.

⋆ **The Hidden Connection to Odds.** We can algebraically rearrange the likelihood equation to expose the **odds ratio**. By factoring out $(1-p)$, we get: $\boxed{p^y(1-p)^{1-y} = (1-p)\left(\frac{p}{1-p}\right)^y}$

Term inside the parentheses: $\frac{p}{1-p}$. That is exactly the definition of **Odds**!

Odds Vs Likelihood: **Odds** is about a single event, ratio of the probability of an event happening to the probability of it not happening, while **Likelihood** is about your entire dataset and your model. It is the joint probability of observing your entire dataset given a specific set of rules (parameters).

The Connection to Log-Likelihood. If you look closely at the formula, the term inside the brackets is exactly the log of the Bernoulli likelihood we derived earlier!

- **Statisticians** want to *maximize* the log-likelihood function.
- **Computer Scientists** want to *minimize* a cost function.

Because maximizing a function is mathematically identical to minimizing its negative, Minimizing Cross Entropy and Maximizing Log-Likelihood are the exact same operation under the hood. The cross entropy simply slaps a negative sign on the log-likelihood so that gradient descent has a "valley" to roll down into, heavily penalizing the model when it confidently predicts the wrong class.

⋆ **Binary Cross Entropy = Negative Log-Likelihood.** In statistics, the goal is to **maximize the log-likelihood of the data**. In machine learning, algorithms like Gradient Descent are designed to **minimize a loss function**. These two approaches are mathematically identical.

Log-Likelihood (Statistical view). The log-likelihood $\ell(\beta)$ measures how well the model explains the data. We want to maximize this sum:

$$\log \ell(\beta) = \sum_{i=1}^{N} [y_i \log(\hat{p}_i) + (1-y_i)\log(1-\hat{p}_i)]$$

Binary Cross Entropy (Machine Learning view). To turn this into a loss function that we can minimize, we multiply by -1 (flipping the "peak" of the mountain into the "bottom" of a valley) and average it over the N samples:

$$\text{BCE}(\beta) = -\frac{1}{N}\sum_{i=1}^{N} [y_i \log(\hat{p}_i) + (1-y_i)\log(1-\hat{p}_i)]$$

The Exact Relationship. Comparing the two equations side-by-side reveals their direct mathemat-

ical connection:

$$\boxed{\text{BCE}(\beta) = -\frac{1}{N}\ell(\beta)}$$

Minimizing the Binary Cross Entropy is exactly the same operation as maximizing the Log-Likelihood. The negative sign simply flips the optimization direction (so we can find a minimum), and the $\frac{1}{N}$ scaling factor ensures the loss doesn't artificially inflate just because we added more data points.

2.2 Class Imbalance

Classification dataset is **imbalanced** when one or more classes are rare (few observations), but those rare cases may contain the most important information.

$$\boxed{\text{Class imbalance } \not\Rightarrow \text{ class irrelevance}}$$

Types of imbalance (core distinction). This distinction matters because the data can be globally imbalanced (rare labels overall) or locally easy/hard depending on the predictor values.

1. Marginal imbalance. A dataset has **marginal imbalance** when one class is rare in the overall class distribution:

$$\boxed{\Pr(Y = 1) \approx 0 \quad \text{(or very small)}}$$

for binary **label** $Y \in \{0, 1\}$. This is the simplest imbalance notion: the positive class is rare overall. It describes class proportions without using features X.

2. Conditional imbalance. A dataset has **conditional imbalance** when labels are highly predictable for most predictor values, so conditional probabilities are near 0 or 1. For example, if $X \in \{0, 1\}$,

$$\boxed{\Pr(Y = 1 \mid X = 0) \approx 0 \qquad \text{and} \qquad \Pr(Y = 1 \mid X = 1) \approx 1.}$$

The issue is not just rare classes overall, but extreme class probabilities given features. The classification problem becomes very uneven across regions of the feature space.

Solutions:

+ Subsampling (down-sampling) A common strategy is to **down-sample** the majority class in the training set to enrich the rare class proportion.

$$\boxed{\text{Training set: reduce majority class count to improve balance}}$$

Critical evaluation rule for down-sampling. If the training set is artificially balanced by random sampling, the **test set** should still reflect the real class imbalance (the state of nature).

+ Upsampling idea. Another strategy is to **up-sample** the minority class by sampling minority examples with replacement until classes are approximately balanced.

$$\boxed{\text{Training set: duplicate minority cases (with replacement) to increase balance}}$$

⋆ **SMOTE (Synthetic Minority Over-sampling Technique).** **SMOTE** is a data re-sampling procedure for **class imbalance** that combines **up-sampling** of the minority class (by creating synthetic points) and, if desired, **down-sampling** of the majority class.

Core SMOTE idea (minority synthesis). To create a new minority-class point, SMOTE:
1) randomly selects a minority observation x_i,

2) finds its K-nearest minority neighbors,
3) generates a synthetic point as a random combination along local neighbor directions.

$$\boxed{X_{\text{new}} = \frac{w_1 x_1 + \cdots + w_K x_K}{w_1 + w_2 + \cdots + w_K} \quad w_i \sim U[0, 10]}$$

A common operational choice is $K = 5$ (often a default).

The new point is built from nearby minority points, so it stays in a region where minority cases already exist. This helps expand minority support more smoothly than exact duplication.

⋆ **Case-control sampling and logistic regression.** In **case-control sampling**, the sample class proportion is intentionally different from the population prevalence (e.g., oversampling cases). Let

$$\pi = \Pr(Y = 1) \quad \text{(population prevalence)}, \qquad \tilde{\pi} = \Pr_{\text{sample}}(Y = 1) \quad \text{(sample case fraction)}.$$

This is common when cases are rare: we sample more cases than nature provides so estimation becomes feasible. The sample is then useful for learning relationships, but its class proportion is no longer the true prevalence.

Effect on logistic regression coefficients

- **Slope effects:** $\hat{\beta}_1, \ldots, \hat{\beta}_p$ are not biased by case-control sampling
  The slopes describe how predictors change log-odds relative to each other, and that information is retained.
- **Intercept effects:** $\hat{\beta}_0$ is generally incorrect for population probabilities
  The intercept depends on the baseline class prevalence, which the sampling scheme changes on purpose.

Intercept correction: If $\hat{\beta}_0$ is the intercept estimated from the case-control sample and π is the true population prevalence while $\tilde{\pi}$ is the sample prevalence, then a corrected intercept is

$$\boxed{\hat{\beta}_0^* = \hat{\beta}_0 + \log\left(\frac{\pi}{1-\pi}\right) - \log\left(\frac{\tilde{\pi}}{1-\tilde{\pi}}\right)}$$

This replaces the sample baseline odds with the population baseline odds. It fixes probability calibration while preserving the estimated predictor effects.

2.3 Bayes' Rule and the Connection to Discriminant Analysis

⋆ **Law of total probability (partition form).** Let $\{H_1, \ldots, H_n\}$ be a **partition** of the sample space Ω, i.e. $H_i \cap H_j = \varnothing$ **for** $i \neq j$ **and** $\bigcup_{i=1}^n H_i = \Omega$.
For any **event** E, $E = (E \cap H_1) \cup (E \cap H_2) \cup \cdots \cup (E \cap H_n)$, and these pieces are disjoint. Therefore,

$$\boxed{P(E) = \sum_{i=1}^{n} P(E \cap H_i).}$$

Conditional probability and multiplication rule. For any hypothesis/event H_i with $P(H_i) > 0$,

$$\boxed{P(E \mid H_i) = \frac{P(E \cap H_i)}{P(H_i)}} \quad \Longleftrightarrow \quad \boxed{P(E \cap H_i) = P(E \mid H_i) P(H_i).}$$

Substituting into the law of total probability gives $\boxed{P(E) = \sum_{i=1}^{n} P(E \mid H_i)\, P(H_i).}$

This rewrites total probability in a form that separates **within-case likelihood** $P(E \mid H_i)$ and **case frequency** $P(H_i)$. It is the key bridge to Bayes classification.

⋆ **Bayes' rule (posterior from likelihood and prior).** For a hypothesis H and evidence E with $P(E) > 0$, $\quad P(H \mid E) = \frac{P(E|H)\,P(H)}{P(E)}$.

More generally, for a partition $\{H_1, \ldots, H_n\}$, $\boxed{P(H_i \mid E) = \dfrac{P(E \mid H_i)\,P(H_i)}{\sum_{j=1}^{n} P(E \mid H_j)\,P(H_j)}.}$

Bayes' rule updates belief in a hypothesis after observing evidence. The **posterior is proportional to likelihood times prior**, then normalized so probabilities across all hypotheses sum to 1.

⋆ **Posterior, likelihood, and prior (classification interpretation).** In Bayes' rule,

$$\boxed{P(H_i \mid E) \propto P(E \mid H_i)P(H_i),} \quad \Rightarrow \quad \boxed{\arg\max_i P(H_i \mid E) = \arg\max_i \Big(P(E \mid H_i)P(H_i)\Big).}$$

- $P(H_i \mid E)$ is the **posterior probability**,
- $P(E \mid H_i)$ is the **likelihood**, how compatible the observed evidence is with each class.
- $P(H_i)$ is the **prior probability**, how plausible each class/hypothesis was before seeing evidence.

Denominator often does not matter for classification. When comparing which hypothesis is most likely after observing the same evidence E, the denominator $P(E)$ is the same for all H_i.

⋆ **Connection to Discriminant Analysis (preview).** In **classification**, treat each class as a hypothesis H_i, and let the observed features be the evidence E (later written as $X = x$). Then Bayes' rule becomes $\boxed{P(H_i \mid E) = \dfrac{P(E \mid H_i)P(H_i)}{\sum_{j=1}^{n} P(E \mid H_j)P(H_j)}}$: core logic behind Bayesian/disc. classification.

Discriminant analysis computes class posteriors by modeling class priors and class-conditional feature distributions. Bayes' rule is the engine that turns those ingredients into classification probabilities.

⋆ **Why discriminant analysis? (core idea) Discriminant analysis** can be more stable than logistic regression when classes are well-separated or when n is small and class-conditional predictors are approximately Gaussian. *It uses a structured model for $X \mid Y$, which can reduce estimation instability compared with directly fitting $P(Y \mid X)$.*

Well-separated classes (logistic instability). With (nearly) linearly separable data, logistic regression can drive coefficients to very large magnitudes: $\boxed{\text{separable data} \;\Rightarrow\; |\hat{\beta}_j| \text{ unstable / very large}}$

The classifier boundary may still look reasonable, but coefficient inference (SEs, p-values) becomes unreliable.

Small n + Gaussian class structure. If n is small and $X \mid Y = k$ is approximately Gaussian, LDA is often more stable

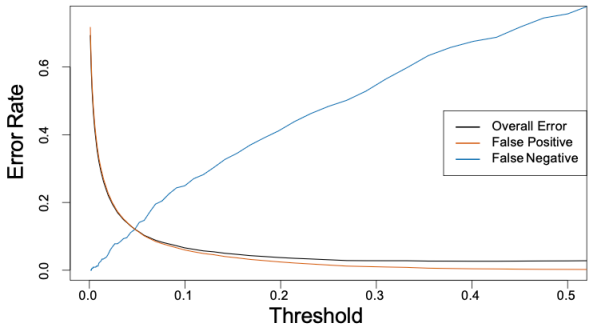
Multiclass advantage. LDA is also popular for $K > 2$ classes because it naturally handles multiclass classification and provides low-dimensional discriminant views. *So LDA is useful both for prediction and for understanding class separation geometry.*

2.4 Measures for Different Types of Error

⋆ **Binary classification error types (threshold-based).** After choosing a threshold (e.g. classify positive if $\hat{p}(x) \geq t$), predictions produce:

- **FP, Type I error (False Positive)** : actual negative predicted positive,
- **FN, Type II error (False Negative)**: actual positive predicted negative.
- **FPR**: tells how often you raise false alarms on negatives; $\boxed{\text{FPR} = \dfrac{FP}{FP + TN}}$
- **FNR**: tells how often you miss true events. $\boxed{\text{FNR} = \dfrac{FN}{FN + TP}}$

Threshold controls the error tradeoff. For probabilistic classifiers, predictions depend on the threshold t: $\boxed{\hat{y} = \begin{cases} 1, & \hat{p}(x) \geq t, \\ 0, & \hat{p}(x) < t. \end{cases}}$ Changing t changes FPR, FNR, recall, precision, and overall error.



Lowering the threshold usually increases recall (fewer false negatives) but also **increases false positives.** Raising the threshold does the opposite.

Sensitivity / Recall / True Positive Rate (TPR). Measures how many actual positives your model successfully detects. **High recall means few misses (low false negatives).**

$$\boxed{\text{Recall} = \text{Sensitivity} = \text{TPR} = \frac{TP}{TP + FN}}$$

Specificity / True Negative Rate (TNR). Measures how many actual negatives your model correctly rejects. **High specificity means few false alarms (low false positives).**

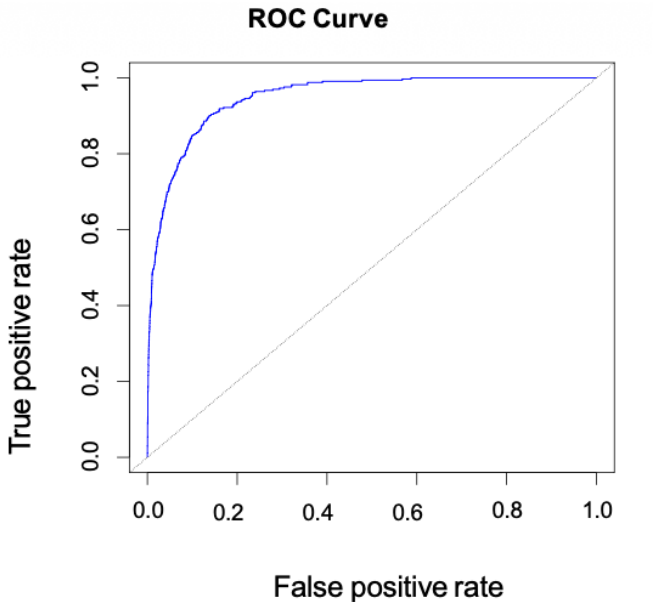
$$\boxed{\text{Specificity} = \text{TNR} = \frac{TN}{TN + FP}}$$

ROC curve and AUC (threshold-free ranking view).

The **ROC curve** plots $\boxed{(\text{FPR}(t),\ \text{TPR}(t))}$ as the threshold t varies. The **AUC** (area under the ROC curve) summarizes ranking performance.

Higher AUC ⇒ better class ranking

ROC shows the tradeoff between catching positives and creating false alarms across thresholds. AUC is useful when you care about ranking quality more than one fixed threshold.



Precision: Positive Predictive Value (PPV) $\boxed{\text{Precision} = \text{PPV} = \dfrac{TP}{TP + FP}}$

Precision answers: of the cases predicted positive, how many were truly positive? High precision means your positive alerts are trustworthy.

Negative Predictive Value (NPV) $\boxed{\text{NPV} = \dfrac{TN}{TN + FN}}$

NPV answers: of the cases predicted negative, how many were truly negative? It is the negative-side analog of precision.

Precision–Recall (PR) Curve. The **PR curve** plots precision against recall as the threshold varies:

$$\boxed{(TPR(t), PPV(t)) \quad = \quad (\text{Recall}(t),\ \text{Precision}(t))}$$

PR curves are especially informative under class imbalance, where ROC curves can look good even when positive-class precision is poor. Use PR when the positive class is rare and important.

F1 score (balance between precision and recall). $\boxed{F_1 = 2 \cdot \dfrac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}}$

F_1 **is the harmonic mean**, so it is **high only when both precision and recall are high.** It is often better than accuracy when classes are imbalanced and both false positives and false negatives matter.

F_β score (skewed precision–recall tradeoff). $\boxed{F_\beta = \dfrac{\beta^2 + 1}{\dfrac{\beta^2}{\text{recall}} + \dfrac{1}{\text{precision}}}}$

Interpretation: $\boxed{\beta > 1 \Rightarrow \text{recall weighted more}, \qquad \beta < 1 \Rightarrow \text{precision weighted more.}}$

Use F_β when the costs are asymmetric. For example, choose larger β when missing positives is much worse than false alarms.

⋆ **Multiple classes: averaging is needed.** Precision/recall/F_β are defined per class (treating one class as "positive" in a one-vs-rest view), so multiclass settings require aggregation.

Macro average (weights classes equally). For a per-class **metric** M_k (e.g. precision or recall),

$$\boxed{M_{\text{macro}} = \frac{1}{K} \sum_{k=1}^{K} M_k}$$

Micro average (weights instances equally). Micro averaging pools counts across classes first, then computes the metric. For precision:

$$\boxed{\text{Precision}_{\text{micro}} = \frac{\sum_{k=1}^{K} TP_k}{\sum_{k=1}^{K} TP_k + \sum_{k=1}^{K} FP_k}}$$

(similarly for recall using pooled FN_k).

Micro averaging emphasizes overall instance-level performance, so large classes have more influence. It is often closer to what you get by pooling all predictions into one big confusion table.

2.5 Linear Regression for Classification

⋆ **Linear regression for binary classification.** We can code the categorical **outcome** as a dummy variable: $\boxed{Y = \begin{cases} 0 & \text{if No} \\ 1 & \text{if Yes} \end{cases}}$ We then perform a standard linear regression of Y on X and classify the observation as **Yes** if the fitted value $\boxed{\hat{Y} > 0.5}$.

Expected value as probability. For a 0/1 coded binary variable, the population expected value is mathematically equivalent to the probability of the event occurring: $\boxed{\mathbb{E}(Y \mid X = x) = \Pr(Y = 1 \mid X = x)}$ *In the strictly binary case, linear regression acts as a reasonable classifier and is mathematically equivalent to linear discriminant analysis. However, it can produce wrong values less than zero or greater than one.*

⋆ **Linear regression for multiclass outcomes.** Suppose we have a qualitative response with **three or more possible values** (e.g., patient symptoms in an ER): $Y = \begin{cases} 1 & \text{if stroke} \\ 2 & \text{if drug overdose} \\ 3 & \text{if epileptic seizure} \end{cases}$ **Linear Regress Fails:** The coding $1, 2, 3$ implies that the mathematical difference (or distance) between a stroke and a drug overdose is identical to the difference between a drug overdose and a seizure. **Because qualitative categories have no inherent order or spacing, linear regression is fundamentally invalid here.**

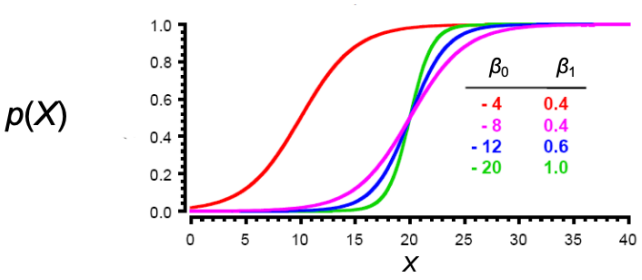
2.6 Logistic Regression

⋆ **Logistic Regression Formulation.** Let $\boxed{p(X) = \Pr(Y = 1 \mid X)}$ represent the probability of an event (e.g., default $Y = 1$) given feature X (e.g., balance). To avoid predicting probabilities outside the valid range, logistic regression models this probability using the **Logistic (sigmoid) function**:

$$\boxed{p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}}$$

The Sigmoid Function. The parameters of the logistic regression model dictate the exact shape and position of the S-shaped curve:

- β_0 controls the location of the curve's midpoint.
- β_1 controls the slope (steepness) of the rise.



Finding the midpoint. Based on the formula, the curve reaches exactly $p(X) = 0.5$ (or $\frac{1}{2}$) when the exponential term equals 1:

$e^{\beta_0+\beta_1 X} = 1 \implies \beta_0 + \beta_1 X = 0$

Solving for X gives the exact midpoint location on the x-axis: $\boxed{X_{mid} = \frac{-\beta_0}{\beta_1}}$

⋆ **Log Odds or Logit Transformation:** Monotone transformation known as the **log odds** or **logit** of $p(X)$. It shows that while the **relationship between X and the probability $p(X)$ is non-linear** (S-shaped), the **relationship between X and the log-odds is strictly linear**.

By manipulating the logistic function, we can find the probability of the opposite event ($Y = 0$):

$$1 - p(X) = \frac{1}{1 + e^{\beta_0+\beta_1 X}} = \Pr(Y = 0 \mid X)$$

Ratio of the probability of the event occurring, ($Y = 1$) to it not occurring ($Y = 0$) gives

Odds: $\boxed{\frac{p(X)}{1 - p(X)} = e^{\beta_0+\beta_1 X} = \frac{\Pr(Y = 1 \mid X)}{\Pr(Y = 0 \mid X)}}$

By taking the natural logarithm of both sides, we arrive at a

Log Odds: $\boxed{\log\left(\frac{p(X)}{1 - p(X)}\right) = \beta_0 + \beta_1 X = \log[O(Y = 1 \mid X = x)]}$ *(linear equation)* →

→ which is **Logit Model.** It is formulated by applying the **logit transformation** to the probability $p(X) = \Pr(Y = 1 \mid X)$. Which makes the model **linear in the log-odds**.

Interpretation of coefficients. Because the **model predicts log-odds rather than raw probabilities or raw outcomes**, the coefficients have specific interpretations:
- β_0: The log-odds of the event occurring when $X = 0$.
- β_1: A one-unit increase in X increases the log-odds by exactly β_1.

Effect on odds. By exponentiating both sides, we can see the multiplicative effect on the odds:

$\boxed{\frac{p(X)}{1 - p(X)} = e^{\beta_0} \cdot e^{\beta_1 X}}$ → one-unit increase in X multiplies the odds by e^{β_1}. *Unlike in OLS, a **one-unit change in X does not correspond to a constant change in the probability** $p(X)$. The actual amount of probability changes depends entirely on where you are on the S-curve.*

⋆ **Connecting Bernoulli to the Logistic function.** Because the target variable is binary $Y_i \in \{0, 1\}$, the outcome of each observation is naturally modeled as a **Bernoulli random variable**. However, the probability of success $p_i = \Pr(Y_i = 1 \mid x_i)$ must strictly lie in the interval $[0, 1]$.

$$\boxed{Y_i \mid X_i = x_i \sim \text{Bernoulli}(p(x_i))}$$

Substitution into the Joint Bernoulli PMF. Substitute $p(x_i)$ into the Bernoulli PMF for each observation. Multiplying these independent **marginals** together yields the **joint probability,** allowing us to express the **likelihood function** entirely in terms of the unknown parameters β_0 and β_1:

$$L(\beta_0, \beta_1) = \prod_{i=1}^{N} \left(\frac{e^{\beta_0+\beta_1 x_i}}{1 + e^{\beta_0+\beta_1 x_i}} \right)^{y_i} \left(1 - \frac{e^{\beta_0+\beta_1 x_i}}{1 + e^{\beta_0+\beta_1 x_i}} \right)^{1-y_i}$$

Likelihood: $$\boxed{\ell(\beta_0, \beta_1) = \prod_{i:\, y_i=1}^{N} p(x_i) \prod_{i:\, y_i=0}^{N} \Big(1 - p(x_i)\Big).}$$

Log-Likelihood: $$\boxed{\log \ell(\beta_0, \beta_1) = \sum_{i=1}^{N} \Big[y_i \log p(x_i) + (1 - y_i) \log\Big(1 - p(x_i)\Big) \Big].}$$

By plugging the sigmoid into the Bernoulli joint PMF, we successfully connect the raw feature data and linear coefficients directly to the probability of observing our specific dataset.

⋆ **Connecting MLE to Logistic Regression.** To find the best fitting parameters β_0 and β_1, we use **Maximum Likelihood Estimation (MLE)**. Because working with products of tiny probabilities is numerically unstable, we maximize the **Log-Likelihood** (ℓ) instead.

$$\max_{\beta_0,\beta_1} \ell(\beta_0, \beta_1) \;\equiv\; \max \log \ell(\beta_0, \beta_1) \;\equiv\; \min\Big(-\log \ell(\beta_0, \beta_1)\Big) \rightarrow$$

$\rightarrow$ Binary Cross Entropy = Negative log-likelihood

$\max_{\beta_0,\beta_1} \ell(\beta_0, \beta_1) = \min_{\beta_0,\beta_1} -\log(\prod_{i=1}^{n} [p(x_i)]^{y_i} [1 - p(x_i)]^{1-y_i})$
$= \min_{\beta_0,\beta_1} -\sum_{i=1}^{n} \Big[y_i \log p(x_i) + (1 - y_i) \log(1 - p(x_i)) \Big]$

$$\therefore \quad \text{Individual loss (for each data point)} = \quad -\Big[y_i \log p(x_i) + (1 - y_i) \log(1 - p(x_i)) \Big]$$

Model not dependent on form of $p(x_i)$

We can rearrange the terms inside the sum algebraically:

$$y_i \log(p(x_i)) - y_i \log(1 - p(x_i)) + \log(1 - p(x_i)) = y_i \log \left(\frac{p(x_i)}{1 - p(x_i)} \right) + \log(1 - p(x_i))$$

Notice the first term contains our **Log-Odds**! We can substitute our linear model $\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_i$ directly into the equation: $$\boxed{\log \ell(\beta_0, \beta_1) = \sum_{i=1}^{N} \Big[y_i(\beta_0 + \beta_1 x_i) - \log(1 + e^{\beta_0+\beta_1 x_i}) \Big]}$$

⋆ **Logistic regression with several variables.** For binary **classification** with **feature vector** $X = (X_1, \ldots, X_p)$ and **label** $Y \in \{0, 1\}$, logistic regression models the **log-odds** as a linear function of the predictors:

$$\boxed{\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p} \qquad \text{where}$$

$$\boxed{p(X) = \Pr(Y = 1 \mid X_1, \ldots, X_p) = \frac{e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}{1 + e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}}$$

The linear combination $\beta_0 + \sum_{j=1}^{p} \beta_j X_j$ is the model score, and the sigmoid maps that score into a valid probability. Large positive scores push the probability toward 1, and large negative scores push it toward 0.

⋆ **Decision boundary of logistic regression is a linear classifier.** Using the standard classification rule $\hat{y} = 1$ if $p(X) \geq \frac{1}{2}$, the **decision boundary** is determined by $p(X) = \frac{1}{2}$.

Substitute the logistic form:

$$\frac{e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}{1 + e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}} = \frac{1}{2} \quad \textit{which implies} \quad e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p} = 1 \quad \Longleftrightarrow \quad \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p = 0.$$

Hence, the decision boundary is $\boxed{\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p = 0,}$ which is **linear** in the predictors.

Even though the predicted probability is nonlinear (sigmoid-shaped), the **separating surface at threshold** 0.5 **is linear.** In two predictors it is a line, in three predictors a plane, and in higher dimensions a hyperplane.

Classification rule: $\boxed{\hat{y} = \begin{cases} 1, & p(X) \geq \frac{1}{2}, \\ 0, & p(X) < \frac{1}{2}, \end{cases} \quad \Longleftrightarrow \quad \hat{y} = \begin{cases} 1, & \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p \geq 0, \\ 0, & \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p < 0. \end{cases}}$

This shows the practical logic clearly: compute a linear score, then classify by which side of the boundary the point falls on. **The probability interpretation comes from the sigmoid, while the boundary geometry comes from the linear score.**

2.6.1 Multinomial Logistic Regression

⋆ **Multinomial Logistic Regression : Logistic regression with more than two classes .** For a multiclass **classification** problem with **label** $Y \in \{1, \dots, K\}$ and **feature vector** $X = (X_1, \dots, X_p)$, we model the **class-conditional probabilities** $\boxed{p_k(x) = \Pr(Y = k \mid X = x), \qquad k = 1, \dots, K}$

A common symmetric parameterization (used in multinomial logistic regression / softmax models) assigns one linear score to each class:

$$\boxed{s_k(x) = \beta_{0k} + \beta_{1k}x_1 + \cdots + \beta_{pk}x_p}$$

and then converts scores to probabilities by

$$\boxed{p_k(x) = \Pr(Y = k \mid X = x) = \frac{e^{s_k(x)}}{\sum_{\ell=1}^{K} e^{s_\ell(x)}} = \frac{e^{\beta_{0k}+\beta_{1k}x_1+\cdots+\beta_{pk}x_p}}{\sum_{\ell=1}^{K} e^{\beta_{0\ell}+\beta_{1\ell}x_1+\cdots+\beta_{p\ell}x_p}}.}$$

This is the multiclass version of logistic regression: **each class gets a linear score, and the softmax turns all scores into probabilities that are positive and sum to 1.** The model is linear in the scores, but nonlinear in the probabilities.

Parameter matrix (one linear function per class). The coefficients can be organized as a matrix where rows are features for column are classes s.t. k, and column k contains the coefficients for class k

$$\beta = \begin{pmatrix} \beta_{01} & \beta_{02} & \cdots & \beta_{0k} \\ \beta_{11} & \beta_{12} & \cdots & \beta_{1k} \\ \beta_{21} & \beta_{22} & \cdots & \beta_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{p1} & \beta_{p2} & \cdots & \beta_{pk} \end{pmatrix},$$

This makes the model structure clear: you are fitting K linear discriminant scores in parallel. **Each class has its own intercept and predictor effects inside its score $s_k(x)$.**

⋆ **Decision rule and decision boundaries.** The predicted class is the class with largest probability, equivalently the largest score: $\boxed{\hat{y}(x) = \arg\max_{k \in \{1,\dots,K\}} p_k(x) = \arg\max_{k \in \{1,\dots,K\}} s_k(x).}$

The **decision boundary** between classes ℓ and k is where their probabilities are equal: $p_\ell(x) = p_k(x)$. Using the softmax form,

$$\frac{e^{s_\ell(x)}}{\sum_{m=1}^{K} e^{s_m(x)}} = \frac{e^{s_k(x)}}{\sum_{m=1}^{K} e^{s_m(x)}} \quad \Longrightarrow \quad e^{s_\ell(x)} = e^{s_k(x)} \quad \Longrightarrow \quad s_\ell(x) = s_k(x).$$

Thus,

$$\boxed{s_\ell(x) - s_k(x) = 0}$$

is a linear equation in x, so pairwise decision boundaries are **linear**.

Relative log-odds are linear. For any two classes k and ℓ,

$$\frac{p_k(x)}{p_\ell(x)} = \frac{e^{s_k(x)}}{e^{s_\ell(x)}} = e^{s_k(x)-s_\ell(x)} \quad \Longrightarrow \quad \boxed{\log\left(\frac{p_k(x)}{p_\ell(x)}\right) = s_k(x) - s_\ell(x)}$$

which is linear in x. This is the multiclass analog of the binary logit idea. The model is easiest to interpret through relative log-odds between classes, and those are linear functions of the predictors.

⋆ **Only $K-1$ independent logits are needed.** Because class probabilities must sum to one, $\sum_{k=1}^{K} p_k(x) = 1$, thus only $K-1$ probabilities are free (the last one is determined by the others). Equivalently, only $K-1$ independent log-odds equations are needed by choosing a **baseline class** K:

$$\boxed{\log\left(\frac{p_k(x)}{p_K(x)}\right) = (\beta_{0k} - \beta_{0K}) + \sum_{j=1}^{p}(\beta_{jk} - \beta_{jK})x_j, \qquad k = 1, \ldots, K-1.}$$

One-hot encoding for multiclass labels. To estimate the coefficients, encode each observed class label with **one-hot indicators** $z_{ik} = \begin{cases} 1, & y_i = k, \\ 0, & y_i \neq k, \end{cases}$ $k = 1, \ldots, K$, so for each observation i, exactly one of $(z_{i1}, \ldots, z_{iK})$ equals 1, and the rest are 0. $\sum_{k=1}^{K} z_{ik} = 1$ for each i.

One-hot encoding turns the multiclass label into a vector representation that matches the vector of predicted class probabilities. This makes the likelihood and loss compact to write and optimize.

⋆ **Multinomial likelihood (from one-hot labels).** For observation i, the model probability of the observed class can be written as $\boxed{\Pr(Y_i = y_i \mid X_i = x_i) = \prod_{k=1}^{K} p_k(x_i)^{z_{ik}}}$

because only the term corresponding to the true class has exponent 1. Over a dataset $\{(x_i, y_i)\}_{i=1}^{N}$, the likelihood is

$$\boxed{\ell(\beta) = \prod_{i=1}^{N}\prod_{k=1}^{K} p_k(x_i)^{z_{ik}}.}$$

This is the direct multiclass extension of the binary likelihood. One-hot exponents let one formula pick out the probability assigned to the correct class for every observation.

Multiclass cross-entropy = negative log-likelihood. Estimating β by maximum likelihood is equivalent to minimizing the **negative log-likelihood**, also called **multiclass cross-entropy loss**:

$$-\log \ell(\beta) = -\sum_{i=1}^{N}\sum_{k=1}^{K} z_{ik} \log p_k(x_i).$$

So the training objective is $\boxed{\min_{\beta} \ \mathrm{NLL}(\beta) = \min_{\beta}\left[-\sum_{i=1}^{N}\sum_{k=1}^{K} z_{ik} \log p_k(x_i)\right].}$

The **individual loss** for observation i is $\boxed{L_i = -\sum_{k=1}^{K} z_{ik} \log p_k(x_i).}$

Since $z_{ik} = 1$ only for the true class, each data point contributes minus the log of the predicted probability assigned to its correct class. The loss is small when the model puts high probability on the correct class, and large when it is confidently wrong.

⋆ **Learning Process.** The model learns class-specific linear scores $s_k(x)$, converts them to probabilities through softmax, and then adjusts coefficients to maximize the probability of observed labels (equivalently minimize cross-entropy).

$$x \longrightarrow (s_1(x), \ldots, s_K(x)) \longrightarrow (p_1(x), \ldots, p_K(x)) \longrightarrow \text{cross-entropy loss.}$$

The key idea is the same as binary logistic regression, just repeated across classes with a shared normalization. The optimization pushes the true class score up relative to competing class scores, which improves both classification and probability fit.

2.6.2 Coefficient Significance (Wald z-Test)

⋆ **Coefficient significance in logistic regression (Wald z-test).** For a logistic model with **dataset** $\{(x_i, y_i)\}_{i=1}^N$, **feature vector** $x_i = (x_{i1}, \ldots, x_{ip})$, and binary **label** $y_i \in \{0, 1\}$, we test whether a coefficient β_j is significantly different from 0 using a ***z*-test** (not a t-test).

$$\boxed{H_0 : \beta_j = 0 \qquad \text{vs.} \qquad H_A : \beta_j \neq 0}$$

At significance level α, reject H_0 if the observed z-statistic falls in either tail:

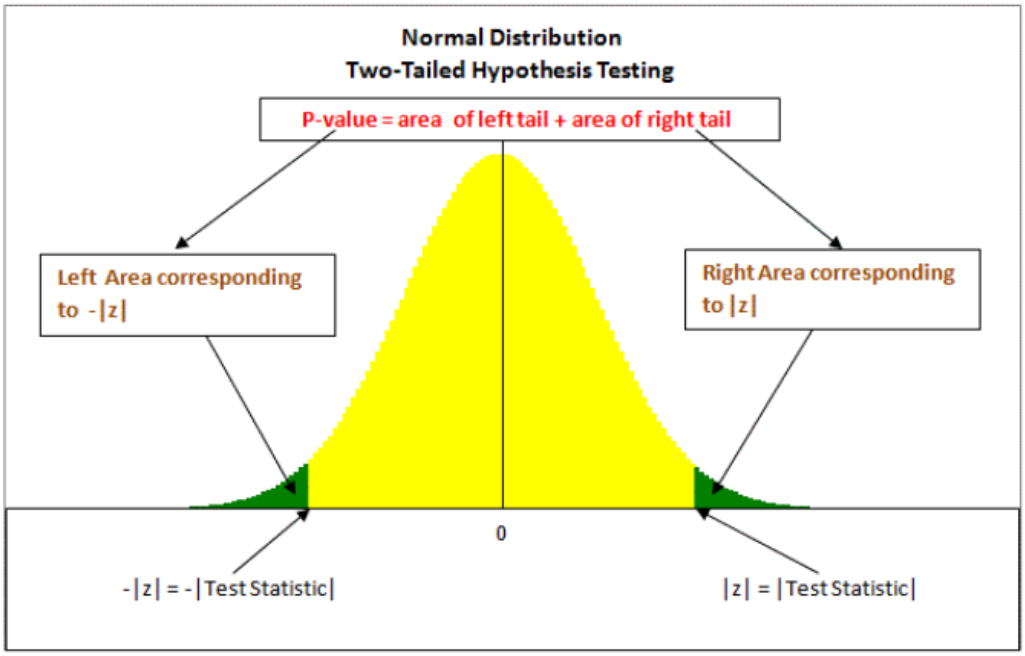
$$\boxed{z > z_{\alpha/2} \qquad \text{or} \qquad z < -z_{\alpha/2}} \qquad \Longleftrightarrow \qquad \boxed{|z| > z_{\alpha/2}}$$

MLE normal approximation (large N). If $\hat{\beta}_j$ is the **maximum likelihood estimate**, then for large sample size, $\boxed{\hat{\beta}_j \approx \mathcal{N}\big(\beta_j,\ \mathrm{SE}(\hat{\beta}_j)^2\big)}$

Hence under $H_0 : \beta_j = \beta_j^{(0)}$ $\boxed{z = \dfrac{\hat{\beta}_j - \beta_j}{\mathrm{SE}(\hat{\beta}_j)} \approx \mathcal{N}(0, 1).}$ (base case for us $H_0 : \beta_j = 0$)

This statistic measures how many standard errors the estimate is away from the null value.

Two-sided p-value. The **p-value** is the probability (under H_0) of observing a statistic at least as extreme as the one obtained: $\boxed{p\text{-value} = \Pr\big(|Z| \geq |z_{\text{obs}}|\big) = 2\ \Pr\big(Z \geq |z_{\text{obs}}|\big) \qquad Z \sim \mathcal{N}(0, 1)}$



2.7 Discriminant Analysis (Bayesian)

⋆ **Bayesian Discriminant analysis: modeling idea.** In **discriminant analysis**, we model the distribution of **features** X separately within each class, then use **Bayes' theorem** to obtain class probabilities $\Pr(Y = k \mid X = x)$.

Model: $\boxed{f_k(x) = p(X = x \mid Y = k) \text{ and } \pi_k = P(Y = k), \quad \text{then compute } P(Y = k \mid X = x)}$

This is a generative classification approach: model how each class generates data, then invert to classify. It differs from logistic regression, which models $P(Y \mid X)$ directly.

Key ingredients and notation. For $k = 1, \ldots, K$:

- $\boxed{\pi_k = P(Y = k)}$: **prior probability** of class k, encode how common classes are before seeing features.
- $\boxed{f_k(x) = p(X = x \mid Y = k)}$: class-$k$ **density** (or likelihood in x), measure how compatible the observed feature vector is with each class.
- $\boxed{P(Y = k \mid X = x)}$: **posterior probability** of class k after observing x, probability output of the classifier for each class.

Generality of the approach. Discriminant analysis is a general Bayes-classification framework and does not require a specific class-conditional distribution in principle. Standard pick is **normal (Gaussian) class densities**, used to model the **class-conditional feature distribution** for each class:

$$\boxed{X \mid Y = k \sim \mathcal{N}_p(\mu_k, \Sigma_k) \qquad \Longrightarrow \qquad f_k(x) = p(X = x \mid Y = k)}$$

The Gaussian is **not** the posterior class probability. It is the model for how features X are distributed inside class k.

The **classifier first models feature behavior within each class, then uses Bayes'.** So the Gaussian appears in $f_k(x)$, not directly in $P(Y = k \mid X = x)$.

⋆ **Bayes theorem for classification.** For class k and observed features $X = x$,

$$\boxed{P(Y = k \mid X = x) = \frac{\pi_k f_k(x)}{\sum_{\ell=1}^{K} \pi_\ell f_\ell(x)}}$$

Posterior probability of class k is proportional to prior $\times$ class density evaluated at x. The denominator just normalizes across all classes so the posteriors sum to 1.

Classification compares Class Scores: Once we specify a **Gaussian** $f_k(x) = p(X = x \mid Y = k)$ **for each class** and **priors** $\pi_k = P(Y = k)$, from posterior class probabilities and proptiality property we get **class scores:**

$$\boxed{\hat{y}(x) = \arg\max_k \pi_k f_k(x).}$$

At a new point x, each class Gaussian tells us how compatible x is with that class. The $\pi_k f_k(x)$ is the **unnormalized posterior score** for class k. **Multiplying by π_k adjusts the class density by how common the class is overall.** A class can fit x well (high density) but still lose if it is much less likely a priori.

The **density** $f_k(x)$ says "how typical is x inside class k," while the **prior** π_k says "how common is class k." The product balances both pieces when deciding the class.

Case: priors are equal ($\pi_1 = \pi_2$), this reduces to comparing densities only: $\hat{y}(x) = \begin{cases} 1, & f_1(x) > f_2(x), \\ 2, & f_2(x) > f_1(x). \end{cases}$

Then **decision boundary is where the two class scores are equal:** $\boxed{f_1(x) = f_2(x)}$

The density says "how typical is x inside class k," while the prior says "how common is class k." The product balances both pieces when deciding the class.

Core classification logic For a new point x: $x \longrightarrow f_1(x), f_2(x) \longrightarrow \pi_1 f_1(x), \pi_2 f_2(x) \longrightarrow$ choose larger score. *The density curves tell you where each class tends to generate data. The priors tilt the decision toward more common classes, and Bayes classification combines both in one comparison.*

⋆ **Build for each class (the actual model objects).** For each class k, we build:

$$\boxed{\pi_k = P(Y = k), \qquad \mu_k = \mathbb{E}[X \mid Y = k], \qquad \Sigma_k = \mathrm{Cov}(X \mid Y = k).}$$

These define the Gaussian class model:

$$\boxed{f_k(x) = \frac{1}{(2\pi)^{p/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_k)^\top \Sigma_k^{-1}(x - \mu_k)\right).}$$

From Data: Given training data $\{(x_i, y_i)\}_{i=1}^N$, let $n_k = \sum_{i=1}^N \mathbf{1}\{y_i = k\}$. Then estimate:

$\boxed{\hat{\pi}_k = \frac{n_k}{N}}$ $\boxed{\hat{\mu}_k = \frac{1}{n_k} \sum_{i:y_i=k} x_i}$ $\boxed{\hat{\Sigma}_k = \frac{1}{n_k - 1} \sum_{i:y_i=k} (x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^\top}$ (QDA-style).

2.7.1 LDA: Linear Discriminant Analysis

⋆ **LDA with $p = 1$:** For one **feature** X and K classes, assume the class-conditional density is Gaussian in each class:

$$\boxed{X \mid Y = k \sim \mathcal{N}(\mu_k, \sigma_k^2)} \qquad \Longrightarrow \qquad \boxed{f_k(x) = \frac{1}{\sqrt{2\pi}\,\sigma_k} \exp\left(-\frac{1}{2}\left(\frac{x - \mu_k}{\sigma_k}\right)^2\right).}$$

For **LDA**, each class has its own center μ_k, but all classes share the same spread aka **equal (constant) variance** across all classes: $\boxed{\sigma_1^2 = \sigma_2^2 = \cdots = \sigma_K^2 = \sigma^2.}$

Posterior class probability (Bayes rule). With prior $\pi_k = P(Y = k)$, the posterior probability is

$$\boxed{p_k(x) = P(Y = k \mid X = x) = \frac{\pi_k f_k(x)}{\sum_{\ell=1}^K \pi_\ell f_\ell(x)} = \frac{\pi_k \dfrac{1}{\sqrt{2\pi}\sigma} \exp\left(-\dfrac{1}{2}\left(\dfrac{x - \mu_k}{\sigma}\right)^2\right)}{\sum_{\ell=1}^K \pi_\ell \dfrac{1}{\sqrt{2\pi}\sigma} \exp\left(-\dfrac{1}{2}\left(\dfrac{x - \mu_\ell}{\sigma}\right)^2\right)}}$$

With winning class

$$\boxed{\hat{y}(x) = \arg\max_{k \in \{1,\ldots,K\}} p_k(x) [= \pi_k f_k(x)] = \arg\max_{k \in \{1,\ldots,K\}} \pi_k \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\left(\frac{x - \mu_k}{\sigma}\right)^2\right)}$$

Apply Log to LDA score from $\pi_k f_k(x)$ ($p = 1$). Take logs and expand:

$$\hat{y}(x) = \arg\max_k \log(\pi_k f_k(x)) = \arg\max_k \left[\log \pi_k + \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{x^2}{2\sigma^2} + \frac{x\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2}\right].$$

Drop terms independent of k, giving the **discriminant score**

$$\boxed{\delta_k(x) = \log(\pi_k) + \frac{x\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2}} \qquad \Rightarrow \qquad \boxed{\hat{y}(x) = \arg\max_{k\in\{1,\dots,K\}} \delta_k(x).}$$

$\log(\pi_k)$ is the prior effect, $\frac{x\mu_k}{\sigma^2}$ is the fit/alignment term, and $-\frac{\mu_k^2}{2\sigma^2}$ is a class offset.

$\delta_k(x)$ is linear in x.

$$\delta_k(x) = \underbrace{\frac{\mu_k}{\sigma^2}}_{\text{slope in class } k} x + \underbrace{\left(\log(\pi_k) - \frac{\mu_k^2}{2\sigma^2}\right)}_{\text{intercept/offset for class } k}.$$

So $\delta_k(x)$ is a **linear function of x**.

The posterior probabilities are nonlinear, but the class comparison is linear in x. This is exactly why it is called **linear** discriminant analysis.

Case for Decision boundary for $K = 2$, balanced priors. If $K = 2$ and $\pi_1 = \pi_2 = 0.5$, the decision boundary is where the two discriminant scores are equal:

$$\delta_1(x) = \delta_2(x).$$

Using $\delta_k(x) = \log(\pi_k) + \frac{x\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2}$ and $\log(\pi_1) = \log(\pi_2)$, we get $\frac{x\mu_1}{\sigma^2} - \frac{\mu_1^2}{2\sigma^2} = \frac{x\mu_2}{\sigma^2} - \frac{\mu_2^2}{2\sigma^2}$.
Multiply by $2\sigma^2$: we get $2x\mu_1 - \mu_1^2 = 2x\mu_2 - \mu_2^2$ $\quad 2x(\mu_1 - \mu_2) = \mu_1^2 - \mu_2^2 = (\mu_1 - \mu_2)(\mu_1 + \mu_2)$.

Hence (for $\mu_1 \neq \mu_2$) $\boxed{x = \frac{\mu_1 + \mu_2}{2}.}$

In the balanced two-class case with equal variance, the boundary is the midpoint of the two class means. This matches the picture intuition: classify by which class center the point is closer to (after accounting for common spread).

⋆ **Estimating LDA parameters from Data ($p = 1$).** Given training data $\{(x_i, y_i)\}_{i=1}^n$, let $n_k = \sum_{i=1}^n \mathbf{1}\{y_i = k\}$: number of observations in class k and n = total sample size

LDA estimates: $\boxed{\hat{\pi}_k = \frac{n_k}{n}}$ $\boxed{\hat{\mu}_k = \frac{1}{n_k}\sum_{i:y_i=k} x_i}$ $\boxed{\hat{\sigma}^2 = \frac{1}{n-K}\sum_{k=1}^K \sum_{i:y_i=k} (x_i - \hat{\mu}_k)^2.}$

$\hat{\sigma}^2$ *(pooled) variance estimate*

LDA estimates class **priors** from class frequencies, class **means** from within-class averages, and one **shared variance** from the pooled within-class spread.

Class-specific within-class variance $\boxed{\hat{\sigma}_k^2 = \frac{1}{n_k - 1}\sum_{i:y_i=k} (x_i - \hat{\mu}_k)^2.}$

Pooled variance can be written as a weighted average of class variances: $\boxed{\hat{\sigma}^2 = \sum_{k=1}^K \frac{n_k - 1}{n - K}\hat{\sigma}_k^2 \approx \sum_{k=1}^K \hat{\pi}_k \hat{\sigma}_k^2.}$

For large samples is common approximation.
The pooled variance combines within-class variability across all classes under the equal-variance assumption. It is the shared spread parameter that makes the discriminant score linear.

⋆ **LDA when $p > 1$:** For **feature vector** $x \in \mathbb{R}^p$, LDA assumes a class-specific Gaussian mean μ_k and a **shared covariance matrix** $\Sigma = Cov(X_i, X_j)$: $\boxed{X \mid Y = k \sim \mathcal{N}_p(\mu_k, \Sigma), \qquad k = 1, \ldots, K.}$

The class-k density is $\boxed{f_k(x) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^\top \Sigma^{-1}(x-\mu_k)\right).}$

Using Bayes classification and dropping terms independent of k, the LDA **discriminant score** is

$$\boxed{\delta_k(x) = x^\top \Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^\top \Sigma^{-1}\mu_k + \log(\pi_k)} \quad \Rightarrow \quad \boxed{\hat{y}(x) = \arg\max_k \delta_k(x)}$$

Even though the Gaussian density looks complicated, the shared covariance assumption makes the $x^\top \Sigma^{-1} x$ term cancel across classes.

Discriminant function is atill Linear in x: $\delta_k(x) = \underbrace{(\Sigma^{-1}\mu_k)^\top}_{c_k^\top} x + \underbrace{\left(-\frac{1}{2}\mu_k^\top \Sigma^{-1}\mu_k + \log \pi_k\right)}_{b_k},$

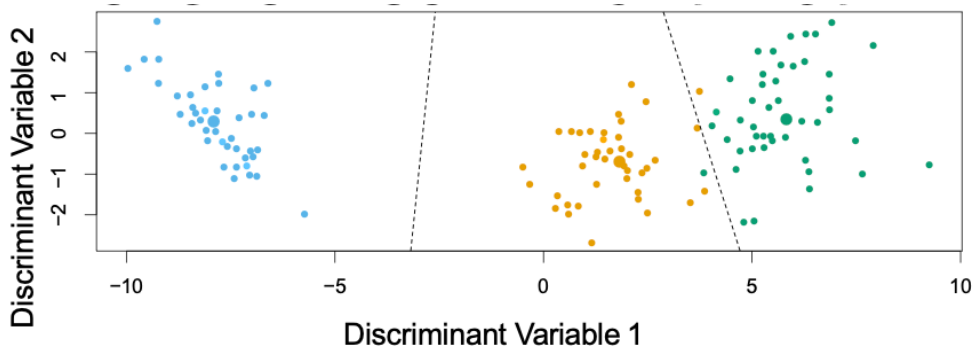
so $\boxed{\delta_k(x) = b_k + c_k^\top x}$ is a **linear function of the predictors $x_1, \ldots, x_p$.**

Relative to a baseline class K (why $K-1$ comparisons). Choosing a **baseline class** K means we use class K as a reference and compare every other class to it through score differences (or relative log-odds), e.g.

$$\boxed{\delta_k(x) - \delta_K(x), \qquad k = 1, \ldots, K-1.}$$

This does **not** change the final classification rule, because LDA only depends on which class score is largest; only relative differences matter. It is a convenient way to remove redundancy, since K class scores contain only $K-1$ independent comparisons.

Each difference is linear in x, so classification depends only on the projection of x onto the subspace spanned by these $K-1$ discriminant directions. Hence LDA can be represented exactly in at most $K-1$ dimensions. **LDA classification rule depends on at most K-1 discriminant coordinates.**



Outputs of LDA K = 3 and p = 2 are seen through
(i) **class assignment regions** separated by the dashed decision boundaries and
(ii) **discriminant coordinates** (Discriminant Variable 1 and 2), which are linear combinations of the original features used for classification.
Each point is plotted after projection into the discriminant subspace, and its predicted class is determined by which side of the boundary it falls on (equivalently, which class has the largest discriminant score $\delta_k(x)$). Intuitively, the plot shows how LDA transforms high-dimensional data into the directions that best separate class centroids while preserving the classification rule in this lower-dimensional view.

⋆ **From LDA discriminant scores to class probabilities.** Once we estimate the **discriminant scores** $\hat{\delta}_k(x)$, we can convert them into estimated class probabilities using a **softmax** transformation which converts the estimated $\hat{\delta}_1(x), \ldots, \hat{\delta}_K(x)$ into estimated class probabilities by exponentiating and normalizing:

$$\boxed{\hat{p}_k(x) = \widehat{\Pr}(Y = k \mid X = x) = \frac{e^{\hat{\delta}_k(x)}}{\sum_{\ell=1}^{K} e^{\hat{\delta}_\ell(x)}}}$$

The discriminant scores are class-specific linear scores, and exponentiating + normalizing turns them into probabilities that are positive and sum to 1. This preserves the same class ranking as the scores.

Maximizing $\hat{\delta}_k(x)$ is the same as maximizing $\hat{p}_k(x)$. Since the exponential function is increasing and the denominator is common across classes, $\boxed{\arg\max_k \hat{\delta}_k(x) = \arg\max_k e^{\hat{\delta}_k(x)} = \arg\max_k \hat{p}_k(x).}$

Hence, the **winning class is the one with the largest discriminant score, equivalently the largest estimated posterior probability.**

2.7.2 QDA: Quadratic Discriminant Analysis

⋆ **Quadratic Discriminant Analysis (QDA): key idea.** QDA uses the same Bayes-classification framework as LDA, $\boxed{P(Y = k \mid X = x) = \frac{\pi_k f_k(x)}{\sum_{\ell=1}^{K} \pi_\ell f_\ell(x)},}$

allows each class to have its own Gaussian covariance matrix:

$$\boxed{X \mid Y = k \sim \mathcal{N}_p(\mu_k, \Sigma_k), \qquad \Sigma_k \text{ can differ across classes.}}$$

QDA is the "different class shape" version of Gaussian discriminant analysis. The class-specific covariance matrices let different classes have different spread/orientation.

⋆ **QDA discriminant score (multivariate form).** Dropping class-independent constants, a common QDA discriminant score is

$$\boxed{\delta_k(x) = -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x - \mu_k)^\top \Sigma_k^{-1}(x - \mu_k) + \log(\pi_k)}$$

with classification $\boxed{\hat{y}(x) = \arg\max_k \delta_k(x).}$

The score combines prior class frequency, a covariance penalty term, and a Mahalanobis-distance fit term. The class with the largest score is the Bayes classifier under the Gaussian QDA model.

QDA in one dimension ($p = 1$): why the score is quadratic. If $X \mid Y = k \sim \mathcal{N}(\mu_k, \sigma_k^2)$, then (up to class-independent constants) $\delta_k(x) = \log(\pi_k) - \log(\sigma_k) - \frac{1}{2}\left(\frac{x-\mu_k}{\sigma_k}\right)^2$. Expanding gives

$$\boxed{\delta_k(x) = \log(\pi_k) - \log(\sigma_k) - \frac{x^2}{2\sigma_k^2} + \frac{x\mu_k}{\sigma_k^2} - \frac{\mu_k^2}{2\sigma_k^2},}$$

which is **quadratic in x** because the coefficient of x^2 depends on k. Henec in QDA, different σ_k values make the x^2 term class-specific, so it stays and creates curved boundaries.

⋆ **Why high-dimensional discriminant analysis becomes hard (idea of $O(p)$ vs. $O(p^2)$).** With p **features**, each class model needs a mean vector $\mu_k \in \mathbb{R}^p$ and (for Gaussian DA) covariance information. The number of parameters grows quickly with p, which increases data requirements (**curse of dimensionality**).

2.8 Naïve Bayes Classifier

Bayesian Classification Rule. For classes $k \in \{1, \ldots, K\}$, Bayes' rule gives

$$\Pr(Y = k \mid X = x) = \frac{\Pr(X = x \mid Y = k)\Pr(Y = k)}{\Pr(X = x)} = \frac{\pi_k f_k(x)}{\sum_{\ell=1}^{K} \pi_\ell f_\ell(x)},$$

Optimal Bayesian classifier requires the full joint class-conditional density

$f_k(x) = \Pr(X_1 = x_1, \ldots, X_p = x_p \mid Y = k)$, which is infeasible to estimate reliably when p is large.

Key difficulty: estimating a high-dimensional joint density suffers from the curse of dimensionality.

Conditional independence assumption (Naïve Bayes). Naïve Bayes assumes that features are conditionally independent given the class.

Conditional independence $\boxed{X_1 \perp X_2 \perp \cdots \perp X_p \,\big|\, Y = k.}$

Means that features are **independent once the class $Y = k$ is known**.

Formally, for any features X_i, X_j, $\Pr(X_i \mid X_j, Y = k) = \Pr(X_i \mid Y = k) \quad \Rightarrow$

$\Pr(X_1, \ldots, X_p \mid Y = k) = \prod_{j=1}^{p} \Pr(X_j \mid Y = k)$.

Under this assumption, the **joint density** factorizes as

$$\boxed{f_k(x) = \Pr(X = x \mid Y = k) \;\approx\; \prod_{j=1}^{p} \Pr(X_j = x_j \mid Y = k) = \prod_{j=1}^{p} f_{kj}(x_j).}$$

Each marginal density $f_{kj}(x_j)$ can be estimated easily from data.

Classification rule (Naïve Bayes): $\boxed{\hat{k} = \arg\max_k \; \pi_k \prod_{j=1}^{p} f_{kj}(x_j)}$

Take Log then **Discriminant score** $\boxed{\delta_k(x) = \log \pi_k + \sum_{j=1}^{p} \log f_{kj}(x_j)}$

Gaussian Naïve Bayes. Each feature is modeled as Gaussian within each class,

$X_j \mid Y = k \sim \mathcal{N}(\mu_{kj}, \sigma_{kj}^2)$, then $\boxed{\delta_k(x) = -\frac{1}{2} \sum_{j=1}^{p} \frac{(x_j - \mu_{kj})^2}{\sigma_{kj}^2} - \sum_{j=1}^{p} \log \sigma_{kj} + \log \pi_k + C.}$

⋆ **Estimating $f_k(x_i) = \Pr(X_i = x_i \mid Y = k)$ from data** In Naïve Bayes, we do *not* estimate the full joint density $f_k(x_1, \ldots, x_p)$, but instead estimate each **one-dimensional conditional density**

$$\boxed{f_k(x_i) = \Pr(X_i = x_i \mid Y = k)}$$

separately and then combine them.

Discrete features: If X_i is categorical, $\boxed{\widehat{\Pr}(X_i = x_i \mid Y = k) = \dfrac{|\{\text{observations in class } k \text{ with } X_i = x_i\}|}{N_k}}$

where N_k is the number of training samples in class k. This is empirical frequency estimation (histograms).

Naïve Bayes zero-frequency problem & smoothing. If any estimated conditional probability is zero, the whole class score becomes zero: $\hat{p}_k(x) \propto \hat{\pi}_k \prod_{j=1}^{p} \hat{p}(x_j \mid C_k)$, $\hat{p}(x_j \mid C_k) = 0 \Rightarrow \hat{p}_k(x) = 0$.

Alternatives for Discrete features: Smoothing adds pseudo-counts so unseen categories do not produce zero probabilities.

Laplace smoothing. Let c be the number of possible categories of X_i $\boxed{\widehat{\Pr}(X_i = x_i \mid Y = k) = \dfrac{N_{ik} + 1}{N_k + c}}$

where N_{ik} is the number of observations in class k with $X_i = x_i$.

m-estimate (general smoothing). $\boxed{\widehat{\Pr}(X_i = x_i \mid Y = k) = \dfrac{N_{ik} + mp}{N_k + m}}$

where p is a prior guess for the probability (often $p = \frac{1}{c}$) and $m > 0$ controls how strongly we shrink toward that prior.

Continuous features. Three common strategies:

- **Discretization:** bin the range of X_i and treat bins as categories.
- **Binary split:** replace X_i by an indicator $\{X_i < v \quad or \quad X_i > v\}$ for some threshold v.
- **Parametric density estimation (most common):** assume $X_i \mid Y = k \sim \mathcal{N}(\mu_{ik}, \sigma_{ik}^2)$, estimate μ_{ik}, σ_{ik}^2 from data,
  N = total number of training observations, and $N_k = |\{n : Y_n = k\}|$ the number of observations belonging to class k.
  For feature X_i in class k, the class-conditional density $f_k(x_i) = \Pr(X_i = x_i \mid Y = k)$ is estimated from the subset $\{X_{ni} : Y_n = k\}$. With
  Sample mean. $\hat{\mu}_{ik} = \frac{1}{N_k} \sum_{n:Y_n=k} X_{ni}$.
  Sample variance. $\hat{\sigma}_{ik}^2 = \frac{1}{N_k} \sum_{n:Y_n=k} \big(X_{ni} - \hat{\mu}_{ik}\big)^2$.
  These estimates are computed separately for each feature i and each class k, and used in

  Gaussian Naïve Bayes joint density: $\boxed{f_k(x_i) = \dfrac{1}{\sqrt{2\pi\sigma_{ik}^2}} \exp\left(-\dfrac{(x_i - \mu_{ik})^2}{2\sigma_{ik}^2}\right)}$

  Class priors: $\boxed{\hat{\pi}_k = \Pr(Y = k) = \dfrac{N_k}{N}}$

  Putting it together (Naïve Bayes rule). Using conditional independence,
  $\boxed{\Pr(Y = k \mid X = x) \propto \pi_k \prod_{i=1}^{p} f_k(x_i),}$ and classification $\boxed{\hat{y} = \arg\max_k \left\{ \log \pi_k + \sum_{i=1}^{p} \log f_k(x_i) \right\}}$

  Naïve Bayes replaces one impossible high-dimensional density estimation problem with many easy one-dimensional ones, trading bias (independence assumption) for massive variance reduction.

Chapter 3

Linear Model Selection and Regularization

3.1 Three Classes of Methods

Alternatives to ordinary least squares for fitting the linear model $Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p + \varepsilon$.

- **Subset Selection.** Identify a subset of the p predictors related to the response and fit via least squares on the reduced set.
  Fewer variables means a simpler, more interpretable model. The challenge is choosing which predictors to keep.
- **Shrinkage (= Regularization).** Fit a model with all p predictors, but **shrink** the estimated coefficients towards zero relative to the OLS estimates. Reduces **variance** and can perform **variable selection**.
  Rather than discarding predictors outright, shrinkage dampens their influence. This trades a small increase in bias for a potentially large decrease in variance.
- **Dimension Reduction.** Project the p predictors into an M-dimensional subspace ($M < p$) by computing M different **linear combinations** (projections) of the variables:

  $$Z_m = f_m(X_1, \ldots, X_p), \qquad m = 1, \ldots, M,$$

  then fit a linear regression on $Z_1, \ldots, Z_M$ by least squares.
  Instead of selecting or shrinking individual predictors, dimension reduction constructs a smaller set of derived features that capture most of the information in the original predictors. If chosen well, $M \ll p$ avoids overfitting.

3.2 Subset Selection Methods

⋆ **Best Subset Selection.** For each model size k, fit *every* possible model with exactly k predictors and keep the best one.

1. Let M_0 denote the **null model** (no predictors). It predicts the sample mean for every observation: $\hat{y} = \hat{\beta}_0 = \bar{y}$.
2. For $k = 1, 2, \ldots, p$:
   - Fit all $\binom{p}{k}$ models containing exactly k predictors.
   - Pick the best among these $\binom{p}{k}$ models and call it M_k. Here **best** = smallest RSS (equivalently largest R^2).
3. Select a single best model from $M_0, \ldots, M_p$ using **cross-validated prediction error**, C_p (AIC), BIC, or adjusted R^2.

Step 2 finds the best model within *each size by training-set fit (RSS/R^2). Step 3 compares* across *sizes using a criterion that estimates test error, because RSS alone always favors larger models.*

Computational cost. Best subset requires fitting $\sum_{k=0}^{p} \binom{p}{k} = 2^p$ models, which grows exponentially and becomes infeasible for large p.

Extensions to other models. The same idea applies beyond least squares (e.g. logistic regression). For a broader class of models the **deviance** (negative two times the maximized log-likelihood) replaces RSS as the within-size comparison criterion.

⋆ **Stepwise Selection.** Best subset selection suffers from two problems when p is large:

- **Computational:** 2^p models is infeasible.
- **Statistical:** a larger search space increases the chance of **overfitting**: finding models that look good on training data but have high variance and poor predictive power.

Stepwise methods explore a far more restricted set of models, searching only $1 + p(p+1)/2$ models instead of 2^p. They are *not* guaranteed to find the globally best subset.

⋆ **Forward Stepwise Selection.** Start from the null model and greedily add one predictor at a time.

1. Let M_0 denote the **null model** (no predictors).
2. For $k = 0, 1, \ldots, p-1$:
   - Consider all $p - k$ models that augment M_k with one additional predictor.
   - Choose the **best** among these $p - k$ models (smallest RSS / highest R^2) and call it M_{k+1}.
3. Select a single best model from $M_0, \ldots, M_p$ using cross-validated prediction error, C_p (AIC), BIC, or adjusted R^2.

At each step the variable giving the greatest additional *improvement to the fit is added. The method is greedy: once a variable enters it is never removed, so earlier choices constrain later ones. This is why forward selection is not guaranteed to find the best size-k model out of all 2^p possibilities.*

Complexity. Forward stepwise fits $p + (p-1) + \cdots + 1 = \mathcal{O}(p^2)$ models: polynomial, not exponential.

Key advantage. Forward stepwise can be used even when $n < p$, since at each step the model being fit has at most $k + 1 \leq n$ predictors.

⋆ **Backward Stepwise Selection.** Start from the full model and greedily remove one predictor at a time.

1. Let M_p denote the **full model** containing all p predictors.
2. For $k = p, p-1, \ldots, 1$:
   - Consider all k models that remove one predictor from M_k, yielding $k-1$ predictors.
   - Choose the **best** among these k models (smallest RSS / highest R^2) and call it M_{k-1}.
3. Select a single best model from $M_0, \ldots, M_p$ using cross-validated prediction error, C_p (AIC), BIC, or adjusted R^2.

Backward selection mirrors forward selection but works top-down: at each step it removes the predictor that harms the model the least. Like forward selection, it searches $1 + p(p+1)/2$ models and is not guaranteed to find the globally best subset.

Requirement: $n > p$. Backward selection requires $n > p$ so that the full model M_p can be fit. When $p \geq n$, least squares on all p predictors is not defined, making backward selection inapplicable.

*In contrast, forward stepwise is the **only viable subset method when p is very large** (including $p > n$), because it never needs to fit more than $k+1$ predictors at once.*

	Best Subset	Forward	Backward
Models searched	2^p	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2)$
Guarantees global best?	Yes	No	No
Works when $n < p$?	No	Yes	No

*All three methods produce a sequence of candidate models $M_0, \ldots, M_p$. Choosing among them requires a criterion that estimates **test error** rather than training error, covered next (Cp, BIC, adjusted R^2, cross-validation).*

3.3 Choosing the Optimal Model

⋆ **Choosing the Optimal Model.** The full model (all p predictors) always has the smallest RSS and largest R^2, since these measure **training error**. But we want low **test error**, and training error is a poor estimate of test error. Therefore RSS and R^2 alone are not suitable for selecting among models of different sizes.

Adding any predictor (even a useless one) can only decrease RSS on the training data. A model selection criterion must penalize complexity to avoid choosing an overfit model.

⋆ **Estimating test error: two approaches.**

- **Indirect (adjustment) approach.** Adjust the training error to account for the **bias due to overfitting**. This yields criteria such as C_p, AIC, BIC, and adjusted R^2. Each adds a **penalty** that grows with model size: if the model is too large, it is penalized.
- **Direct approach.** Estimate test error directly via a **validation set** or **cross-validation**. No adjustment formula needed.

⋆ **Mallow's C_p:** is trying to estimate what your test MSE would be if you used this model on new data.

$$\boxed{C_p = \frac{1}{n}\left(\text{RSS} + 2\,d\,\hat{\sigma}^2\right)} = \underbrace{\frac{\text{RSS}}{n}}_{\text{training error}} + \underbrace{\frac{2\,d\,\hat{\sigma}^2}{n}}_{\text{optimism correction}}$$

where d = total number of parameters used, and $\hat{\sigma}^2$ is an estimate of $\text{Var}(\varepsilon)$.

Select the model with the smallest C_p.

The $2d\hat{\sigma}^2$ term is the penalty for model complexity. A model with more parameters gets a larger penalty, offsetting the fact that its RSS is artificially low on training data.

Estimating the error variance. In simple regression ($p = 1$):

$$\hat{\sigma} = \text{RSE} = \sqrt{\frac{1}{n-2}\,\text{RSS}} = \sqrt{\frac{1}{n-2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}.$$

In general with d parameters: $\boxed{\hat{\sigma} = \sqrt{\frac{1}{n-d-1}\,\text{RSS}}}$

⋆ **AIC (Akaike Information Criterion).** Defined for models fit by **maximum likelihood**:

$$\boxed{\text{AIC} = \underbrace{-2\log L}_{\text{lack of fit}} + \underbrace{2d}_{\text{complexity penalty}}}$$

where L is the maximized value of the likelihood and d is the number of parameters.
Select the model with the smallest AIC.

For the linear model with Gaussian errors, maximum likelihood and least squares coincide, so C_p and AIC are equivalent (proportional). AIC is more general because it applies to any likelihood-based model (e.g. logistic regression, GLMs).

Likelihood L. L is the probability of the observed data given the fitted model:

$$L = \prod_{i=1}^{n} P(y_i \mid x_i,\, \hat{\beta}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \hat{y}_i)^2}{2\sigma^2}\right)$$

Large $L \Rightarrow$ good fit, small $L \Rightarrow$ poor fit. Since $-2\log L$ flips this, minimizing AIC favors models that fit well but are not overly complex.

For the linear model with Gaussian errors, the log-likelihood reduces to a function of RSS, which is why AIC and C_p are proportional in that setting.

⋆ **BIC (Bayesian Information Criterion).** BIC works like C_p (it starts with training error and adds a penalty for each parameter) but it charges $\log(n)$ per parameter instead of 2, so the penalty grows with sample size. This makes BIC increasingly reluctant to add variables as you get more data, favoring the simplest model that still explains the data well.

$$\boxed{\text{BIC} = \frac{1}{n}\left(\text{RSS} + \log(n)\,d\,\hat{\sigma}^2\right)}$$

Select the model with the smallest BIC.

*BIC replaces the $2d\hat{\sigma}^2$ penalty in C_p with $\log(n)\, d\, \hat{\sigma}^2$. Since $\log n > 2$ for any $n > 7$, BIC penalizes large models more heavily than C_p/AIC and therefore tends to select **smaller models**. BIC is considered very conservative.*

Penalty comparison: C_p vs. BIC. $\underbrace{2d\hat{\sigma}^2}_{C_p \text{ penalty}}$ vs. $\underbrace{\log(n)\, d\, \hat{\sigma}^2}_{\text{BIC penalty}}$ Both scale linearly in d, but BIC's factor of $\log(n)$ replaces the constant 2, giving a stronger penalty whenever $n > 7$.

⋆ **Adjusted R^2:** $\boxed{\text{Adjusted } R^2 = 1 - \dfrac{\text{RSS}/(n-d-1)}{\text{TSS}/(n-1)}}$

where $\text{TSS} = \sum_{i=1}^{n}(y_i - \bar{y})^2$ is the total sum of squares.

Unlike C_p, AIC, and BIC, a **large** adjusted R^2 indicates a model with low test error.

Recall the ordinary R^2: $R^2 = 1 - \dfrac{\text{RSS}}{\text{TSS}}$.

*Maximizing adjusted R^2 is equivalent to minimizing $\text{RSS}/(n-d-1)$. While RSS always decreases as d increases, dividing by $(n-d-1)$ means the denominator shrinks as variables are added. If a new variable does not reduce RSS enough to compensate for the lost degree of freedom, adjusted R^2 decreases. Thus, unlike plain R^2, the adjusted version **penalizes unnecessary variables**.*

Criterion	Formula (linear model)	Select by
C_p	$\frac{1}{n}(\text{RSS} + 2d\hat{\sigma}^2)$	minimize
AIC	$-2\log L + 2d$	minimize
BIC	$\frac{1}{n}(\text{RSS} + \log(n)\, d\, \hat{\sigma}^2)$	minimize
Adj. R^2	$1 - \frac{\text{RSS}/(n-d-1)}{\text{TSS}/(n-1)}$	maximize

⋆ **Indirect vs. direct estimation of test error.** Both approaches ultimately measure the same thing: average squared prediction error. They differ in *where* that error is computed.

- **Indirect (Cp, AIC, BIC, Adj. R^2):** $\widehat{\text{Test Error}} = \underbrace{\dfrac{\text{RSS}}{n}}_{\substack{\text{squared errors on} \\ \textbf{training data} \\ \text{(biased low)}}} + \underbrace{\text{penalty}(d, \hat{\sigma}^2)}_{\substack{\text{formula that corrects} \\ \text{the optimistic bias}}}$

  Training error underestimates test error because the model was fit to the same data. The penalty formula adds back the estimated gap. Requires $\hat{\sigma}^2$ and d; if either is wrong or hard to define, the correction is unreliable.

- **Direct (Cross-Validation):** $\widehat{\text{Test Error}} = \underbrace{\dfrac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i^{(-i)})^2}_{\substack{\text{squared errors on} \\ \textbf{held-out data} \\ \text{(already unbiased)}}}$

  The held-out observations were not used to fit the model, so there is **no optimistic bias** and **no correction needed**. The "test-like" behavior comes from the data split, not from a formula. No estimate of σ^2 or knowledge of d is required.

The metric is the same (squared error); the difference is where you compute it. Indirect methods measure error on training data and patch it with a penalty. Direct methods measure error on data the model has never seen, so no patch is needed. CV is more general and assumption-free, but computationally more expensive (the model must be refit multiple times). The formula-based criteria are cheap (one number from one fit) but rely on $\hat{\sigma}^2$ and d being correct.

3.3.1 Validation, Cross-Validation, and the Model Selection Pipeline

⋆ **Validation and Cross-Validation.** Each subset selection procedure returns a sequence of models $M_0, M_1, \ldots, M_p$ indexed by size k. Our goal is to select k^* and return model M_{k^*}.

Compute the **validation set error** or **cross-validation error** for each M_k, and select the k with the smallest estimated test error.

Cross-validation has several advantages over the adjustment-based criteria:

- It provides a **direct estimate** of test error rather than an indirect adjustment.
- It does **not** require an estimate of the error variance σ^2.
- It applies even when the effective **degrees of freedom** are hard to define (e.g. complex or non-linear models).

⋆ **Model Selection Pipeline.** Two jobs must be completed: (1) which predictors for each size, (2) which size is best. Job 1 is always done by subset selection. Job 2 is done by *either* a formula criterion *or* cross-validation, not both.

Pipeline A: Subset Selection + Formula Criterion (e.g. BIC, C_p). All steps use the **full dataset**: no splitting required.

1. **Run subset selection** (forward stepwise, backward, or best subset) on all n observations. This produces a sequence of models with specific variables already chosen: $M_0,\ M_1,\ M_2,\ \ldots,\ M_p$.
2. **Compute the criterion** (BIC, C_p, AIC, or Adj. R^2) for each M_k using the same training data. This is valid because the formula's penalty corrects for the optimistic bias mathematically.
3. **Pick** k^* = size with smallest BIC/C_p/AIC (or largest Adj. R^2).
4. **Return** M_{k^*} as the final model.

Fast (one pass, no refitting), but relies on a good estimate of σ^2 and a well-defined number of parameters d. If either is unreliable, use Pipeline B.

Pipeline B: Subset Selection + Cross-Validation. Data must be split; subset selection is **redone inside each fold** to avoid leaking information.

1. **Split** the data into K folds (e.g. $K = 5$ or 10).
2. **For each fold** $j = 1, \ldots, K$:
   (a) Hold out fold j as test data.
   (b) Run subset selection **from scratch** on the remaining $K - 1$ folds. This produces a new sequence $M_0^{(j)}, M_1^{(j)}, \ldots, M_p^{(j)}$. (The specific variables chosen may differ across folds; that is fine.)
   (c) For each size k, predict on the held-out fold j using $M_k^{(j)}$. Record the squared errors.
3. **Average** the squared errors across all K folds for each size k:
   $\mathrm{CV}(k) = \frac{1}{n} \sum_{j=1}^{K} \sum_{i \in \text{fold } j} (y_i - \hat{y}_i^{(j,k)})^2$.
4. **Pick** k^* = size with smallest $\mathrm{CV}(k)$.
5. **Refit:** run subset selection on the **full dataset** and take the model of size k^*. This is your final model.

The held-out fold never participates in subset selection for that fold, so there is no data leaking. The final refit on all data ensures the returned model uses every available observation. CV is more expensive (subset selection runs K times) but requires no estimate of σ^2 and no penalty formula.

3.4 Shrinkage Methods

⋆ **Shrinkage Methods.** Instead of selecting a subset of predictors, shrinkage fits a model with **all** p predictors but **constrains (regularizes)** the coefficient estimates, shrinking them towards zero relative to the OLS estimates.

The two main shrinkage methods are **Ridge Regression** (ℓ_2 penalty) and the **Lasso** (ℓ_1 penalty).
Core logic: OLS minimizes RSS alone:

$$\hat{\beta}^{\text{OLS}} = \arg\min_{\beta} \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2.$$

Shrinkage adds a **penalty term** that punishes large coefficients:

$$\hat{\beta}^{\text{shrink}} = \arg\min_{\beta} \left\{ \underbrace{\sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2}_{\text{fit the data (RSS)}} + \underbrace{\lambda \cdot \text{Penalty}(\beta)}_{\text{keep coefficients small}} \right\}$$

The model must now balance two competing goals: fitting the data well (low RSS) and keeping coefficients small (low penalty). The **tuning parameter** $\lambda \geq 0$ controls the trade-off.

Role of λ.

- $\lambda = 0$: no penalty $\Rightarrow$ reduces to OLS.
- $\lambda \to \infty$: penalty dominates $\Rightarrow$ all coefficients shrink to zero.
- Intermediate λ: trades a small increase in **bias** for a potentially large decrease in **variance**.

Why does shrinking coefficients help? OLS is **unbiased** but can have **high variance**, especially when p is large or predictors are correlated. By accepting a little bias (coefficients pulled towards zero), the estimates become much more stable across different samples. This is the **bias-variance trade-off**: a slightly wrong but stable model often predicts better on new data than an unbiased but noisy one.

3.4.1 Ridge Regression

⋆ **Ridge Regression:** It modifies OLS by adding an ℓ_2 **penalty** on the coefficients.

The ridge estimates $\hat{\beta}^R$ minimize: $\hat{\beta}^R = \arg\min_{\beta} \left\{ \underbrace{\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2}_{\textbf{RSS: fit the data}} + \underbrace{\lambda \sum_{j=1}^{p} \beta_j^2}_{\textbf{shrinkage penalty} = \lambda \|\beta\|_2^2} \right\}$

where $\lambda \geq 0$ is a **tuning parameter** and $\|\beta\|_2 = \sqrt{\sum_{j=1}^{p} \beta_j^2}$ is the ℓ_2 **norm** (Euclidean norm) of the coefficient vector.
Note: β_0 **is not penalized**: it is just an overall mean level of the response and should not be shrunk.
The objective balances two competing goals. The RSS term wants coefficients that explain the data; the penalty term wants coefficients close to zero. The model cannot make any coefficient large without paying a quadratic price in the penalty, which forces it to spread influence more evenly and avoid extreme estimates.

Role of λ: interpolating between full and null models.

- $\boxed{\lambda = 0}$: penalty vanishes $\Rightarrow \hat{\beta}^R = \hat{\beta}^{\text{OLS}}$ (full model, low bias, high variance).
- $\boxed{\lambda \to \infty}$: penalty dominates $\Rightarrow \hat{\beta}^R \to 0$ (null model, high bias, low variance).
- $\boxed{\textbf{Intermediate} \quad \lambda}$: adjusts between these two extremes.

Moving λ from 0 to ∞ traces a continuous path from the OLS solution to the zero vector. Choosing λ is itself a **model selection problem**: too small and we overfit, too large and we underfit. **Cross-validation** is used to select the best λ.

Ridge coefficient path: two views.

- **Plot x-axis** $= \lambda$: each curve shows one coefficient $\hat{\beta}_j^R$ as λ increases from 0 to ∞. All coefficients shrink smoothly towards zero but never reach it.
- **Plot x-axis** $= \|\hat{\beta}_\lambda^R\|_2 / \|\hat{\beta}^{\textbf{OLS}}\|_2$: same curves, but the x-axis is the **fraction of the original OLS coefficient size remaining** after shrinkage:

$$\frac{\|\hat{\beta}_\lambda^R\|_2}{\|\hat{\beta}^{\text{OLS}}\|_2} = \frac{\sqrt{\sum_{j=1}^p (\hat{\beta}_j^R)^2}}{\sqrt{\sum_{j=1}^p (\hat{\beta}_j^{\text{OLS}})^2}} \qquad \begin{cases} = 1 & \text{no shrinkage } (\lambda = 0) \\ = 0 & \text{full shrinkage } (\lambda \to \infty) \end{cases}$$

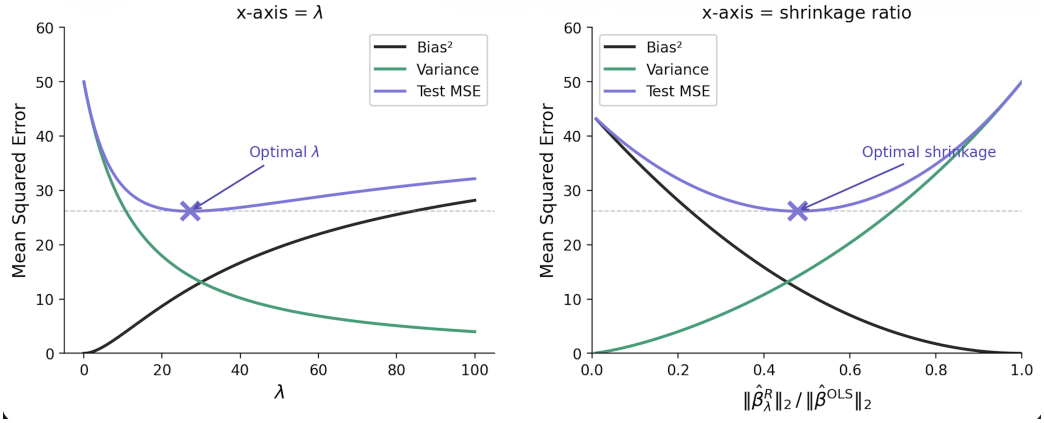
λ is in arbitrary units and hard to interpret across datasets. The ratio normalizes shrinkage to a 0–1 scale: "I have shrunk my model to 30% of its original size" is immediately meaningful regardless of the data.

⋆ **Bias-Var Trade-off in Ridge Regression:** $\underbrace{\text{MSE}(\hat{\beta}^R)}_{\text{test error}} = \underbrace{\text{Bias}^2}_{\text{increases with } \lambda} + \underbrace{\text{Variance}}_{\text{decreases with } \lambda}$

As λ increases from zero:

- **Bias increases:** coefficients are pulled away from their true values.
- **Variance decreases:** estimates become more stable across different samples.
- **Test MSE first decreases** (variance reduction outweighs bias), **reaches a minimum, then increases** (bias dominates).

OLS is unbiased but can have very high variance, especially when p is close to n or predictors are correlated. Ridge accepts a small bias to buy a large reduction in variance. The optimal λ is the point where test MSE is minimized, marked by the purple crosses in the bias-variance plot.



Black: *Bias^2 increases with λ as coefficients are pulled from their true values.*
Green: *Variance decreases as estimates become more stable.*
Purple: *Test MSE = Bias^2 + Variance, with a minimum (purple cross) at the optimal λ. The dashed line marks the minimum achievable MSE. Left panel uses λ on the x-axis; right panel uses the shrinkage ratio $\|\hat{\beta}_\lambda^R\|_2 / \|\hat{\beta}^{\text{OLS}}\|_2$, where 1 = no shrinkage (OLS) and 0 = full shrinkage (null model).*

⋆ **Scaling of Predictors:**

OLS is scale equivariant (scale does not matter): multiplying predictor X_j by a constant c scales $\hat{\beta}_j^{\text{OLS}}$ by $1/c$, so the fitted values $X_j\hat{\beta}_j$ are unchanged.

Ridge regression is not scale equivariant: Multiplying X_j by c changes the penalty contribution $\lambda\beta_j^2$, so the ridge estimates can change **substantially** depending on how predictors are scaled.

Before fitting ridge regression, standardize each predictor: $\boxed{\tilde{x}_{ij} = \dfrac{x_{ij} - \bar{x}_j}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \bar{x}_j)^2}}}$

After standardization every predictor has mean zero and variance one, so the penalty treats all predictors equally.

Key limitation of ridge regression: it pushes all coefficients towards zero but never sets any exactly to zero, so the final model always keeps all p predictors, the Lasso fixes this by producing sparse models where irrelevant variables are completely eliminated (i.e. automatic variable selection).

3.4.2 Lasso

⋆ **The Lasso:** Least Absolute Shrinkage and Selection Operator replaces ridge's ℓ_2 penalty with an ℓ_1 **penalty**.

Lasso estimates $\hat{\beta}^L$ minimize: $\boxed{\hat{\beta}^L = \arg\min_{\beta} \left\{ \underbrace{\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2}_{\text{RSS: fit the data}} + \underbrace{\lambda\sum_{j=1}^{p}|\beta_j|}_{\text{sparsity penalty} = \lambda\|\beta\|_1} \right\}}$

where $\|\beta\|_1 = \sum_{j=1}^{p}|\beta_j|$ is the ℓ_1 **norm** (Manhattan norm) of the coefficient vector.

The critical difference from ridge: the ℓ_1 **penalty forces some coefficients to be exactly zero when λ is sufficiently large**. The Lasso therefore performs **variable selection** automatically, yielding **sparse** models that involve only a subset of the predictors.

Role of λ: As with ridge, λ is selected by **cross-validation**.

- $\boxed{\lambda = 0}$: no penalty $\Rightarrow \hat{\beta}^L = \hat{\beta}^{\text{OLS}}$ (full model).
- $\boxed{\lambda \to \infty}$: penalty dominates $\Rightarrow$ all $\hat{\beta}_j^L = 0$ (null model).
- $\boxed{\textbf{Intermediate} \quad \lambda}$: some coefficients are exactly zero, others are shrunk. As λ increases, more coefficients are eliminated.

Lasso coefficient path. Plotting $\hat{\beta}_j^L$ as a function of λ (or $\|\hat{\beta}_\lambda^L\|_1/\|\hat{\beta}^{\text{OLS}}\|_1$):

- Unlike ridge, coefficients hit **exactly zero** at finite values of λ and stay there.
- As λ increases, variables are eliminated one by one; the path traces a sequence of progressively sparser models.
- The x-axis ratio uses the ℓ_1 norm (not ℓ_2): $\|\hat{\beta}_\lambda^L\|_1/\|\hat{\beta}^{\text{OLS}}\|_1$, since the Lasso constraint is in ℓ_1.

⋆ **Penalized vs. Constrained Form.** The penalized and constrained formulations are equivalent: for every λ there exists a unique budget s (and vice versa) that gives the same solution.

Budget s and penalty λ: The budget s is the maximum total size the coefficients are allowed to have: $\sum_{j=1}^{p} |\beta_j| \leq s$ for Lasso, $\sum_{j=1}^{p} \beta_j^2 \leq s$ for Ridge.
There is a one-to-one inverse mapping between s and λ:
large s (loose budget) $\Leftrightarrow$ small λ (weak penalty) $\Leftrightarrow$ close to OLS; small s (tight budget) $\Leftrightarrow$ large λ (strong penalty) $\Leftrightarrow$ more shrinkage.

Lasso (constrained):
$$\min_{\beta} \ \text{RSS}(\beta) = \min_{\beta} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 \quad \text{s.t.} \quad \sum_{j=1}^{p} |\beta_j| \leq s$$

Ridge (constrained):
$$\min_{\beta} \ \text{RSS}(\beta) = \min_{\beta} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 \quad \text{s.t.} \quad \sum_{j=1}^{p} \beta_j^2 \leq s$$

λ as Lagrangian The constraint $\|\beta\|_1 \leq s$ can be rewritten as $s - \|\beta\|_1 \geq 0$. Using a Lagrange multiplier λ:

$$\min_{\beta} \ \text{RSS} - \lambda(s - \|\beta\|_1) = \min_{\beta} \ \text{RSS} + \lambda\|\beta\|_1 - \underbrace{\lambda s}_{\text{constant}}$$

which is equivalent to minimizing $\text{RSS} + \lambda\|\beta\|_1$. So the penalized and constrained forms are two views of the same problem.

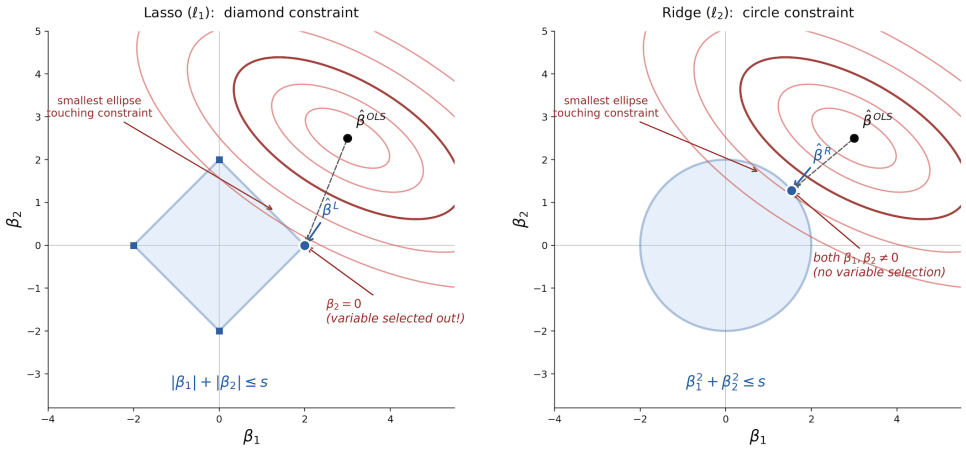
***Think of s as a budget**: the total absolute size of all coefficients cannot exceed s.*
***Large** $s \Rightarrow$ loose budget $\Rightarrow$ small $\lambda \Rightarrow$ close to OLS.*
***Small** $s \Rightarrow$ tight budget $\Rightarrow$ large $\lambda \Rightarrow$ more coefficients forced to zero.*

⋆ **Geometric Intuition: Why Lasso Produces Sparsity and Ridge Does Not.**

The constrained forms of Lasso and Ridge both minimize RSS within a **budget** s:

Lasso (ℓ_1): $\min_{\beta}$ RSS s.t. $|\beta_1| + |\beta_2| \leq s \Rightarrow$ constraint region is a **diamond**.

Ridge (ℓ_2): $\min_{\beta}$ RSS s.t. $\beta_1^2 + \beta_2^2 \leq s \Rightarrow$ constraint region is a **circle**.



- **Red ellipses** = **RSS contours**. Each ellipse is the set of all (β_1, β_2) combinations that give the same RSS value. Different points on the same ellipse are different coefficient pairs but with identical fit. The center $\hat{\beta}^{\text{OLS}}$ is the unconstrained minimum. Smaller (inner) ellipses = lower RSS = better fit. The ellipses are tilted because the predictors are correlated.

- **Blue shape** = **constraint region**. All (β_1, β_2) satisfying the budget s live inside.

Finding the solution. Imagine expanding the RSS ellipses outward from $\hat{\beta}^{\text{OLS}}$, like inflating a balloon. The constrained solution is the first point where an ellipse **touches** the constraint boundary: the smallest achievable RSS within the budget.

Why they differ.

- **Lasso (diamond):** the ellipse almost always hits a **corner** of the diamond. Corners sit on the coordinate axes, where one or more $\beta_j = 0$. In the figure, $\hat{\beta}^L$ lands on the β_1-axis, so $\beta_2 = 0 \Rightarrow$ **variable selection**.
- **Ridge (circle):** the ellipse touches the **smooth surface** of the circle at a generic point where both $\beta_1 \neq 0$ and $\beta_2 \neq 0 \Rightarrow$ **no variable selection**.

Sparsity comes from sharp corners. *The diamond has corners on the coordinate axes; the circle does not. In higher dimensions ($p > 2$), the Lasso constraint is an octahedron with corners, edges, and faces: all places where one or more $\beta_j = 0$. The ridge constraint is a smooth sphere with no such features. This is the fundamental geometric reason the Lasso performs variable selection and ridge does not.*

⋆ **General ℓ_q Constraints and Sparsity.** The constraint region $\sum_{j=1}^{p} |\beta_j|^q \leq s$ changes shape with q:

- $q = 2$: circle/sphere (ridge): smooth, no sparsity.
- $q = 1$: diamond/octahedron (lasso): sharp corners, sparsity.
- $q < 1$: even sharper, more sparsity, but the region is **nonconvex** $\Rightarrow$ optimization becomes combinatorially hard (NP-hard).

Two requirements for a useful sparsity-inducing constraint:

1. The constraint region must have **sharp corners** (so the solution lands on an axis where some $\beta_j = 0$).
2. The constraint region must be **convex and closed around zero** (so the optimization problem is tractable).

The ℓ_1 norm ($q = 1$) is the sweet spot: it is the smallest value of q for which the constraint region is still convex, while having corners sharp enough to produce sparse solutions. This is why the Lasso is so widely used: it gives variable selection "for free" within a convex optimization framework.

3.4.3 Selecting the Tuning Parameter λ for Ridge and Lasso

⋆ **Selecting the Tuning Parameter λ for Ridge and Lasso.** In subset selection, CV chose the best model **size** k^* from a sequence $M_0, M_1, \ldots, M_p$. For ridge/lasso, there is no sequence of discrete models; instead there is a **continuous dial (make it a finite grid with higher density around most likely value) λ that controls shrinkage.** CV now chooses the best λ^* from a grid of candidate values.

The role of λ in ridge/lasso is analogous to the role of k in subset selection: both control model complexity. The only difference is that k is discrete (1, 2, 3, ...) while λ is continuous, so we discretize it into a grid.

⋆ **CV Pipeline :**

Step 1: Choose a grid of λ values: Select a grid of candidate values, typically on a log scale:

$$\log_{10} \lambda \in \{-6, -5, \ldots, -1, 0, 1, \ldots, 6\}$$

We use a log scale because the effect of λ is multiplicative: the difference between $\lambda = 0.01$ and $\lambda = 0.1$ matters much more than between $\lambda = 100$ and $\lambda = 100.09$.

Step 2: Cross-validate each λ: For each candidate λ in the grid, run K-fold CV:

1. Split data into K folds (e.g. $K = 10$).
2. For each fold $j = 1, \ldots, K$:
   - Fit ridge or lasso with this specific λ on the $K - 1$ training folds, using **all p predictors**. The optimization
   $$\min_{\beta} \ \text{RSS} + \lambda \|\beta\|$$
   is solved internally; this is where the shrinkage happens. You do not choose which variables to include; the penalty handles it.
   - Predict on the held-out fold j, record the squared errors.
3. Average the errors across all K folds:

$$\text{CV}(\lambda) = \frac{1}{n} \sum_{j=1}^{K} \sum_{i \in \text{fold } j} (y_i - \hat{y}_i^{(j,\lambda)})^2$$

Now you have one CV error for each λ in the grid.

Step 3: Pick λ^*:

$$\boxed{\lambda^* = \arg\min_{\lambda} \ \text{CV}(\lambda)}$$

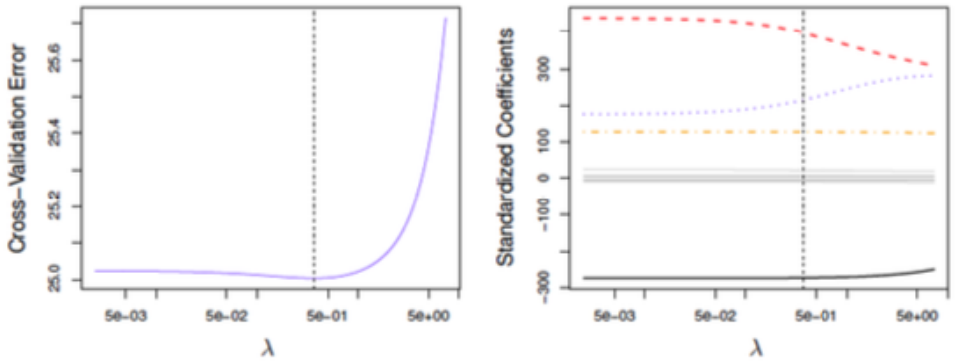
The λ with the smallest cross-validation error wins.

Step 4: Refit on full data: Fit the final model on **all n observations** using λ^*:

$$\boxed{\hat{\beta}^{\text{final}} = \arg\min_{\beta} \{\text{RSS} + \lambda^* \|\beta\|\}}$$

This is your final model. For ridge, all p coefficients are nonzero (just shrunk). For lasso, some coefficients are exactly zero; those variables are automatically excluded.

Notice the key difference from subset selection: you never manually decide which variables to keep. You give the model all p predictors and let the penalty do the work. The optimization algorithm (internal to the software) finds the $\hat{\beta}$ that minimizes RSS + penalty for each λ. Cross-validation just tells you which λ gives the best predictions on new data.



Reading the CV plot (ridge example).

- **Left panel:** CV error (y-axis) vs. λ (x-axis). The curve dips to a minimum then rises. The minimum is λ^* (marked by a vertical dashed line). Too small λ = overfitting (high variance), too large λ = underfitting (high bias).
- **Right panel:** coefficient paths vs. λ. The vertical dashed line shows where λ^* falls; this determines how much each coefficient is shrunk in the final model.

The two panels together tell the full story: the left panel picks λ^*, the right panel shows what the coefficients look like at that λ^*. For lasso, some coefficient paths hit zero before λ^*; those variables are eliminated from the final model.

3.4.4 Elastic Net

The Lasso's problem with correlated variables. If X_j has coefficient $\hat{\beta}_j > 0$ and we add an identical copy $X_{j'} = X_j$, Lasso is indifferent to any split $\tilde{\beta}_j + \tilde{\beta}_{j'} = \hat{\beta}_j$ because both the RSS and the ℓ_1 penalty are unchanged. The coefficients for the pair are not uniquely defined.

A quadratic (ℓ_2) penalty, on the other hand, would divide $\hat{\beta}_j$ exactly equally between the two twins. In practice we rarely have identical variables, but groups of highly correlated predictors are common (e.g. genes in the same biological pathway).

⋆ **Elastic Net Penalty.** The Elastic Net combines ℓ_1 and ℓ_2 penalties into a single regularizer:

$$\hat{\beta}^{EN} = \arg\min_{\beta} \left\{ \text{RSS} + \lambda \left[\underbrace{\alpha \|\beta\|_1}_{\text{Lasso (sparsity)}} + \underbrace{\frac{1-\alpha}{2} \|\beta\|_2^2}_{\text{Ridge (grouping)}} \right] \right\}$$

where $\alpha \in [0, 1]$ is the **mixing parameter** that controls the blend:

- $\alpha = 1$: pure Lasso (ℓ_1 only).
- $\alpha = 0$: pure Ridge (ℓ_2 only).
- $0 < \alpha < 1$: a mix of both: sparsity from ℓ_1 plus stability from ℓ_2.

*The ℓ_1 component provides **variable selection** (drops irrelevant predictors). The ℓ_2 component provides the **grouping effect**: correlated variables receive similar coefficients and enter or leave the model together, rather than one being arbitrarily eliminated. The Elastic Net has two tuning parameters (λ and α), both selected by cross-validation.*

Grouping effect (intuition from coefficient paths): With six variables in two correlated groups of three:

- **Lasso** ($\alpha = 1$): coefficient paths within each group are erratic: variables in the same group cross, diverge, and flip as λ changes.
- **Elastic Net** ($\alpha = 0.3$): coefficient paths within each group are pulled together: correlated variables move in sync and are included or excluded as a group.

By adding a ridge component to the Lasso penalty, the Elastic Net automatically controls for within-group correlations, producing more stable and interpretable coefficient paths.

Constraint region geometry. The Elastic Net constraint region is a hybrid of the ℓ_1 diamond and ℓ_2 sphere:

- It has **sharp corners and edges** from the ℓ_1 part $\Rightarrow$ encourages sparsity (variable selection).
- It has **curved contours** from the ℓ_2 part $\Rightarrow$ encourages sharing (correlated variables get similar coefficients).

The shape looks like a diamond with rounded edges: pointy enough to zero out coefficients, smooth enough to keep correlated variables together. As α moves from 1 to 0, the shape morphs continuously from a pure diamond (Lasso) to a pure sphere (Ridge).

	Ridge	**Lasso**	**Elastic Net**
Penalty	$\lambda\|\beta\|_2^2$	$\lambda\|\beta\|_1$	$\lambda[\alpha\|\beta\|_1 + \frac{1-\alpha}{2}\|\beta\|_2^2]$
Variable selection?	No	Yes	Yes
Handles correlated vars?	Yes	Poorly	Yes
Constraint shape	Sphere	Diamond	Rounded diamond
Tuning parameters	λ	λ	λ, α

3.5 Dimension Reduction Methods

⋆ **Dimension Reduction Methods.** Instead of selecting or shrinking the original predictors $X_1, X_2, \ldots, X_p$, dimension reduction methods **transform** them into a smaller set of new variables and then fit a regression on the transformed variables. These are **feature transformation** methods, not feature selection methods: every original predictor contributes to the new variables.

Idea: Construct $M < p$ new variables $Z_1, \ldots, Z_M$, each a **linear combination** of the original predictors:

$$\boxed{Z_m = \sum_{j=1}^{p} \phi_{mj} X_j, \qquad m = 1, \ldots, M}$$

for some constants (weights) $\phi_{m1}, \phi_{m2}, \ldots, \phi_{mp}$. Then fit OLS on the new variables:

$$\boxed{y_i = \theta_0 + \sum_{m=1}^{M} \theta_m z_{im} + \varepsilon_i, \qquad i = 1, \ldots, n}$$

Instead of estimating p coefficients in the original space, we estimate only $M \ll p$ coefficients in the reduced space. Fewer parameters means lower variance, at the cost of some bias from compressing the information.

⋆ **Connection to the Original Linear Model.** The dimension-reduced model is a **special case** of the original linear model with a constraint on the coefficients. Substituting $Z_m = \sum_j \phi_{mj} X_j$ into the regression:

$$\sum_{m=1}^{M} \theta_m z_{im} = \sum_{m=1}^{M} \theta_m \sum_{j=1}^{p} \phi_{mj} x_{ij} = \sum_{j=1}^{p} \underbrace{\left(\sum_{m=1}^{M} \theta_m \phi_{mj} \right)}_{\beta_j} x_{ij} = \sum_{j=1}^{p} \beta_j x_{ij}$$

So the effective coefficient on the original predictor X_j is: $\boxed{\beta_j = \sum_{m=1}^{M} \theta_m \phi_{mj}}$

*This is the key insight: dimension reduction does not change the model form: it is still $\sum \beta_j X_j$. But it **constrains** the β_j to be expressible as a linear combination of only M weight vectors. With $M < p$, the β_j's are no longer free: they live in an M-dimensional subspace. This constraint reduces variance, just as shrinkage does, and can win in the bias-variance trade-off if the weights ϕ_{mj} are chosen wisely.*

How does this reduce variance?

- OLS on the original p predictors estimates p free parameters ⇒ high variance when p is large.
- OLS on the M derived variables estimates only M free parameters $(\theta_1, \ldots, \theta_M)$, while the ϕ_{mj} weights are fixed before regression ⇒ far fewer parameters to estimate ⇒ lower variance.

*The critical question is: how to choose the weights ϕ_{mj}? Different answers give different methods: **Principal Components Regression** (PCR) chooses them to capture maximal variance in X, while **Partial Least Squares** (PLS) chooses them to capture maximal covariance between X and Y.*

3.5.1 Principal Components Analysis (PCA)

⋆ **Principal Components Analysis (PCA).** PCA defines the weights ϕ_{mj} by finding directions of **maximum variance** in the predictor space. It is an **unsupervised** method: it uses only the predictors X, not the response Y.

First principal component Z_1. Find the loading vector $\phi_1 = (\phi_{11}, \ldots, \phi_{1p})$ that maximizes the variance of the projection:

$$\boxed{z_{i1} = \sum_{j=1}^{p} \phi_{1j}\, x_{ij}, \qquad \phi_1 = \arg\max_{\|\phi\|_2=1} \mathrm{Var}(Z_1) = \arg\max_{\|\phi\|_2=1} \frac{1}{n-1} \sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_j\, x_{ij} \right)^2}$$

The normalization constraint $\|\phi_1\|_2^2 = \sum_{j=1}^{p} \phi_{1j}^2 = 1$ prevents the trivial solution of making ϕ_1 arbitrarily large. The data are assumed **centered** ($\bar{x}_j = 0$), so $\bar{Z}_1 = 0$ and the variance simplifies to $\frac{1}{n-1} \sum_i z_{i1}^2$.

*Geometrically, ϕ_1 is the direction along which the data cloud is most "stretched out." Projecting all n observations onto this line captures the single most informative one-dimensional summary of the predictor space. Equivalently, this is the line that minimizes the sum of squared **perpendicular** distances from each data point to the line (not vertical distances as in OLS).*

Second principal component Z_2. Find the loading vector ϕ_2 that maximizes variance among all directions **orthogonal** (uncorrelated) to the first:

$$\boxed{z_{i2} = \sum_{j=1}^{p} \phi_{2j}\, x_{ij}, \qquad \|\phi_2\|_2 = 1, \qquad \phi_2 \perp \phi_1}$$

General m-th component. Each subsequent PC maximizes variance subject to being orthogonal to all previous loading vectors:

$$\phi_m = \arg \max_{\substack{\|\phi\|_2=1 \\ \phi \perp \phi_1, \ldots, \phi_{m-1}}} \mathrm{Var}(Z_m)$$

Together, $Z_1, \ldots, Z_p$ form a new orthogonal coordinate system for the data. The first few PCs capture most of the total variance when the original predictors are correlated. If the first M PCs explain, say, 95% of the total variance, then the remaining $p - M$ directions contribute very little information and can be safely discarded.

Properties of principal components.

- $Z_1, Z_2, \ldots, Z_p$ are mutually **uncorrelated**: $\mathrm{Cov}(Z_m, Z_k) = 0$ for $m \neq k$.
- $\mathrm{Var}(Z_1) \geq \mathrm{Var}(Z_2) \geq \cdots \geq \mathrm{Var}(Z_p)$: variance is non-increasing.
- The loading vectors $\phi_1, \ldots, \phi_p$ are the **eigenvectors** of the sample covariance matrix of X, and the variances $\mathrm{Var}(Z_m)$ are the corresponding **eigenvalues**.
- With all p components, no information is lost: we simply rotated the coordinate system.

3.5.2 Principal Components Regression (PCR)

⋆ **Principal Components Regression (PCR).** PCR uses the first M principal components as predictors in an OLS regression:

PCR Algorithm.

1. **Standardize** all p predictors (mean zero, unit variance).
2. Compute the first M principal components $Z_1, \ldots, Z_M$ from the standardized predictors. Note: the response Y is **not used** in this step.
3. Regress Y on $Z_1, \ldots, Z_M$ by OLS:

$$\boxed{\hat{Y} = \hat{\theta}_0 + \hat{\theta}_1 Z_1 + \cdots + \hat{\theta}_M Z_M}$$

4. Select M by **cross-validation**.

PCR replaces the p correlated original predictors with M uncorrelated principal components that capture their joint variation. Since $M < p$, we estimate fewer parameters, which reduces variance. When $M = p$, PCR is exactly equivalent to OLS on the original predictors: no dimension reduction occurs.

Why standardize? PCA maximizes variance. Without standardization, variables measured in large units (e.g. income in dollars) dominate the first PCs simply because they have larger numerical variance, regardless of their importance. Standardizing puts all predictors on the same scale so that variance reflects actual variability, not measurement units. This is the same issue as in ridge and lasso.

M as a tuning parameter. M controls model complexity, analogous to k in subset selection or λ in ridge/lasso:

- $M = 0$: null model (intercept only, high bias, low variance).
- $M = p$: full OLS (no reduction, low bias, high variance).
- Intermediate M: bias-variance trade-off.

The bias-variance trade-off in PCR mirrors ridge regression. Few components = aggressive compression = high bias, low variance. Many components = little compression = low bias, high variance. The optimal M is where test MSE is minimized, identified by CV.

Key assumption and limitation. PCR assumes that the directions of **maximum variance in X** are also the directions most associated with the **response Y**. This is **not guaranteed**: PCA is **unsupervised** (it never looks at Y).

A direction with high predictor variance but no relationship to Y would be included as an early PC, while a low-variance direction that is highly predictive of Y might be excluded. This is the fundamental weakness of PCR: what matters for explaining variance in X may not be what matters for predicting Y. Partial Least Squares addresses this directly.

PCR is not feature selection. PCR uses **all** p original predictors: each PC is a linear combination of all of them. It reduces the number of **regression coefficients** (M instead of p), but no original variable is dropped. This makes PCR harder to interpret than subset selection or lasso.

3.5.3 Partial Least Squares (PLS)

⋆ **Partial Least Squares (PLS).** PLS is a **supervised** dimension reduction method: unlike PCR, it uses **both** the predictors X **and** the response Y to construct the new variables $Z_1, \ldots, Z_M$.

Motivation. PCR finds directions of maximum variance in X, hoping they are also predictive of Y. PLS directly finds directions that have high variance in X **and** are strongly correlated with Y.

PLS asks: "which linear combination of predictors best explains the response?" rather than "which linear combination has the most variance?" This makes PLS more targeted but also more prone to overfitting, since it uses Y during feature construction.

PLS Algorithm.

1. **Standardize** all p predictors and the response.
2. **Compute the first direction Z_1:** Set the weight of predictor X_j in the first PLS direction equal to the **simple regression coefficient** of Y on X_j:

$$\boxed{\phi_{1j} = \langle X_j, Y \rangle \qquad \Rightarrow \qquad Z_1 = \sum_{j=1}^{p} \phi_{1j} X_j}$$

   where $\langle X_j, Y \rangle$ is the inner product (proportional to the correlation between X_j and Y when both are standardized).

   This gives the highest weight to predictors most strongly correlated with the response. Unlike PCA, which only looks at predictor variance, PLS directly incorporates the relationship with Y.
3. **Compute subsequent directions $Z_2, Z_3, \ldots$:** To find Z_m, first **residualize** each predictor X_j by regressing it on $Z_1, \ldots, Z_{m-1}$ and taking the residuals. Then set the weights of Z_m using the simple regression coefficients of Y on these residualized predictors.

   Residualizing removes the information already captured by previous directions, ensuring each new direction adds genuinely new predictive information. This is analogous to the orthogonality constraint in PCA, but driven by the response.
4. **Regress Y on $Z_1, \ldots, Z_M$** by OLS:

$$\boxed{\hat{Y} = \hat{\theta}_0 + \hat{\theta}_1 Z_1 + \cdots + \hat{\theta}_M Z_M}$$

5. Select M by **cross-validation**.

PCA vs. PCR vs. PLS: comparison.

	PCA	**PCR**	**PLS**
What is it?	feature extraction (no Y involved)	regression method (PCA + OLS)	regression method (supervised PCA + OLS)
Goal	compress X into fewer dimensions	predict Y using PCA directions	predict Y using Y-aware directions
Uses Y?	No (unsupervised)	No (for components)	Yes (supervised)
Components maximize	max variance in X	same as PCA	max covariance between X and Y
Weights ϕ_{mj} set by	eigenvectors of $\mathrm{Cov}(X)$	same as PCA	$\phi_{1j} = \langle X_j, Y \rangle$ (residualized for $m > 1$)
Key formula	$Z_m = \sum_j \phi_{mj} X_j$ $\|\phi_m\| = 1,\ \phi_m \perp \phi_{<m}$	$\hat{Y} = \hat{\theta}_0 + \sum_{m=1}^{M} \hat{\theta}_m Z_m$ (where Z_m from PCA)	$\hat{Y} = \hat{\theta}_0 + \sum_{m=1}^{M} \hat{\theta}_m Z_m$ (where Z_m from PLS)
Tuning param.	none (M if truncated)	M (via CV)	M (via CV)
Feature selection?	No	No	No
Needs standard.?	Yes	Yes	Yes
Risk	may discard low-var but useful directions	include high-var but irrelevant directions	overfit by using Y in component construction
$M = p$ gives	full rotation of X	same as OLS	same as OLS

PCA is a general-purpose tool for compressing predictors; PCR plugs those compressed features into a regression. PLS modifies the compression step to target the response directly. In practice, PLS often reduces bias relative to PCR (by targeting Y-relevant directions), but the supervision can increase variance (by fitting noise in Y). Both give similar performance in many standard settings; PLS tends to help most when the dominant variance directions in X are not predictive of Y.

Practical note. When $M = p$, PLS produces the same fit as OLS on the original predictors, just as PCR does. The dimension reduction only occurs when $M < p$. Both methods require standardization and use CV to choose M. Neither performs variable selection: all p predictors contribute to every component.

Chapter 4

Tree-Based Methods

4.1 Basics of Decision Trees

⋆ **Tree-Based Methods.** **Decision trees** partition the **predictor space** $\mathcal{X} = \{(X_1, \ldots, X_p)\}$ into distinct non-overlapping rectangular regions, and make a single flat prediction in each region. They apply to both **regression** (predict a continuous Y) and **classification** (predict a class label $Y \in \{1, \ldots, K\}$).

*A single tree is simple and highly interpretable, but typically weak in prediction accuracy. This motivates ensemble methods (**bagging**, **random forests**, and **boosting**) which grow many trees and combine them into a single consensus prediction, dramatically improving accuracy at the cost of some interpretability.*

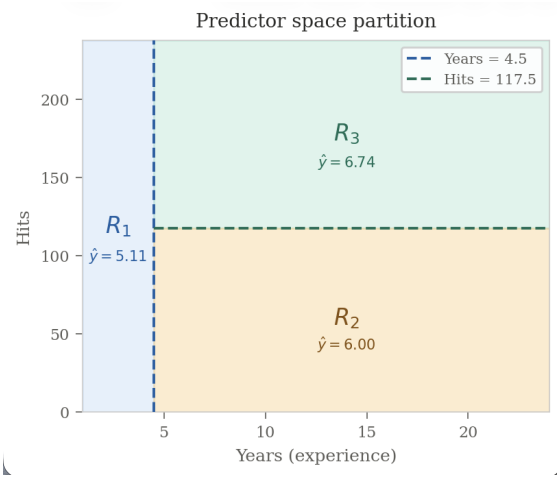
⋆ **Tree Terminology.**

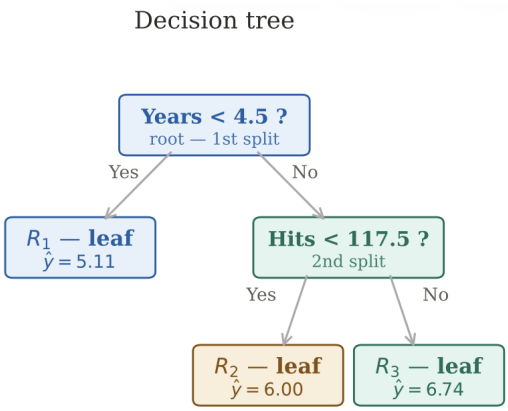
- **Internal node**: a point in the tree where the predictor space is split using a rule of the form $X_j < t_k$. The **left branch** takes $X_j < t_k$ and the **right branch** takes $X_j \geq t_k$.
- **Terminal node** (leaf): a final region R_j at the bottom of the tree. Each leaf stores the prediction for observations that fall there.
- Trees are drawn **upside down**: the root (first split) is at the top and the leaves are at the bottom.

⋆ **Data to Tree:** Consider predicting log(Salary) from two predictors: **Years** (experience) and **Hits** (hits last year). The tree-building procedure finds the best axis-aligned cuts in the (Years, Hits) plane:

1. **First split**: a vertical cut at $\texttt{Years} = 4.5$ divides the plane into left (R_1: low experience) and right (experienced players). This is the **root node**.
2. **Second split**: within the right half only, a horizontal cut at $\texttt{Hits} = 117.5$ creates two sub-regions: R_2 (experienced, few hits) and R_3 (experienced, many hits).

The result is three **terminal nodes** R_1, R_2, R_3, each predicting the **mean response** of training observations in that region:
$R_1 = \{X \mid \texttt{Years} < 4.5\}$,
$R_2 = \{X \mid \texttt{Years} \geq 4.5,\ \texttt{Hits} < 117.5\}$,
$R_3 = \{X \mid \texttt{Years} \geq 4.5,\ \texttt{Hits} \geq 117.5\}$
with predictions $\hat{y}_{R_1} = 5.11$, $\hat{y}_{R_2} = 6.00$, $\hat{y}_{R_3} = 6.74$ (in log-salary).





The scatter plot and the tree are two representations of the same object. Each split in the tree corresponds to a straight line cutting the predictor space in the scatter plot. Reading the tree top-down traces a path through those rectangles; a new observation simply follows the decision rules from root to leaf and receives that leaf's prediction.

⋆ **Regression tree: prediction rule.** Partition the predictor space $\{(X_1, \ldots, X_p)\}$ into J distinct, non-overlapping **rectangular regions** (boxes) $R_1, \ldots, R_J$. For every observation falling in region R_j, predict the **mean response** of all training observations in that box:

$$\boxed{\hat{y}_{R_j} = \frac{1}{|R_j|} \sum_{i \in R_j} y_i = \bar{y}_{R_j}}$$

⋆ **RSS criterion for tree building.** The goal is to find boxes $R_1, \ldots, R_J$ that minimize the **residual sum of squares** across all regions:

$$\boxed{\text{RSS} = \sum_{j=1}^{J} \sum_{i \in R_j} \left(y_i - \hat{y}_{R_j}\right)^2}$$

Each squared error measures how far a training observation is from its region's mean. Minimizing this pushes the tree to form regions that are internally homogeneous in Y.

⋆ **Recursive binary splitting.** Exhaustive search over all partitions into J boxes is computationally infeasible. Instead, we use a **top-down**, **greedy** procedure:

- **Top-down:** start at the root and successively split the predictor space downward; each split produces two new branches.
- **Greedy:** at each step, choose the **best split available now** (the predictor X_j and cutpoint s such that splitting into $\{X \mid X_j < s\}$ and $\{X \mid X_j \geq s\}$ gives the greatest reduction in RSS) without looking ahead to future splits.

Greedy splitting does not guarantee a globally optimal tree. A locally weak split early on might have enabled much better splits later, but finding the global optimum requires evaluating all possible trees, which is intractable. This limitation motivates pruning and ensemble methods.

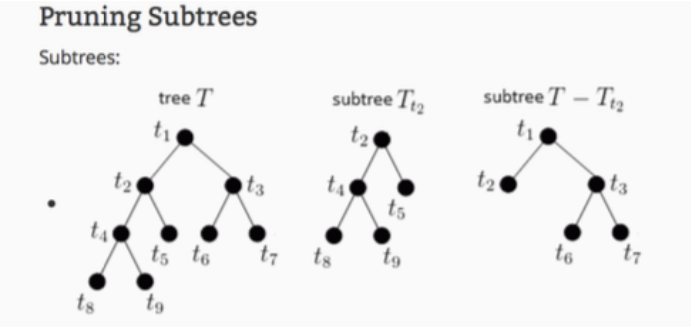
⋆ **Why pruning is needed.** A fully grown tree fits the training data very well but **overfits**: it creates too many regions, each capturing noise rather than signal. The result is low bias but very high variance: poor test performance. A **smaller tree with fewer splits trades a little bias for a large reduction in variance.**

⋆ **Pruning procedure** means cutting branches off a tree that do not help. It is a **post-processing algorithm**: grow a very large tree T_0 first, then **cut it back** by repeatedly collapsing the **weakest link** (the internal node whose removal hurts the training fit the least) until only a single root remains. This produces a nested sequence of subtrees

$$T_0 \supset T_1 \supset T_2 \supset \cdots \supset \{\text{root}\}$$

each one simpler than the last. The **cost-complexity criterion** $\sum\sum(y_i - \bar{y}_{R_m})^2 + \alpha|T|$ is not the pruning procedure itself; it is the **scoring function** that tells you which subtree in that sequence is best for a given penalty α. Cross-validation then picks the optimal $\hat{\alpha}$.

The analogy to OLS is exact: RSS is the equation you minimize, but $(X^\top X)^{-1}X^\top y$ is the algorithm that achieves it. Similarly, the cost-complexity equation defines what a good subtree looks like, and weakest-link pruning is the algorithm that efficiently finds it, without ever checking every possible subtree, which would be infeasible.



⋆ **Cost complexity pruning (weakest link pruning).** The correct strategy: grow a very large tree T_0 first, then **prune** it back to find the best **subtree**. For a tuning parameter $\alpha \geq 0$, find the subtree $T \subset T_0$ that minimizes the **cost-complexity criterion**:

$$\underbrace{\sum_{m=1}^{|T|} \sum_{i:\, x_i \in R_m} \left(y_i - \bar{y}_{R_m}\right)^2}_{\text{total RSS across all } |T| \text{ leaves (training fit)}} + \underbrace{\alpha}_{\text{penalty rate}} \underbrace{|T|}_{\text{num. leaves}}$$

The penalty $\alpha|T|$ is an L_0 penalty: it charges a flat cost α per leaf added, penalizing model complexity directly as a discrete count rather than through coefficient magnitudes as in Lasso or Ridge.

- $|T|$: number of **terminal nodes** (leaves) of subtree T. Controls tree size.
- R_m: the rectangular region of predictor space corresponding to the m-th leaf.
- $\bar{y}_{R_m}$: mean response of all training observations in region R_m.
- $\sum_{i:\, x_i \in R_m}(y_i - \bar{y}_{R_m})^2$: RSS within leaf m; how well the tree fits the data in that region.
- $\sum_{m=1}^{|T|}(\cdots)$: total RSS summed over all leaves; the overall training fit.
- $\alpha|T|$: **penalty term**: charges a cost α for each additional leaf. Discourages overly complex trees.

The criterion balances two opposing forces. The RSS term wants more leaves (better fit); the penalty $\alpha|T|$ wants fewer leaves (simpler tree). When $\alpha = 0$, there is no penalty and $T = T_0$ is optimal. As α increases, the penalty grows and the algorithm prunes leaves whose RSS reduction does not justify the complexity cost, collapsing the tree until only the most important splits survive.

Role of α. The **tuning parameter** $\alpha \geq 0$ controls the bias-variance trade-off:

- $\alpha = 0 \Rightarrow$ no penalty $\Rightarrow$ full tree T_0 (low bias, high variance)
- $\alpha \to \infty \Rightarrow$ heavy penalty $\Rightarrow$ null single-node tree (high bias, zero variance)
- Optimal $\hat{\alpha}$ chosen by **cross-validation**

Each value of α corresponds to a different subtree: a nested sequence from the full tree down to the root. Pruning does not search all possible subtrees; it efficiently finds the best one for each α by collapsing the weakest links first (the splits that contribute least RSS reduction per added leaf). Grow big, then cut back smartly.

4.2 Regression Tree Algorithm

⋆ **Recursive binary splitting, how the large tree is grown.** At each node, search exhaustively over all p predictors and all possible cutpoints. For predictor X_j and cutpoint s, define the two candidate regions:

$$R_L(j,s) = \{X \mid X_j < s\} \qquad R_R(j,s) = \{X \mid X_j \geq s\}$$

Choose the pair (j^*, s^*) that minimizes the split RSS:

$$\boxed{(j^*, s^*) = \underset{j \in \{1,\ldots,p\},\ s}{\arg\min} \left[\sum_{i:\, x_i \in R_L(j,s)} \left(y_i - \bar{y}_{R_L}\right)^2 + \sum_{i:\, x_i \in R_R(j,s)} \left(y_i - \bar{y}_{R_R}\right)^2 \right]}$$

Recurse on each resulting region until a stopping criterion is met (e.g. $|R_j| < 5$).

At every node all p predictors are tried from scratch; a predictor can appear multiple times, at different levels, or never. The tree discovers importance purely from data: predictors that win the RSS competition higher up the tree are more informative about Y.

⋆ **Regression tree algorithm: full procedure.**

Step 1: Grow a large tree, overfit Use **recursive binary splitting** on the training data. Stop only when each terminal node contains fewer than some minimum number of observations. This produces a large, overfit tree T_0.

Step 2: Prune via cost-complexity. Apply **weakest link pruning** to T_0. For each value of α this yields the subtree $T(\alpha) \subset T_0$ minimizing:

$$\boxed{\sum_{m=1}^{|T|} \sum_{i:\, x_i \in R_m} \left(y_i - \bar{y}_{R_m}\right)^2 + \alpha |T|}$$

Varying α from 0 to ∞ produces a nested sequence $T_0 \supset T_1 \supset T_2 \supset \cdots$

Step 3: Choose α by K-fold cross-validation.
For each fold $k = 1, \ldots, K$:

- Repeat Steps 1–2 on the $\frac{K-1}{K}$ fraction of training data excluding fold k
- Evaluate **MSE** on the left-out fold k as a function of α

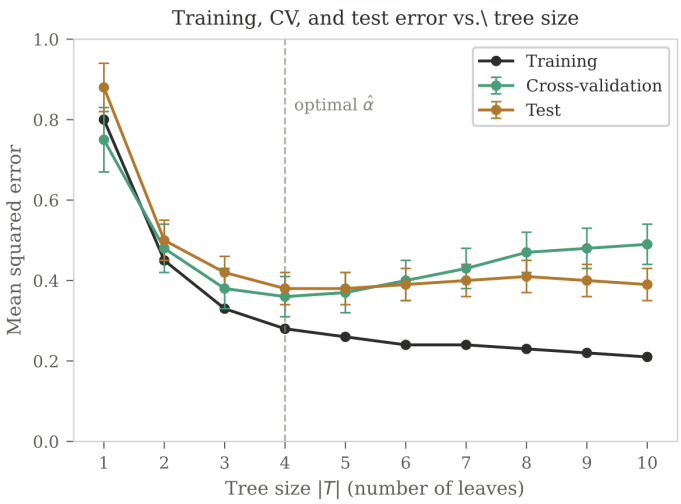
Average across folds; pick $\hat{\alpha}$ minimizing the average CV error.

Step 4: Return final subtree. Return the subtree from Step 2 corresponding to $\hat{\alpha}$.

Steps 1–2 run once on the full training set to produce the candidate sequence. Step 3 reruns Steps 1–2 on each CV fold independently to estimate test error, never touching the held-out fold during fitting. Step 4 then retrieves the right subtree from the full-data sequence using the $\hat{\alpha}$ found by CV.

What the error curves tell us.

- **Training error** decreases monotonically as the tree grows: more leaves always fit training data better.
- **CV and test error** decrease at first (underfitting regime) then flatten or rise (overfitting regime).
- The **optimal tree size** $\hat{\alpha}$ is where CV error is minimized, typically a small tree, well before training error bottoms out.



Training error falls with every added leaf. CV and test error improve up to the optimal size then rise; the vertical dashed line marks $\hat{\alpha}$, the point cross-validation selects.

4.3 Classification Trees

Classification prediction: how it works. For region R_m containing n_m training observations, the **class proportion** for class k is:

$$\hat{p}_{mk} = \frac{1}{n_m} \sum_{i:\, x_i \in R_m} \mathbf{1}(y_i = k)$$

that is, simply **count how many observations in R_m belong to class k, divide by total observations in R_m.** Then predict the majority class: $\boxed{\hat{y}_{R_m} = \text{argmax}_k \ \hat{p}_{mk}}$

Example. Suppose R_m contains 10 observations: 7 with $y = \text{Yes}$, 3 with $y = \text{No}$. Then:

$$\hat{p}_{m,\text{Yes}} = \tfrac{7}{10} = 0.7 \qquad \hat{p}_{m,\text{No}} = \tfrac{3}{10} = 0.3 \quad \hat{y}_{R_m} = \text{argmax}_k \ \hat{p}_{mk} = \text{Yes}$$

$\hat{p}_{mk}$ is nothing more than an empirical frequency: no model, no estimation, just counting. The prediction rule is pure majority vote within the region. Note $\sum_{k=1}^{K} \hat{p}_{mk} = 1$ always, since every observation in R_m belongs to exactly one class.

$\star$ **Classification trees.** Identical structure to a regression tree, but predicts a **qualitative response** $Y \in \{1, \ldots, K\}$. The prediction for any observation falling in region R_m is the **most commonly occurring class** among training observations in that region:

$$\boxed{\hat{y}_{R_m} = \text{argmax}_k \ \hat{p}_{mk}}$$

where $\hat{p}_{mk}$ is the proportion of training observations in region R_m belonging to class k.

⋆ **Split criterion: why RSS fails.** The tree is grown using **recursive binary splitting** as before, but RSS cannot be used as the split criterion: Y is categorical so squared errors are meaningless. Instead, three alternatives are used:

Classification error rate. Fraction of observations in region R_m not belonging to the majority class:

$$E_m = 1 - \max_k(\hat{p}_{mk})$$

$\max_k(\hat{p}_{mk})$ *is the proportion of the dominant class; subtracting from 1 gives the misclassification rate. If one class owns 90% of a node,* $E_m = 0.10$. *A pure node (one class only) gives* $E_m = 0$. *In practice, error rate is* ***not sensitive enough*** *for growing trees: small improvements in purity produce negligible changes in* E_m*, so the two measures below are preferred.*

4.3.1 Gini and Deviance (CART)

⋆ **Gini index (CART):** (Classification and Regression Trees) Measures **node purity: total variance across all K classes in region R_m:**

$$\boxed{G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})}$$

Gini takes as input the class proportions $\hat{p}_{mk}$ (the fraction of observations in region R_m belonging to each class k) and outputs a single number in $[0, 0.5]$ measuring how mixed the node is: zero when the node is perfectly pure and maximal when all classes are equally represented.

Interpretation.: Each **term $\hat{p}_{mk}(1-\hat{p}_{mk})$ is the variance of a Bernoulli for class k.** G is small when all $\hat{p}_{mk}$ are near 0 or 1, meaning the node is dominated by one class (**pure**). G is maximized when all classes are equally represented. Used in **CART** (Classification and Regression Trees).

Think of $\hat{p}_{mk}(1-\hat{p}_{mk})$ *as asking: if I randomly picked two observations from this node, how likely are they to be from different classes? Low Gini means most observations share the same label: a clean, useful node.*

⋆ **Cross-entropy / deviance**: Measures **uncertainty** or disorder in a node aka node impurity :

$$\boxed{D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}}$$

Asks: how uncertain am I about the class of a randomly drawn observation from this node?

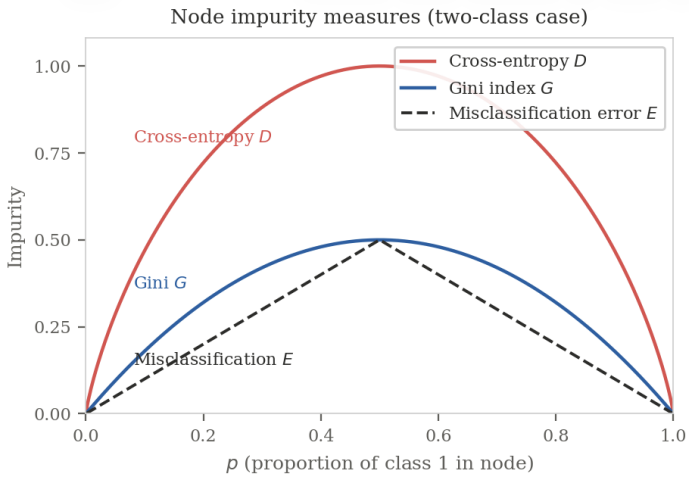
Interpretation: $D = 0$ when the node is pure ($\hat{p}_{mk} = 1$ for one class). D is maximized when all K classes are equally likely. Numerically very similar to the Gini index; both penalize mixed nodes and reward pure ones.

Cross-entropy comes from information theory: it measures the average number of bits needed to encode a class label drawn from this node's distribution. A pure node needs zero bits (you already know the answer); a maximally mixed node needs the most bits.

⋆ **Comparison of split criteria.**

- **Misclassification error** $E = 1 - \max_k(\hat{p}_{mk})$: linear in p, too insensitive for growing trees: small purity improvements cause negligible change. Used only for **pruning evaluation**.
- **Gini** and **cross-entropy**: both strictly concave, more sensitive to node composition, strongly preferred for **growing** the tree. Both peak at $p = 0.5$ (maximum impurity) and equal zero at $p \in \{0, 1\}$ (pure node).

All three measures equal zero at $p = 0$ and $p = 1$ (pure nodes) and peak at $p = 0.5$ (maximally mixed). Gini and cross-entropy are more curved: they are more sensitive to partial purity improvements, making them better split criteria than misclassification error.



⋆ **Weighted impurity for split selection:** A split divides region R_m into two child regions R_L and R_R. To evaluate the split fairly, impurity is **weighted** by the fraction of observations in each child:

$$\boxed{I_{\text{split}} = \frac{|R_L|}{|R_m|} I(R_L) + \frac{|R_R|}{|R_m|} I(R_R)}$$

where $I(\cdot)$ is any impurity measure (G, D, or E), and $|R|$ denotes the number of observations in region R. Choose the split (j^*, s^*) that minimizes I_{split}.

Without weighting, a split that creates one tiny pure child and one large impure child would look artificially good: the pure child dominates the average. Weighting by size forces the algorithm to account for how many observations actually benefit from the purity gain.

Information gain. Before a split, region R_m has n_m observations and impurity G_m (Gini index). A split divides it into two children R_L (n_L obs, impurity G_L) and R_R (n_R obs, impurity G_R).

The **information gain** of the split is: $\boxed{\Delta G = G_m - \left(\frac{n_L}{n_m} G_L + \frac{n_R}{n_m} G_R\right)}$

- G_m: impurity of the parent node before splitting
- $\frac{n_L}{n_m} G_L + \frac{n_R}{n_m} G_R$: **weighted impurity** after the split
- $\Delta G > 0$ means the split improved purity; choose the split maximizing ΔG

Unweighted vs. weighted (from example). With 400 total obs, left child $n_L = 300$, right child $n_R = 100$:

$$\text{Unweighted: } G_1 + G_2 \qquad \text{Weighted: } \frac{300}{400} G_1 + \frac{100}{400} G_2$$

Weighting is necessary for fairness: a split that creates one tiny pure child and one large impure child looks artificially good without weights. Multiplying each child's impurity by its size fraction ensures larger regions contribute more to the overall score, reflecting their greater impact on predictions.

Trees versus linear models. Neither approach dominates; the better model depends on the true decision boundary:

- **True boundary is linear:** a linear model fits cleanly with one equation; a tree must approximate it with many axis-aligned rectangular cuts, wasteful and less accurate.
- **True boundary is non-linear:** a tree captures it naturally with a few splits; a linear model fundamentally cannot represent it without manual feature engineering.

4.4 Bagging

⋆ **Bagging (Bootstrap Aggregation).** is a general-purpose **ensemble method**: a **wrapper** that can be applied to any statistical learning method.
Idea: a single tree has high variance (small changes in data produce very different trees). If we could average many trees trained on independent datasets, variance would drop by $1/B$. Since we only have one training set, we **simulate** multiple datasets via **bootstrap sampling**.

- **Input:** one training set $\{(x_i, y_i)\}_{i=1}^n$, a number of trees B
- **Output:** a single aggregated prediction $\hat{f}_{\text{bag}}(x)$ with lower variance than any single tree

Variance of the mean. For n independent observations $Z_1, \ldots, Z_n$ each with variance σ^2:

$$\text{Var}(\bar{Z}) = \frac{\sigma^2}{n}$$

Averaging reduces variance. Bagging exploits this: instead of one tree, average B trees.

⋆ **Bagging procedure.**

1. Draw B **bootstrap samples** from the training data (sample n observations **with replacement**)
2. Fit a full unpruned tree $\hat{f}^{*b}(x)$ to the b-th bootstrap sample, $b = 1, \ldots, B$
3. Average all predictions:

$$\boxed{\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)}$$

Prediction rule by task:

- **Regression:** average the B numerical predictions: the formula above directly
- **Classification:** **majority vote**: each tree votes for a class, predict the most commonly occurring class across all B trees

Each bootstrap sample contains roughly $\frac{2}{3}$ of the original observations (some appear multiple times, some not at all). The trees are grown deep and not pruned: high variance individually, but averaging cancels the variance out. Bias stays the same as a single deep tree; only variance drops.

⋆ **Out-of-Bag (OOB) error estimation.** Each bootstrap tree uses $\approx \frac{2}{3}$ of observations. The remaining $\approx \frac{1}{3}$ not used to fit tree b are its **out-of-bag** (OOB) observations. For each observation x_i, predict using only the $\approx B/3$ trees for which x_i was OOB, then average:

$$\boxed{\hat{f}_{\text{OOB}}(x_i) = \frac{1}{|\mathcal{B}_i|}\sum_{b \in \mathcal{B}_i} \hat{f}^{*b}(x_i)}$$

where $\mathcal{B}_i = \{b : x_i \text{ was OOB in bootstrap sample } b\}$.

OOB error is the average prediction error over all n observations using their respective OOB predictions. For large B this is essentially the **leave-one-out cross-validation error**, computed at no extra cost, with no need to refit the model.

4.5 Random Forests

⋆ **Random Forests. Random forests** improve over bagging by a single tweak: at each split, instead of considering all p predictors, only a **random subset of m predictors** is chosen as candidates. This **decorrelates** the trees, reducing variance further when averaged.

Why bagging alone is insufficient. In bagging all B trees see all p predictors at every split. If one predictor is very strong, it will dominate the root split of nearly every tree, making all trees look similar and highly correlated. Averaging correlated trees gives limited variance reduction:

$$\mathrm{Var}\left(\frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x)\right) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

where ρ is the **pairwise correlation** between trees. Even as $B \to \infty$, variance floors at $\rho\sigma^2$. **Reducing ρ reduces this floor**; that is exactly what random forests do.

Bagging is Monte Carlo averaging of correlated samples. Random forests break the correlation by forcing each tree to ignore most predictors at each split, making trees genuinely different from each other, so averaging them actually kills variance.

⋆ **Random forest algorithm.** At each node, instead of searching over all p predictors:

1. Draw bootstrap sample (same as bagging)
2. At each split, randomly select $m \le p$ predictors as candidates
3. Find the best split using **only those m predictors**
4. Grow the full tree; repeat B times
5. Aggregate: average (regression) or majority vote (classification)

$$\boxed{\hat{f}_{\mathrm{RF}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x), \qquad m \approx \sqrt{p}}$$

A fresh random subset of m predictors is drawn **independently at every split**, not once per tree.

Choice of m. The key tuning parameter:

- $m = p \;\Rightarrow$ reduces to **bagging** (no decorrelation)
- $m = \sqrt{p} \;\Rightarrow$ standard **random forest** (default for classification)
- $m = p/3 \;\Rightarrow$ common alternative for regression

Smaller $m \Rightarrow$ more decorrelated trees $\Rightarrow$ lower variance, but each individual tree is weaker (higher bias). Cross-validation can be used to tune m.

The $m = \sqrt{p}$ rule is purely empirical: it works well across many problems. With $p = 500$ predictors (gene expression example), each split only considers ≈ 22 predictors, so even the strongest predictor is frequently excluded, forcing other predictors to carry the split and diversifying the trees.

⋆ **Bagging vs. random forests: comparison.**

- **Same:** bootstrap sampling, B trees, same aggregation rule, OOB error available
- **Different:** at each split, RF restricts candidates to $m < p$ predictors $\Rightarrow$ decorrelated trees $\Rightarrow$ lower variance
- **Result:** random forests always outperform or match bagging; the gain is largest when a few predictors strongly dominate

Method	Row randomness	Column randomness
Single tree	None	None: all p predictors
Bagging	Bootstrap ($\frac{2}{3}$ of rows)	None: all p predictors
Random forest	Bootstrap ($\frac{2}{3}$ of rows)	Random $m \approx \sqrt{p}$ predictors per split

⋆ **Alternative version of random forests.** Instead of bootstrap sampling rows, one can:

- Build each of the B trees on a **random subset of m features** (columns) directly
- Use a **subsample with replacement** of size different from n, equivalent to a bootstrap sample of a different size

Both versions decorrelate trees; the standard version (bootstrap rows + random m features at each split) is most common.

4.6 Boosting for Regression

⋆ **Boosting.** Like bagging, **boosting** is a general ensemble algorithm for regression and classification. The key difference from bagging: trees are grown **sequentially**: each tree is fit to the **residuals** of the current model, not to the original response. No bootstrap sampling. The model learns **slowly and adaptively**.

Bagging builds independent trees in parallel on random data copies and averages them. Boosting builds small trees one at a time, each correcting the mistakes of all previous trees, like a student who reviews only the questions they got wrong before each exam.

⋆ **Boosting algorithm for regression trees.**

Initialise: set $\hat{f}(x) = 0$ and residuals $r_i = y_i$ for all i.

For $b = 1, 2, \ldots, B$ **repeat:**

1. Fit a small tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r)
2. **Update** the model by adding a shrunken version of the new tree:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$

3. **Update** the residuals:

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$$

Output the boosted model:

$$\boxed{\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)}$$

Terms in the algorithm:

- r_i: **residual** for observation i: what the current model has not yet explained. Initialized to y_i, updated after every tree.
- $\hat{f}^b(x)$: the b-th small tree, fit to current residuals (not to Y)
- λ: **shrinkage parameter**: scales down each tree's contribution before adding it. Slows learning deliberately.
- d: number of splits per tree; controls how complex each weak learner is
- $\hat{f}(x)$: the running model, built up incrementally across all B trees

At each step the tree does not try to predict Y; it tries to predict what the current model got wrong. Adding $\lambda \hat{f}^b(x)$ nudges the model in the right direction, then the residuals shrink. The next tree attacks what remains. The shrinkage λ ensures no single tree dominates; the model inches toward the solution rather than leaping, which prevents overfitting.

⋆ **Key idea: learning slowly.** A single large tree fits the data hard and overfits. Boosting instead:

- Fits many **small** trees (stumps if $d = 1$) sequentially to residuals
- Each tree corrects errors in areas where $\hat{f}$ performs poorly

- λ slows the process further, allowing more diverse trees to contribute

The analogy: rather than one expert who claims to know everything (single large tree), boosting assembles a committee of many humble specialists, each focused on the hardest remaining cases.

Boosting for classification. Same spirit as regression boosting but more complex: each tree predicts a class and weights are assigned to observations based on misclassification. Implemented in the `gbm` package in R (gradient boosted models).

4.6.1 Tuning Parameters for Boosting

⋆ **Tuning parameters for boosting.** Boosting has three key tuning parameters, all chosen by **cross-validation**:

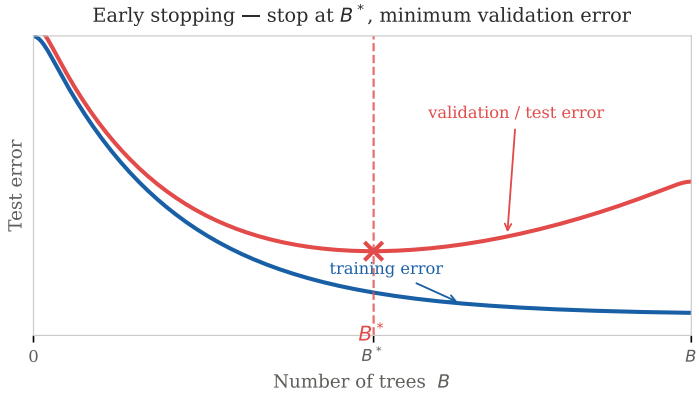
- **1. Number of trees B.** Unlike bagging/RF, boosting **can overfit** if B is too large, though overfitting tends to occur slowly. Cross-validation selects B.

$$B \text{ too small} \Rightarrow \text{underfitting}; \qquad B \text{ too large} \Rightarrow \text{slow overfitting}$$

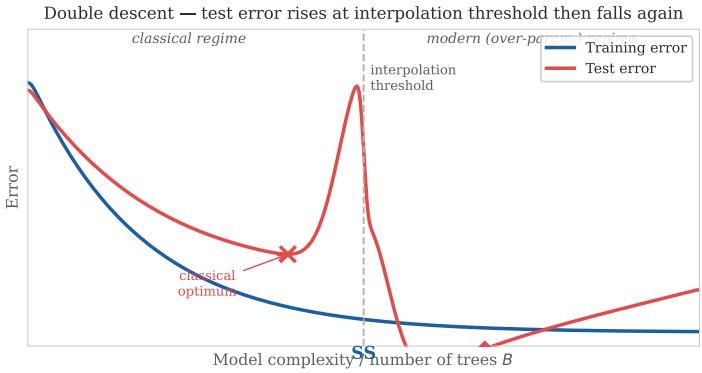
- **2. Shrinkage parameter $\lambda > 0$.** Controls the **learning rate**. Typical values: $\lambda \in \{0.01, 0.001\}$.

$$\lambda \text{ small} \Rightarrow \text{slower learning} \Rightarrow \text{need larger } B$$

  Very small λ paired with large B often gives best performance at the cost of computation.
- **3. Number of splits d per tree.** Controls **interaction depth**: with d splits, each tree involves at most d variables, capturing d-way interactions.
  - $d = 1 \Rightarrow$ each tree is a **stump** (single split) $\Rightarrow$ purely additive model, no interactions
  - $d = 2 \Rightarrow$ two-way interactions allowed
  - Often $d = 1$ or $d = 2$ works well in practice

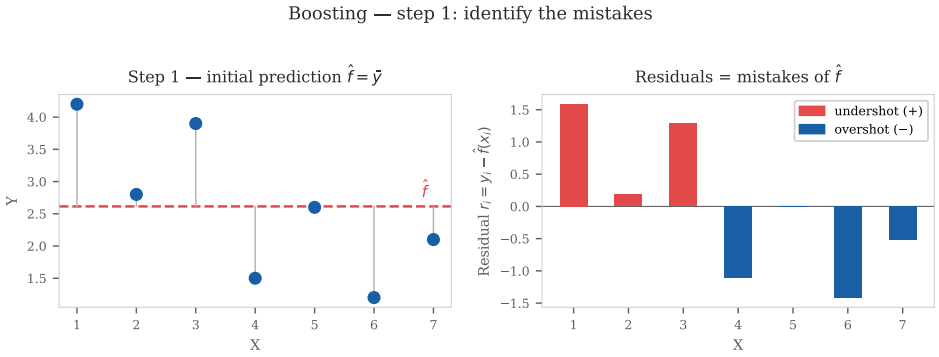


Training error falls monotonically as B grows. Validation error has a U-shape: it improves then rises as boosting overfits. B^* marks the minimum validation error; stop here. Adding more trees past B^* hurts generalization.

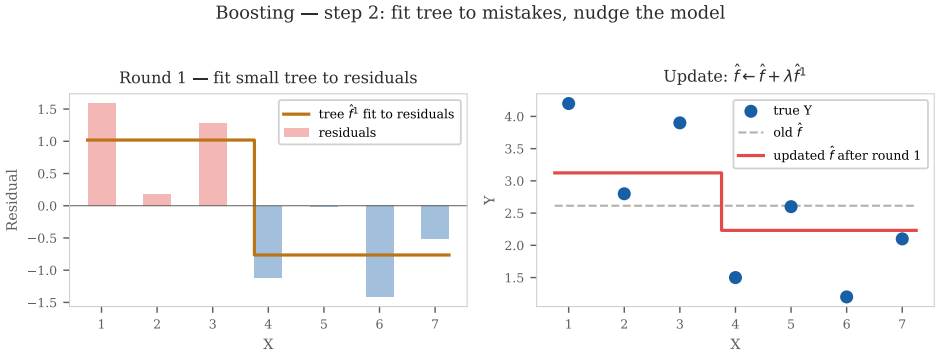


Test error descends twice. In the classical regime it forms a U-shape with a clear optimum. At the interpolation threshold (SS) the model exactly fits training data and test error spikes. In the modern over-parameterised regime, further complexity paradoxically reduces test error again, sometimes below the classical optimum.

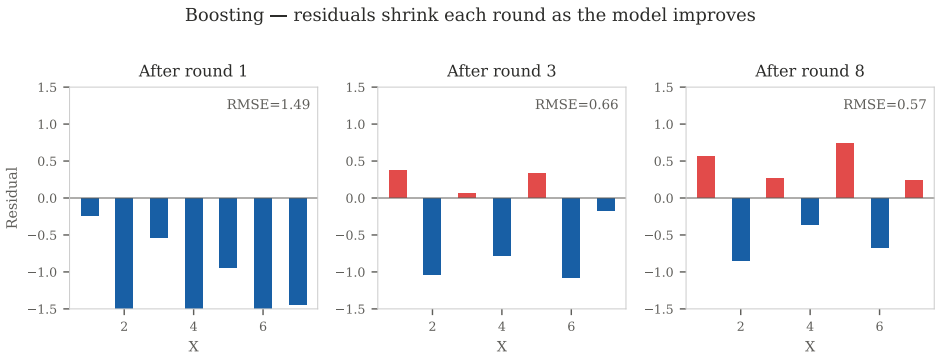
Boosting Core Logic:



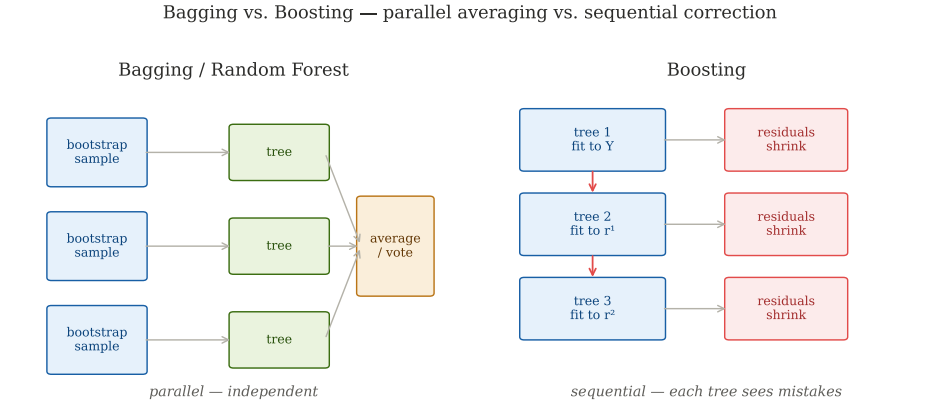
Step 1: fit an initial flat prediction $\hat{f} = \bar{y}$. The residuals $r_i = y_i - \hat{f}(x_i)$ are the mistakes: red bars undershoot, blue bars overshoot.



Round 1: fit a small tree $\hat{f}^1$ to the residuals (not to Y). Add a shrunken version to the model: $\hat{f} \leftarrow \hat{f} + \lambda \hat{f}^1$. The model nudges toward the truth without overcorrecting.



Each round the residuals shrink; the model has less to correct. RMSE falls with every tree added. Eventually residuals approach zero and the model has learned the signal.



*Bagging builds independent trees in parallel on bootstrap samples and averages them: reduces **variance**. Boosting chains trees sequentially, each correcting the mistakes of the last: reduces **bias**.*

4.6.2 Variable Importance and Method Comparison

⋆ **Variable importance measure.** Ensemble methods lose the interpretability of a single tree: you can no longer read off which predictors matter from one diagram. Instead, **variable importance** is measured by how much each predictor reduces the split criterion, averaged across all B trees:

- **Regression (bagging/RF):** record the total **RSS decrease** due to splits on predictor X_j, averaged over all B trees:

$$\boxed{\mathrm{VI}(X_j) = \frac{1}{B}\sum_{b=1}^{B} \sum_{\text{splits on } X_j \text{ in tree } b} \Delta\mathrm{RSS}}$$

- **Classification (bagging/RF):** same idea but sum the total **Gini index decrease** due to splits on X_j, averaged over all B trees.

A large value of $\mathrm{VI}(X_j)$ *means that splitting on* X_j *consistently reduced impurity or RSS by a large amount across many trees: strong evidence that* X_j *is genuinely informative about* Y*. A predictor that never wins splits, or wins only weak ones, gets a near-zero importance score.*

⋆ **Bagging vs. random forests vs. boosting: key distinctions.**

	Bagging	**Random Forest**	**Boosting**
Tree growth	Parallel	Parallel	**Sequential**
Data per tree	Bootstrap	Bootstrap	Full data (residuals)
Tree size	Large (deep)	Large (deep)	Small (d splits)
Randomness	Row sampling	Row + column sampling	None (deterministic)
Can overfit?	No	No	**Yes** (if B large)
Key parameter	B	B, m	B, λ, d

Boosting is the most powerful of the three but also the most sensitive to tuning. Random forests are often the best default choice: robust, parallelizable, and require minimal tuning. Boosting can outperform random forests but requires careful selection of B*,* λ*, and* d*.*

⋆ **Summary: tree-based methods.**

- **Single decision tree:** simple, interpretable, handles regression and classification, but typically lower predictive accuracy than competing methods due to high variance.
- **Bagging, random forests, boosting:** improve accuracy dramatically by growing many trees on the training data and combining their predictions into a single ensemble. The price is loss of interpretability.
- **Random forests and boosting** are among the **state-of-the-art methods** for supervised learning: strong default choices in practice across both regression and classification.

The progression through this chapter reflects a fundamental trade-off: as models become more powerful (single tree → bagging → random forest → boosting), they become harder to interpret. Variable importance plots partially recover interpretability by summarizing which predictors drove the ensemble's decisions across all B *trees.*

4.7 Extra Trees

⋆ **Extremely Randomized Trees .** Proposed by Geurts et al. (2006). The core idea: make individual trees **even weaker** than in random forests, and compensate with a larger ensemble. Three sources of randomness:

- **Features and samples** shown to each tree are randomized (as in random forests)
- **Split thresholds** are also randomized: instead of searching for the best cutpoint, a random threshold is drawn for each candidate feature

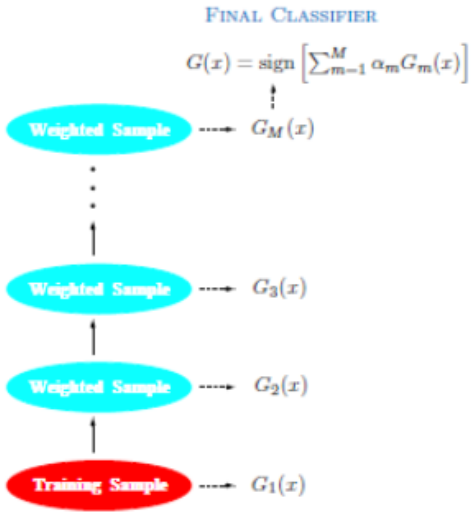
Randomizing the split point makes each tree faster to train. Output: aggregate via majority vote (classification) or average (regression). Implemented as `sklearn.ensemble.ExtraTreesClassifier`.

Random forests randomize which features compete at each split but still find the best cutpoint among them. Extra Trees go further: the cutpoint itself is random. This makes each tree weaker individually but further decorrelates the ensemble, often matching or beating random forest accuracy at lower computational cost.

4.8 Boosting for Classification

⋆ **Boosting for Classification:** iteratively learns a sequence of **weak classifiers** (each with error rate only slightly better than random guessing) and combines them into one strong classifier via a **weighted majority vote**:

$$\boxed{G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]}$$



- $G_m(x)$: the m-th weak classifier, fit to a **reweighted** version of the training data
- α_m : scalar weight assigned to G_m; larger for more accurate classifiers
- $\sum_{m=1}^{M} \alpha_m G_m(x)$: weighted vote; $\text{sign}(\cdot)$ converts to a class label $\in \{-1, +1\}$

Weight update rule. After each round m:

- **Up-weight** observations misclassified by G_m; the next classifier focuses harder on them
- **Down-weight** observations correctly classified; they are "easy" and need less attention

Each round re-exposes the same training set but with a shifted probability distribution. Hard examples accumulate weight across rounds, forcing successive classifiers to specialize on the cases that the ensemble keeps getting wrong.

4.8.1 AdaBoost.M1 Algorithm

⋆ **AdaBoost.M1 Algorithm.** The canonical boosting algorithm for binary classification ($y_i \in \{-1, +1\}$):

Initialize: observation weights $w_i = \frac{1}{N}$, $i = 1, \ldots, N$.

For $m = 1$ **to** M:

1. Fit classifier $G_m(x)$ to training data using current weights $\{w_i\}$
2. Compute **weighted error rate**:

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i \cdot \mathbf{1}(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$$

3. Compute **classifier weight**:

$$\alpha_m = \log \frac{1 - \text{err}_m}{\text{err}_m}$$

4. **Update** observation weights:

$$w_i \leftarrow w_i \cdot \exp\Big[\alpha_m \cdot \mathbf{1}(y_i \neq G_m(x_i))\Big], \quad i = 1, \ldots, N$$

Output: $\boxed{G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]}$

When err_m *is small,* α_m *is large: that round's classifier gets a loud vote. Misclassified observations have their weights multiplied by* $e^{\alpha_m} > 1$*, so they loom larger in the next round. A perfect classifier (*$\text{err}_m = 0$*) would get infinite weight and end the process in one step; a useless one (*$\text{err}_m = 0.5$*) gets* $\alpha_m = 0$ *and contributes nothing.*

Common classification boosting variants: AdaBoost, LPBoost, BrownBoost, LogitBoost, and Gradient Boosting.

Boosting Intuition.

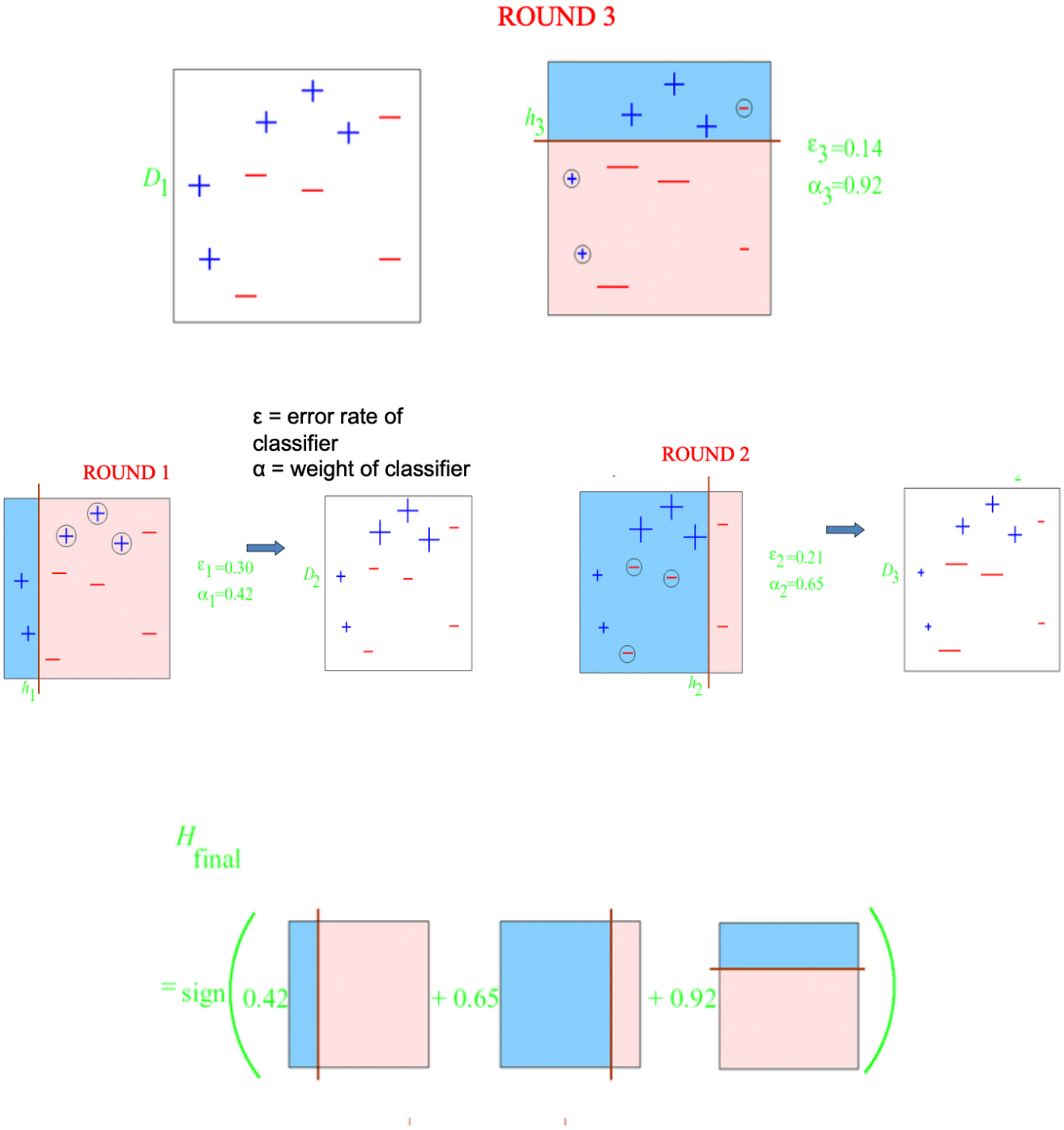
- Each round fits a new simple classifier on the **reweighted** dataset: hard cases get higher weight
- Classifiers with low training error get high α_m: more influence in the final vote
- Stopping: monitor a held-out set (cross-validation) to choose M

Properties and limitations of boosting.

- **Resistant to overfitting:** test error often keeps falling even after training error reaches zero: the margin keeps improving
- **General:** improves the performance of many base learning algorithms
- **Fails when:** not enough data; base learner is too weak (can't learn signal) or too strong (overfits each round); data is very noisy (up-weighting noise examples hurts)

The overfitting resistance is surprising: classical bias-variance intuition suggests that driving training error to zero should overfit. Boosting escapes this because the margin (confidence of correct classification) continues to grow even after the error is zero, and larger margins generalize better.

AdaBoost: Illustrative Example.

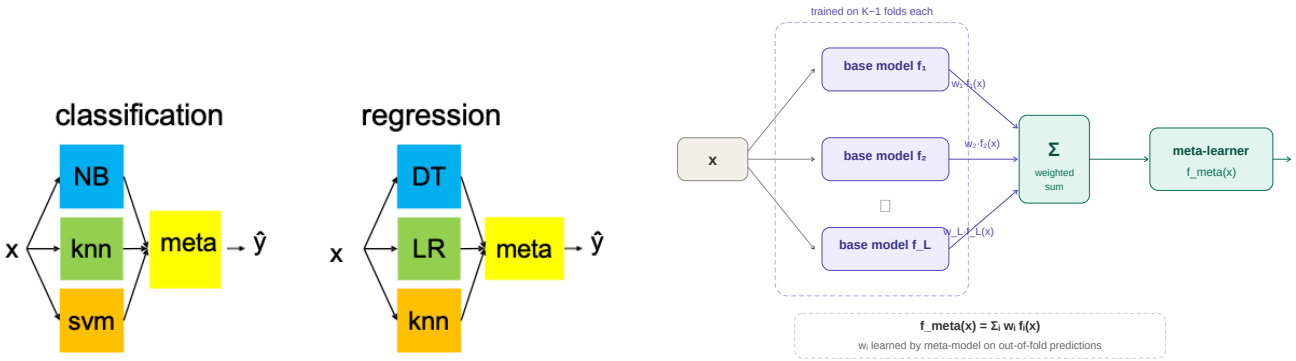


4.9 Ensemble Methods: Stacking and Mixture of Experts

$$\underbrace{\text{Ensemble Methods}}_{\text{combine multiple models into one prediction}} \supset \underbrace{\text{Static Combiners}}_{\text{fixed weights}} \supset \underbrace{\text{Adaptive Meta-Learners}}_{\text{weights depend on } x} \supset \underbrace{\text{Mixture of Experts}}_{\text{specific adaptive model}}$$

4.9.1 Stacking

⋆ **Stacking.** Use the output of L **base models** $f_1(x), \ldots, f_L(x)$ as input to a **meta-model** (meta-learner), which produces the final prediction. The meta-model is stacked on top of the base models.



⋆ **Naïve approach: regression case.** For regression, the meta-learner can be a linear combination of base model outputs: $\boxed{f_{\text{meta}}(\mathbf{x}) = \sum_{i=1}^{L} w_i f_i(\mathbf{x})}$

- If w_i could be chosen to minimize **true error**, the stacked model is always at least as good as any single base model: worst case, set all $w_i = 0$ except for the best base model
- **Problem:** minimizing **training error** instead causes the meta-model to prefer **overfitted base models**: their outputs have been adapted to the labels, so they look artificially good at train time

⋆ **Cross-Validated Stacking** To avoid preference for overfitted models, the inputs fed to the meta-model during training **must not have seen the labels** for the data points they correspond to. The solution: generate base-model outputs via **cross-validation**.

- Split training data into K folds; for each fold, train base models on the other $K-1$ folds and predict on the held-out fold
- These **out-of-fold predictions** are the inputs to the meta-learner; they reflect true test-time behavior since each prediction was made without seeing that point's label
- Train the meta-model on $\{(f_1(x_i), \ldots, f_L(x_i)), y_i\}$ using these cross-validated outputs
- If the meta-model has free parameters, use the same folds to cross-validate them

This is exactly the same logic as OOB estimation in bagging: each base model prediction is made on data it was not trained on, so the meta-learner receives uncontaminated inputs.

Meta-model choice. Choose simple models: linear regression or logistic regression. Empirical rule: do not use the same model class as the base learners. If base learners include logistic regression, use a different meta-learner (e.g. SVM).

⋆ **Testing the stacked model.** Always set aside a **test set from the beginning**, never used during base model training or meta-model training. At test time, two approaches for obtaining base model

predictions:

- **Retrain** all base models on the full training set, then predict on the test set: slightly different input distribution to the meta-model (since training used fold subsets)
- **Average** the K fold-trained versions of each base model: avoids retraining but costs time

Then feed the base model test predictions into the trained meta-model to get $\hat{y}$.

⋆ **Stacking vs. model selection.**

- If the meta-learner is forced to use exactly one base model ($w_j = 1$, all others $= 0$), stacking reduces to **cross-validation model selection**, picking the best single model
- A more expressive meta-model (linear/logistic regression) can exploit the **relative strengths** of multiple models: different base models may dominate in different regions of the input space
- A **too complex** meta-model (e.g. deep decision tree) can overfit at the meta level; apply cross-validation on the meta level if needed

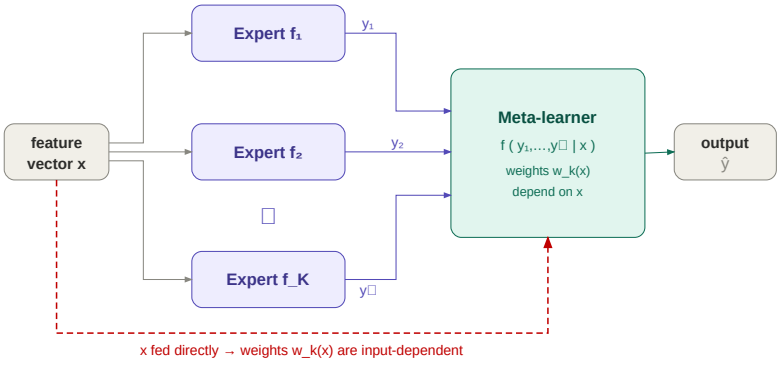
Stacking is the most general ensemble strategy: unlike bagging (same model, different data) or boosting (same model family, sequential reweighting), stacking combines heterogeneous models (each can be any learning algorithm) and lets the meta-learner discover the optimal blend.

4.9.2 Adaptive Meta-Learners & Mixture of Experts

⋆ **Adaptive Meta-Learners.** In standard stacking the meta-model uses a **fixed** combination rule: the same weights w_i apply to all inputs x. An **adaptive meta-learner** instead makes the combination rule a **function of** x: different regions of input space get different weights, so each base model specializes where it performs best.

Standard stacking asks "which model is best overall?" **Adaptive meta-learning asks "which model is best here, for this specific input?"** A classifier that dominates on easy examples can coexist with one that handles edge cases; neither needs to be globally good.

- Each **expert** (base model) only needs to perform well in its **region of expertise**, not across all inputs
- The ensemble implements a **divide-and-conquer** strategy: partition the input/output space, assign specialists to each region
- Results in **modular ensembles** where simple classifiers **specialize** in different parts of the I/O space f
- Key contrast with static combiners (bagging, stacking): individual experts here need **not** perform well for all inputs, only locally



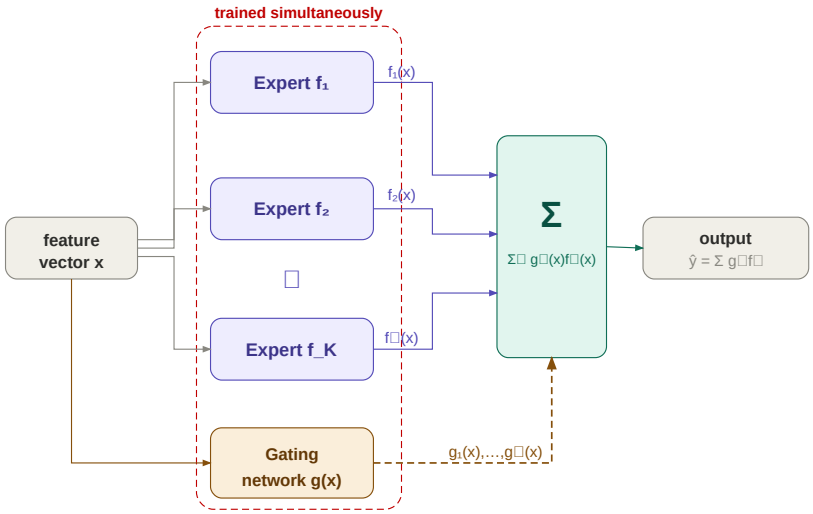
*The feature vector x is fed to all K experts simultaneously. Each expert k produces output y_k. The meta-expert sees both the expert outputs **and** the original feature vector x; this is what makes it adaptive: the combination weights depend on where x lives in feature space.*

⋆ **Mixture of Experts (ME).** The classical adaptive ensemble. A **gating network** $g(x)$ takes the input x and outputs a probability distribution over experts; these are the **input-dependent weights**. The final output is:

$$f_{\text{ME}}(x) = \sum_{k=1}^{K} g_k(x)\, f_k(x), \qquad \sum_{k=1}^{K} g_k(x) = 1, \quad g_k(x) \geq 0$$

- $g_k(x)$: gating function (logistic, etc.) outputs build **gating network** for expert k; how much expert k is trusted at input x. (Typically a softmax over a linear function of x)
- $f_k(x)$: output of the k-th expert (base model)
- The gating network **partitions feature space** into regions, with one expert responsible for each region
- **Key:** experts and gating network are **trained simultaneously**, unlike stacking where base models are trained first

The gating network is a learned router. As training proceeds, experts and gate co-adapt: the gate learns to send inputs to the expert that handles them best, while each expert learns to specialize on the inputs it receives most often. This joint training is what distinguishes ME from stacking.



Hierarchical ME. ME can be extended to a **multi-level** tree structure where each node is itself a Mixture of Experts. The gating network at each level routes inputs to sub-mixtures rather than directly to experts, allowing finer-grained specialization.

ME vs. Stacking

	Stacking	**Mixture of Experts**
Combination weights	Fixed (w_i same for all x)	Adaptive ($g_k(x)$ depends on x)
Training order	Sequential (base then meta)	**Simultaneous**
Expert requirement	Good globally	Good locally only
Meta-model input	Base model outputs only	Outputs **and** x
Specialization	None enforced	Explicit via gating

Chapter 5

Support Vector Machines

SVMs approach **two-class classification** by finding a **hyperplane** that separates the classes in feature space. The core idea: if you can draw a dividing surface between classes, classify new points by which side they fall on.

When perfect separation is impossible, two strategies:

- **Soften** the notion of separation: allow some misclassifications via a **soft margin**
- **Enlarge** the feature space via transformations so that separation becomes possible in the higher-dimensional space

SVMs are fundamentally geometric: rather than modeling class probabilities, they look for the boundary that best divides the two classes. The power comes from the kernel trick, which lets you find complex nonlinear boundaries without ever explicitly computing the high-dimensional feature space.

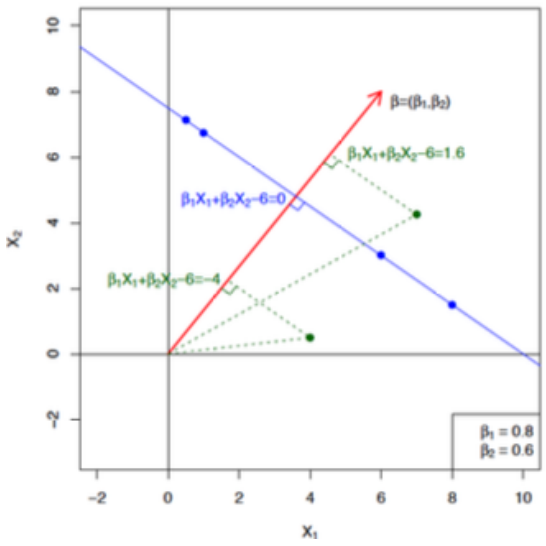
5.1 Hyperplanes & Separating Hyperplanes

⋆ **Hyperplane** in p dimensions is a flat affine subspace of dimension $p-1$. Defined by:

$$\boxed{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p = 0}$$

- $\beta_0 = 0 \Rightarrow$ hyperplane passes through the origin; otherwise shifted
- $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_p)$ is the **normal vector**: points orthogonal to the hyperplane surface
- In $p = 2$: a hyperplane is a line. In $p = 3$: a plane. In general: a $(p-1)$-dimensional flat surface.

Think of β as the direction the hyperplane "faces." Any point X satisfying the equation sits exactly on the surface; points not on it fall on one side or the other.



⋆ **Separating Hyperplane.** For $f(X) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$, the sign of $f(X)$ tells you which side of the hyperplane a point is on: $\boldsymbol{f(X) > 0}$ **one side**, $\boldsymbol{f(X) < 0}$ **other side**

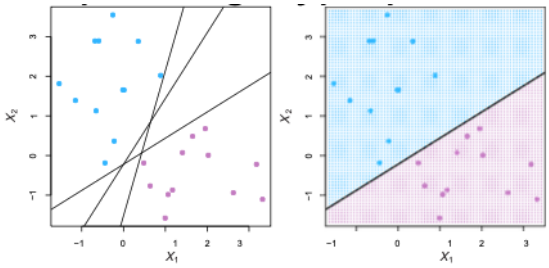
Code class labels as $Y_i \in \{+1, -1\}$.

Then $f(X) = 0$ defines a **separating hyperplane** if: $\boxed{Y_i \cdot f(X_i) > 0 \quad \forall\, i}$

i.e. every point is on the correct side: positive class where $f > 0$, negative class where $f < 0$.

Purple $Y_i = -1$ *and Blue* $Y_i = 1$

There are infinitely many separating hyperplanes for any linearly separable dataset (left figure). The right figure shows one chosen boundary that cleanly divides the two classes. The SVM picks the best one: the one with the largest margin.

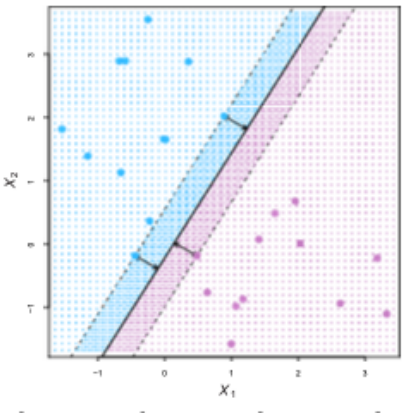


5.2 Maximal Margin Classifier

⋆ **Maximal Margin Classifier.** Among all separating hyperplanes, find the one that makes the **biggest gap** = the **margin**, between the two classes. This is a **constrained optimization problem**:

$$\boxed{\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{maximize}}\ M \quad \text{subject to} \quad \sum_{j=1}^{p} \beta_j^2 = 1, \quad y_i(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}) \geq M \quad \forall\, i}$$

The margin M is the perpendicular distance from the hyperplane to the nearest points on either side. Maximizing it gives the classifier most confident about its boundary. The problem is a convex quadratic program: solvable efficiently with standard solvers.



Role of each constraint:

- $\sum_{j=1}^{p} \beta_j^2 = 1$: **normalization constraint**: since scaling β by any nonzero k still defines the same hyperplane, this pins a unique solution for β_i's. It also gives geometric meaning to M; with this constraint, the perpendicular distance from observation i to the hyperplane is exactly: $\boxed{y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip})}$

- $y_i(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}) \geq M$: **margin constraint**: requires every observation to be on the **correct side** of the hyperplane, with a cushion of at least M. Simply being on the correct side would require > 0; requiring $\geq M > 0$ adds the cushion.

The margin constraint is strictly stronger than just "correct side." It says: not only must $y_i \cdot f(x_i) > 0$, it must be at least M; so the classifier is confident by a margin, not just barely right. Maximizing M pushes the boundary as far as possible from both classes simultaneously.

⋆ **Why the hard margin fails.** Two problems with the maximal margin classifier:

- **Non-separable data:** no separating hyperplane exists at all: classes overlap in feature space. This is the common case unless $N < p$.
- **Noisy data:** data may be separable, but a single outlier forces the margin to collapse and the boundary to contort: extremely high variance.

The maximal margin classifier is brittle: it demands every point be on the correct side with a cushion.

One misplaced point can destroy the solution. The fix is to allow violations: trade a little purity for a lot of stability.

5.3 Soft Margin and the Support Vector Classifier

⋆ **Support Vector Classifier:** Relax the hard margin by introducing **slack variables** $\varepsilon_i \geq 0$, one per observation.

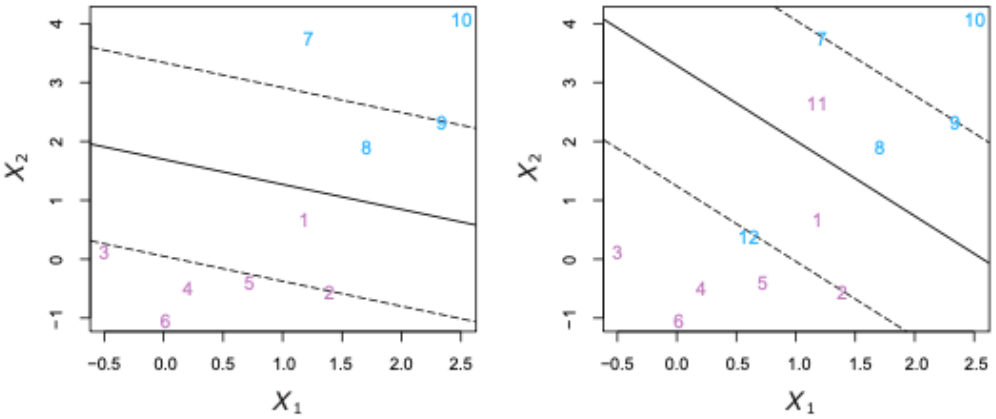
The optimization becomes: $\boxed{\underset{\beta_0,\ldots,\beta_p,\varepsilon_1,\ldots,\varepsilon_n}{\text{maximize}} \; M \quad \text{subject to} \quad \sum_{j=1}^{p} \beta_j^2 = 1,}$

$$\boxed{y_i(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}) \geq M(1 - \varepsilon_i)} \qquad \boxed{\varepsilon_i \geq 0, \quad \sum_{i=1}^{n} \varepsilon_i \leq C}$$

- $\varepsilon_i \geq 0$: **slack variable** for observation i: measures how far the point has violated the margin. $\varepsilon_i = 0$: on correct side of margin. $\varepsilon_i > 0$: violated the margin. $\varepsilon_i > 1$: on the **wrong side of the hyperplane (misclassified).**
- $M(1 - \varepsilon_i)$: the effective cushion for point i. When $\varepsilon_i = 0$ this reduces to the hard margin $\geq M$. As ε_i grows, the required cushion shrinks; the point is allowed to be closer to or past the boundary.
- $\sum_{i=1}^{n} \varepsilon_i \leq C$: **budget constraint**: $C \geq 0$ is the total amount of margin violation permitted across all observations. It caps the aggregate slack, not individual violations.
- C: **tuning parameter** controlling the bias-variance trade-off, chosen by cross-validation.

Think of C as a budget for mistakes. **Each observation can spend some of that budget to violate the margin ($\varepsilon_i > 0$) or even cross the hyperplane ($\varepsilon_i > 1$), but the total bill across all observations cannot exceed C.** A larger budget means a wider, more tolerant margin; a smaller budget means a tighter, stricter one.

Obs. 3, 4, 5, 6 (purple): correct side of margin $\Rightarrow \varepsilon_i = 0$
Obs. 2 (purple): on the margin $\Rightarrow \varepsilon_i = 0$
Obs. 1 (purple): wrong side of margin, correct side of hyperplane $\Rightarrow 0 < \varepsilon_i < 1$
Obs. 11, 12 (right panel): wrong side of hyperplane $\Rightarrow \varepsilon_i > 1$ (misclassified)



Slack variable ε_i - what it measures. $\varepsilon_i \geq 0$ encodes the position of observation i relative to both the margin and the hyperplane:

- $\varepsilon_i = 0$: observation is on the **correct side** of the margin, no violation
- $0 < \varepsilon_i \leq 1$: observation is on the **wrong side of the margin** but still on the correct side of the hyperplane: **margin violated**, not misclassified
- $\varepsilon_i > 1$: observation is on the **wrong side of the hyperplane**: **misclassified**

Think of ε_i as a continuous "how wrong" score. Zero means perfect. Crossing the margin line costs $\varepsilon_i \in (0, 1]$. Crossing the decision boundary itself costs $\varepsilon_i > 1$. The budget C caps the total cost across all observations.

⋆ **Role of C - the budget parameter.** $C \geq 0$ bounds $\sum_{i=1}^{n} \varepsilon_i$ s.t. $\sum_{i=1}^{n} \varepsilon_i \leq C$ and thus controls the **number and severity** of violations tolerated:

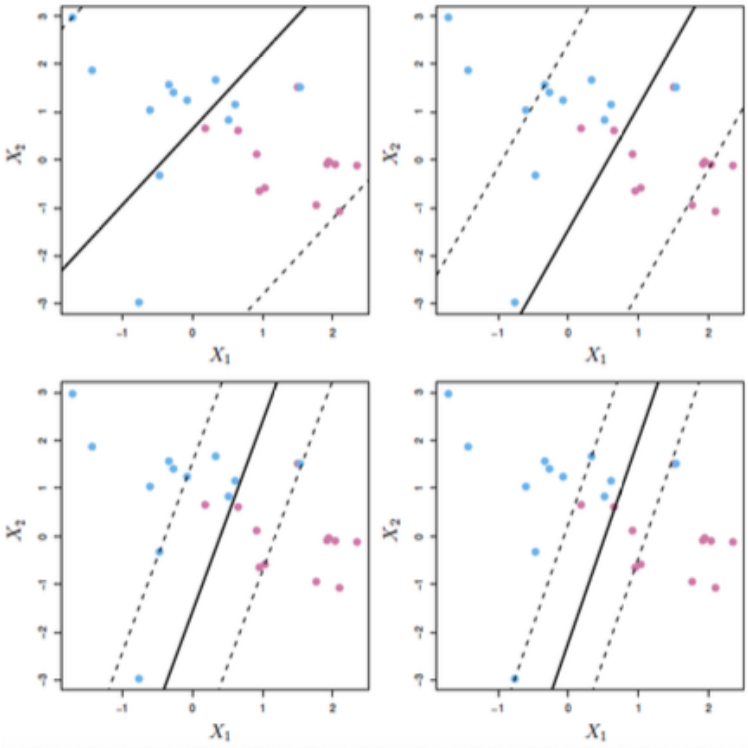
- $C = 0 \Rightarrow$ all $\varepsilon_i = 0 \Rightarrow$ no violations allowed $\Rightarrow$ reduces to the **maximal margin classifier**
- $C > 0 \Rightarrow$ at most $\lfloor C \rfloor$ observations can be misclassified (since each misclassified point contributes $\varepsilon_i > 1$, so the sum exceeds C with more than C such points)
- $C \uparrow \Rightarrow$ more violations tolerated $\Rightarrow$ **margin widens** $\Rightarrow$ more points become support vectors
- $C \downarrow \Rightarrow$ fewer violations tolerated $\Rightarrow$ **margin narrows** $\Rightarrow$ fewer support vectors, tighter fit

C is the SVM's regularization knob. It is chosen by cross-validation and directly controls the bias-variance trade-off: exactly like λ in Ridge or Lasso.

C and the bias-variance trade-off:

- **Small C:** narrow margin, rarely violated; classifier hugs the training data tightly. **Low bias, high variance.** Sensitive to individual points.
- **Large C:** wide margin, many violations tolerated; classifier fits less hard. **High bias, low variance.** More robust, more generalizable.

$$\boxed{C \uparrow \Rightarrow \text{ wider margin } \Rightarrow \text{ more bias, less variance}}$$



The four panels show the same dataset with decreasing C (top-left to bottom-right).
The top-left panel has the largest C: the widest margin with many points inside it.
As C shrinks the margin tightens, fewer points violate it, and the boundary becomes more sensitive to individual

⋆ **Support Vectors.** are the observations that lie directly on the margin, or on the wrong side of the margin for their class ($\varepsilon_i > 0$). Formally:

$$\boxed{\text{support vector} \iff \varepsilon_i > 0 \quad \text{(on or past the margin boundary)}}$$

- Only support vectors affect the position of the hyperplane; observations strictly on the correct side of the margin ($\varepsilon_i = 0$) have **zero influence**
- Moving a non-support-vector point anywhere within the correct side of the margin leaves the classifier completely unchanged
- As C increases, more observations become support vectors (wider margin, more violations allowed)

The decision rule depends on only a small subset of training points: the ones closest to or past the boundary. Points far from the hyperplane are entirely irrelevant. **This makes SVMs robust to outliers that are far away, but sensitive to outliers near the boundary.**

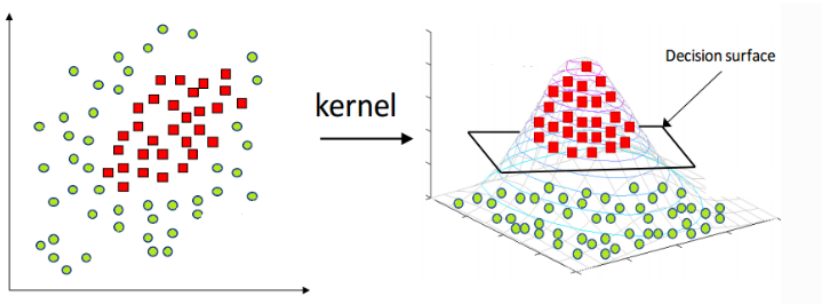
Comparison:

- **SVC:** decision rule depends only on **support vectors**: a potentially small subset of training observations. Points far from the hyperplane have zero influence.
- **LDA:** decision rule depends on the **mean of all observations** within each class and the **within-class covariance matrix** computed from all observations: every single point contributes equally.
- **Logistic regression:** like SVC, has low sensitivity to observations far from the decision boundary: more similar to SVC than LDA in this respect.

The contrast is fundamental: LDA is a global method: every point shapes the boundary. SVC is a local method: only the points near the boundary matter. This is why SVC is more robust than LDA when the class distributions are non-Gaussian or when there are outliers far from the boundary.

5.4 Feature Expansion & Nonlinear Boundaries

When the linear boundary fails. Sometimes no linear boundary separates the classes, regardless of C. The SVC is fundamentally limited to **linear decision boundaries** in the original feature space. **Two solutions: soften separation (SVC) or enlarge the feature space**.



⋆ **Feature Expansion.** Enlarge the predictor space from p dimensions to $M > p$ dimensions by adding **transformations** of the original features: powers, cross-products, etc.:

$$X_1, X_2 \longrightarrow X_1, X_2, X_1^2, X_1^3, X_1X_2, X_1X_2^2, \ldots$$

Then fit a standard SVC in the enlarged M-dimensional space. The boundary is **linear in the enlarged space** but **nonlinear in the original space**: it curves to fit class structure that no straight line could capture.

The key insight: a straight hyperplane in a higher-dimensional space looks curved when projected back down. Squaring a feature lets the boundary bend; adding cross-products lets it twist. The SVC does all its usual work: it just sees more features.

5.4.1 FE via Polynomials

Quadratic expansion example: Replace (X_1, X_2) with $(X_1,\ X_2,\ X_1^2,\ X_2^2,\ X_1X_2)$: going from 2 to 5 features. The decision boundary becomes:

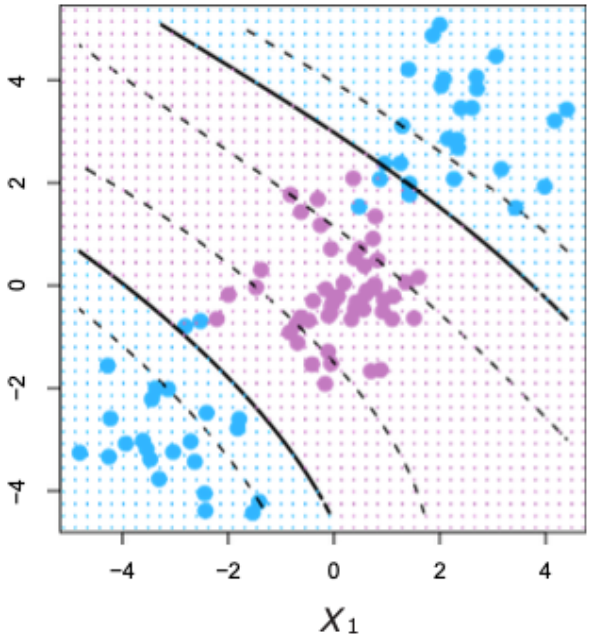
$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1^2 + \beta_4 X_2^2 + \beta_5 X_1 X_2 = 0$$

This is a **quadratic conic section** in the original space: an ellipse, parabola, or hyperbola; not a straight line.

Cubic expansion example: Expand (X_1, X_2) to 9 features using cubic polynomials:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1^2 + \beta_4 X_2^2 + \beta_5 X_1 X_2 + \beta_6 X_1^3 + \beta_7 X_2^3 + \beta_8 X_1 X_2^2 + \beta_9 X_1^2 X_2 = 0$$

The figure shows the cubic expansion boundary in the original 2D space: the **solid lines are the decision boundary** $f(x) = 0$ and the **dashed lines are the margin boundaries** $f(x) = \pm M$.
The boundary curves: it is nonlinear in (X_1, X_2), even though it was a flat hyperplane in the 9-dimensional expanded space. Blue points (class $+1$) are separated from purple points (class -1) by the curved boundary, something no straight line could achieve.



Feature expansion works but has a serious problem: **with p original features and degree-d polynomials, the expanded space grows as $O(p^d)$: exponentially.** For $p = 100$ and $d = 3$ this is millions of features, making computation intractable. This motivates the kernel trick, which achieves the same effect without ever explicitly computing the expanded features.

5.4.2 Inner Products

⋆ **Why inner products:** SVC can be re-expressed entirely in terms of **inner products** between observations.

The **inner product** between two vectors $x_i, x_{i'}$ is: $\boxed{\langle x_i, x_{i'} \rangle = \sum_{j=1}^{p} x_{ij}\, x_{i'j}}$

The linear SVC classifier function can be written as: $\boxed{f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x_i \rangle, \quad s.t. \quad \sum_{i=1}^{n} \alpha_i = 0}$

- α_i: one parameter per training observation (n total). To evaluate $f(x)$ at a new point, compute its inner product with **every** training point x_i
- β_0: intercept, computed using any support vector x_j with label y_j: $\boxed{\beta_0 = \frac{1}{y_j} - \sum_{i=1}^{n} \alpha_i \langle x_i, x_j \rangle}$

- **Key sparsity: $\alpha_i \neq 0$ only for support vectors**. All other $\alpha_i = 0$. So the sum collapses to only $|\mathcal{S}|$ terms where $\mathcal{S}$ is the support set:

$$\boxed{f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i \langle x, x_i \rangle, \qquad \beta_0 = \frac{1}{y_j} - \sum_{i \in \mathcal{S}} \hat{\alpha}_i \langle x_i, x_j \rangle}$$

This representation shows that the **classifier only depends on inner products: dot products between pairs of points**. Non-support vectors drop out entirely since their $\alpha_i = 0$.
To fit the model, all we need are the $\binom{n}{2}$ inner products between all training pairs; not the raw features themselves. This is the key that unlocks the kernel trick.

⋆ **Extending to nonlinear feature spaces.** To use an expanded feature space, apply a **transformation $u = \varphi(x)$ where $\varphi : \mathbb{R}^p \to \mathbb{R}^M$ with $M > p$** (possibly $M = \infty$).

The classifier in the new space becomes: $\boxed{f_1(x) = f(\varphi(x)) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i \langle \varphi(x), \varphi(x_i) \rangle}$

The inner products $\langle x, x_i \rangle$ are simply replaced by $\langle \varphi(x), \varphi(x_i) \rangle$: inner products in the **expanded** space.

Computing $\varphi(x)$ explicitly for high-dimensional expansions is expensive or impossible (e.g. cubic polynomials in $p = 100$ features gives $M \approx 10^6$). But notice: **we only ever need $\langle \varphi(x), \varphi(x_i) \rangle$, never $\varphi(x)$ itself.** This is where kernels come in.

Inner product: geometric interpretation. The inner product between two vectors u, v has a clean geometric meaning:

$$\boxed{\langle u, v \rangle = \|u\| \, \|v\| \cos\theta}$$

where θ is the angle between them. This means the inner product is a **similarity measure**:

- $\theta = 0$ (parallel, same direction) $\Rightarrow \cos\theta = 1 \Rightarrow \langle u, v \rangle = \|u\|\|v\|$: maximally similar
- $\theta = 90$ř (perpendicular) $\Rightarrow \cos\theta = 0 \Rightarrow \langle u, v \rangle = 0$: completely dissimilar
- $\theta = 180$ř (opposite directions) $\Rightarrow \cos\theta = -1 \Rightarrow \langle u, v \rangle = -\|u\|\|v\|$: maximally dissimilar

This connects directly to SVMs: the classifier $f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i \langle x, x_i \rangle$ is computing how similar the new point x is to each support vector x_i. A new point gets classified positive if it is geometrically close (small angle) to the positive support vectors and far (large angle) from the negative ones. Kernels generalize this: replacing $\langle x, x_i \rangle$ with $K(x, x_i)$ is still measuring similarity, just in a richer, potentially infinite-dimensional space.

5.5 Kernel

Kernel function takes two points in the original space and returns a single number measuring their similarity in some higher-dimensional space; without ever going there.

Kernel function: formal definition. A function $K : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}$ is a **kernel** if it can be written as an inner product in some (possibly infinite-dimensional) feature space:

$$K(x_i,\, x_{i'}) = \langle \varphi(x_i),\, \varphi(x_{i'}) \rangle = \sum_j \varphi_j(x_i)\, \varphi_j(x_{i'})$$

for some mapping $\varphi : \mathbb{R}^p \to \mathcal{H}$. Kernels **generalize the inner product**: they measure similarity between two points in the original space, implicitly computing $\langle \varphi(x_i), \varphi(x_{i'}) \rangle$ without ever constructing $\varphi(x)$ explicitly.

It computes the inner product (angle/similarity) between two points as if they had been mapped to a higher-dimensional space, without actually mapping them there.

⋆ **The Kernel Trick** A **kernel** $K(x_i, x_{i'})$ is a function that quantifies the **similarity** of two observations. It can be shown that certain kernels can be represented as inner products in some (possibly infinite-dimensional) feature space:

$$\boxed{K(x_i, x_{i'}) = \langle \varphi(x_i), \varphi(x_{i'}) \rangle}$$

The trick: **replace every inner product in the classifier with** K; without ever computing $\varphi(x)$:

$$\boxed{f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i\, K(x, x_i)}$$

- $K(x, x_i)$: kernel evaluation between new point x and support vector x_i; measures how similar they are in the (implicit) expanded space
- We get all the benefits of working in the high-dimensional space $\varphi(x)$ **without ever computing it**
- **Linear kernel**: $K(x_i, x_{i'}) = \langle x_i, x_{i'} \rangle$; this is just the standard SVC, where $\varphi(x) = x$

The kernel trick is the central insight of SVMs. **Instead of explicitly mapping data into a high-dimensional space and computing inner products there, we compute a single kernel function that implicitly does both steps at once.** For some kernels (like the RBF kernel), the implicit feature space is infinite-dimensional: something we could never compute directly.

⋆ **Mercer's Condition.** A function $K(x_i, x_j)$ is a valid **kernel**; i.e. it can be written as an inner product in some feature space:

$$K(x_i, x_j) = [\varphi(x_i)]^\top [\varphi(x_j)]$$

if and only if it satisfies **Mercer's condition**: for all finite sequences $(x_1, x_2, \ldots, x_n)$ and all real numbers $(c_1, c_2, \ldots, c_n)$:

$$\boxed{\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) \geq 0}$$

Mercer's condition is the mathematical guarantee that K behaves like a real inner product, specifically that the kernel matrix is positive semi-definite.

Intuitively: if you compute all pairwise similarities between n points using K, the resulting $n \times n$ matrix must have no negative eigenvalues. This ensures there exists some valid φ such that K is truly computing inner products in that space, not just any arbitrary function of two points.

5.5.1 Polynomial Kernel Method

The **polynomial kernel** of degree d is: $\boxed{K(x_i, x_{i'}) = (1 + \langle x_i, x_{i'} \rangle)^d} = \sum_{k=0}^{d} \binom{d}{k} \langle x_i, x_{i'} \rangle^k$
It fits directly into the kernel framework:

- Satisfies **Mercer's condition** $\Rightarrow$ valid kernel, positive semi-definite kernel matrix
- There exists a φ such that $K(x_i, x_{i'}) = \langle \varphi(x_i), \varphi(x_{i'}) \rangle$; that φ corresponds to **all polynomial terms up to degree** d, i.e. the explicit polynomial feature expansion

$\star$ **Polynomial Kernel.** The **polynomial kernel** of degree d is: $\boxed{K(x_i,\, x_{i'}) = \left(1 + \sum_{j=1}^{p} x_{ij}\, x_{i'j}\right)^d}$

This single function implicitly computes the inner product in the space of **all polynomial basis functions up to degree** d; without ever constructing them. The SVM classifier becomes:

$$\boxed{f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i\, K(x, x_i) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i\, (1 + \langle x, x_i \rangle)^d}$$

Instead of explicitly building the $O(p^d)$-dimensional expanded feature space, the kernel evaluates a cheap scalar function. You get the expressive power of high-degree polynomial features at the cost of computing n dot products.

Example: $p = 2,\ d = 2$**.** Let $u = (u_1, u_2)^\top$ and $v = (v_1, v_2)^\top$. Expanding the kernel directly:

$$\begin{aligned} K(u, v) &= (1 + \langle u,\, v \rangle)^2 = (1 + u_1 v_1 + u_2 v_2)^2 = 1 + (u_1 v_1 + u_2 v_2)^2 + 2(u_1 v_1 + u_2 v_2) \\ &= 1 + u_1^2 v_1^2 + u_2^2 v_2^2 + 2 u_1 u_2 v_1 v_2 + 2 u_1 v_1 + 2 u_2 v_2 \end{aligned}$$

This is identical to the inner product $\langle \varphi(u),\, \varphi(v) \rangle$ where the **implicit feature map** is:

$$\varphi(u) = \begin{bmatrix} 1 \\ \sqrt{2}\, u_1 \\ \sqrt{2}\, u_2 \\ \sqrt{2}\, u_1 u_2 \\ u_1^2 \\ u_2^2 \end{bmatrix}, \qquad \varphi(v) = \begin{bmatrix} 1 \\ \sqrt{2}\, v_1 \\ \sqrt{2}\, v_2 \\ \sqrt{2}\, v_1 v_2 \\ v_1^2 \\ v_2^2 \end{bmatrix} \quad \Rightarrow \quad K(u, v) = \langle \varphi(u),\, \varphi(v) \rangle$$

$\varphi(u) = \left(1, ; \sqrt{2}, u_1, ; \sqrt{2}, u_2, ; \sqrt{2}, u_1 u_2, ; u_1^2, ; u_2^2\right)$

The $\sqrt{2}$ factors appear from the binomial expansion cross-terms ($2u_1 v_1 = \sqrt{2}u_1 \cdot \sqrt{2} v_1$) to make the inner product work out exactly. So with $p = 2$ original features and $d = 2$, the kernel implicitly operates in a **6-dimensional** space; and does it with a single scalar computation instead of explicitly constructing 6 features per point.

5.5.2 Radial Basis Function (RBF) Kernel

⋆ **Radial Kernel:** The **RBF kernel** (also called the Gaussian or radial kernel) is defined as:

$$\boxed{K(x_i,\, x_{i'}) = \exp\left(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2\right) = \exp\left(-\gamma \left\|x_i - x_{i'}\right\|_2^2\right)}$$

where $\gamma > 0$ is a tuning parameter.
The exponent is the **squared Euclidean distance** between x_i and $x_{i'}$. As with all kernels, this implicitly computes $\langle \varphi(x_i), \varphi(x_{i'}) \rangle$ in some feature space; but here the implicit φ maps into an **infinite-dimensional** space. The classifier takes the standard kernel form: $f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i\, K(x,\, x_i)$

Locality of the RBF kernel.

- If x^* is **far** from x_i in Euclidean distance $\Rightarrow$ $\|x^* - x_i\|_2^2$ is large $\Rightarrow$ exponent is very negative $\Rightarrow$ $K(x^*, x_i) \approx 0 \Rightarrow$ x_i contributes **nothing** to $f(x^*)$

- If x^* is **close** to $x_i \Rightarrow K(x^*, x_i) \approx 1 \Rightarrow$ x_i has **full influence**

The RBF kernel is therefore **purely local**: classification of x^* is determined only by its nearby support vectors. Distant training points are automatically silenced. *(Similar to KNN: influence decays with distance.)*

Role of γ.

- $\gamma \uparrow$: distance penalty sharpens $\Rightarrow$ influence decays faster $\Rightarrow$ **narrower neighborhoods, higher variance**, more flexible boundary
- $\gamma \downarrow$: influence decays slowly $\Rightarrow$ **smoother, more global** boundary, higher bias

Choosing γ. γ is selected by **cross-validation**, jointly with C via a **2D grid search** over (C, γ) pairs; they cannot be tuned independently since they interact directly in controlling the bias-variance trade-off.

A common **initialization heuristic** before grid search: $\boxed{\gamma_0 = \frac{1}{p}}$ or $\boxed{\gamma_0 = \frac{1}{p \cdot \mathrm{Var}(X)}}$ where p is the number of features. This scales the kernel to the spread of the data so no single feature dominates the distance computation. Cross-validation then searches around this starting point.

Why the RBF kernel is infinite-dimensional. Expand the RBF kernel using the Taylor series $e^z = \sum_{n=0}^{\infty} \frac{z^n}{n!}$:

$$K(x_i, x_{i'}) = \exp\left(-\gamma \|x_i - x_{i'}\|^2\right) = \exp(-\gamma \|x_i\|^2) \cdot \exp(-\gamma \|x_{i'}\|^2) \cdot \exp(2\gamma \langle x_i, x_{i'} \rangle)$$

Expanding the last factor:

$$\exp(2\gamma \langle x_i, x_{i'} \rangle) = \sum_{n=0}^{\infty} \frac{(2\gamma)^n}{n!} \langle x_i,\, x_{i'} \rangle^n = 1 + 2\gamma \langle x_i, x_{i'} \rangle + \frac{(2\gamma)^2}{2!} \langle x_i, x_{i'} \rangle^2 + \cdots$$

Each term $\langle x_i, x_{i'} \rangle^n$ corresponds to a **degree-n polynomial kernel**; so the RBF kernel implicitly sums over polynomial feature spaces of **every degree** $n = 0, 1, 2, \ldots, \infty$ simultaneously. The implicit feature map $\varphi(x)$ therefore lives in an **infinite-dimensional** space: it can never be written down explicitly. The parameter γ controls how fast the higher-degree terms are down-weighted, not the dimensionality of the space.

5.6 Multi-Class and Multi-Label SVMs

Multiclass classification. A classification task with $K > 2$ classes where each sample is assigned to **exactly one** label.

⋆ **Multilabel classification.** Each sample is assigned a **set of labels** simultaneously: the labels are not mutually exclusive. A document can be about politics *and* finance at the same time; an image can contain a cat *and* a dog.

Multiclass forces a single winner among K options; multilabel allows any subset of labels to fire at once. Standard binary classifiers handle neither directly, both require adaptation.

5.6.1 One-vs-All and One-vs-One

⋆ **One-vs-All (OVA).** Fit K separate binary SVMs, each pitting class k against all others:

$$\hat{f}_k(x), \quad k = 1, \ldots, K$$

Classify x^* to the class with the **largest decision function value**: $\boxed{\hat{y} = \arg\max_k \hat{f}_k(x^*)}$

⋆ **One-vs-One (OVO).** Fit all $\binom{K}{2}$ pairwise binary classifiers $\hat{f}_{kl}(x)$ between every pair of classes (k, l). Classify x^* to the class that **wins the most pairwise duels**: majority vote across all $\binom{K}{2}$ classifiers.

- **OVO** is preferred when K is not too large; each pairwise classifier trains on a smaller, cleaner subset of the data (only two classes)
- **OVA** forces each classifier to handle highly imbalanced problems (1 class vs. $K - 1$).

OVA is cheaper to train (K classifiers vs. $\binom{K}{2}$) but each sub-problem is harder. OVO trains more classifiers but each one is simpler. For SVMs specifically, OVO tends to work better in practice.

5.6.2 Problem Transformation: Binary Relevance, Classifier Chains, Label Powerset

Three strategies to handle multilabel output; all reduce it to problems standard classifiers can solve.

⋆ **Binary Relevance (BR).** Decompose the multilabel problem into L independent binary classification problems: one per label. Train a separate classifier for each label Y_ℓ using the full feature set X. At test time, run all L classifiers independently and collect whichever labels fire.

$$\boxed{(X, Y_1),\ (X, Y_2),\ \ldots,\ (X, Y_L) \quad \longrightarrow \quad L \text{ independent classifiers}}$$

Any classifier (SVM, logistic regression, Naïve Bayes) can be plugged in. Each label may produce a binary or multi-class sub-problem depending on the label structure.

BR is the simplest approach and easy to implement, but it treats labels as completely independent: it ignores any correlation between them. If knowing $Y_1 = 1$ makes $Y_2 = 1$ more likely, BR throws away that information entirely.

⋆ **Classifier Chains (CC).** Like binary relevance, but chain the classifiers so each one **conditions on the predictions of all previous classifiers** in the chain. Classifier 1 sees only X; Classifier 2 sees X and $\hat{Y}_1$; Classifier ℓ sees $X, \hat{Y}_1, \ldots, \hat{Y}_{\ell-1}$:

$$\boxed{\hat{Y}_1 = f_1(X), \quad \hat{Y}_2 = f_2(X, \hat{Y}_1), \quad \ldots \quad \hat{Y}_L = f_L(X, \hat{Y}_1, \ldots, \hat{Y}_{L-1})}$$

CC preserves label correlations that BR discards; if label 1 is a good predictor of label 2, that signal propagates forward through the chain. The cost is that early errors compound: a wrong $\hat{Y}_1$ corrupts all downstream classifiers.

⋆ **Label Powerset (LP).** Map every unique combination of labels observed in training to a single unique class ID, then train **one multi-class classifier** on the transformed single-label problem.

$$(Y_1, Y_2, \ldots, Y_L) \longrightarrow \text{one composite class} \in \{1, 2, \ldots, M\}$$

where M is the number of distinct label combinations seen in training. At test time, the classifier predicts a composite class, which is decoded back to the full label vector.

LP fully captures label co-occurrence by treating each combination as an atomic class. The weakness: it only predicts combinations seen in training: unseen label combinations at test time cannot be predicted. The number of classes can also explode exponentially with L.

Label Powerset: concrete example with labels $\{A, O, K, L\}$. Scan training data and assign a unique class ID to every distinct label combination:

X	A	O	K	L	Class ID
x_1	1	1	0	0	1
x_2	0	0	1	0	2
x_3	1	1	0	0	1
x_4	0	1	1	0	3

The class ID is just a name for each unique fingerprint: x_1 and x_3 share the same combination $(1, 1, 0, 0)$ so they get the same ID. The decoding table is stored:

$\boxed{1 \leftrightarrow (A{=}1,\ O{=}1,\ K{=}0,\ L{=}0)}$, $\boxed{2 \leftrightarrow (A{=}0,\ O{=}0,\ K{=}1,\ L{=}0)}$, $\boxed{3 \leftrightarrow (A{=}0,\ O{=}1,\ K{=}1,\ L{=}0)}$

Train a standard multiclass SVM on $(X, \text{Class ID})$: the model learns $X \to \{1, 2, 3\}$ and never sees A, O, K, L individually. At test time:

$$x^* \xrightarrow{\text{SVM}} \text{Class } 1 \xrightarrow{\text{decode}} (A{=}1,\ O{=}1,\ K{=}0,\ L{=}0)$$

The label structure is entirely hidden inside the ID encoding during training. Prediction is instant: one classifier call followed by a table lookup. The weakness is that any label combination not seen in training has no ID and cannot be predicted.

5.6.3 Adapted Algorithms: ML-kNN and Ensemble Methods

⋆ **ML-kNN (Multilabel k-Nearest Neighbours).** Applies kNN **independently for each label** ℓ: find the k nearest neighbors to the test point, count how many carry label ℓ, and use a **prior probability** (estimated from training) to make the final 0/1 decision via a maximum a posteriori rule.

What differentiates ML-kNN from plain binary relevance with kNN is the incorporation of prior probabilities: it does not just majority-vote the neighbors, it weighs the neighborhood evidence against the overall prevalence of each label. ML-kNN can also produce a label ranking (confidence scores per label) rather than just a binary decision.

⋆ **Ensemble Algorithms.** Standard ensemble methods can be extended to the multilabel setting. The main examples are **AdaBoost.MH** (minimizes Hamming loss: treats each label independently across boosting rounds) and **AdaBoost.MR** (minimizes ranking loss: focuses on ranking labels correctly relative to each other for each example).

AdaBoost.MH is closer in spirit to binary relevance; AdaBoost.MR is more sensitive to the relative ordering of labels and is appropriate when the goal is label ranking rather than exact binary recovery.

Summary:

- **Problem Transformation** (BR, CC, LP): reduce multilabel to single-label sub-problems; any off-the-shelf classifier applies. Trade-off between simplicity (BR), label correlation (CC), and combinatorial expressiveness (LP).
- **Adapted Algorithms** (ML-kNN): modify an existing algorithm to output label sets directly; uses prior information to improve label decisions.
- **Ensemble approaches** (AdaBoost.MH/MR): combine multiple classifiers; choice of loss function determines whether independence or label ordering is prioritized.

5.6.4 Evaluation Metrics for Multi-Label Classification

⋆ **Exact Match (Subset Accuracy).** Standard accuracy cannot be used for multilabel output; partial label matches are neither fully right nor fully wrong. The **exact match** metric requires the **entire predicted label vector** to match the true label vector:

$$\boxed{\text{Exact Match} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{1}\left(\hat{Y}_i = Y_i\right)}$$

where $\hat{Y}_i = (\hat{y}_{i1}, \ldots, \hat{y}_{iL})$ and $Y_i = (y_{i1}, \ldots, y_{iL})$ are the full predicted and true label vectors. This is the strictest possible metric: one wrong label in a vector of L counts as a complete failure.

⋆ **Hamming Loss.** A softer metric that counts the **fraction of individual label positions** that are wrong across all N samples and L labels:

$$\boxed{\text{Hamming Loss} = \frac{1}{NL}\sum_{i=1}^{N}\sum_{j=1}^{L}\mathbf{1}(\hat{y}_{ij} \neq y_{ij})}$$

where N is the number of samples and L is the number of labels. Each of the NL individual label predictions contributes equally: getting label j wrong for sample i costs $\frac{1}{NL}$ regardless of the other labels.

Exact match is all-or-nothing and very harsh: a prediction with $L-1$ correct labels scores the same as one with zero correct. Hamming loss gives partial credit and is more informative when L is large. The choice of metric should reflect the application: if all labels must be jointly correct, use exact match; if label-level accuracy matters independently, use Hamming loss.

5.7 SVC vs. Logistic Regression

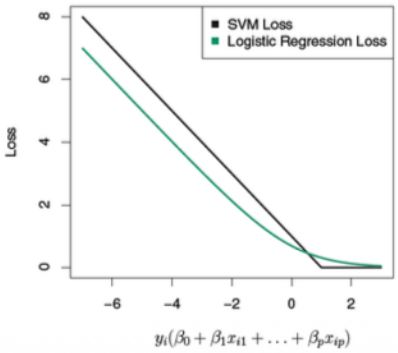
5.7.1 Hinge Loss and the Loss + Penalty Form

⋆ **SVC as loss plus penalty.** With $f(x) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$, the SVC optimization can be rephrased as:

$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{minimize}} \left\{ \sum_{i=1}^{n} \max\Big[0,\, 1 - y_i f(x_i)\Big] + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}$$

This has the standard **loss + penalty** form. The loss term $\max[0, 1 - y_i f(x_i)]$ is the **hinge loss**: it is zero when the point is correctly classified with sufficient margin, and grows linearly as the point moves toward or past the boundary. The penalty $\lambda \sum_j \beta_j^2$ is a **ridge (L2) penalty**, controlling model complexity exactly as in Ridge regression.

The hinge loss is the SVM's alternative to logistic regression's negative log-likelihood. Both penalize wrong predictions, but hinge loss is exactly zero for correctly classified points beyond the margin: it ignores them entirely. Logistic loss is never exactly zero, always pulling the boundary slightly toward every point.



⋆ **Logistic regression in $\{-1, +1\}$ encoding.** When labels are coded $y_i \in \{-1, +1\}$ (instead of $\{0, 1\}$), logistic regression becomes:

$$P(Y = y_i \mid X = x_i) = \frac{1}{1 + \exp(-y_i f(x_i))}$$

and the **negative log-likelihood loss** (what LR minimizes) is:

$$-\log P(Y = y_i \mid X = x_i) = \log\Big[1 + \exp(-y_i f(x_i))\Big]$$

Both losses are functions of the **margin** $y_i f(x_i)$: when the margin is large and positive (confident correct prediction) both losses are small; when it is negative (misclassification) both losses grow. The key difference is shape: hinge loss has a kink at $y_i f(x_i) = 1$ and is exactly flat beyond it; logistic loss is smooth and always positive.

⋆ **L1 regularization for SVC.** The default SVC penalty is L2 (ridge). Replacing it with an **L1 penalty** gives:

$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{minimize}} \left\{ \sum_{i=1}^{n} \max\Big[0,\, 1 - y_i f(x_i)\Big] + \lambda \sum_{j=1}^{p} |\beta_j| \right\}$$

Like Lasso, the L1 penalty drives some β_j exactly to zero: performing **variable selection** simultaneously with classification.

Note: because the hinge loss is **not differentiable** (it has a kink at $y_i f(x_i) = 1$), solution paths as a function of λ may have **jumps**. Some authors replace hinge loss with the **squared hinge loss** $\max[0, 1 - y_i f(x_i)]^2$ to restore differentiability. Different combinations of {hinge, squared hinge} $\times$ {L1, L2} penalty produce different SVM variants.

The coefficient paths for L1-SVM and L1 logistic regression are strikingly similar in practice: both select the same features and shrink coefficients at similar rates, especially when $p \gg n$.

5.7.2 liblinear and Class Probabilities

liblinear: loss × penalty combinations. The **liblinear** package implements linear classification by mixing two choices of **loss** with two choices of **penalty**:

Loss		Penalty	
L_1	hinge: $\max[0, 1 - y_i f(x_i)]$	L_1	$\sum_j \lvert\beta_j\rvert$
L_2	squared hinge: $\max[0, 1 - y_i f(x_i)]^2$	L_2	$\sum_j \beta_j^2$ (Elastic net?)

Also supports **CE** (cross-entropy) loss (i.e. logistic regression) with the same penalty options. Different combinations produce different SVM variants: L1 loss + L2 penalty is the standard SVC; L2 loss + L1 penalty adds variable selection; L2 loss + L2 penalty is fully smooth and differentiable.

Extracting class probabilities from SVM. The SVM outputs a raw **decision score** $f(x) = \beta_0 + \boldsymbol{\beta}^\top x$; not a probability. However, since SVM and logistic regression produce similar $f(x)$, one can plug the SVM score directly into the **logistic sigmoid** to obtain an approximate conditional probability. With labels $y_i \in \{-1, +1\}$, logistic regression gives:

$$P(Y = y_i \mid X = x_i) = \frac{1}{1 + \exp(-y_i f(x_i))}$$

Substituting the SVM's $f(x)$ in place of the logistic regression's $f(x)$:

$$\boxed{\hat{P}(Y = +1 \mid x) = \frac{1}{1 + \exp(-f(x))}}$$

The distance of $f(x)$ from zero encodes confidence:

- $f(x) \gg 0 \Rightarrow \hat{P}(Y = +1 \mid x) \to 1$: far on the positive side, very confident
- $f(x) \ll 0 \Rightarrow \hat{P}(Y = +1 \mid x) \to 0$: far on the negative side, very confident
- $f(x) \approx 0 \Rightarrow \hat{P}(Y = +1 \mid x) \approx 0.5$: on the boundary, maximum uncertainty

Distance from the hyperplane is the SVM's natural confidence measure. The sigmoid transformation converts that geometric distance into a probability scale, exploiting the fact that SVM and logistic regression optimize nearly identical objectives and produce near-identical decision functions.

5.7.3 SVM or Logistic Regression?

- **Nearly separable classes:** **SVM** (or LDA) outperforms LR; the margin-based objective directly exploits the gap between classes.
- **Overlapping classes:** **LR with ridge penalty** and SVM give very similar results; both reduce to smooth penalized loss minimization.
- **Probability estimates needed:** use **LR**; it directly models $P(Y = 1 \mid X)$. SVM does not model probabilities, though distances from the hyperplane can be used as a proxy.
- **Nonlinear boundaries:** **kernel SVM** is the natural choice; kernels find a feature space where the boundary is linear. Kernels can also be applied to LR and Bayesian LDA but at greater computational cost. Note that kernels are philosophically more natural for SVM since LR models class probabilities, not boundaries.

- **High-dimensional settings ($p \gg n$):** **SVM** is popular; training complexity is $O(pn^2)$ for both linear and nonlinear kernels, independent of p, making it efficient even when features vastly outnumber observations.

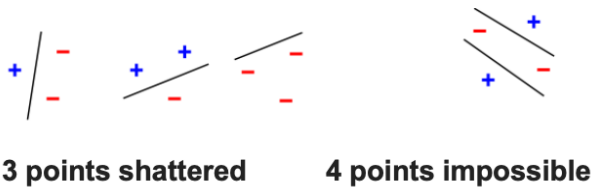
5.8 Vapnik-Chervonenkis (VC) Dimension

⋆ **VC Dimension.** The **VC dimension** is a measure of the **capacity** (complexity, expressive power, and flexibility) of the space of functions that can be learned by a classification algorithm. Formally it is the maximum number of points that can be **shattered** by the model f.

⋆ **Shattering.** A model f with parameter vector $\boldsymbol{\beta}$ is said to **shatter** a set of points $(x_1, x_2, \ldots, x_n)$ if for **every possible** $\{+1, -1\}$ label assignment to those points, there exists a $\boldsymbol{\beta}$ such that f makes **zero errors** on that labeling.
With n points there are 2^n possible label assignments; shattering requires f to perfectly separate all 2^n of them.

$$\boxed{\text{VC dim}(f) = \max\Big\{n : \exists\ (x_1, \ldots, x_n) \text{ shattered by } f\Big\}}$$

Shattering is a worst-case concept: it does not require every arrangement of n points to be shatterable, only that there exists at least one arrangement that is. The VC dimension is the largest n for which such an arrangement exists.



Example: linear classifier in $\mathbb{R}^2$. A straight line (perceptron) in 2D can shatter any 3 non-collinear points: for all $2^3 = 8$ label assignments, a separating line exists. However no arrangement of 4 points can be shattered by a line. Therefore: VC dim(linear classifier in $\mathbb{R}^2$) = 3

With 3 non-collinear points, no matter how you label them + or −, you can always draw a line to separate them. With 4 points, there always exists at least one labeling (e.g. XOR-like pattern) that no single line can separate.

⋆ **VC Dimension of affine classifiers.** For affine classifiers of the form $f(x) = \boldsymbol{\beta}^\top x + \beta_0$ with $\boldsymbol{\beta} \in \mathbb{R}^p$, which includes the linear SVC, the VC dimension is:

$$\boxed{\text{VC dim} = p + 1}$$

The intuition: in p dimensions, a hyperplane has $p + 1$ free parameters (p slopes + intercept), giving it enough flexibility to shatter exactly $p + 1$ points. Adding one more feature adds one more degree of freedom and increases VC dimension by one.

VC dimension and the bias-variance trade-off.

- **Low VC dim** ⇒ restricted model ⇒ **high bias, low variance**: cannot fit complex patterns but generalizes reliably
- **High VC dim** ⇒ very flexible model ⇒ **low bias, high variance**: can fit anything but risks memorizing the training data

This is why the RBF kernel SVM requires careful tuning of C and γ; its infinite VC dimension means it can perfectly memorize any training set if left unconstrained. Regularization artificially limits the effective complexity actually used.

VC Dimension of kernel SVMs.

- **Linear SVM** ($\boldsymbol{\beta} \in \mathbb{R}^p$): VC dim $= p + 1$: finite, grows with the number of features

- **RBF kernel SVM**: VC dim $= \infty$; the implicit feature space is infinite-dimensional, so the classifier can in principle shatter any finite set of points

An infinite VC dimension means the RBF-SVM is expressive enough to fit any training set perfectly; but this does not mean it will generalize well. The regularization parameter C and the kernel width γ jointly constrain the effective complexity actually used, preventing the model from exploiting its full infinite capacity.

5.9 Support Vector Regression (SVR)

⋆ **Support Vector Regression.** SVR extends the SVM framework to **regression** problems. Recall that least squares regression minimizes the **residual sum of squares**:

$$\min_{\beta_0,\ldots,\beta_p} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

SVR instead minimizes a different loss where **only residuals larger than some threshold $\epsilon > 0$ contribute to the loss**; small residuals inside a tube around the prediction are treated as zero error:

$$\boxed{\min_{\beta_0,\ldots,\beta_p} \sum_{i=1}^{n} \max\Big[0,\ |y_i - f(x_i)| - \epsilon\Big] + \lambda \sum_{j=1}^{p} \beta_j^2}$$

This is the **ϵ-insensitive loss**: the regression analog of the hinge loss. Predictions within ϵ of the true value cost nothing; only those outside the ϵ-tube are penalized, and penalized linearly in the excess.

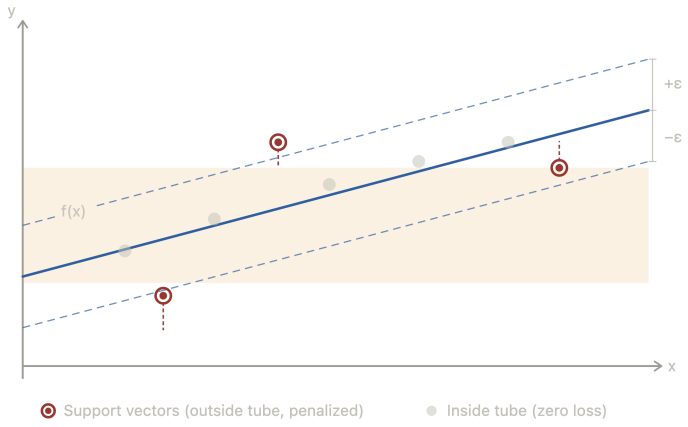
SVR extends the margin concept from classification to regression: instead of a margin separating two classes, there is an ϵ-tube around the regression function. Points inside the tube are support-free: they don't influence the fit at all. Only points outside the tube become support vectors and determine the regression line, making SVR robust to small noise and outliers within the tube.

ϵ-insensitive loss: shape. The loss function has a **dead zone** of width 2ϵ centered at zero residual:

$$L_\epsilon(y_i, f(x_i)) = \max\Big[0,\ |y_i - f(x_i)| - \epsilon\Big] = \begin{cases} 0 & \text{if } |y_i - f(x_i)| \le \epsilon \\ |y_i - f(x_i)| - \epsilon & \text{if } |y_i - f(x_i)| > \epsilon \end{cases}$$

The loss is exactly zero inside the ϵ-tube $[-\epsilon, +\epsilon]$ and grows linearly outside it; analogous to the hinge loss in classification. The **squared** version e^2 (parabola shape) penalizes large residuals more aggressively.

Support vectors in SVR. The **support vectors** are the observations that lie **outside** the ϵ-tube; i.e. those with $|y_i - f(x_i)| > \epsilon$. Points inside the tube contribute zero to the loss and have no influence on the fitted regression function. Only the outside points determine the boundary of the tube and the regression line, exactly as in SVC where only margin-violating points determine the hyperplane. The tube width ϵ is selected by **cross-validation**.


y
+ε
−ε
f(x)
x
Support vectors (outside tube, penalized)
Inside tube (zero loss)

Chapter 6

Unsupervised Learning

6.1 Unsupervised vs. Supervised Learning

⋆ **Supervised Learning.** Observe both features $X_1, X_2, \ldots, X_p$ **and** a response Y for each observation. The goal is to **predict** Y from $X_1, \ldots, X_p$: a well-defined objective with a clear performance criterion (test error, accuracy, etc.).

⋆ **Unsupervised Learning.** Observe **only** the features $X_1, X_2, \ldots, X_p$: **no response variable** Y. The goal shifts from prediction to **discovery**: find informative structure, patterns, or subgroups in the data without a labeled target to guide the search.

Two core methods covered:

- **Principal Components Analysis (PCA)**: finds a low-dimensional representation of the data that captures as much variance as possible. Used for **visualization** and as a **pre-processing step** before supervised methods.
- **Clustering**: partitions observations into **subgroups** (clusters) such that observations within a group are similar to each other and dissimilar from those in other groups.

Supervised learning has a scoreboard: test error tells you how well you're doing. Unsupervised learning has no scoreboard. You're looking for structure that is "interesting," which is inherently subjective. This makes it harder to validate and easier to overinterpret. **Why unsupervised learning is harder.**

- **No gold standard:** no response variable Y means no ground truth to evaluate against.
- **No single objective:** unlike minimizing prediction error, the goal (discover subgroups? visualize variation?) is ambiguous and domain-dependent.
- **More subjective:** results depend heavily on method choices: distance metric, number of clusters, number of components.

Practical motivation: why it matters despite being harder.

- **Unlabeled data is cheap, labeled data is expensive.** Collecting $X_1, \ldots, X_p$ from sensors or instruments is automatic; assigning labels Y typically requires costly human intervention (e.g., deciding if a movie review is positive or negative).
- Real applications: grouping breast cancer patients by gene expression, segmenting shoppers by purchase history, clustering movies by viewer ratings: all settings where Y is absent or undefined.

Unsupervised methods are important both on their own (market segmentation, anomaly detection) and as a pre-processing tool: PCA-reduced features fed into a supervised classifier often generalize better than raw high-dimensional inputs.

6.2 Clustering

⋆ **Clustering.** A broad class of techniques for finding **subgroups (clusters)** in a dataset, a **partition** of observations into distinct groups such that observations **within** a group are similar to each other and observations **between** groups are different.

Key design decision: clustering requires explicitly defining what **similarity** and **dissimilarity mean between observations.** This is **domain-specific**, there is no universal right answer. The choice of distance/similarity measure shapes the entire solution.

Unlike supervised learning where the task is fixed (predict Y), clustering forces you to make a judgment call upfront: what does "close" mean for your data? Euclidean distance, correlation, and other measures can give radically different clusterings on the same dataset.

Market segmentation. Given measurements $X_1, \ldots, X_p$ (income, occupation, location, etc.) across n people, **market segmentation** means identifying subgroups more receptive to a particular advertisement or product. The task reduces directly to clustering the n observations, no response Y is needed or available.

Two clustering methods:

- **K-means clustering**: partitions observations into a **pre-specified** number of clusters K. You must choose K before fitting.
- **Hierarchical clustering**: does not require pre-specifying K. Produces a **dendrogram**: a tree-like representation encoding the clustering for **every** possible K from 1 to n simultaneously. The user cuts the tree at the desired level of resolution.

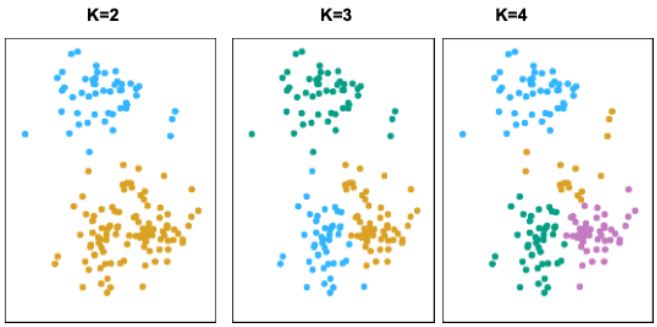
K-means is simpler and faster but demands you commit to K upfront. Hierarchical clustering defers that commitment: you get the full picture and choose K by inspecting the dendrogram.

6.3 K-Means Clustering

⋆ **K-Means Clustering.** Partition n observations into a **pre-specified** number of clusters K. Let $C_1, \ldots, C_K$ denote the sets of indices of observations in each cluster. A valid partition satisfies:

- **Coverage:** $C_1 \cup C_2 \cup \cdots \cup C_K = \{1, \ldots, n\}$: every observation belongs to at least one cluster
- **Non-overlap:** $C_k \cap C_{k'} = \emptyset$ for all $k \neq k'$: no observation belongs to more than one cluster

If observation i is in cluster k, then $i \in C_k$.



Objective: minimize within-cluster variation. A good clustering minimizes internal disagreement. Define $\text{WCV}(C_k)$ as the **within-cluster variation** of cluster C_k: how much observations inside C_k differ from each other. The K-means objective:

$$\underset{C_1,\ldots,C_K}{\text{minimize}} \left\{ \sum_{k=1}^{K} \text{WCV}(C_k) \right\}$$

Minimizing total WCV forces each cluster to be tight internally. There is no reward for separating clusters from each other, only for compressing them inward.

Euclidean WCV. Standard choice: squared Euclidean (L_2 **norm**: $\sum_{j=1}^{p}(x_{ij} - x_{i'j})^2 = \|x_i - x_{i'}\|_2^2$) , distance averaged over all pairs in the cluster:

$$\text{WCV}(C_k) = \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2$$

where $|C_k|$ is the number of observations in cluster k. The full K-means optimization problem:

$$\underset{C_1,\ldots,C_K}{\text{minimize}} \left\{ \sum_{k=1}^{K} \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2 \right\}$$

The $\frac{1}{|C_k|}$ normalization measures average pairwise spread, not total, otherwise large clusters would always dominate the objective regardless of how scattered they are. This problem is NP-hard in general (exponential in n), so an efficient greedy algorithm is used instead.

Lloyd's Algo Terms

- **Descent:** the objective function **never increases** between steps, each update moves to an equal or lower value, monotonically approaching a minimum
- **Coordinate:** a block of variables being optimized, here either the **centroid positions** $\{\bar{x}_k\}$ or the **cluster assignments** $\{C_k\}$
- **Coordinate Descent:** optimize over one coordinate block at a time while holding all others fixed, each block is solved **exactly**, guaranteeing descent

⋆ **K-Means Algorithm** or **Lloyd's Algorithm.** A **coordinate descent** procedure that alternates between two exact minimization steps, fixing assignments and optimizing centroids, then fixing centroids and optimizing assignments, guaranteed to decrease the K-means objective at every step until convergence to a local minimum.

1. **Initialize:** randomly assign each observation a label from $\{1, \ldots, K\}$.
2. **Iterate** until cluster assignments stop changing:
   (a) **Compute centroids:** for each C_k, compute the **centroid** $\bar{x}_{kj} = \frac{1}{|C_k|} \sum_{i \in C_k} x_{ij}$: the mean of feature j over all points in cluster k.
   (b) **Reassign:** assign each observation to the cluster whose centroid is closest in Euclidean distance.

Each step has a clear local role: the centroid step finds the best center given current membership; the reassignment step finds the best membership given current centers. Together they alternate minimization, a coordinate descent on the objective.

Properties of the algorithm.

Guaranteed to decrease the objective at each step. The key identity that explains why:

$$\frac{1}{|C_k|}\sum_{i,i'\in C_k}\sum_{j=1}^{p}(x_{ij}-x_{i'j})^2 = 2\sum_{i\in C_k}\sum_{j=1}^{p}(x_{ij}-\bar{x}_{kj})^2$$

Step 2a minimizes the right side exactly (mean is the minimizer of squared deviations). Step 2b can only decrease it further by moving points to closer centroids.

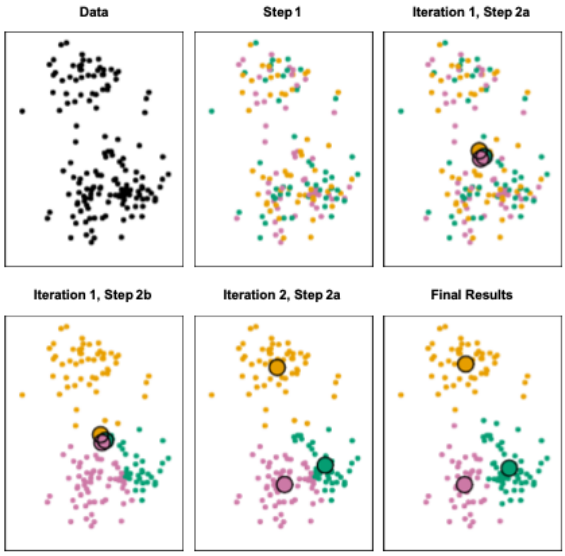
This identity is the reason the centroid (and not any other point) is used: the mean uniquely minimizes sum of squared distances, making step 2a a true minimization step.

Not guaranteed to reach the global minimum. Convergence is to a **local optimum**: the final solution depends entirely on the random initialization.

Lloyd's Algorithm, one run with $K = 3$.

- **Data:** raw unlabeled observations, no cluster structure imposed
- **Step 1:** random initialization, each point randomly assigned to one of $K = 3$ clusters; centroids nearly overlap since assignment is spatially uninformed
- **Iteration 1, Step 2a:** centroids computed from random assignments
- **Iteration 1, Step 2b:** each point reassigned to nearest centroid; clusters begin to reflect geometric structure
- **Iteration 2, Step 2a:** centroids recomputed from updated assignments; centroids shift substantially toward true cluster centers
- **Final:** converged solution after 10 iterations

Steps 2a and 2b form one complete iteration of the inner loop: centroid update then reassignment, repeated until convergence. The figure shows two consecutive iterations to illustrate how rapidly centroids stabilize once assignments become spatially coherent.



Centroid of cluster C_k. The **centroid** $\bar{x}_k \in \mathbb{R}^p$ is the mean vector of all observations in the cluster:

$$\bar{x}_k = \frac{1}{|C_k|}\sum_{i\in C_k} x_i \qquad \text{i.e.} \qquad \bar{x}_{kj} = \frac{1}{|C_k|}\sum_{i\in C_k} x_{ij} \quad \forall\, j = 1,\ldots,p$$

The centroid is the unique point in $\mathbb{R}^p$ that minimizes the sum of squared L_2 distances to all points in the cluster, this is exactly why step 2a of Lloyd's algorithm is a true descent step.

Medoid of C_k.

6.3.1 K-Medoids Clustering

⋆ **K-Medoids Clustering.** A robust variant of K-means. Instead of the **centroid** (mean vector, not necessarily a real point), each cluster is represented by its **medoid**, the actual data point with the

smallest average dissimilarity to all others in the cluster:

$$\boxed{\text{medoid}(C_k) = \arg\min_{i \in C_k} \frac{1}{|C_k|} \sum_{i' \in C_k} d(x_i, x_{i'})}$$

Medoid is the actual data point $x_i \in C_k$ that minimizes the average dissimilarity to all other points in the cluster:

The centroid is an abstract average; it may not correspond to any real observation. The medoid is always a real data point, making cluster representatives directly interpretable.

- **Works with any dissimilarity measure** $d(\cdot, \cdot)$, applicable to non-numeric data (categorical, ordinal) where a mean is undefined
- **More robust to outliers**: an extreme point cannot drag the medoid away from the cluster bulk the way it can distort a centroid
- **Computationally heavier**: finding the medoid costs $O(|C_k|^2)$ pairwise evaluations per cluster per iteration vs. $O(|C_k| \cdot p)$ for the centroid

Use K-medoids when your dissimilarity is non-Euclidean, your data is non-numeric, or you need cluster centers that are real, interpretable observations. Use K-means when speed matters and Euclidean distance is appropriate.

6.4 Hierarchical Clustering

⋆ **Hierarchical Clustering.** An alternative to K-means that does **not require pre-specifying** K. Produces a **dendrogram**: a tree-like representation encoding the clustering for every possible number of clusters from 1 to n simultaneously. The user selects K by **cutting the dendrogram** at a chosen height after the fact.

K-means forces a commitment to K before seeing any results. Hierarchical clustering defers that decision entirely: you fit once and read off any K you want from the same tree.

⋆ **Agglomerative (Bottom-Up) Clustering.** The most common variant. Builds the dendrogram starting from the **leaves** (individual points) and merging upward to the **trunk** (single cluster containing all n points).

Hierarchical Clustering as Tree Construction. Agglomerative clustering is equivalent to building a **binary tree** bottom-up: each of the $n - 1$ merge operations adds one internal node, until the root is reached:

- **Leaves** (n): individual observations, the starting singletons
- **Internal nodes** $(n - 1)$: each represents one merge event
- **Root**: the single cluster containing all n points
- **Edge height**: the dissimilarity $d(C_k, C_{k'})$ at which that merge occurred

*This is the same tree-building logic used throughout ML, decision trees split nodes top-down, hierarchical clustering merges them bottom-up. The dendrogram is simply the binary tree made visible, with height encoding dissimilarity rather than information gain. The key difference from decision trees: here the tree is built on **observations**, not features, each leaf is a data point, not a feature threshold.*

Algorithm (Algorithm 10.2).

1. **Initialize:** treat each of the n observations as its own singleton cluster. Compute all $\binom{n}{2} = \frac{n(n-1)}{2}$ pairwise dissimilarities.
2. **For** $i = n, n - 1, \ldots, 2$:

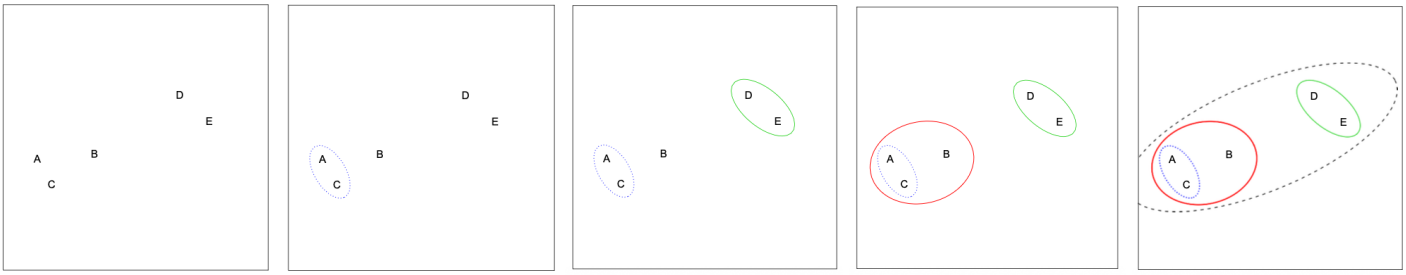
(a) Examine all pairwise inter-cluster dissimilarities among the current i clusters. **Fuse** the two most similar (least dissimilar) clusters into one. The dissimilarity at which they merge determines the **height** of that fusion node in the dendrogram.
(b) Recompute all pairwise inter-cluster dissimilarities among the $i-1$ remaining clusters.

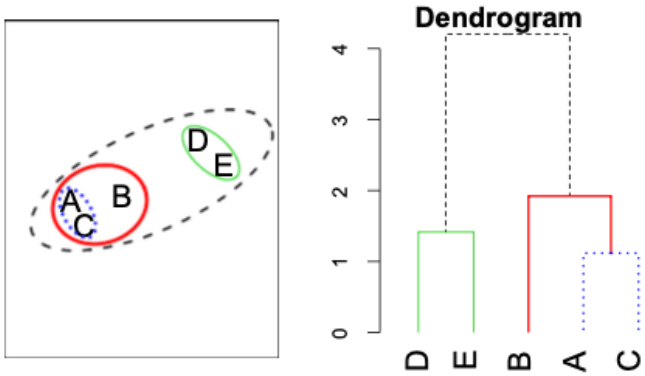
Each merge is irreversible: once two observations are placed in the same cluster they stay together forever. This is the fundamental constraint of hierarchical clustering: it cannot undo an early bad merge the way K-means can reassign points across iterations.

Bottom-up merging, illustrated with 5 points $\{A, B, C, D, E\}$.

- Step 1: A and C are closest $\Rightarrow$ merge into $\{A, C\}$
- Step 2: D and E are next closest $\Rightarrow$ merge into $\{D, E\}$
- Step 3: B joins $\{A, C\}$ $\Rightarrow$ merge into $\{A, B, C\}$
- Step 4: $\{A, B, C\}$ and $\{D, E\}$ merge $\Rightarrow$ single cluster $\{A, B, C, D, E\}$

The nested structure is visible in the dendrogram: $\{A, C\}$ fuses at low height (very similar), $\{D, E\}$ fuses slightly higher, B joins the $\{A, C\}$ group at intermediate height, and the two macro-groups merge at the top. The height encodes how reluctantly each merger happened.





- **Leaves** (bottom): individual observations
- **Height of a node:** the value of the **inter-cluster dissimilarity** at which the two daughter clusters were fused; taller fusions indicate more dissimilar groups being merged
- **Cutting at height h:** draw a horizontal line at h; the number of branches it crosses is the number of clusters K. Lower cuts give more clusters; higher cuts give fewer.

⋆ **Nested Clusters, the key limitation.** Clusters obtained by cutting the dendrogram at height h are always **nested within** clusters from any higher cut. This is forced by the algorithm's irreversibility: a merger made early can never be undone.

Nesting Constraint. Any clustering obtained by cutting the dendrogram at a lower height is always a **refinement** of a clustering obtained by cutting at a higher height, clusters can only be split going down, never reorganized.

- This assumption of nested structure may be **unrealistic** for many datasets

- Example: for a dataset of males/females evenly split among Americans, Japanese, and French: the best $K = 2$ split is by **gender**; the best $K = 3$ split is by **nationality**. These two solutions are **not nested**: nationality groups do not arise by splitting a gender group. Hierarchical clustering cannot represent this.
- Consequence: for a given K, hierarchical clustering can yield **worse results** than K-means, which optimizes directly for K clusters without the nesting constraint.

Hierarchical clustering buys flexibility in K at the cost of the nesting constraint. When the true cluster structure is genuinely hierarchical (e.g. biological taxonomies), this is fine. When the best $K = 2$ and best $K = 3$ solutions are unrelated, hierarchical clustering will get at least one of them wrong.

6.4.1 Linkage

Dissimilarity. A function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{\geq 0}$ that measures how **different** two observations $x_i, x_{i'} \in \mathbb{R}^p$ are, satisfying:

$$\boxed{d(x_i, x_{i'}) \geq 0, \qquad d(x_i, x_i) = 0, \qquad d(x_i, x_{i'}) = d(x_{i'}, x_i)}$$

Small $d(x_i, x_{i'})$: observations are **similar** and **Large** $d(x_i, x_{i'})$: observations are **different**
Unlike a true metric, dissimilarity does **not** require the triangle inequality
$d(x_i, x_{i'}) \leq d(x_i, x_k) + d(x_k, x_{i'})$

Dissimilarity is the raw ingredient every clustering algorithm consumes: K-means uses it to assign points to centroids, hierarchical clustering uses it to decide which clusters to merge. The choice of d encodes what "different" means in your domain, and this choice shapes the entire clustering solution.

⋆ **Linkage.** The **linkage** function extends pairwise observation dissimilarity $d(x_i, x_{i'})$ to a dissimilarity between two **groups** of observations C_A and C_B, required by the hierarchical algorithm to decide which two clusters to merge at each step. Four standard types:

- **Complete linkage:** $d(C_A, C_B) = \max_{i \in C_A, i' \in C_B} d(x_i, x_{i'})$: **maximal** pairwise dissimilarity. Merges clusters that are close even at their most distant points. Produces **compact, balanced** clusters.
  Distance between the two farthest points across clusters *Complete linkage is conservative: two clusters only merge when their farthest members are close. This tends to produce roughly equal-sized, well-separated clusters.*

- **Single linkage:** $d(C_A, C_B) = \min_{i \in C_A, i' \in C_B} d(x_i, x_{i'})$: **minimal** pairwise dissimilarity. Merges clusters as soon as any two members are close. Can produce **chaining**: long, trailing clusters.
  Distance between the two closest points across clusters*Single linkage is aggressive: one close pair of points is enough to fuse two large clusters. This makes it sensitive to noise and prone to producing elongated chains rather than compact blobs.*

- **Average linkage:** $d(C_A, C_B) = \frac{1}{|C_A||C_B|} \sum_{i \in C_A} \sum_{i' \in C_B} d(x_i, x_{i'})$: **mean** of all pairwise dissimilarities. A compromise between complete and single.
  Mean distance over all cross-cluster point pairs *Average linkage is the most commonly used in practice: less sensitive to outliers than single, less rigid than complete.*

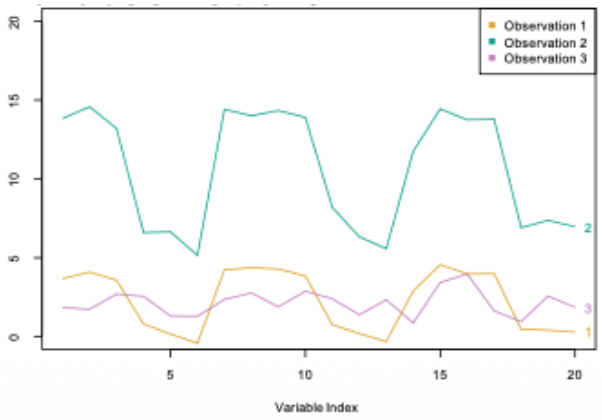
- **Centroid linkage:** $d(C_A, C_B) = d(\bar{x}_A, \bar{x}_B)$: dissimilarity between the **centroids** of the two clusters. Can produce **inversions**: a fusion placed at a height **lower** than one or both of the clusters being merged, violating the monotonicity of the dendrogram and making it uninterpretable.
  Distance between the two cluster centroids *Inversions occur because the centroid of a*

merged cluster can be closer to another cluster's centroid than the original sub-cluster centroids were, a geometric artifact with no clean fix. Avoid centroid linkage unless there is a strong reason to use it.

⋆ **Choice of Dissimilarity Measure** All linkage types require a pairwise dissimilarity $d(x_i, x_{i'})$ between observations. Two main choices:

- **Euclidean distance:** $d(x_i, x_{i'}) = \|x_i - x_{i'}\|_2$: measures **magnitude** of difference. Two observations are similar if their feature values are numerically close.
- **Correlation-based distance:** $d(x_i, x_{i'}) = 1 - \text{corr}(x_i, x_{i'})$: measures **shape** of the observation profile across features, ignoring magnitude. Two observations are similar if their features move up and down together across variables, regardless of their absolute values.

*This is an unusual use of correlation, normally computed between variables, here it is computed between pairs of **observations** treated as vectors over p feature indices. Obs. 1 and 3 may be numerically close (small Euclidean distance) but have uncorrelated profiles (dissimilar shapes): Euclidean calls them close, correlation calls them far. Obs. 1 and 2 may be far apart in magnitude but track the same pattern, correlation calls them close. The right choice depends entirely on whether you care about absolute values or relative patterns.*



⋆ **Practical Issues in Clustering**

Key decisions required before fitting any clustering model:

- **Standardization:** should features be centered (mean zero) and scaled (standard deviation one) before clustering? Without scaling, features with large numeric ranges dominate the distance computation regardless of their actual importance.
- **Dissimilarity measure:** Euclidean or correlation-based? The choice changes which observations are considered similar and thus which clusters form.
- **Linkage type** (hierarchical only): complete, single, average, or centroid? Each produces qualitatively different dendrograms from the same data.
- **Number of clusters K:** how many clusters to extract? No agreed-upon method. A hard, open problem, discussed next.

None of these choices has a universally correct answer. Clustering results can change dramatically with different decisions. This is the core subjectivity of unsupervised learning; always explore multiple settings and validate against domain knowledge.

6.5 Choosing K

6.5.1 Within-Cluster Variation $W(K)$ and Between-Cluster Variation $B(K)$

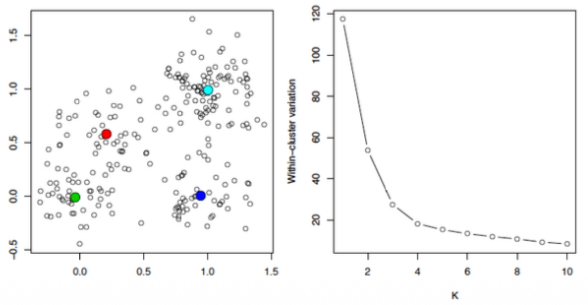
⋆ $W(K)$ **Total Within-Cluster Variation.** The K-means objective written explicitly as a function of K: $\boxed{W(K) = \sum_{k=1}^{K} \sum_{C(i)=k} \|X_i - \bar{X}_k\|_2^2}$ where $\bar{X}_k$ is the centroid of cluster k and $C(i) = k$ means observation i is assigned to cluster k.

$W(K)$ is monotonically decreasing in K by construction: more clusters always fit tighter. At $K = n$

every point is its own cluster and $W = 0$. Minimizing $W(K)$ directly over K is therefore useless as a model selection criterion.

Why choosing K is hard. In most exploratory applications K is unknown. The naive approach: minimize $W(K)$ over K, **fails immediately:** $W(K) = \sum_{k=1}^{K} \sum_{C(i)=k} \|X_i - \bar{X}_k\|_2^2$
$W(K)$ is **monotonically decreasing** in K, adding more clusters can only decrease within-cluster variation. At $K = n$ every point is its own cluster and $W = 0$. Minimizing $W(K)$ directly always selects the largest K, which is useless.

This is the fundamental tension: more clusters always fits better by construction, but overfitting is meaningless here since there is no response Y to overfit to. We need a criterion that balances tightness against parsimony, which is exactly what the Gap Statistic, CH index, and Silhouette analysis provide in the next section.



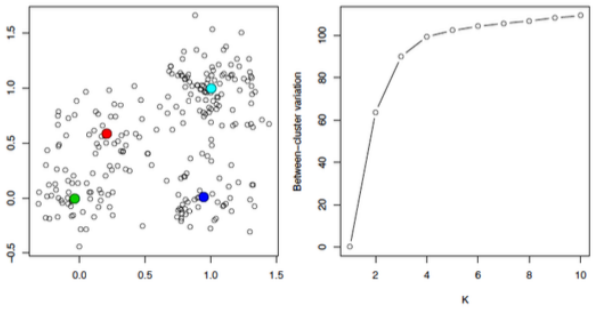
⋆ **Between-Cluster Variation $B(K)$.** $W(K)$ alone fails because it always decreases with K. What is missing is a measure of how **spread apart** the clusters are from each other. Define:

$$B(K) = \sum_{k=1}^{K} n_k \|\bar{X}_k - \bar{X}\|_2^2$$

where $n_k = |C_k|$, $\bar{X}_k = \frac{1}{n_k} \sum_{C(i)=k} X_i$ is the centroid of cluster k, and $\bar{X} = \frac{1}{n} \sum_{i=1}^{n} X_i$ is the **overall mean**. Each term measures how far cluster k's centroid is from the global center, weighted by cluster size.

*$B(K)$ is the between-cluster analog of $W(K)$: while W measures how tight each cluster is internally, B measures how far apart the cluster centers are from the global mean. A good clustering has small W **and** large B simultaneously.*

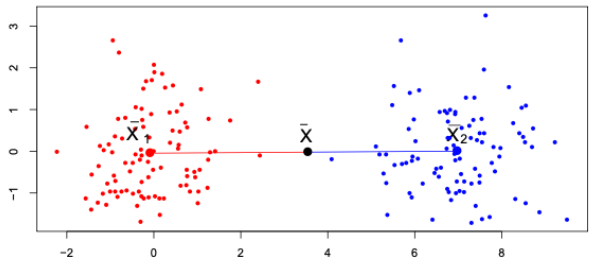
Why $B(K)$ alone also fails. $B(K)$ is **monotonically increasing** in K: more clusters always pushes centroids farther from the global mean. Maximizing $B(K)$ directly always selects the largest K, just as minimizing $W(K)$ does. Neither criterion alone is useful for selecting K.



Both $W(K)$ and $B(K)$ are individually degenerate as selection criteria: one always goes down, one always goes up. The solution is to combine them into a ratio that rewards large separation relative to small within-cluster spread.

- B measures how far each cluster centroid is from the global center $\bar{X}$: the horizontal arrows
- W measures how spread each point is around its own cluster centroid: the internal scatter

Good clustering maximizes B (centroids far from global mean) and minimizes W (points tight around their centroid) simultaneously, exactly the ratio the CH index optimizes.



$$B = n_1\|\bar{X}_1 - \bar{X}\|_2^2 + n_2\|\bar{X}_2 - \bar{X}\|_2^2$$

$$W = \sum_{C(i)=1} \|X_i - \bar{X}_1\|_2^2 + \sum_{C(i)=2} \|X_i - \bar{X}_2\|_2^2$$

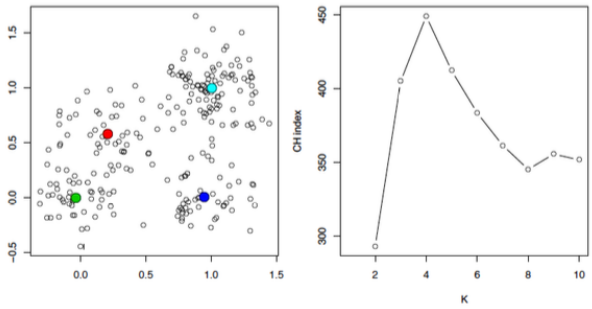
6.5.2 Calinski-Harabasz (CH) Index

⋆ **Calinski-Harabasz (CH) Index.** The **CH index** (Calinski & Harabasz, 1974) combines W and B into a single ratio that simultaneously rewards tight clusters and large separation:

$$\boxed{\mathrm{CH}(K) = \frac{B(K)/(K-1)}{W(K)/(n-K)}}$$

- $B(K)/(K-1)$: between-cluster variance, normalized by degrees of freedom $K-1$
- $W(K)/(n-K)$: within-cluster variance, normalized by degrees of freedom $n-K$
- **Larger** $\mathrm{CH}(K)$ is better: clusters are both tight and well-separated

The CH index is a variance ratio: the same logic as an F-statistic. A large numerator means cluster centers are far apart; a small denominator means points are close to their own center. The ratio peaks where this trade-off is most favorable, identifying the best K.



How to use CH to select K. Fix a maximum $K_{\max}$ (e.g. $K_{\max} = 20$), compute $\mathrm{CH}(K)$ for $K = 2, \ldots, K_{\max}$, and select: $\boxed{\hat{K} = \arg\max_{K \in \{2,\ldots,K_{\max}\}} \mathrm{CH}(K)}$

Limitation $\mathrm{CH}(K)$ is undefined at $K = 1$ with one cluster $B = 0$ and $K - 1 = 0$, making the ratio degenerate. The CH index can therefore never select the null model ($K = 1$), which is a structural blind spot.

6.5.3 Gap Statistic

⋆ **Gap Statistic.** The key observation: although $W(K)$ always decreases, **how much it drops at each K** is informative. The gap statistic formalizes this by comparing the observed $W(K)$ to a **null reference distribution**, the within-cluster variation $W_{\mathrm{unif}}(K)$ we would expect if the data had **no cluster structure** (points distributed uniformly over a bounding box):

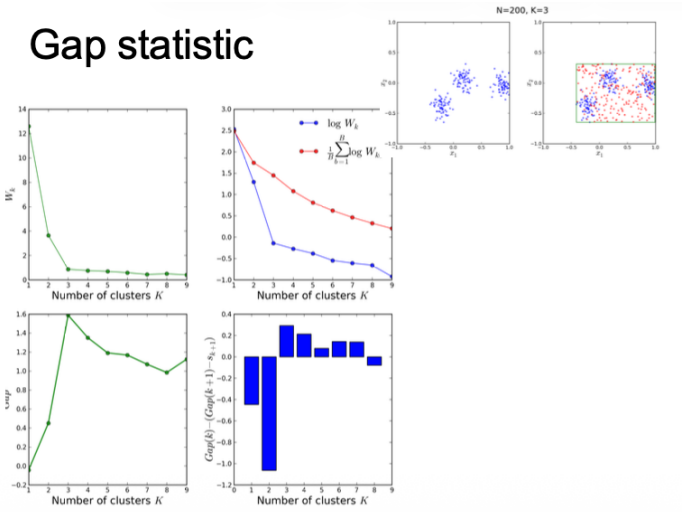
$$\boxed{\mathrm{Gap}(K) = \log W_{\mathrm{unif}}(K) - \log W(K)}$$

If the data have genuine cluster structure at K clusters, $W(K)$ will be much smaller than $W_{\text{unif}}(K)$, a big drop that the null wouldn't explain. The gap measures this excess reduction. When the data have no structure, $W(K) \approx W_{\text{unif}}(K)$ and $\text{Gap}(K) \approx 0$.
is our data's clustering meaningfully tighter than what pure randomness would produce at the same n? If yes, the gap is large and K clusters are real. If no, we are just fitting noise.

Reference distribution and $s(K)$.
$W_{\text{unif}}(K)$ is estimated by Monte Carlo: generate B datasets sampled uniformly from the **bounding box** of the original data, run K-means on each, and average their log within-cluster variations:

$$\log W_{\text{unif}}(K) \approx \frac{1}{B}\sum_{b=1}^{B} \log W^{(b)}(K)$$

$$s(K) = \text{sd}\Big(\{\log W^{(b)}(K)\}_{b=1}^{B}\Big)\sqrt{1+\tfrac{1}{B}}$$



Monte Carlo: repeatedly sample fake datasets from the null (uniform over the bounding box), run K-means on each, and average the results. Each of the B draws is a fresh finite sample of n points (same n as real data) drawn uniformly, so K-means on each produces a slightly different $W^{(b)}(K)$ purely from finite-sample randomness, not from any structure. The average $\frac{1}{B}\sum_b \log W^{(b)}(K)$ converges to $\mathbb{E}_{\text{unif}}[\log W(K)]$, the expected log within-cluster variation under a structureless null; $s(K)$ captures the sampling variability of this estimate. **The gap is then how far the real data's $\log W(K)$ sits below this null expectation, a large gap means real structure, not just finite-sample compression.**

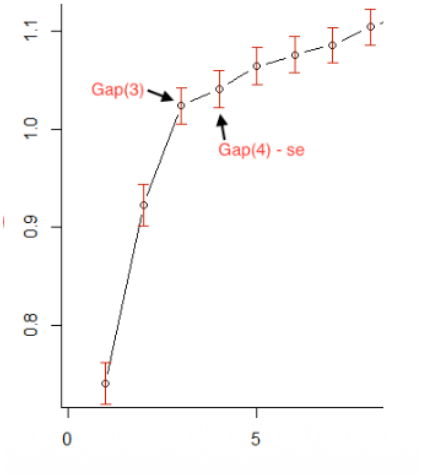
K Selection rule from Gap:

$$\hat{K} = \min_{K\in\{1,\ldots,K_{\max}\}} \{\text{Gap}(K) \geq \text{Gap}(K+1) - s(K+1)\}$$

Choose the **smallest** K such that the gap at K is already within one standard error of the gap at $K+1$. Adding one more cluster doesn't give a meaningfully larger gap.

$s(K)$ is the standard deviation of the B simulated $\log W^{(b)}(K)$ values, is also standard error on the null baseline estimate. The selection rule $\text{Gap}(K) \geq \text{Gap}(K+1) - s(K+1)$ uses it as a tolerance: don't move to $K+1$ unless the improvement in gap exceeds one standard error, the same 1-SE logic as in cross-validation model selection.

The rule is a one-standard-error (1-SE) criterion: don't increase K unless the next value gives a gap that is significantly larger.
This penalizes complexity and tends toward parsimonious solutions. The gap statistic is particularly powerful when the true $K = 1$ (null case) since the null reference *is* the uniform distribution, Gap correctly stays near zero for all K and selects $K = 1$.



6.5.4 Silhouette Analysis

⋆ **Silhouette Analysis.** A method for **interpretation and validation** of consistency within clusters. For any clustering (K-means or otherwise), it assigns each point x_i a scalar score $s_i \in [-1, 1]$ measuring how well it sits within its assigned cluster relative to the next-best alternative.

Building blocks for point x_i (after clustering into K clusters).

- a_i: average distance from x_i to all other points **in its own cluster** measures how well x_i fits internally. Smaller a_i = tighter fit.
- b_i: lowest average distance from x_i to all points in **any other cluster** (of which x_i is not a member), the **neighboring cluster** is whichever other cluster achieves this minimum. Larger $b_i = x_i$ is far from its next-best option.

Silhouette Score s_i. $\boxed{s_i = \dfrac{b_i - a_i}{\max(a_i,\, b_i)}} \quad \Rightarrow \quad s_i = \begin{cases} 1 - a_i/b_i & a_i < b_i \\ 0 & a_i = b_i \\ b_i/a_i - 1 & a_i > b_i \end{cases}$ (Range $-1 \le s_i \le 1$)

- $s_i \approx 1$: requires $a_i \ll b_i$; x_i is tight within its cluster and far from the neighboring cluster. **Well clustered.**
- $s_i \approx 0$: $a_i \approx b_i$; x_i sits on the **border** between two clusters.
- $s_i \approx -1$: $a_i \gg b_i$; x_i is closer to its neighboring cluster than its own. **Misclassified.**

s_i simultaneously encodes both how tight x_i's own cluster is (a_i) and how separated it is from the next-best cluster (b_i). It is a per-point verdict: positive means the assignment is correct, negative means the point belongs elsewhere.

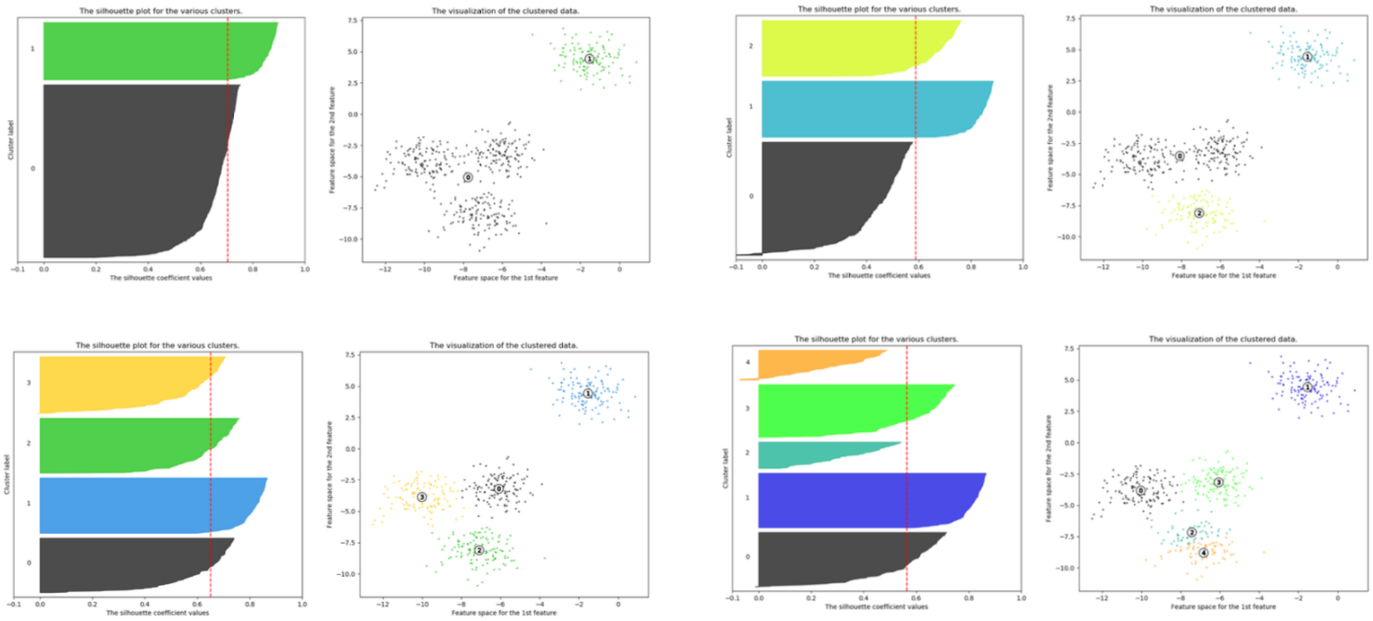
Aggregation and use for selecting K.

- **Average s_i over a cluster**: measures how tightly grouped that cluster is.
- **Average s_i over the full dataset**: a global measure of how appropriate the entire clustering is - maximize this over K to select the best number of clusters.
- **Silhouette plot**: sorts s_i values within each cluster and plots them as horizontal bars. A good K gives all clusters with **above-average width** and **uniform size**. A bad K produces some clusters with narrow or negative bars.

Rescaling features using cluster-specific weights can further improve silhouette scores at the correct K, though this is an optional refinement.

When silhouette analysis is uncertain, combine it with CH index or gap statistic; three methods agreeing on the same K is strong evidence. Three methods disagreeing signals a genuinely ambiguous cluster structure in the data.

$K = 3, 5, 6$ are bad picks: some clusters display **below-average silhouette scores** and **wide size fluctuations** across clusters. Silhouette analysis is **ambivalent between** $K = 2$ **and** $K = 4$. *Cluster must be above red threshold to be accepted.*

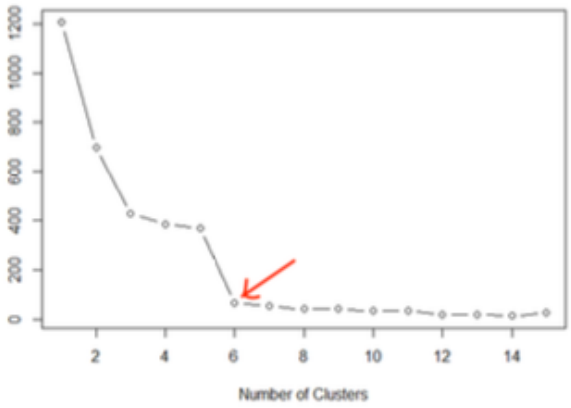


6.5.5 Cross-Validation for Selecting K

⋆ **Cross-Validation for Selecting K.** The problem: we need to pick K but have no response Y to validate against. CV adapts the supervised idea, hold out data and measure how well the model generalizes, to the unsupervised setting by using a geometric proxy instead of prediction error.

1. Partition the n observations into v folds.
2. For each fold $f = 1, \ldots, v$: fit K-means on the $v-1$ training folds, then assign held-out points to the nearest centroid and compute the objective (sum of squared distances to assigned centroids) on the test fold.
3. Average the v test-fold objectives $\rightarrow$ one number per K.
4. Repeat for $K = 1, \ldots, K_{\max}$ and plot: pick $\hat{K}$ at the **elbow**, the point where further increases in K yield only marginal improvement.

The key asymmetry with supervised CV: centroids are learned on training folds but the test-fold objective (distance to those centroids) is a proxy for fit quality, not true generalization error. There is no Y to be wrong about. The elbow heuristic is therefore still subjective; CV narrows the range but rarely gives a sharp answer.



6.6 Variable Selection for Clustering

⋆ **Variable Selection for Clustering.** Find the **smallest feature subset** that best uncovers natural cluster structure. Not all features help; two failure modes to watch for:

- **Redundant features**: two features carrying the same cluster-discriminating information. Including both inflates distances without adding signal.
- **Irrelevant features**: features that provide no separation between clusters: pure noise in the distance computation. In high dimensions these dominate, washing out true structure.

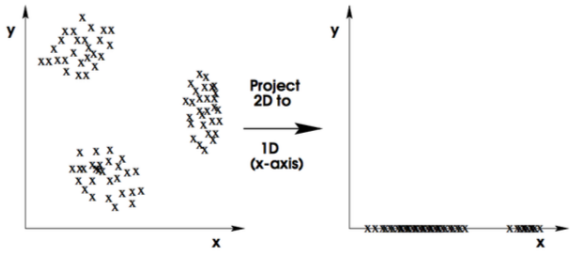
Different feature subsets can reveal different numbers of clusters in the same data, so K must be re-estimated at every step of the subset search, not fixed globally. This is the core complication that makes variable selection for clustering harder than for supervised methods.

6.6.1 Subset Selection for Clustering

Two paradigms for feature subset selection.

- **Unified criterion**: the selection criterion is the **same** as the clustering criterion, select features that directly optimize the clustering objective (e.g. minimize $W(K)$).
- **Separate criteria**: the selection criterion **need not match** the clustering criterion: e.g. select features by mutual information or variance, then cluster by K-means. More flexible but risks misalignment between selection and clustering goals.

Core complication: K depends on the feature subset. Different feature subsets reveal different numbers of clusters. Example: in 2D (x, y) three clusters are visible; projecting onto x alone collapses them into two. **A fixed K applied across all feature subsets incorrectly models the data** in each subspace; the best K must be re-optimized at every step of the search.



⋆ **K-means with feature weighting.** Most K-means variants for variable selection follow a two-step loop:

1. Cluster data into K clusters.
2. Assign weight $w_j \geq 0$ to each feature j: features that **minimize within-cluster distance** or **maximize between-cluster distance** receive higher weight. The weighted distance replaces standard Euclidean: $d_w(x_i, x_{i'}) = \sum_{j=1}^{p} w_j (x_{ij} - x_{i'j})^2$ Iterate until convergence.

⋆ **Sparse Clustering (Witten & Tibshirani).** Reformulate K-means with a weight vector $w \in \mathbb{R}^p$ and an L_1 penalty to produce **exact zeros**, hard feature removal rather than soft down-weighting:

$$\underset{C_1,\ldots,C_K,\, w}{\text{maximize}} \sum_{j=1}^{p} w_j \cdot \text{BCSS}_j \quad \text{s.t.} \quad \|w\|_1 \leq s,\ \|w\|_2 \leq 1,\ w_j \geq 0$$

where $\text{BCSS}_j = \frac{1}{n}\sum_{i \neq i'} (x_{ij} - x_{i'j})^2 - \sum_{k=1}^{K} \frac{1}{|C_k|} \sum_{i,i' \in C_k} (x_{ij} - x_{i'j})^2$ is the **between-cluster sum of squares** for feature j (total variance minus within-cluster variance). The L_1 bound s controls sparsity: smaller $s \Rightarrow$ fewer active features.

This is the Lasso applied to clustering: the L_1 constraint is the only mechanism that produces exact zeros. The L_2 constraint $\|w\|_2 \leq 1$ prevents the trivial solution of concentrating all weight on one feature. Together they identify the minimal informative feature subset automatically, inside the clustering objective.

6.7 Principal Components Analysis (PCA) and the SVD

⋆ **Principal Components Analysis.** Given an $n \times p$ data matrix $\mathbf{X}$, PCA finds a sequence of **mutually uncorrelated** linear combinations of the p features that have **maximal variance**: **produces a low-dimensional representation that captures as much variation as possible.**
Two uses: (1) **visualization** of high-dimensional data, (2) **pre-processing** before supervised methods.

Core idea: instead of working in $\mathbb{R}^p$, rotate the coordinate system to align axes with the directions of maximum spread. The first new axis (PC1) captures the most variance, PC2 the next most while being perpendicular to PC1, and so on. The first $M \ll p$ axes often capture most of the information.

⋆ **First Principal Component: PC1** The **first principal component** Z_1 is the normalized linear combination of $X_1, \ldots, X_p$ with the largest variance:

$$\boxed{Z_1 = \varphi_{11} X_1 + \varphi_{21} X_2 + \cdots + \varphi_{p1} X_p}$$

where $\varphi_1 = (\varphi_{11}, \varphi_{21}, \ldots, \varphi_{p1})^T \in \mathbb{R}^p$ is the **first loading vector** (also called the first right singular vector of $\mathbf{X}$), normalized so that:

$$\boxed{\sum_{j=1}^{p} \varphi_{j1}^2 = 1}$$

The normalization constraint is essential; without it, inflating any φ_{j1} arbitrarily would produce arbitrarily large variance, making the problem unbounded.

Meanings of PC1:

- $\varphi_1 \in \mathbb{R}^p$: the **loading vector**: defines the **direction** of PC1 in feature space. This is what you solve for: p unknown weights, one per feature **Length: p in direction in feature space**
- $\mathbf{z}_1 = \mathbf{X}\varphi_1 \in \mathbb{R}^n$: the **score vector**: the actual values of PC1 for each of your n observations. One number per data point: its coordinate along the φ_1 direction. Falls out for free once φ_1 is known. **Length: n projection of data onto that direction**

Idea: The loading vector φ_1 tells you **how** to combine features. The score vector $\mathbf{z}_1$ gives you the **result** of that combination for every observation.

Optimization problem. Assume $\mathbf{X}$ is **column-centered**. Then $z_{i1} = \sum_{j=1}^{p} \varphi_{j1} x_{ij}$ has mean zero, its sample variance is $\frac{1}{n}\sum_{i=1}^{n} z_{i1}^2$. The first loading vector solves:

$$\boxed{\underset{\varphi_1}{\text{maximize}} \underbrace{\frac{1}{n}\sum_{i=1}^{n}\left(\sum_{j=1}^{p}\varphi_{j1}x_{ij}\right)^2}_{=\mathrm{Var}(Z_1)=\varphi_1^T\mathbf{S}\varphi_1} \quad \text{subject to} \quad \|\varphi_1\|^2 = 1}$$

This is the **Rayleigh quotient** of $\mathbf{S} = \frac{1}{n}\mathbf{X}^T\mathbf{X}$: find the unit-norm direction maximizing projected spread, solved exactly by the **largest eigenvector of S**, computed efficiently via **Singular Value Decomposition (SVD)** of $\mathbf{X}$ directly:

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$$

- $\mathbf{V}_{p\times p}$: **right singular vectors**: columns are the loading vectors $\varphi_1, \varphi_2, \ldots$ (eigenvectors of $\mathbf{S}$)
- $\mathbf{D}_{p\times p}$: diagonal singular values $d_1 \geq \cdots \geq d_p \geq 0$, giving component variances $\lambda_m = d_m^2/n$ (eigenvalues of $\mathbf{S}$)
- $\mathbf{U}_{n\times p}$: **left singular vectors**: directions in observation space

Scores fall out immediately: $\mathbf{Z} = \mathbf{X}\mathbf{V} = \mathbf{U}\mathbf{D}$. SVD avoids forming $\mathbf{X}^T\mathbf{X}$ explicitly: numerically more stable when p is large.

Geometry of PCA. The loading vector φ_1 defines a **direction in** $\mathbb{R}^p$ along which the data vary the most. Projecting all n points $x_1, \ldots, x_n$ onto this direction yields exactly the scores $z_{11}, \ldots, z_{n1}$:

$$z_{i1} = \varphi_1^T x_i = \sum_{j=1}^{p} \varphi_{j1} x_{ij}$$

PC scores are orthogonal projections, each z_{i1} is the signed length of x_i along the direction of maximum variance. PCA literally rotates the data so the new first axis points where the cloud is most elongated.

Dimensionality reduction means observation i is compressed from p features down to $M \ll p$ scores:

$$x_i = \begin{pmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{pmatrix} \in \mathbb{R}^p \quad \xrightarrow{\text{PCA}} \quad z_i = \begin{pmatrix} z_{i1} \\ z_{i2} \\ \vdots \\ z_{iM} \end{pmatrix} \in \mathbb{R}^M$$

⋆ **Further Principal Components.** The ***m*-th principal component** Z_m is the linear combination of $X_1, \ldots, X_p$ with maximal variance among all combinations **uncorrelated with** $Z_1, \ldots, Z_{m-1}$:

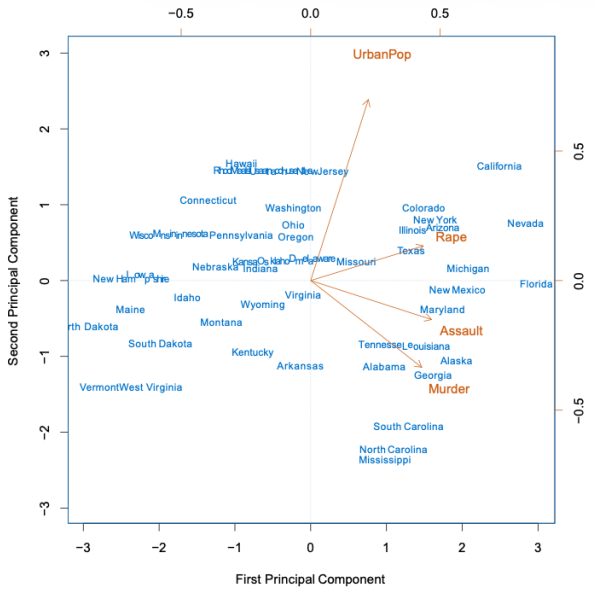
$$z_{im} = \varphi_{1m} x_{i1} + \varphi_{2m} x_{i2} + \cdots + \varphi_{pm} x_{ip}$$

where $\varphi_m = (\varphi_{1m}, \ldots, \varphi_{pm})^T$ is the m-th loading vector.
Uncorrelated with $Z_{m-1} \iff \varphi_m \perp \varphi_{m-1}$ (orthogonal directions) The loading vectors $\varphi_1, \varphi_2, \ldots$ are the **ordered right singular vectors** of $\mathbf{X}$; variances of components are $\frac{1}{n}$ times the squared singular values. There are at most $\min(n-1, p)$ principal components.

Orthogonality of loading vectors is not just a constraint, it is a consequence of the uncorrelatedness requirement. Each new PC finds the direction of maximum residual variance in the subspace orthogonal to all previous PCs. The PCs partition the total variance of $\mathbf{X}$ into non-overlapping pieces.

Illustration: USArrests ($n = 50$, $p = 4$). Features: Assault, Murder, Rape (arrests per 100k residents), UrbanPop (% urban population). PCA performed after standardizing each variable to mean 0, sd 1.

PC1 loading: approximately equal weight on Assault, Murder, Rape: roughly an overall crime rate index. States with large positive PC1 scores (e.g. California, Florida) have high crime rates; negative scores (e.g. North Dakota) have low crime rates.
PC2 loading: dominated by UrbanPop: roughly an urbanization index. California scores high (urban); Mississippi scores low (rural).
*The **biplot** displays both scores (observations in PC space) and loadings (feature arrows) simultaneously. Arrow direction shows which features a PC emphasizes; arrow length shows how much variance that feature contributes to those two PCs.*



PCA biplot key ideas:

- **Loading arrows reveal feature correlations**: features whose arrows point in similar directions are correlated: they move together across observations. Features with orthogonal arrows are uncorrelated.

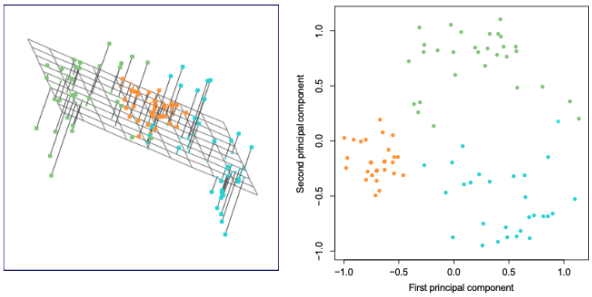
- **Loading magnitude = contribution**: a large $|\varphi_{jm}|$ means feature j drives component m. Near-zero loading means that feature is irrelevant to that component.
- **Scores reveal observation structure**: observations far in the same direction share the same underlying signal. Observations near the origin are average on all components.
- **Components are interpretable**: each PC often aligns with a latent concept: e.g. PC1 $\approx$ overall crime level, PC2 $\approx$ urbanization. PCA discovers these automatically without being told.
- **Biplot = scores + loadings simultaneously**: the two coordinate systems (scores on bottom/left axes, loadings on top/right axes) live in the same plot; you can read off both observation positions and feature contributions at once.

The core lesson: PCA does not just compress data; it exposes latent structure. Correlated features get bundled into one component; independent signals become separate components. The loadings tell you what each component means; the scores tell you where each observation stands on that meaning.

⋆ **PCA as Closest Hyperplane.** An equivalent geometric interpretation: the first M principal components span the M**-dimensional hyperplane closest to the** n **observations** in terms of average squared Euclidean distance.

- $M = 1$: φ_1 defines the **line** in $\mathbb{R}^p$ closest to all n points; maximizing variance and minimizing reconstruction error are the same problem.
- $M = 2$: φ_1, φ_2 span the **plane** closest to the n points.
- $M = M$: the first M loading vectors span the closest M-dimensional **hyperplane**.

*Two views of PCA: same solution. Maximizing projected variance is equivalent to minimizing the average squared distance from each point to its projection onto the subspace. The loading vectors simultaneously point where data spreads most **and** where the subspace fits best.*



Hyperplane reconstruction formulation. The M-dimensional hyperplane spanned by $\varphi_1, \ldots, \varphi_M$ approximates each observation $x_i \in \mathbb{R}^p$ by its projection:

$$\hat{x}_i = \sum_{m=1}^{M} z_{im}\varphi_m = \mathbf{\Phi}_M z_i \in \mathbb{R}^p$$

where $z_{im} = \varphi_m^T x_i$ are the scores and $\mathbf{\Phi}_M = [\varphi_1 | \cdots | \varphi_M]$ is the $p \times M$ loading matrix. PCA solves simultaneously:

$$\underset{\varphi_1,\ldots,\varphi_M}{\text{minimize}} \frac{1}{n}\sum_{i=1}^{n} \|x_i - \hat{x}_i\|_2^2 \quad \text{subject to} \quad \mathbf{\Phi}_M^T\mathbf{\Phi}_M = \mathbf{I}_M$$

The constraint $\mathbf{\Phi}_M^T\mathbf{\Phi}_M = \mathbf{I}_M$ enforces orthonormality of the loading vectors: columns of $\mathbf{\Phi}_M$ must be unit length and mutually orthogonal. In matrix form the reconstruction error is:

$$\frac{1}{n}\|\mathbf{X} - \mathbf{X}\mathbf{\Phi}_M\mathbf{\Phi}_M^T\|_F^2$$

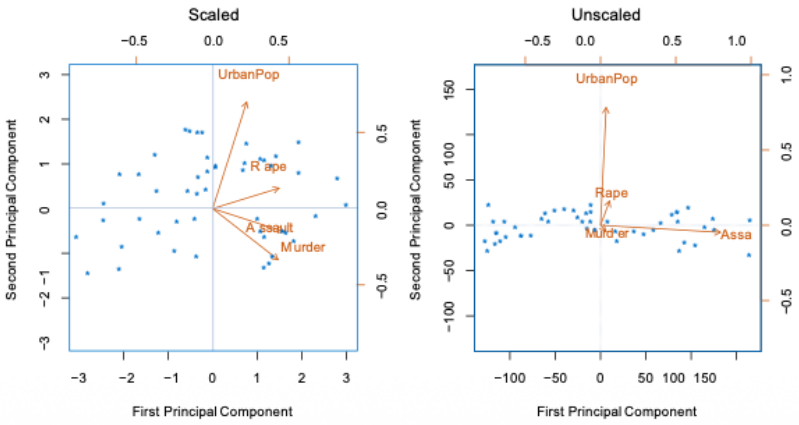
where $\|\cdot\|_F$ is the Frobenius norm and $\mathbf{\Phi}_M\mathbf{\Phi}_M^T$ is the **projection matrix** onto the M-dimensional subspace. **Minimizing reconstruction error and maximizing variance are the same problem**, both are solved by the top M right singular vectors of $\mathbf{X}$.

$\mathbf{\Phi}_M\mathbf{\Phi}_M^T \in \mathbb{R}^{p\times p}$ projects any point in $\mathbb{R}^p$ onto the hyperplane. The residual $x_i - \hat{x}_i$ is the component of x_i orthogonal to the hyperplane, PCA minimizes the total squared length of these residuals over all n points simultaneously.

⋆ **Scaling of Variables.** PCA is **not scale-invariant**: loadings depend on the units of measurement. A feature with large variance will dominate PC1 regardless of whether that variance is scientifically meaningful or just a consequence of units.

Why scaling matters. PCA maximizes $\varphi_1^T \mathbf{S} \varphi_1$; the covariance matrix $\mathbf{S}$ is directly inflated by features with large raw variance. A feature measured in small units (large numerical values) will have artificially large variance and hijack the first PC.

Example: Assault has variance 6945 *vs. Murder's* 19*, on unscaled data PC1 is almost entirely Assault, not because assault is more important but because its units produce larger numbers. Dividing by 1000 would collapse its variance to near zero. The loading vector should not depend on an arbitrary unit choice.*



Standard practice for Scaling.

- **Different units**: always standardize each feature to mean 0, standard deviation 1 before PCA. This puts all features on equal footing and decomposes the **correlation matrix** $\mathbf{R}$ instead of the covariance matrix $\mathbf{S}$.
- **Same units**: standardization is optional, if raw variance is meaningful (e.g. gene expression counts where higher variance = genuine biological variation) you may want to preserve it.
- **When in doubt, scale**: unscaled PCA is sensitive to an arbitrary choice (units of measurement) that has nothing to do with data structure. Scaled PCA makes all features compete on equal footing.
- **No prior importance**: if you have no domain reason to believe one feature should matter more, scaling prevents high-variance features from hijacking PC1 purely due to units.

Standardizing replaces $\mathbf{S}$ *with* $\mathbf{R}$ *as the matrix being decomposed; eigenvectors of* $\mathbf{R}$ *reflect genuine correlation structure, not measurement scale. Eigenvectors of* $\mathbf{S}$ *reflect both structure and units, making them harder to interpret.*

6.7.1 Proportion of Variance Explained (PVE)

⋆ **Proportion of Variance Explained.** How much information is lost by keeping only M components? Quantified by the **PVE** of each component: what fraction of total data variance it captures.

Total variance (assuming centered data, mean zero per feature):

$$\text{Total Var} = \sum_{j=1}^{p} \text{Var}(X_j) = \sum_{j=1}^{p} \frac{1}{n} \sum_{i=1}^{n} x_{ij}^2$$

This is the sum of all feature variances, the total spread in the original p-dimensional space.

Variance explained by the m-th PC: $\boxed{\text{Var}(Z_m) = \frac{1}{n}\sum_{i=1}^{n} z_{im}^2 = \lambda_m = \frac{d_m^2}{n}}$

where $z_{im} = \varphi_m^T x_i$ are the scores and λ_m is the m-th eigenvalue of $\mathbf{S}$.
And d_m is the m-th diagonal entry of $\mathbf{D}$ in $\mathbf{X} = \mathbf{UDV}^T$ (the m-th **singular value** of $\mathbf{X}$, ordered $d_1 \geq d_2 \geq \cdots \geq d_p \geq 0$), and n is the sample size. The equality $\lambda_m = d_m^2/n$ follows from $\mathbf{S} = \frac{1}{n}\mathbf{X}^T\mathbf{X} = \frac{1}{n}\mathbf{VD}^2\mathbf{V}^T$, eigenvalues of $\mathbf{S}$ are exactly the squared singular values of $\mathbf{X}$ divided by n.

PCA partitions total variance exactly: $\boxed{\sum_{j=1}^{p} \text{Var}(X_j) = \sum_{m=1}^{M} \text{Var}(Z_m), \quad M = \min(n-1, p)}$

No variance is created or destroyed: PCA is a rotation. The total variance in the original p features equals the total variance across all M components. This identity is what makes PVE a valid fraction.

Expanding both sides explicitly, numerator is variance of PCm scores, denominator is total raw data variance across all features and observations:

$$\text{PVE}_m = \frac{\text{Var}(Z_m)}{\sum_{k=1}^{M}\text{Var}(Z_k)} = \frac{\frac{1}{n}\sum_{i=1}^{n} z_{im}^2}{\sum_{j=1}^{p}\sum_{i=1}^{n} x_{ij}^2} \quad \Rightarrow \quad \sum_{m=1}^{M}\text{PVE}_m = \frac{\sum_{m=1}^{M}\text{Var}(Z_m)}{\sum_{k=1}^{M}\text{Var}(Z_k)} = 1$$

Summing over all M components confirms PVEs are a valid partition of total variance.

The denominator is the same for every component: it is a fixed constant (total data variance). So PVE_m is just how large λ_m is relative to the total eigenvalue sum. All PVEs sum to 1 because PCA is a rotation: it redistributes, not creates or destroys, variance.

⋆ **PVE of the m-th component.** The fraction of total variance captured by PCm:

$$\boxed{\text{PVE}_m = \frac{\sum_{i=1}^{n} z_{im}^2}{\sum_{j=1}^{p}\sum_{i=1}^{n} x_{ij}^2} = \frac{\lambda_m}{\sum_{j=1}^{p}\lambda_j} \in [0,1]}$$

Numerator: variance of PCm scores. Denominator: total variance across all features. PVEs sum to one: $\sum_{m=1}^{M}\text{PVE}_m = 1$.

PVE_m tells you: if I keep only PCm, what fraction of the original data's spread do I retain? PC1 always has the largest PVE by construction: it was the direction of maximum variance. Each subsequent PC has smaller PVE since it explains residual variance in an orthogonal subspace.

Scree plot and cumulative PVE. Plot PVE_m vs. m (scree plot) and $\sum_{m'=1}^{m}\text{PVE}_{m'}$ vs. m (cumulative). Use the **elbow** of the scree plot to choose M: the point where additional components give diminishing returns in explained variance.

The scree plot is the PCA analog of the $W(K)$ plot in clustering; both decrease monotonically and both require an elbow heuristic rather than a sharp criterion. The cumulative PVE is more interpretable: "I need $M = 2$ components to explain 86% of variance" is a clear statement.

⋆ **Choosing M: how many components to keep.**

- **Unsupervised PCA**: no CV available, there is no response Y to validate against. Use the **scree plot elbow**: keep components up to where the PVE curve bends, adding more gives little gain. Inherently subjective.
- **Supervised PCA** (e.g. PCA + regression/classification): M becomes a **tuning parameter**, treat it like any other hyperparameter and select via **cross-validation** on the downstream prediction error. CV is valid here because there is a response Y to evaluate against.

The reason CV cannot select M in pure unsupervised PCA is the same reason it cannot select K in clustering, there is no held-out prediction to evaluate. The scree plot elbow is the best available heuristic: look for where the marginal gain in explained variance drops sharply, and stop there.

Both are unsupervised and operate only on $\mathbf{X}$, but they answer different questions: PCA asks "where does the data live?" clustering asks "who belongs together?". They are complementary: PCA scores are often fed into clustering as a denoised, lower-dimensional input.

6.8 Fisher's Linear Discriminant Analysis (LDA)

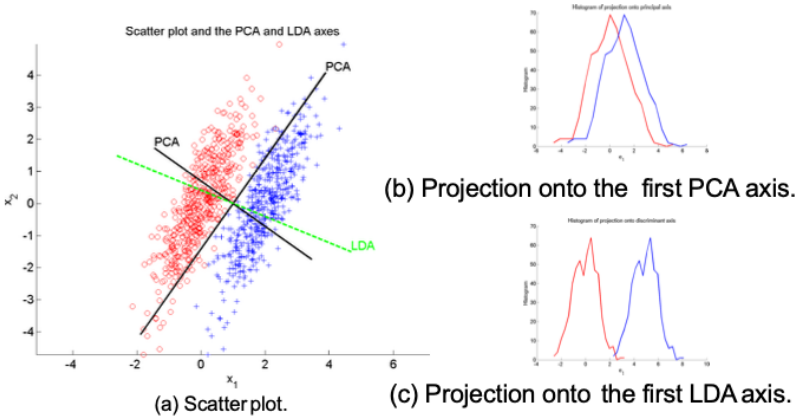
⋆ **Fisher's LDA.** The **supervised analog of PCA**. While PCA finds directions of maximum variance (ignoring class labels), **LDA finds directions of maximum class separation, projections that make different classes as distinguishable as possible.**

Setup. Given $x_1, \ldots, x_n \in \mathbb{R}^p$ divided into two classes $\mathcal{D}_1$ and $\mathcal{D}_2$, find a projection direction $\beta \in \mathbb{R}^p$ such that the scalar projections:

$$\boxed{y = \beta^T x}$$

produce **maximum separation between the two classes.** Here $y = \beta^T x \in \mathbb{R}$ is the **projected score**: the coordinate of observation x along direction β. It is the 1D compression of x used for classification, the LDA analog of a PCA score $z_{i1} = \varphi_1^T x_i$.

Each observation x_i maps to a scalar $y_i = \beta^T x_i$; same dot-product structure as PCA scores, but β is solved for discrimination not variance.



	PCA	LDA
Type	Unsup.	Sup.
Goal	Max variance	Max separation
Uses Y?	No	Yes
Output	Spread	Discrimination

⋆ **Fisher's Criterion.** The best projection maximizes the difference between class means relative to within-class variance, a **signal-to-noise ratio** in the projected space. Three equivalent forms, from intuitive to matrix:

$$\boxed{J(\beta) = \frac{|\tilde{m}_1 - \tilde{m}_2|}{\tilde{s}_1^2 + \tilde{s}_2^2} = \frac{(\mu_1 - \mu_2)^2}{\sigma_1^2 + \sigma_2^2} = \frac{\beta^T \mathbf{S}_B \beta}{\beta^T \mathbf{S}_W \beta}} = \frac{\text{distance between projected class centers}}{\text{spread of each class around its own center}}$$

Building blocks, from projection to criterion. Starting from $y = \beta^T x$ (projection of $x \in \mathbb{R}^p$ to scalar):

- **Projected class means:** $\tilde{m}_k = \frac{1}{|\mathcal{D}_k|} \sum_{x \in \mathcal{D}_k} \beta^T x = \beta^T \bar{x}_k$ where $\bar{x}_k = \frac{1}{|\mathcal{D}_k|} \sum_{i \in \mathcal{D}_k} x_i$ is the class mean in $\mathbb{R}^p$. After projection, $\tilde{m}_k \in \mathbb{R}$ tells you where class k lands on the line.

- **Numerator: between-class separation:** $|\tilde{m}_1 - \tilde{m}_2| = |\beta^T(\bar{x}_1 - \bar{x}_2)|$ is the **Euclidean distance** between the two projected class means on the line. In $\mathbb{R}^p$ the class means were separated by $\|\bar{x}_1 - \bar{x}_2\|_2$; projection collapses this to one scalar. In matrix form: $(\mu_1 - \mu_2)^2 = \beta^T \underbrace{(\bar{x}_1 - \bar{x}_2)(\bar{x}_1 - \bar{x}_2)^T}_{\mathbf{S}_B} \beta$. **Want large.**

- **Denominator: within-class spread:** $\tilde{s}_k^2 = \sum_{x \in \mathcal{D}_k} (\beta^T x - \tilde{m}_k)^2 = \beta^T \mathbf{S}_k \beta$ is the unnormalized variance of $\beta^T x$ within class k. Total: $\tilde{s}_1^2 + \tilde{s}_2^2 = \beta^T \mathbf{S}_W \beta$. **Want small.**

- **Between-class scatter matrix:** $\mathbf{S}_B = (\bar{x}_1 - \bar{x}_2)(\bar{x}_1 - \bar{x}_2)^T \in \mathbb{R}^{p \times p}$: rank-1 matrix encoding the direction and magnitude of class mean separation.

- **Within-class scatter matrix:** $\mathbf{S}_W = \sum_{k=1}^{2} \sum_{i \in \mathcal{D}_k} (x_i - \bar{x}_k)(x_i - \bar{x}_k)^T \in \mathbb{R}^{p \times p}$: total spread of all points around their own class mean.

Every term lives in $\mathbb{R}$ after projecting via $\beta^T x$: the p-dimensional problem collapses entirely to scalars. Maximizing $J(\beta)$ simultaneously pushes the two class centers apart and keeps each class tight internally. The denominator prevents the trivial solution of making $\|\beta\|$ arbitrarily large, same role as the constraint $\|\varphi\| = 1$ in PCA.

Optimization problem and solution. $J(\beta)$ is a **generalized Rayleigh quotient**:

$$\boxed{\underset{\beta}{\text{maximize}}\ J(\beta) = \frac{\beta^T \mathbf{S}_B \beta}{\beta^T \mathbf{S}_W \beta}}$$

Solved by the leading generalized eigenvector of $\mathbf{S}_W^{-1}\mathbf{S}_B$, giving the closed-form solution:

$$\boxed{\beta^* = \mathbf{S}_W^{-1}(\bar{x}_1 - \bar{x}_2)}$$

The **optimal direction is the difference of class means de-correlated by within-class scatter**: if classes have equal spread $\mathbf{S}_W \propto \mathbf{I}$, this reduces to simply pointing from one class mean to the other.

Connection to PCA, same math, different objective. Both PCA and LDA maximize a Rayleigh quotient; the only difference is what goes in the numerator and denominator:

	Maximize	Numerator	Solved by
PCA	$\frac{\beta^T \mathbf{S} \beta}{\beta^T \beta}$	total variance	largest eigenvector of $\mathbf{S}$
LDA	$\frac{\beta^T \mathbf{S}_B \beta}{\beta^T \mathbf{S}_W \beta}$	between-class scatter	largest eigenvector of $\mathbf{S}_W^{-1}\mathbf{S}_B$

PCA finds directions where the data spreads most, ignoring class labels entirely. LDA finds directions where classes separate most, using labels to define what "separate" means. Same Rayleigh quotient structure, same SVD/eigendecomposition machinery, different matrices being decomposed.

*$\mathcal{D}_1, \mathcal{D}_2$ are **given** from training labels: LDA is supervised so class assignments are known upfront, no iteration needed. This is what separates LDA from K-means: K-means must solve for both assignments and centers simultaneously; LDA only solves for β since the class partition is handed to you by the labels. Once β^* is learned, new unlabeled points are projected via $y = \beta^{*T}x$ and classified by a threshold; no labels needed at test time.*

Solution.

$$\boxed{\beta^* = \mathbf{S}_W^{-1}(\bar{x}_1 - \bar{x}_2)}$$

The optimal direction is the difference of class means de-correlated by the within-class scatter. After projecting, classify x to class 1 if $\beta^{*T}x >$ threshold, else class 2.

Key insight from the figure. PCA finds the direction of maximum spread of all data; it may mix classes if they happen to vary together. LDA finds the direction where the two class histograms overlap least. When classes differ in mean but not total variance, LDA outperforms PCA dramatically.

LDA as pre-processing. Once $\mathbf{X}$ is projected from $\mathbb{R}^p$ to a lower-dimensional subspace via PCA or LDA, any standard classifier (logistic regression, SVM, KNN) can be trained on the reduced representation $\mathbf{Z} = \mathbf{X}\mathbf{\Phi}$ or $\mathbf{Y} = \mathbf{X}\mathbf{B}$. This pipeline (reduce then classify) is standard in high-dimensional settings where $p \gg n$.

PCA + classifier: unsupervised reduction then supervised classification: reduction ignores labels. LDA + classifier: supervised reduction then supervised classification: reduction uses labels to find discriminative directions first. LDA is strictly more powerful for classification when class structure is meaningful, but requires labeled data for the reduction step.

Chapter 7

Semi-Supervised and Active Learning

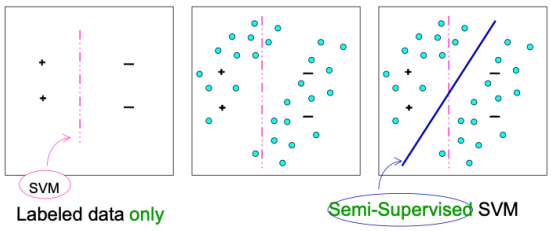
7.1 Semi-Supervised Learning (SSL)

⋆ **Semi-Supervised Learning (SSL). SSL** bridges supervised and unsupervised learning by combining a small **labeled set** $L = \{(x_i, y_i)\}$ **with a large unlabeled set** $U = \{x_j\}$:

$$\text{SSL} = \text{Supervised Learning} + \text{unlabeled data} \iff \text{Unsupervised Learning} + \text{labeled data}.$$

Motivation for SSL.

- **Pragmatic:** Unlabeled data is cheap to collect at scale (e.g., web crawling yields millions of unannotated pages while annotation requires human effort).
- **Methodological:** Unlabeled data reveals the underlying data distribution, helping locate **class separation boundaries** more accurately than L alone.



Key assumption. Examples from the same class follow a **coherent distribution**. Under this assumption, unlabeled points expose the geometry of each class cluster, allowing the decision boundary to be placed in **low-density** regions rather than arbitrarily between the few labeled points.

Without unlabeled data, the boundary is only constrained by the labeled points and can be placed almost anywhere. Adding unlabeled mass reveals where the classes actually sit, shifting the boundary into the gap between clusters.

⋆ **Inductive vs. Transductive SSL.**

- **Transductive:** Uses U to produce labels only for those specific points in U; the output is **not** a general classifier $f : \mathcal{X} \to \mathcal{Y}$. Cannot predict on any new point outside U.
- **Inductive:** Uses U to produce both labels for U *and* a general classifier that generalizes to new, unseen data beyond U.

Think of transductive as memorizing answers for a specific exam: it labels exactly the unlabeled points it sees, but has no mechanism to handle new ones. Inductive SSL is the stronger goal: it uses the unlabeled data to learn a better decision rule that works everywhere, not just on U. In practice, most classifier-based SSL methods (Yarowsky, Co-Training) are inductive.

⋆ **Two Algorithmic Approaches.**

- **Classifier-based methods (Self-Training):** Start from an initial classifier trained on L; iteratively apply it to U, add high-confidence predictions back to L, and retrain. Includes the

Yarowsky algorithm, **Co-Training**, and their combination.

- **Data-based methods:** Discover the intrinsic geometry of $L \cup U$ (manifold structure, density, similarity graph) and exploit it to constrain or regularize the classifier. Examples: Manifold Regularization, Harmonic Mixtures, Information Regularization (not covered here).

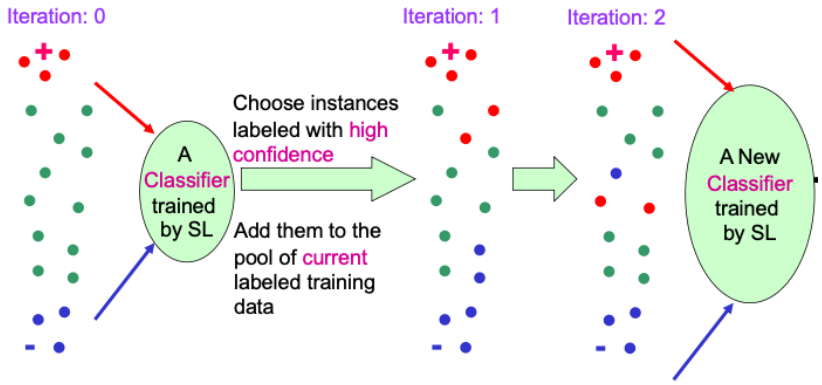
Classifier-based methods are wrappers around any supervised learner and are easy to implement. Data-based methods require modeling the full data geometry but can be more principled when the manifold assumption holds.

7.2 Self-Training: Yarowsky Algorithm and Refinement

⋆ **Self-Training / Yarowsky Algorithm (Bootstrap).** Method that iteratively trains a classifier on L, predicts labels for U, and promotes the most confidently labeled instances from U into L before retraining.
Starting from L, iteratively expand the labeled set using the model's own high-confidence predictions on U:

1. Train a supervised classifier f on current L.
2. Apply f to unlabeled set U; identify the most **confidently** classified instances.
3. Move those instances (with their predicted labels) from U into L.
4. Retrain f on the enlarged L. Repeat until convergence or $U = \emptyset$.

Each iteration, the classifier teaches itself: it labels the easy cases first and gradually incorporates harder ones. The key bet is that high-confidence predictions are likely correct, so adding them to L genuinely improves the next classifier rather than injecting noise.



Refinement. A practical extension: **rather than treating pseudo-labeled data equally to true labeled data, reduce the weight assigned to instances drawn from U during retraining.** This preserves the higher reliability of the original L.

Pseudo-labels are noisier than ground-truth labels. Downweighting them lets the model benefit from the extra coverage of U while preventing confident-but-wrong labels from dominating the loss.

Advantages and Disadvantages of Self-Training.

- **Advantages:** Simplest SSL method; a **wrapper** that wraps around any existing supervised classifier without modifying it; widely used in practice (e.g., NLP tasks).
- **Disadvantages:** **Error reinforcement**, early mistakes get added to L and compound across iterations. Heuristic fix: "un-label" an instance and return it to U if its confidence later drops below a threshold.

7.3 Co-Training

⋆ **Co-Training.** If two independent views of the same instance each contain enough information to predict the label alone, then a classifier confident on view 1 can reliably label instances for a classifier learning on view 2, and vice versa, allowing both to improve each other iteratively using unlabeled data.

Co-Training. Two classifiers C_1, C_2 are trained on two independent **views** x_1, x_2 of the same instance $x = (x_1, x_2)$, and iteratively teach each other by promoting their most confident predictions on U into L.

Independence assumption. The two views are **conditionally independent** given the label y:

$$P(x_1 \mid x_2, y) = P(x_1 \mid y) \qquad \text{and} \qquad P(x_2 \mid x_1, y) = P(x_2 \mid y).$$

Independence means each view carries complementary information about the label. This reduces the probability that both classifiers agree on an incorrect pseudo-label simultaneously, making co-labeling more reliable than a single model.

Consistency assumption. There exist classifiers c_1, c_2 such that the **optimal classifier c^{opt} satisfies:**

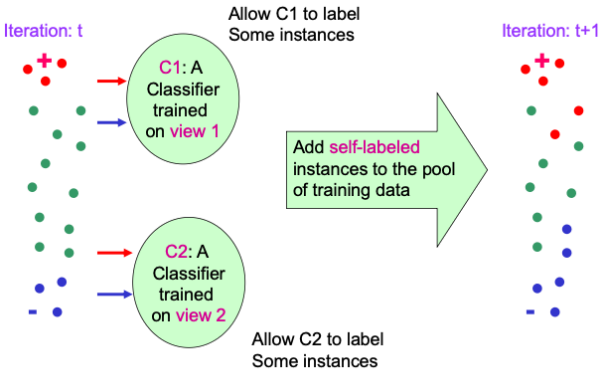
$$\boxed{\exists\, c_1, c_2 : c^{opt}(x) = c_1(x_1) = c_2(x_2).}$$

Either view alone is sufficient to determine the correct label. This guarantees that a classifier trained on x_1 alone and one trained on x_2 alone should agree on the true label; so disagreement signals uncertainty, and agreement signals confidence.

Co-Training algorithm (per iteration t).

1. Train C_1 on view x_1 of L; train C_2 on view x_2 of L.
2. Each classifier labels instances in U with high confidence.
3. Self-labeled instances are added to the pool L.
4. Retrain both classifiers on the enlarged L. Repeat.

C_1 and C_2 teach each other: C_1 labels some unlabeled instances based on view 1, C_2 uses those labels (on view 2) to improve itself, and vice versa. The independence assumption is what makes each teacher's signal genuinely informative to the other.



Example, Webpage classification. Task: classify pages as *student* or *professor*.

- **View 1** (x_1): page content (words, text).
- **View 2** (x_2): hyperlinks pointing to the page.

Both views are sufficient and roughly independent given the true class.

⋆ **Yarowsky + Co-Training.** A hybrid: runs like the Yarowsky algorithm but **switches the active view** at each iteration. Key properties:

- Uses **all** unlabeled data U (probabilistically labeled) for training, not just the most confident subset.
- In each iteration, only **one** classifier labels the data (alternating between C_1 and C_2).
- Has been applied successfully with **neural networks** and **SVMs**.

Switching views each round prevents one classifier from dominating. Using all of U probabilistically (rather than hard-labeling only the confident instances) provides a richer training signal and reduces the risk of error reinforcement from early hard decisions.

⋆ **General Multiview Learning.** A generalization of Co-Training to more than two views or models. Train multiple **diverse** models on L; instances in U on which **most models agree** are promoted to L:

$$x \in U \xrightarrow{\text{consensus}} L \quad \text{if most classifiers assign the same label to } x.$$

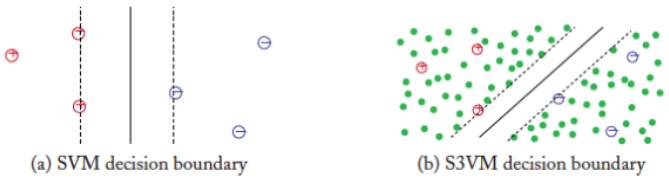
Requiring agreement from a majority of diverse models is a stronger confidence signal than a single model's output. An instance that fools one model is unlikely to fool many independently trained ones, so consensus-based promotion is more robust to individual classifier errors.

7.4 Semi-Supervised SVM (S3VM)

⋆ **Semi-Supervised SVM (S3VM).** Extends the standard SVM by maximizing the **margin** over both L and U jointly. The decision surface is placed in **low-density** regions of the full data distribution rather than just separating the few labeled points. **Key assumptions:**

- Classes are **well-separated**: a low-density gap exists between them.
- The class proportion on each side of the boundary in U should be **consistent** with the labeled proportion in L.

A standard SVM ignores U entirely and may place the boundary through a dense unlabeled region. S3VM uses unlabeled points to "see" where the data actually lives, pushing the margin into the natural gap between clusters. If no such gap exists, S3VM offers little benefit over a standard SVM.



Standard SVM on L only minimizes:

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i \quad \text{s.t.} \quad y_i(w^\top x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0.$$

S3VM extends this to $L \cup U$ by assigning **pseudo-labels** $\tilde{y}_j \in \{-1, +1\}$ to unlabeled points and jointly optimizing:

$$\boxed{\min_{w,b,\xi,\tilde{y}} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i + C^*\sum_{j=1}^{m}\tilde{\xi}_j}$$

$$\text{s.t.} \quad y_i(w^\top x_i + b) \geq 1 - \xi_i, \quad \tilde{y}_j(w^\top x_j + b) \geq 1 - \tilde{\xi}_j, \quad \xi_i, \tilde{\xi}_j \geq 0$$

where C controls labeled slack and C^* controls unlabeled slack. The decision surface is placed in **low-density** regions by forcing unlabeled points to also lie outside the margin.

Class balance constraint. To prevent degenerate solutions (all $\tilde{y}_j = +1$), one can add:

$\frac{1}{m}\sum_{j=1}^{m}\tilde{y}_j \approx \frac{1}{n}\sum_{i=1}^{n} y_i$ enforcing that the **class proportion** in U matches that observed in L.

S3VM is non-convex due to the discrete pseudo-label variables $\tilde{y}_j$. In practice it is solved by continuation methods: start with small C^* (unlabeled points barely constrained), gradually increase it, and switch pseudo-labels when a labeled point crosses the margin. If no low-density gap exists between classes, the margin constraint on U offers no useful signal.

7.5 Cluster-and-Label Approach

⋆ **Cluster-and-Label Approach.** A **data-based** SSL method resting on the assumption that **cluster boundaries coincide with decision boundaries**. Algorithm:

1. Cluster all of $L \cup U$ together (e.g., K-means).
2. For each cluster, train a local classifier using only the **labeled points** within that cluster.
3. Use that local classifier to label all unlabeled points in the same cluster.
4. Train a final global model on the now fully labeled dataset.

The idea is that points in the same cluster are likely in the same class, so a classifier fit on a cluster's labeled subset can reliably propagate labels to its unlabeled members. If the cluster-boundary assumption fails, clusters are mixed-class then approach breaks down and can introduce systematic labeling errors into the final model.

⋆ **Cluster-and-Label as a Preprocessing Step.** use **unsupervised clustering** on $L \cup U$ to discover geometric structure in the data **before** supervised learning. The cluster membership then acts as a prior for label propagation and prediction.

1. **Cluster** all of $L \cup U$ **without** considering labels, purely based on feature similarity. Obtain K clusters $S_1, \ldots, S_K$.
2. **Build a local model** f_k for each cluster S_k, trained only on the labeled points $\{(x_i, y_i) : x_i \in S_k \cap L\}$.
3. **Prediction:** given a test point x^*, first determine which cluster S_k it belongs to (e.g., nearest centroid), then apply the local model f_k for that cluster:

$$\hat{y}^* = f_k(x^*), \qquad k = \arg\min_j \|x^* - \mu_j\|^2.$$

The intuition is that nearby points in feature space are likely from the same class. By clustering first, we partition the input space into coherent regions, then fit a specialized classifier per region using only the labeled points that fall there. A test point inherits the decision rule of its local neighborhood rather than a global model that may be distorted by the overall class imbalance or complex global geometry. This works well when clusters align with class boundaries, and fails when they do not.

7.6 Active Learning

⋆ **Active Learning.** Unlike **passive supervised learning**, where the learner receives a **random sample** of labeled instances from an oracle, an **active learner** inspects U itself, selects the most **informative** unlabeled instances, and queries the oracle only for those:

$$\text{Passive:} \quad U \xrightarrow{\text{random sample}} \text{oracle} \xrightarrow{\langle x_i, y_i \rangle} \text{learner}$$

$$\text{Active:} \quad U \xrightarrow{\text{inspect}} \text{learner} \xrightarrow{\langle x_i, ? \rangle} \text{oracle} \xrightarrow{\langle x_i, y_i \rangle} \text{learner}$$

Not all labeled examples are equally useful. A random label wastes budget on easy, already-certain instances. An active learner spends its budget only where uncertainty is highest, achieving better accuracy with far fewer queries than passive learning.

Active Learning loop (per round).

1. Maintain a **current model** f_t trained on all labeled data seen so far.
2. Use f_t and a **query selection strategy** to rank unlabeled instances in U by **informativeness**.
3. Query the oracle for the label(s) of the most informative instance(s).
4. Add the newly labeled example(s) to L, retrain f_{t+1}, repeat until **budget** is exhausted.

⋆ **Types of Active Learning.**

- **Stream-Based:** Instances arrive one at a time from a continuous source. For each x_i, the learner decides to **query** its label or **ignore** it and move on. No access to the full pool.
- **Pool-Based:** A large pool U is available upfront. The learner ranks **all** instances by informativeness and queries the most informative one(s). More common in practice.

Stream-based is suited to real-time data pipelines where you cannot revisit examples. Pool-based is more powerful since global ranking over all of U finds the single best query at each round rather than making a local accept/reject decision.

7.7 Query Selection Strategies

⋆ **Query Selection Strategies.** Every active learning algorithm requires a **query selection strategy**, a scoring function $s(x)$ that measures how informative querying x would be. Key strategies:

- **Uncertainty Sampling**: query the instance the current model is most uncertain about.
- **Query By Committee (QBC)**: query the instance with maximum **disagreement** among a committee of models.
- **Expected Model Change**: query the instance whose label would change the model the most.
- **Expected Error Reduction**: query the instance that reduces generalization error the most.
- **Variance Reduction**: query the instance that maximally reduces model variance (via Fisher information).
- **Density Weighted Methods**: weight informativeness by similarity to the rest of U, penalizing outliers.

All strategies share the same goal: spend the labeling budget on instances that teach the model the most. They differ in what "informative" means: uncertainty, disagreement, model sensitivity, or error reduction, and in computational cost.

7.7.1 Uncertainty Sampling and Query By Committee

⋆ **Uncertainty Sampling.** Query the instance the current model f_t is **most uncertain** about. Two common uncertainty measures:

- **Margin-based (SVM):** query x^* with smallest distance to the decision hyperplane:

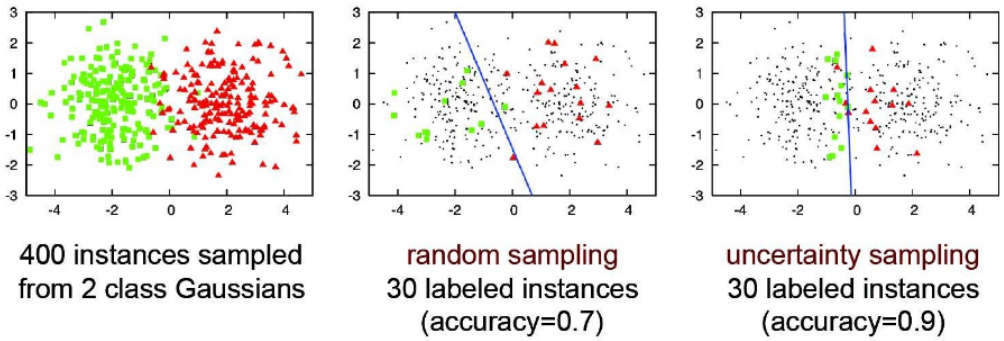
$$x^* = \arg\min_{x \in U} \; |w^\top x + b|$$

  Points close to the hyperplane are hardest to classify, the model is nearly 50/50 between classes.
- **Probabilistic (Logistic Regression, etc.):** query x^* whose posterior is closest to uniform:

$$x^* = \arg\min_{x \in U} \; \max_y \; P(y \mid x)$$

  A confident model gives $P \approx 1$ for one class; an uncertain one gives $P \approx 0.5$ for all classes.

With the same budget of 30 labeled instances, uncertainty sampling (accuracy = 0.9) substantially outperforms random sampling (accuracy = 0.7) because it concentrates labels near the decision boundary where they are most informative.



⋆ **Query By Committee (QBC).** Maintain a **committee** of M diverse models $\{f_1, \ldots, f_M\}$, all trained on the current L. Each model votes its prediction on every $x \in U$; query the instance with **maximum disagreement**:

$$x^* = \arg\max_{x \in U} \; D(f_1(x), \ldots, f_M(x))$$

where D is a disagreement measure (e.g., vote entropy). After querying, **all** committee members are retrained on the updated L.

Committee construction. Two common approaches:
(1) sample diverse models from the posterior over model parameters;
(2) use **ensemble methods** such as bagging or boosting to produce diverse classifiers from the same L.
A single model's uncertainty can be misleading, it might be uncertain for the wrong reason (e.g., a poorly initialized model). A committee provides a more robust signal: if many independently trained models disagree, the instance is genuinely hard and worth labeling.

7.7.2 Outliers and Model-Impact Methods

⋆ **Effect of Outliers on Uncertainty Sampling and QBC.** Both methods score instances by model uncertainty or disagreement, but an **outlier** (an atypical, isolated point far from the data distribution)

can also appear highly uncertain or cause high disagreement, even though labeling it provides almost no generalizable information.

An outlier near the decision boundary looks maximally uncertain, but it represents no other points in U,labeling it shifts the boundary for one irrelevant region while wasting the entire query budget for that round. This is the fundamental weakness of pure uncertainty-based methods.

Robust alternative: model-impact methods. Instead of measuring the model's **confidence on** x, measure how much a labeled x would **change the model itself**. The instance that maximally perturbs the model is the most informative:

$$x^* = \arg\max_{x \in U} \ \Delta(f, x)$$

where $\Delta(f, x)$ quantifies model change after including x (various instantiations below).

⋆ **Expected Model Change.** Select the instance whose inclusion causes the **largest change** in model parameters, measured by the gradient of the loss $\mathcal{L}$ w.r.t. parameters θ:

$$x^* = \arg\max_{x \in U} \ \mathbb{E}_y[\|\nabla_\theta \mathcal{L}(\theta; x, y)\|]$$

Since the true label is unknown, the expectation is computed using the **expected labels**, the current model's predicted probabilities. For binary classification:

$$\boxed{\text{Average model change} = \Delta E^+ \cdot \Pr(+) + \Delta E^- \cdot \Pr(-)}$$

where ΔE^+ and ΔE^- are the model changes if the instance were labeled $+1$ or -1 respectively, weighted by the current model's confidence $\Pr(+)$ and $\Pr(-)$. A large gradient means the model would update substantially: the instance is highly informative.

⋆ **Expected Error Reduction.** Select the instance that minimizes **expected generalization error** on the remaining unlabeled pool after being labeled:

$$x^* = \arg\min_{x \in U} \ \mathbb{E}_y \left[\sum_{x' \in U \setminus \{x\}} \mathcal{L}(f_{L \cup \{(x,y)\}}, \ x') \right]$$

The expectation over y uses the current model's predicted label probabilities $P(y \mid x)$ as a proxy for the unknown true label.

⋆ **Variance Reduction.** Select the instance that **minimizes model variance** the most; equivalently, **maximizes Fisher information**:

$$x^* = \arg\min_{x \in U} \ \operatorname{tr}\left(\mathcal{I}(\theta)^{-1}\right) \quad \text{or} \quad x^* = \arg\min_{x \in U} \ \det\left(\mathcal{I}(\theta)^{-1}\right)$$

where $\mathcal{I}(\theta) = -\mathbb{E}[\nabla^2_\theta \log p(y \mid x, \theta)]$ is the **Fisher information matrix**, computed from the log-likelihood. Minimizing the trace or determinant of $\mathcal{I}^{-1}$ reduces overall parameter uncertainty.

⋆ **Density Weighted Methods.** Weight the informativeness score $s(x)$ of any strategy by the **average similarity** of x to the rest of U:

$$x^* = \arg\max_{x \in U} \ s(x) \cdot \left(\frac{1}{|U|} \sum_{x' \in U} \operatorname{sim}(x, x') \right)^\beta$$

where $\operatorname{sim}(\cdot, \cdot)$ is a similarity measure (e.g., RBF kernel) and $\beta \geq 0$ controls the density penalty. An **outlier** has low average similarity to U and therefore receives a low overall score regardless of its raw informativeness $s(x)$.

Density weighting fixes the outlier problem: an isolated point that looks uncertain is down-weighted because it is dissimilar to most of U: labeling it would only help classify that one atypical region. A representative point near the center of a dense cluster gets a high density weight, ensuring the query budget is spent on instances that generalize to many similar unlabeled points.

7.7.3 Active Learning with Selective Sampling

⋆ **Active Learning with Selective Sampling.** A **stream-based** active learning algorithm using a margin-based classifier. Instead of querying every instance, it queries probabilistically, the closer an instance is to the decision boundary, the more likely it is queried.

Algorithm. Input: parameter $b > 0$ controlling query aggressiveness.

1. For $n = 1 : N$, receive x_n, compute margin score $p_n = w^\top x_n + b$.
2. Predict $\hat{y}_n = \text{sign}(p_n)$.
3. Draw Bernoulli random variable $Z \in \{0, 1\}$ with query probability:

$$\boxed{P(Z = 1) = \frac{b}{b + |p_n|}} \quad \textbf{probability that instance } x_n \textbf{ gets queried}$$

4. If $Z = 1$: query the true label y_n from the oracle and update w.
   If $Z = 0$: ignore x_n, do not update w.

- $|p_n|$ is the **absolute margin** of x_n: how far the point lies from the decision hyperplane.
- **Large** $|p_n| \Rightarrow$ small query probability: the model is already confident, no need to query.
- **Small** $|p_n| \Rightarrow$ query probability $\to 1$: the point is near the boundary, maximally uncertain: always query.
- b controls aggressiveness: large $b \Rightarrow$ query more instances; small $b \Rightarrow$ query only the most uncertain.

This is the stream-based analog of uncertainty sampling: instead of ranking all of U and picking the best, it makes an online accept/reject decision per instance using a randomized threshold. The Bernoulli draw means no instance is ever permanently ignored: even a confident instance has a small chance of being queried, preventing the model from becoming overconfident in any region.

Open questions and limitations.

- **Dataset reuse**: can an actively labeled dataset be reused to train a new, different model? Generally not directly, the sampling is tied to the original model's uncertainty, not the new model's needs.
- **Sampling bias**: the actively labeled dataset **does not reflect the true training/test data distribution**, it overrepresents instances near the decision boundary. Consequences: a model trained on this biased L may generalize poorly; importance weighting by $1/P(Z = 1)$ can partially correct for this.
- **Active learning for regression**: uncertainty and informativeness are less obvious without discrete class boundaries, variance reduction and Fisher information criteria extend naturally to the regression setting.

The sampling bias is the fundamental tension of active learning: you query where the model is uncertain, but uncertainty is not the same as representativeness. A region that is uncertain for the current model may be easy for a different model architecture, making actively labeled datasets partially model-specific and not freely transferable.

Chapter 8

Neural Networks and Deep Learning

⋆ **Neural Networks and Deep Learning.** A **neural network** is a computational model built from layers of interconnected units (neurons), each applying a weighted sum followed by a nonlinear **activation function**, stacked so that each layer learns increasingly abstract representations of the input. **Deep learning** refers to neural networks with many such layers: deep architectures that can approximate virtually any function and are trained end-to-end via **backpropagation**.

8.1 The Perceptron

Notation. Throughout this lecture, matrices and vectors are denoted by **bold** letters. Vectors of functions (e.g. $\mathbf{f}$) are also bold and, unless otherwise stated, act on vectors **element-wise**, meaning the function is applied independently to each component.

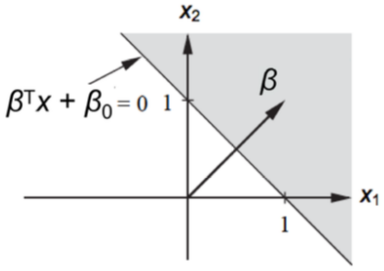
⋆ **Perceptron.** A binary classifier that uses a **linear prediction function** combined with a **step (threshold) activation function**:

$$f(\mathbf{x}) = \begin{cases} +1 & \boldsymbol{\beta}^T\mathbf{x} + \beta_0 \geq 0 \\ -1 & \boldsymbol{\beta}^T\mathbf{x} + \beta_0 < 0 \end{cases}$$

where $\mathbf{x} = (x_1, \ldots, x_p)^T \in \mathbb{R}^p$ is the feature vector, $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_p)^T$ are the **weights** (also called parameters), and β_0 is the **bias** (intercept). By convention:

- Classes are always encoded as $\{+1, -1\}$.
- Ties are broken in favor of the positive class: if $\boldsymbol{\beta}^T\mathbf{x} + \beta_0 = 0$ exactly, output $+1$.

Think of $\boldsymbol{\beta}^T\mathbf{x} + \beta_0$ as a signed score. The decision boundary is the hyperplane where the score is exactly zero. Points on the positive side get $+1$; points on the negative side get -1. The weights $\boldsymbol{\beta}$ are unknown: learning the perceptron means finding weights that best separate the two classes.



Structure of a perceptron. With $\beta_1 = 1$, $\beta_2 = 1$, $\beta_0 = -1$ and $p = 2$, the decision boundary is the line $x_1 + x_2 - 1 = 0$. The weight vector $\boldsymbol{\beta}$ always points **perpendicular to the boundary**, in the direction of the positive class. Points in the shaded region satisfy $\boldsymbol{\beta}^T\mathbf{x} + \beta_0 \geq 0$ and are classified as $+1$.

8.1.1 Learning the Weights

⋆ **Perceptron Learning Algorithm.** The weights $\boldsymbol{\beta}$ are unknown and must be learned from training data. The algorithm is **online**: it processes one training point at a time and updates the weights immediately upon misclassification.

Algorithm.

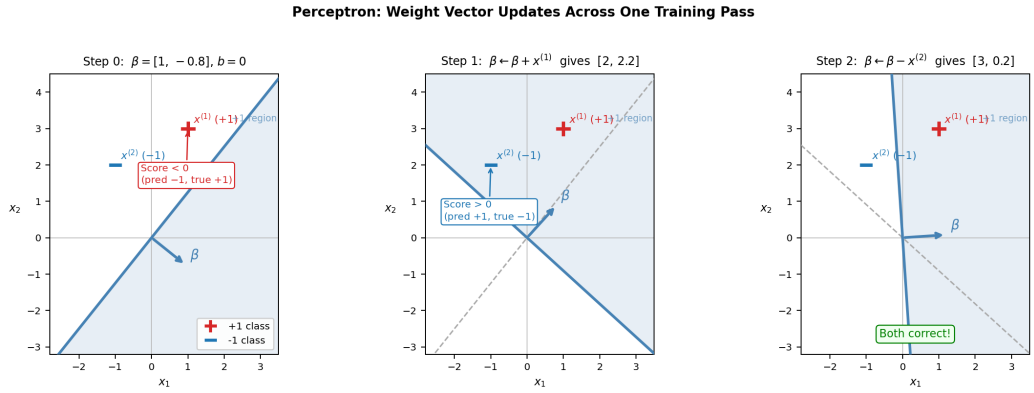
1. **Initialize** all weights $\boldsymbol{\beta}$ to zero (or randomly).
2. **Iterate** through the training data. For each instance $\mathbf{x}^{(i)}$:
   (a) **Classify** it using the current weights: compute $f(\boldsymbol{\beta}^T\mathbf{x}^{(i)} + \beta_0)$.
   (b) If prediction is **correct**: do nothing: leave the weights alone.
   (c) If prediction is **wrong**: apply the **update rule** to adjust the weights.
3. **Repeat** step 2 for a fixed number of iterations (epochs).

The intuition is Hebbian: if the classifier is wrong, push the weights in the direction that would have made the correct prediction. If the prediction was right, trust the current weights and move on.

What does the update rule do geometrically?

- If the classifier predicted -1 but should have been $+1$: the score $\boldsymbol{\beta}^T\mathbf{x} + \beta_0$ is currently negative but needs to be non-negative. **Move $\boldsymbol{\beta}$ toward $\mathbf{x}$** to increase the score.
- If the classifier predicted $+1$ but should have been -1: the score is positive but should be negative. **Move $\boldsymbol{\beta}$ away from $\mathbf{x}$** to decrease the score.

The weight vector $\boldsymbol{\beta}$ lives in the same space as the data points. Adding $\mathbf{x}$ to $\boldsymbol{\beta}$ rotates the decision boundary toward correctly classifying $\mathbf{x}$; subtracting $\mathbf{x}$ rotates it away.



⋆ **Perceptron Update Rule.** The full update rule for weights and bias after presenting training instance $\mathbf{x}^{(i)}$ with true label $y^{(i)} \in \{+1, -1\}$:

$$\boxed{\boldsymbol{\beta}^{(i+1)} = \boldsymbol{\beta}^{(i)} + \frac{1}{2}\left[y^{(i)} - f\left(\boldsymbol{\beta}^{T(i)}\mathbf{x}^{(i)} + \beta_0^{(i)}\right)\right]\mathbf{x}^{(i)}}$$

$$\boxed{\beta_0^{(i+1)} = \beta_0^{(i)} + \frac{1}{2}\left[y^{(i)} - f\left(\boldsymbol{\beta}^{T(i)}\mathbf{x}^{(i)} + \beta_0^{(i)}\right)\right]}$$

Define the **error** at step i as:

$$e^{(i)} = y^{(i)} - f\left(\boldsymbol{\beta}^{T(i)}\mathbf{x}^{(i)} + \beta_0^{(i)}\right)$$

Then both update rules compress to:

$$\boldsymbol{\beta}^{(i+1)} = \boldsymbol{\beta}^{(i)} + \tfrac{1}{2}\,e^{(i)}\,\mathbf{x}^{(i)}, \qquad \beta_0^{(i+1)} = \beta_0^{(i)} + \tfrac{1}{2}\,e^{(i)}$$

Three cases for $e^{(i)}$:

1. $y^{(i)} = f(\cdot)$ (correct prediction): $e^{(i)} = 0 \Rightarrow$ **no update**. Weights unchanged.

2. $y^{(i)} = +1$, $f(\cdot) = -1$ (missed positive): $e^{(i)} = +2$, so $\frac{1}{2}e^{(i)} = +1 \Rightarrow \boldsymbol{\beta}$ += $\mathbf{x}^{(i)}$. Weight vector **moves toward** $\mathbf{x}^{(i)}$, making the score $\boldsymbol{\beta}^T\mathbf{x}^{(i)} + \beta_0$ more positive.

3. $y^{(i)} = -1$, $f(\cdot) = +1$ (missed negative): $e^{(i)} = -2$, so $\frac{1}{2}e^{(i)} = -1 \Rightarrow \boldsymbol{\beta}$ −= $\mathbf{x}^{(i)}$. Weight vector **moves away** from $\mathbf{x}^{(i)}$, making the score more negative.

The factor $\frac{1}{2}$ (equivalently, a learning rate of $\alpha = 0.5$) controls the step size of each correction. The bias update $\beta_0^{(i+1)} = \beta_0^{(i)} + \frac{1}{2}e^{(i)}$ shifts the decision boundary up or down (parallel to itself) to correct misclassifications; it does not rotate the boundary, only translates it.

Bias update geometric intuition. The bias β_0 controls where the hyperplane $\boldsymbol{\beta}^T\mathbf{x} + \beta_0 = 0$ sits along the direction of $\boldsymbol{\beta}$. Increasing β_0 shifts the boundary in the direction that classifies more points as $+1$; decreasing it has the opposite effect. The weight vector $\boldsymbol{\beta}$ controls the **orientation** (rotation) of the boundary; β_0 controls its **position** (translation).

Example Let $\beta_1 = 1$, $\beta_2 = -1$, $\beta_0 = 0$, and suppose $\mathbf{x}^{(1)} = [0\ 1]^T$ with true label $y^{(1)} = +1$.

Compute the score: $\boldsymbol{\beta}^T\mathbf{x}^{(1)} + \beta_0 = (1)(0) + (-1)(1) + 0 = -1 < 0$, so $f = -1$. Error: $e^{(1)} = +1 - (-1) = 2$.

Weight update: $\boldsymbol{\beta}^{\text{new}} = [1,\ -1]^T + \frac{1}{2}(2)[0,\ 1]^T = [1,\ 0]^T$. Bias update: $\beta_0^{\text{new}} = 0 + \frac{1}{2}(2) = 1$.

The update rotated and shifted the boundary to give $\mathbf{x}^{(1)}$ a positive score: $[1, 0] \cdot [0, 1]^T + 1 = 1 > 0$. One step corrected this single misclassification.

Notation switch. The slides use $\boldsymbol{\beta}$ for weights following regression convention. From this point forward, the standard **neural network notation** is used instead:

- $\boldsymbol{\beta} \longrightarrow \mathbf{w}$ (**weight vector**)
- $\beta_0 \longrightarrow b$ (**bias**)

The perceptron becomes $f(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$ and the update rule becomes:

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} + \tfrac{1}{2}\, e^{(i)}\, \mathbf{x}^{(i)}, \qquad b^{(i+1)} = b^{(i)} + \tfrac{1}{2}\, e^{(i)}$$

Nothing about the math changes, only the symbol names. The $\mathbf{w}, b$ notation is universal in the deep learning literature and will be used for all networks going forward.

⋆ **Perceptron vs. SVC** Both are **linear classifiers**: they both find a separating hyperplane. The key difference is **how** they learn it:

- **SVC** is a **batch** algorithm: its optimization requires all n training points simultaneously to find the maximum-margin hyperplane. You cannot update it one point at a time.
- **Perceptron** is an **online** algorithm: it processes one point at a time and updates weights immediately. No need to see the full dataset before making a correction.

Online algorithms are useful when data arrives sequentially (e.g. streams) or when the dataset is too large to hold in memory. The price is that the perceptron has no margin guarantee: it just finds some *separating hyperplane, not the best one.*

8.1.2 Linear Separability and Convergence

⋆ **Linear Separability.** A dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$ is **linearly separable** if there exists a hyperplane $\mathbf{w}^T\mathbf{x} + b = 0$ that perfectly separates the two classes: all $+1$ points on one side, all -1 points on the other.

Perceptron Convergence Theorem.

- **If linearly separable:** the perceptron algorithm is **guaranteed to converge** in a finite number of steps: it will find weights $\mathbf{w}$ such that every training instance is correctly classified.
- **If not linearly separable:** the algorithm **never converges**: there will always be some misclassified points and the weights will keep updating forever.

Stopping criteria (non-separable case). Since the algorithm runs forever on non-separable data, a **stopping criterion** must be imposed:

- Run for a fixed maximum number of **iterations** (one update step per training point).
- Run for a fixed maximum number of **epochs** (one full pass through the training set = one epoch, containing N iterations).

One epoch is a single complete pass through the entire training set : all N training instances have been presented to the network exactly once.

Linear separability is the perceptron's fundamental assumption. When it holds, the algorithm is provably correct. When it doesn't, which is almost always in practice, you need either a stopping rule or a better model. This limitation directly motivates multi-layer networks.

8.1.3 Multi-Class Perceptron

⋆ **Multi-class Perceptron Architecture.** The binary perceptron uses a single hyperplane to split $\mathbb{R}^p$ into 2 regions. With S **hyperplanes,** $\mathbb{R}^p$ **is partitioned into up to** 2^S **regions,** each of which can represent a distinct class. The maximum number of representable classes is:

$$\boxed{K = 2^S}$$

Each additional hyperplane doubles the number of separable regions. Two hyperplanes in $\mathbb{R}^2$ create 4 quadrant-like regions, enough for 4 classes. This is the architectural basis: stack S binary perceptrons in parallel, one hyperplane each.

Class label encoding. Each class is encoded as a binary vector $\mathbf{y} \in \{-1, +1\}^S$ with S elements:

$$\boxed{\mathbf{y} = [y_1, y_2, \ldots, y_S]^T, \quad y_k \in \{-1, +1}$$

Each of the S perceptrons is responsible for predicting one element y_k of this label vector. A class is defined by a unique pattern of $+1$s and -1s across all S outputs.

Example with $S = 2$**,** $K = 4$ **classes.** Two perceptrons produce two binary outputs, giving $2^2 = 4$ distinct label vectors:

$$[+1, +1]^T, \quad [+1, -1]^T, \quad [-1, +1]^T, \quad [-1, -1]^T$$

Each pattern corresponds to one of the 4 regions carved out by the 2 hyperplanes.

The encoding is arbitrary: the class that gets $[+1, -1]^T$ vs. $[-1, +1]^T$ is just a labeling convention. What matters is that each class gets a unique binary string so the S perceptrons can jointly identify it.

⋆ **Weight Matrix and Bias Vector.** With S hyperplanes and p features, the weights are collected into a **weight matrix** $\mathbf{W}$ and **bias vector** $\mathbf{b}$:

$$\boxed{\mathbf{W} = [\mathbf{w}_1 \,|\, \mathbf{w}_2 \,|\, \cdots \,|\, \mathbf{w}_S] \in \mathbb{R}^{p\times S}, \qquad \mathbf{b} = [b_1,\, b_2,\, \ldots,\, b_S]^T \in \mathbb{R}^S}$$

The s-th column $\mathbf{w}_s \in \mathbb{R}^p$ is the weight vector of the s-th perceptron, and b_s is its bias. Each perceptron s computes the score $\mathbf{w}_s^T\mathbf{x} + b_s$ for its own hyperplane independently.

⋆ **Multi-class Perceptron Learning Rule.** Let $\mathbf{e}(i) \in \mathbb{R}^S$ be the **error vector** at step i: the vector of per-perceptron errors. The weight matrix and bias vector are updated simultaneously:

$$\boxed{\mathbf{W}(i+1) = \mathbf{W}(i) + \tfrac{1}{2}\,\mathbf{x}(i)\,\mathbf{e}^T(i)} \qquad \boxed{\mathbf{b}^T(i+1) = \mathbf{b}^T(i) + \tfrac{1}{2}\,\mathbf{e}^T(i)}$$

where the error vector is: $\boxed{\mathbf{e}(i) = \mathbf{y}(i) - \mathbf{f}\Big(\mathbf{W}^T(i)\,\mathbf{x}(i) + \mathbf{b}(i)\Big) \in \mathbb{R}^S}$

Here $\mathbf{y}(i) \in \{-1, +1\}^S$ is the true label vector, $\mathbf{W}^T(i)\mathbf{x}(i) + \mathbf{b}(i) \in \mathbb{R}^S$ is the vector of S linear scores (one per hyperplane), and $\mathbf{f}$ applies the **sign function element-wise** to each score.

Why this is just S scalar rules stacked. The matrix update $\mathbf{W}(i+1) = \mathbf{W}(i) + \frac{1}{2}\mathbf{x}(i)\mathbf{e}^T(i)$ is exactly applying the single-perceptron rule independently to each column $\mathbf{w}_s$. To see this, look at column s of both sides:

$$\mathbf{w}_s(i+1) = \mathbf{w}_s(i) + \tfrac{1}{2}\,e_s(i)\,\mathbf{x}(i)$$

This is identical to the scalar update rule from before, applied to perceptron s with its own error $e_s(i) = y_s(i) - f(\mathbf{w}_s^T(i)\mathbf{x}(i) + b_s(i))$. The matrix form is just a compact way of running all S perceptrons in parallel with one equation.

The outer product $\mathbf{x}(i)\mathbf{e}^T(i)$ is a $p \times S$ matrix, exactly the shape of $\mathbf{W}$. Each column s of this outer product is $e_s(i)\,\mathbf{x}(i)$, which is the scalar update contribution for perceptron s. The matrix rule and the S scalar rules are mathematically identical.

Expanding the multi-class output (professor's notes). The output of the multi-class perceptron is a vector of S scalar predictions, one per perceptron. Written explicitly:

$$\text{output} = \begin{bmatrix} f(\mathbf{w}_1^T\mathbf{x} + b_1) \\ f(\mathbf{w}_2^T\mathbf{x} + b_2) \\ \vdots \\ f(\mathbf{w}_S^T\mathbf{x} + b_S) \end{bmatrix} = \mathbf{f}\Big(\mathbf{W}^T\mathbf{x} + \mathbf{b}\Big)$$

The compact form $\mathbf{f}(\mathbf{W}^T\mathbf{x} + \mathbf{b})$ hides all S scalar computations inside one expression: $\mathbf{f}$ acts **element-wise**. To see why, expand $\mathbf{W}^T\mathbf{x} + \mathbf{b}$ explicitly:

$$\mathbf{W}^T\mathbf{x} + \mathbf{b} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_S^T \end{bmatrix} \mathbf{x} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^T\mathbf{x} + b_1 \\ \mathbf{w}_2^T\mathbf{x} + b_2 \\ \vdots \\ \mathbf{w}_S^T\mathbf{x} + b_S \end{bmatrix} \in \mathbb{R}^S$$

Each row s of $\mathbf{W}^T$ is $\mathbf{w}_s^T$, so the s-th element of the product is exactly $\mathbf{w}_s^T\mathbf{x} + b_s$, the score of perceptron s. Applying $\mathbf{f}$ element-wise then gives:

$$\mathbf{f}(\mathbf{W}^T\mathbf{x} + \mathbf{b}) = \begin{bmatrix} f(\mathbf{w}_1^T\mathbf{x} + b_1) \\ \vdots \\ f(\mathbf{w}_S^T\mathbf{x} + b_S) \end{bmatrix}$$

$\mathbf{W}^T$ *has shape* $S \times p$*: each of its* S *rows is one perceptron's weight vector transposed. Multiplying by* $\mathbf{x} \in \mathbb{R}^p$ *produces an* S*-vector of scores: one dot product per perceptron, all computed simultaneously in one matrix-vector product.*

Error vector expanded . The error vector $\mathbf{e}(i) \in \mathbb{R}^S$ is computed element-wise: each entry is the scalar error of one perceptron:

$$\mathbf{e}(i) = \begin{bmatrix} y_1 - f(\mathbf{w}_1^T\mathbf{x} + b_1) \\ \vdots \\ y_S - f(\mathbf{w}_S^T\mathbf{x} + b_S) \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_S \end{bmatrix} - \begin{bmatrix} f(\mathbf{w}_1^T\mathbf{x} + b_1) \\ \vdots \\ f(\mathbf{w}_S^T\mathbf{x} + b_S) \end{bmatrix} = \mathbf{y} - \mathbf{f}(\mathbf{W}^T\mathbf{x} + \mathbf{b})$$

So the compact form $\mathbf{e}(i) = \mathbf{y}(i) - \mathbf{f}(\mathbf{W}^T(i)\mathbf{x}(i) + \mathbf{b}(i))$ is simply the label vector minus the prediction vector, subtracted **entry by entry**. Entry s is zero if perceptron s was correct, ± 2 if it was wrong.

This confirms that the matrix update rule $\mathbf{W}(i+1) = \mathbf{W}(i) + \frac{1}{2}\mathbf{x}(i)\mathbf{e}^T(i)$ *is nothing but the scalar rule applied to all* S *perceptrons in parallel: column* s *of* $\mathbf{W}$ *gets updated by* $\frac{1}{2}e_s(i)\mathbf{x}(i)$*, exactly as in the binary case.*

Example:

Matrix rule reduces to scalar rule: column-by-column . The matrix update $\mathbf{W}(i+1) = \mathbf{W}(i) + \frac{1}{2}\mathbf{x}(i)\mathbf{e}^T(i)$ can be expanded column by column. Looking at both sides:

$$[\mathbf{w}_1(i+1) \,|\, \cdots \,|\, \mathbf{w}_j(i+1) \,|\, \cdots \,|\, \mathbf{w}_S(i+1)] = [\mathbf{w}_1(i) \,|\, \cdots \,|\, \mathbf{w}_j(i) \,|\, \cdots \,|\, \mathbf{w}_S(i)] + \tfrac{1}{2}\,\mathbf{x}(i)\,[e_1(i) \;\cdots\; e_j(i) \;\cdots\; e_S(i)]$$

The outer product $\mathbf{x}(i)\mathbf{e}^T(i) \in \mathbb{R}^{p \times S}$ has j-th column equal to $e_j(i)\,\mathbf{x}(i)$. Isolating column j:

$$\boxed{\mathbf{w}_j(i+1) = \mathbf{w}_j(i) + \tfrac{1}{2}\,e_j(i)\,\mathbf{x}(i)}$$

This is exactly the binary perceptron rule applied to perceptron j *alone with its own scalar error* $e_j(i)$*. The matrix form is not a new rule: it is* S *independent scalar rules written compactly. Each perceptron updates only its own column using only its own error, completely independently of all others.*

Full worked example. $S = 2$ perceptrons, $p = 2$ features, biases fixed at zero. Initial weight matrix (columns = weight vectors of each perceptron):

$$\mathbf{W}(0) = [\mathbf{w}_1(0) \,|\, \mathbf{w}_2(0)] = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \mathbf{w}_1(0) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{w}_2(0) = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Two training points and their class encodings:

$$\mathbf{x}(1) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{y}(1) = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \; (C_1) \qquad \mathbf{x}(2) = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad \mathbf{y}(2) = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \; (C_0)$$

Step 1, present $\mathbf{x}(1) = [1, 0]^T$**, true label** $\mathbf{y}(1) = [1, -1]^T$**:**

Compute scores (one dot product per perceptron):

$$\mathbf{W}^T(0)\,\mathbf{x}(1) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Apply sign function element-wise: $\mathbf{f}([1, 1]^T) = [+1, +1]^T$.

Compute error: subtract prediction from true label:

$$\mathbf{e}(1) = \mathbf{y}(1) - \mathbf{f}(\mathbf{W}^T(0)\mathbf{x}(1)) = \begin{bmatrix} 1 \\ -1 \end{bmatrix} - \begin{bmatrix} +1 \\ +1 \end{bmatrix} = \begin{bmatrix} 0 \\ -2 \end{bmatrix}$$

Perceptron 1: correct ($e_1 = 0$, no update). Perceptron 2: wrong ($e_2 = -2$, update needed).

Apply matrix update rule:

$$\mathbf{W}(1) = \mathbf{W}(0) + \tfrac{1}{2}\,\mathbf{x}(1)\,\mathbf{e}^T(1) = \begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} + \frac{1}{2}\begin{bmatrix}1\\ 0\end{bmatrix}\begin{bmatrix}0 & -2\end{bmatrix} = \begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} + \begin{bmatrix}0 & -1\\ 0 & 0\end{bmatrix} = \begin{bmatrix}1 & 0\\ 1 & -1\end{bmatrix}$$

The first column is unchanged ($e_1 = 0$ so $\mathbf{w}_1$ gets no update). Only $\mathbf{w}_2$ changes: $\mathbf{w}_2(1) = [1,-1]^T + \frac{1}{2}(-2)[1,0]^T = [0,-1]^T$. The column-by-column rule is visible directly.

Step 2, present $\mathbf{x}(2) = [0,-1]^T$, true label $\mathbf{y}(2) = [-1,-1]^T$:

Compute scores using updated $\mathbf{W}(1)$:

$$\mathbf{W}^T(1)\,\mathbf{x}(2) = \begin{bmatrix}1 & 1\\ 0 & -1\end{bmatrix}\begin{bmatrix}0\\ -1\end{bmatrix} = \begin{bmatrix}-1\\ 1\end{bmatrix}$$

Apply sign function element-wise: $\mathbf{f}([-1,1]^T) = [-1,+1]^T$.

Compute error:

$$\mathbf{e}(2) = \mathbf{y}(2) - \mathbf{f}(\mathbf{W}^T(1)\mathbf{x}(2)) = \begin{bmatrix}-1\\ -1\end{bmatrix} - \begin{bmatrix}-1\\ +1\end{bmatrix} = \begin{bmatrix}0\\ -2\end{bmatrix}$$

Perceptron 1: correct again ($e_1 = 0$). Perceptron 2: wrong again ($e_2 = -2$).

Apply matrix update rule:

$$\mathbf{W}(2) = \mathbf{W}(1) + \tfrac{1}{2}\,\mathbf{x}(2)\,\mathbf{e}^T(2) = \begin{bmatrix}1 & 0\\ 1 & -1\end{bmatrix} + \frac{1}{2}\begin{bmatrix}0\\ -1\end{bmatrix}\begin{bmatrix}0 & -2\end{bmatrix} = \begin{bmatrix}1 & 0\\ 1 & -1\end{bmatrix} + \begin{bmatrix}0 & 0\\ 0 & 1\end{bmatrix} = \begin{bmatrix}1 & 0\\ 1 & 0\end{bmatrix}$$

Final weights after one full epoch: $\mathbf{w}_1 = [1,1]^T$ (never changed across either step), $\mathbf{w}_2 = [0,0]^T$.

Perceptron 1 was correct on every training point so its column was never touched, exactly as the column-by-column rule predicts: $e_1 = 0$ at every step means $\mathbf{w}_1$ receives a zero update every time. Perceptron 2 was wrong on both points and was corrected twice. Further epochs would continue refining $\mathbf{w}_2$ until convergence (or the stopping criterion is met).

8.1.4 Learning Rate

⋆ **Learning Rate.** A scalar hyperparameter $\alpha > 0$ added to the update rule that controls the **step size** of each weight correction. The general update rules become:

$$\boxed{\mathbf{W}(i+1) = \mathbf{W}(i) + \alpha\,\mathbf{x}(i)\,\mathbf{e}^T(i)} \qquad \boxed{\mathbf{b}^T(i+1) = \mathbf{b}^T(i) + \alpha\,\mathbf{e}^T(i)}$$

The scalar rule per column j is correspondingly:

$$\mathbf{w}_j(i+1) = \mathbf{w}_j(i) + \alpha\,e_j(i)\,\mathbf{x}(i)$$

The earlier rule with $\frac{1}{2}$ is the special case $\alpha = 0.5$, a common default. α scales how aggressively the weight vector moves toward or away from each misclassified point.

Every time the perceptron makes an error, it corrects itself by adding $\alpha\,e_j(i)\,\mathbf{x}(i)$ to $\mathbf{w}_j$. If α is large, the correction is large: fast but potentially destabilizing. If α is small, each correction is tiny: stable but slow. Choosing α well is one of the most important practical decisions in training any neural network.

How to choose α; tradeoffs:

- **Too small:** each update makes negligible progress. The weight vector moves so slowly that many epochs are needed to converge: **slow training**.
- **Too large:** each update **overshoots** the correct direction. A correction that fixes one misclassification may flip previously correct points to wrong; the algorithm oscillates and may **never converge** even on linearly separable data.

Learning rate decay schedule. A practical compromise: start with a **large** α for fast initial progress, then **reduce** it over time so updates become finer as the solution is approached. One standard decay schedule:

$$\boxed{\alpha(i) = \frac{\alpha_0}{i + c}}$$

where $\alpha_0 > 0$ is the initial learning rate, i is the current iteration index, and $c > 0$ is a constant that prevents α from blowing up at $i = 0$. As $i \to \infty$, $\alpha(i) \to 0$: updates shrink to zero and the weights stabilize.

The decay schedule gives the best of both worlds: large steps early (explore quickly) and small steps late (converge precisely). This same idea, large α early, small α late, carries forward to all neural network training, not just perceptrons. The exact schedule (polynomial decay, exponential decay, cosine annealing) is still an active area of practice.

8.1.5 Historical Context

McCulloch & Pitts (1943). Proposed the first mathematical model of a neuron: a unit with a *binary threshold activation function*: it fires (+1) if its weighted inputs exceed a threshold, stays silent (−1) otherwise. They showed that such units could implement any logical function (AND, OR, NOT, etc.), establishing that computation could emerge from neuron-like elements.

Hebb's Rule (1949). Donald Hebb observed a biological learning principle: *when two neurons fire together, the synaptic connection between them is strengthened.* This became known as *Hebbian learning*: "neurons that fire together, wire together." He proposed it as a fundamental mechanism of learning and memory in biological systems.

The Hebb rule is the biological inspiration for the perceptron update: when an input $\mathbf{x}$ *causes a correct firing, the weight connecting* $\mathbf{x}$ *to the neuron increases: the connection is reinforced. This is exactly what* $\mathbf{w} \leftarrow \mathbf{w} + \alpha\, e\, \mathbf{x}$ *does when the error* $e > 0$.

Rosenblatt (late 1950s). Frank Rosenblatt combined the McCulloch-Pitts neuron with Hebb's principle to produce the first **Perceptron Learning Rule**: a neuron that learns by adjusting its weights based on prediction errors. The perceptron learns in the Hebbian sense: weights grow toward inputs that the network correctly responds to, and shrink away from inputs that produce errors. This was the direct precursor to all modern neural networks.

8.2 The Perceptron as a Neural Architecture

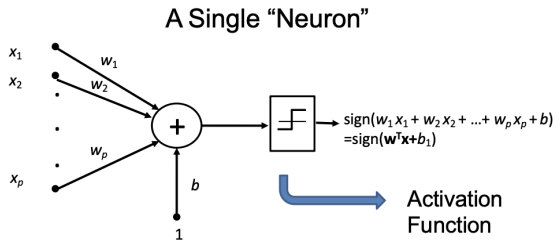
A perceptron is modeled as a single **neuron**: it computes a weighted sum of inputs plus a bias; the **net input** $n = \mathbf{w}^T\mathbf{x} + b$, then passes it through an **activation function** f to produce an output $a = f(n)$. Stacking S such neurons in parallel gives the **multi-class perceptron**, where the scalar weights generalize to a weight matrix $\mathbf{W} \in \mathbb{R}^{p\times S}$ and all S outputs are computed simultaneously as $\mathbf{a} = \mathbf{f}(\mathbf{W}^T\mathbf{x} + \mathbf{b})$.

⋆ **Single Neuron.** A single perceptron maps to the following computational diagram: inputs $x_1, \ldots, x_p$ each scaled by weights $w_1, \ldots, w_p$, summed together with bias b, then passed through an **activation function** f:

$$\text{net input:} \quad n = \mathbf{w}^T\mathbf{x} + b = \sum_{j=1}^{p} w_j x_j + b \qquad \text{output:} \quad a = f(n) = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$$

The components are: $\mathbf{x} \in \mathbb{R}^p$ (input), $\mathbf{w} \in \mathbb{R}^p$ (weights/connections), b (bias, fed from a constant input of 1), f (activation function, here the sign function). The bias node "1" feeds into the sum with weight b, shifting the threshold up or down.

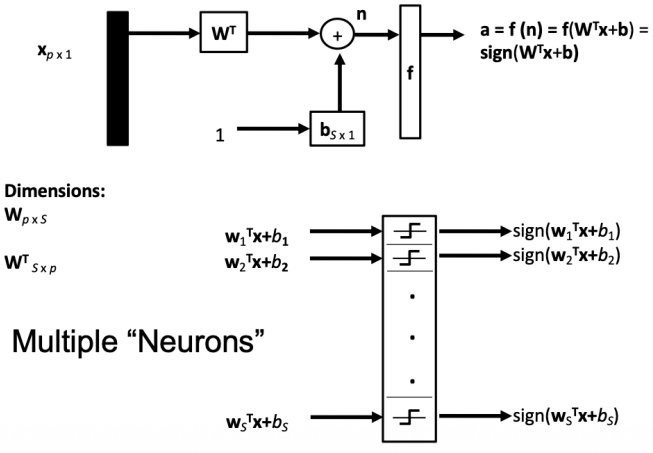
Think of the neuron as a biological cell: incoming signals x_j travel along synapses with strengths w_j. The cell body sums them all up. If the total exceeds the threshold (set by b), the cell fires ($+1$); otherwise it stays silent (-1). The weights w_j are the learned synaptic strengths.



⋆ **Multiple neurons, matrix form.** With S neurons in parallel, the single diagram scales up: input $\mathbf{x}_{p\times 1}$ is multiplied by $\mathbf{W}^T_{S\times p}$, the bias vector $\mathbf{b}_{S\times 1}$ is added, and the activation $\mathbf{f}$ is applied element-wise:

$$\mathbf{n} = \mathbf{W}^T\mathbf{x} + \mathbf{b} \in \mathbb{R}^S$$

$$\mathbf{a} = \mathbf{f}(\mathbf{n}) = \mathbf{f}(\mathbf{W}^T\mathbf{x} + \mathbf{b}) \in \mathbb{R}^S$$



Key dimensions. $\mathbf{W} \in \mathbb{R}^{p\times S}$ (each column = one neuron's weight vector), $\mathbf{W}^T \in \mathbb{R}^{S\times p}$ (each row = one neuron's weights). The output $\mathbf{a}$ is the vector of S individual neuron outputs:

$$\mathbf{a} = \begin{bmatrix} \text{sign}(\mathbf{w}_1^T\mathbf{x} + b_1) \\ \vdots \\ \text{sign}(\mathbf{w}_S^T\mathbf{x} + b_S) \end{bmatrix}$$

The block diagram is just a compact picture of the math: a single matrix-vector multiply $\mathbf{W}^T\mathbf{x}$ computes all S dot products simultaneously, then $+\mathbf{b}$ shifts each one, then $\mathbf{f}$ applies the threshold to each independently. One diagram, S neurons.

Adaptation in neural systems. The perceptron learning rule is a model of neural adaptation: the network **adapts** its weights in response to the **error** (mismatch between actual output and desired output). This error-driven weight adjustment is the foundational principle of supervised learning across the entire class of neural networks, from perceptrons to deep networks trained with backpropagation.

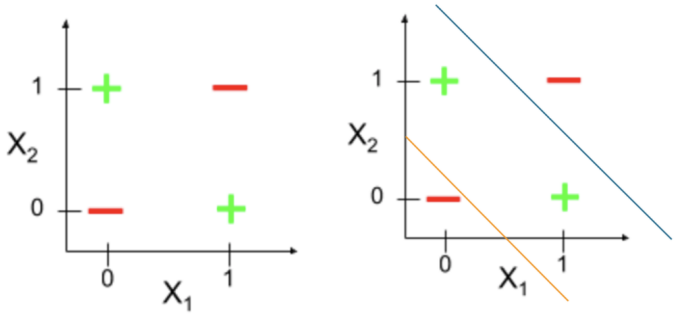
8.2.1 Limitation: Linear Separability and the XOR Problem

Linear Separability The fundamental limitation of a single-layer perceptron: if the training data is **not linearly separable, the perceptron rule never converges**, the weights keep updating forever and classification error stays large. No single hyperplane can correctly separate all points, so the algorithm oscillates endlessly.

The fix is to **map the data into a new feature space where it becomes linearly separable.** If we can find a transformation $\phi : \mathbb{R}^p \to \mathbb{R}^d$ such that $\phi(\mathbf{x})$ is linearly separable, a perceptron in the new space solves the problem.
The key insight: **use layers of perceptrons to learn ϕ automatically**: earlier layers transform the data, the final layer classifies in the transformed space.

Instead of hand-engineering the feature map ϕ, we let the network learn it from data. Each hidden layer applies a learned nonlinear transformation, progressively reshaping the input space until the classes are separable. This is the core idea of deep learning.

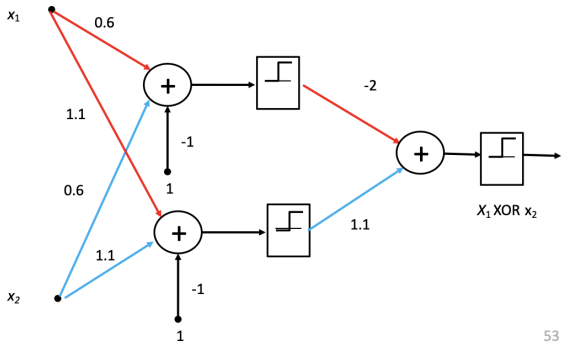
⋆ **The XOR Problem.** XOR is the canonical example of a problem that is **not linearly separable**. The XOR function on two binary inputs:



x_1	x_2	$y = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

The four points sit at the corners of a unit square. The $+1$ class occupies $(0, 1)$ and $(1, 0)$, diagonally opposite corners. The -1 class occupies $(0, 0)$ and $(1, 1)$, the other two corners. **No single straight line can separate these two classes**: any line that separates $(0, 1)$ from $(0, 0)$ will fail to separate $(1, 0)$ from $(1, 1)$, and vice versa.

Two-layer solution. XOR **can** be solved by a two-layer network: a **hidden layer** of 2 neurons computes two intermediate hyperplanes, and the **output layer** combines their outputs. One set of weights that works:
- **Hidden neuron 1**: $\text{sign}(0.6\,x_1 + 1.1\,x_2 - 1)$: fires when approximately $x_2 > x_1$ (separates one diagonal)
- **Hidden neuron 2**: $\text{sign}(1.1\,x_1 + 0.6\,x_2 - 1)$: fires when approximately $x_1 > x_2$ (separates the other diagonal)
- **Output neuron**: $\text{sign}(-2 \cdot h_1 + 1.1 \cdot h_2)$: combines the two hidden outputs to produce XOR



Each hidden neuron draws one hyperplane across the input space. Together, the two hyperplanes carve out the four corners so that the XOR pattern (same-class points on diagonally opposite corners) becomes linearly separable in the hidden representation *space. The output neuron just reads off which region each point landed in. This is the geometric essence of why depth helps: more layers = more hyperplanes = more complex regions.*

Caveat: why hand-designed weights are not enough . The two-layer XOR solution above used **manually chosen weights**. Finding such weights by hand is ad-hoc and quickly becomes intractable for complex functions with many inputs. What is needed is a **principled algorithm** that automatically learns the weights for multiple layers simultaneously; this is exactly what **backpropagation** provides, covered next.

8.3 Multi-Layer Perceptron (MLP)

An MLP solves the fundamental limitation of a single perceptron (that one hyperplane can only separate linearly separable data) by stacking multiple layers of neurons so that each layer **learns a new representation** of the data, progressively reshaping the input space until what was originally inseparable becomes separable in the final layer.

MLP applies the multiple neurons formula $\mathbf{a} = \mathbf{f}(\mathbf{W}^T\mathbf{a}_{\text{prev}} + \mathbf{b})$ repeatedly for M layers in sequence, each layer's output $\mathbf{a}^{(m)}$ becomes the next layer's input $\mathbf{a}^{(m-1)}$, with its own learned $\mathbf{W}^{(m)}, \mathbf{b}^{(m)}, \mathbf{f}^{(m)}$. The full network is the nested composition $g(\mathbf{x}) = g^{(M)}(\cdots g^{(2)}(g^{(1)}(\mathbf{x}))\cdots)$, where each layer progressively transforms the representation until the final output $\mathbf{a}^{(M)}$ is used for classification or regression.

⋆ **Multi-Layer Perceptron (MLP).** An **MLP** stacks multiple layers of neurons sequentially. Each layer has its own **weight matrix $\mathbf{W}^{(m)}$,bias vector $\mathbf{b}^{(m)}$**, and **activation function $\mathbf{f}^{(m)}$.** Each layer may have a different number of neurons, and different activation functions can be used, not just the binary sign function. All weights across all layers are learned jointly from data.

A single perceptron layer draws one set of hyperplanes. Stacking M layers means the output of each layer becomes the input to the next, each layer reshapes the representation so the next layer can do something useful with it. By the final layer, even non-linearly separable data can be classified correctly.

⋆ **MLP as Nested Functions.** An M-layer MLP is mathematically a **composition of functions**, the output of one layer is fed as input to the next:

$$\boxed{g(\mathbf{x}) = g^{(M)}\Big(g^{(M-1)}\Big(\cdots\Big(g^{(2)}\Big(g^{(1)}(\mathbf{x})\Big)\Big)\cdots\Big)}$$

For a 3-layer network: $g(\mathbf{x}) = g^{(3)}(g^{(2)}(g^{(1)}(\mathbf{x})))$. Each $g^{(m)}$ is the function computed by layer m: a matrix multiply, bias add, and activation. The whole network is just one big nested function.

This nested view makes the chain rule (and therefore backpropagation) natural: to compute how the final output depends on the first layer's weights, you just differentiate through the composition layer by layer from outside in.

Layer function $g^{(m)}$. Each $g^{(m)}$ is simply a shorthand for one full layer computation: it takes the previous layer's output, multiplies by the weight matrix, adds the bias, and applies the activation:

$$g^{(m)}(\mathbf{a}^{(m-1)}) = \mathbf{f}^{(m)}\Big(\mathbf{W}^{(m)T}\mathbf{a}^{(m-1)} + \mathbf{b}^{(m)}\Big) = \mathbf{a}^{(m)}$$

The nested composition $g^{(M)}(\cdots g^{(1)}(\mathbf{x})\cdots)$ is then just a compact way of writing "apply layer 1, then layer 2, …, then layer M" in sequence.

Activation vector $\mathbf{a}^{(m)}$. $\mathbf{a}^{(m)} \in \mathbb{R}^{S^{(m)}}$ is the **output vector** of layer m: the $S^{(m)}$ numbers produced by that layer's neurons after applying the activation function, which then serve as the input to layer $m+1$.

⋆ **Layer Notation.** Each layer m ($m = 1, \ldots, M$) consists of:

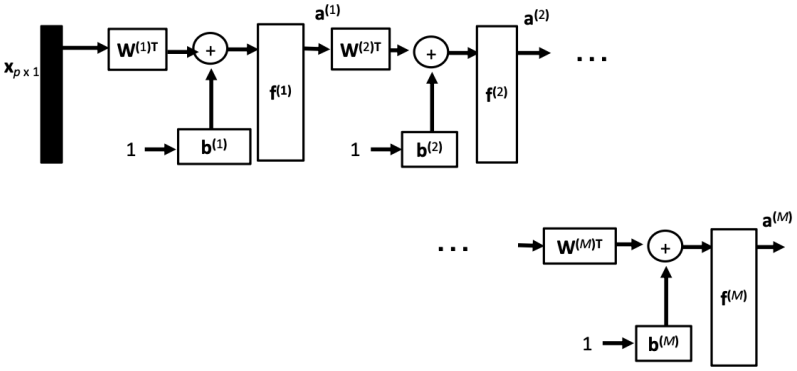
- $\mathbf{W}^{(m)} \in \mathbb{R}^{S^{(m-1)} \times S^{(m)}}$: weight matrix ($S^{(m)}$ = number of neurons in layer m, $S^{(0)} = p$)
- $\mathbf{b}^{(m)} \in \mathbb{R}^{S^{(m)}}$: bias vector
- $\mathbf{f}^{(m)}$: element-wise activation function

The **net input** and **output** of layer m:

$$\boxed{\mathbf{n}^{(m)} = \mathbf{W}^{(m)T}\mathbf{a}^{(m-1)} + \mathbf{b}^{(m)}} \qquad \boxed{\mathbf{a}^{(m)} = \mathbf{f}^{(m)}\left(\mathbf{n}^{(m)}\right) = g^{(m)}\left(\mathbf{a}^{(m-1)}\right)}$$

with the convention $\mathbf{a}^{(0)} = \mathbf{x}$ (the raw input). The output of the full network is $\mathbf{a}^{(M)}$.

Every layer is just the same one-layer perceptron formula applied to the previous layer's output instead of the raw input. Layer 1 sees $\mathbf{x}$*; layer 2 sees* $\mathbf{a}^{(1)}$*, a transformed version of* $\mathbf{x}$*; layer 3 sees* $\mathbf{a}^{(2)}$*, a transformed version of the transformed version. Each layer builds on what the previous one learned.*



Feedforward networks and deep learning (slide 60). MLPs are also called **Feedforward Neural Networks** because information flows in one direction only: forward from input to output, with no loops or feedback. When the number of layers M is large (e.g. $M > 10$), they are called **Deep Feedforward Neural Networks**; this is the origin of the term **deep learning**. The weights in every layer are learned jointly so that the full network minimizes the training objective for classification or regression.

8.3.1 Activation Functions

⋆ **Why nonlinear activations?** Without a nonlinear $\mathbf{f}^{(m)}$, every layer is just a linear map, and a composition of linear maps is still linear regardless of depth, the whole network collapses to a single matrix multiply. **Nonlinear activations are what give MLPs the ability to represent complex, non-linear functions.**

⋆ **Sigmoid.** Historically the standard choice for hidden and output layers:

$$\boxed{f(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}} \quad \in (0,1)$$

Squashes any real input into $(0,1)$, smooth and differentiable everywhere. **Problem:** for large $|x|$ the function **saturates**: the gradient $f'(x) \to 0$ and weight updates vanish. This is the **vanishing gradient** problem that makes deep sigmoid networks hard to train.

Imagine a neuron receiving a very large score: the sigmoid output is nearly 1 and its slope nearly 0. The update rule multiplies by this slope, so almost nothing gets corrected. The neuron is stuck and stops learning.

⋆ **Tanh.** A zero-centered variant of sigmoid:

$$\boxed{f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x}-1}{e^{2x}+1}} \quad \in (-1,1)$$

Output is in $(-1, 1)$ and symmetric around zero. Zero-centering means gradients have no systematic positive or negative bias, making optimization slightly more stable than sigmoid. Still saturates at extreme values: same vanishing gradient problem.

Tanh and sigmoid have the same shape; tanh is just shifted so its output is centered at 0 rather than 0.5. In practice tanh consistently outperforms sigmoid in hidden layers, but both are largely obsolete in deep networks due to saturation.

⋆ **Rectified Linear Unit (ReLU).** The default modern choice for hidden layers:

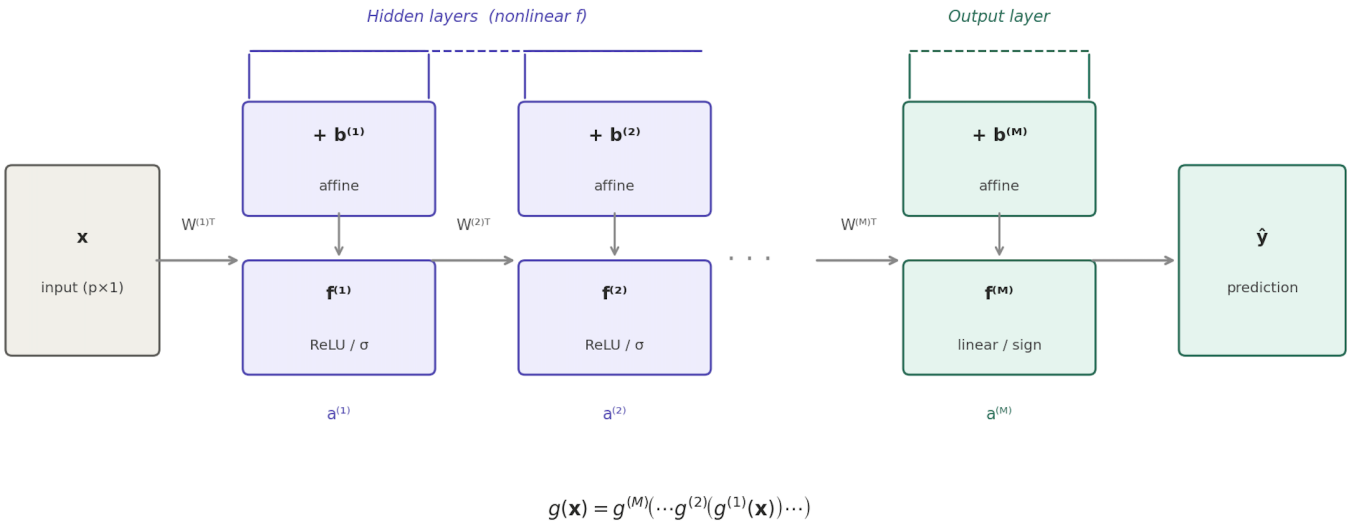
$$\boxed{f(u) = \max(0,\, u)}$$

Zero for negative inputs, identity for positive inputs. Gradient is exactly 1 for $u > 0$: **no saturation, no vanishing gradient** on the positive side. Computationally trivial. The standard choice for all hidden layers in modern deep networks.

Unlike sigmoid/tanh which compress everything into a bounded range and always saturate somewhere, ReLU lets large positive signals pass through with full gradient. Neurons with ReLU either fire (pass the signal unchanged) or are silent (output zero); this sparsity also speeds up computation and acts as implicit regularization.

Summary.

- **Sigmoid**: output $(0, 1)$, saturates, used historically for output layers (probabilities)
- **Tanh**: output $(-1, 1)$, zero-centered, saturates, better than sigmoid but still obsolete for deep hidden layers
- **ReLU**: output $[0, \infty)$, no saturation for $u > 0$, **default for hidden layers in modern deep networks**



8.3.2 MLP as Universal Approximator

⋆ **Universal Approximation Theorem.** It can be proven mathematically that **any smooth function** can be approximated to arbitrary precision using an MLP with only **two layers**:

- A **hidden layer** of smooth nonlinear functions (e.g. sigmoids or ReLUs)
- An **output layer** of smooth functions; even a linear output layer is sufficient

MLPs are **universal approximators**

This does not mean a 2-layer network is always the best choice: it may require an exponentially large number of hidden neurons to approximate complex functions with only 2 layers. Deeper networks can represent the same functions with far fewer total parameters, which is why depth matters in practice.

⋆ **MLP for Regression.** The output layer uses smooth (or even linear) activation functions. The network directly predicts a continuous value $\hat{y} = \mathbf{a}^{(M)}$. The objective is to minimize the mean squared error between $\hat{y}$ and the true response y.

⋆ **MLP for Classification** The standard architecture uses:

- A **hidden layer** of smooth nonlinear activations, learns feature representations
- An **output layer** of linear functions followed by **threshold** (sign) functions, produces class predictions

Class encoding. Classes must be encoded as vectors so the network has a target to predict. Two standard encodings:

- **Binary encoding:** $K = 2^S$ classes encoded as S-bit binary vectors, same as the multi-class perceptron. Each bit is predicted by one output neuron.
- **One-hot encoding:** K classes each represented by a vector of length K with a single 1 in the position corresponding to the class and 0s elsewhere. E.g. class 3 of 4: $[0, 0, 1, 0]^T$. More common in practice.

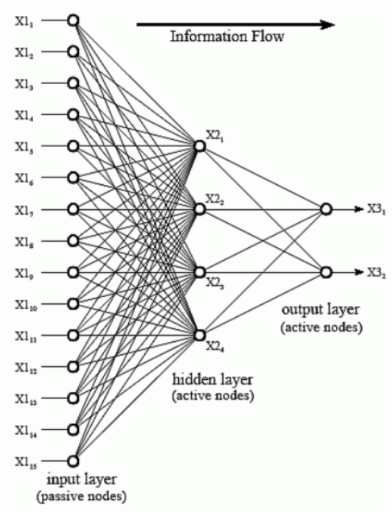
One-hot encoding is preferred over binary because it makes the network's task cleaner: each output neuron is responsible for exactly one class, with no ambiguity about which bit patterns correspond to which classes. The desired output is always a vector, never a scalar, regardless of encoding.

8.3.3 Architectural Considerations

⋆ **Depth and Width.** Two key design decisions must be made before training any MLP:

- **Depth**: the number of layers M: controls how many successive transformations the data passes through.
- **Width**: the number of neurons $S^{(m)}$ in each layer: controls the dimensionality of each intermediate representation.

*The input layer is **passive**: it just holds the raw features* $\mathbf{x}$*, no computation is done. Hidden and output layers are **active**: they apply weights, biases, and activations. Too narrow and the network cannot represent the needed complexity; too wide and it overfits.*



Properties of deeper networks.

- **Fewer units per layer**: deeper networks can represent the same functions with far fewer total neurons per layer than a shallow wide network, exponentially more parameter-efficient for complex functions.
- **Better generalization**: deeper architectures often generalize better to unseen data, because the hierarchical feature learning naturally builds useful inductive biases.
- **Harder to train**: more layers means more complex gradient flow during backpropagation: vanishing gradients, local minima, and longer training times are all more likely.

How to choose the architecture. There is no closed-form formula for the ideal depth and width. The correct procedure is:

1. Train multiple architectures on the training set.
2. Evaluate each on the **validation set**.
3. Select the architecture with the lowest **validation error**.

Architecture selection is itself a hyperparameter search. You cannot use test error to guide this choice: the test set must remain unseen until final evaluation. Validation error is the only honest signal for comparing architectures during development.

8.4 Training MLPs: Backpropagation and Gradient Descent

⋆ **Backpropagation.** MLPs are trained using **gradient-based optimization**: we need $\frac{\partial J}{\partial w_{ij}^{(m)}}$ (how much each individual weight contributes to the total error) then step in the direction that reduces J. **Backpropagation** (Rumelhart, Hinton, Williams, 1980s) is the algorithm that efficiently computes all these gradients by applying the chain rule layer by layer, propagating error signals **backwards** from the output to the input. It is still the foundation of all modern deep learning training.

The key insight of backprop: rather than computing $\frac{\partial J}{\partial w_{ij}^{(m)}}$ separately for every single weight (which would require a separate forward pass per weight), you can reuse intermediate computations by propagating a single error signal backward through all layers simultaneously. One forward pass + one backward pass = gradients for every weight in the network.

⋆ **Setup.** Given a training set of N input-output pairs:

$$\{(\mathbf{x}(1), \mathbf{y}(1)),\ (\mathbf{x}(2), \mathbf{y}(2)),\ \ldots,\ (\mathbf{x}(N), \mathbf{y}(N))\}$$

- $\mathbf{x}(k) \in \mathbb{R}^p$: the k-th input feature vector
- $\mathbf{y}(k) \in \mathbb{R}^{S^{(M)}}$: the k-th desired output vector (true label, encoded as a vector)

The goal: find weight matrices $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(M)}$ and bias vectors $\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(M)}$ that minimize an objective function J measuring total prediction error across all training points.

⋆ **Full MLP notation summary.** For each layer $j = 1, \ldots, M$:

$$\boxed{\mathbf{n}^{(j)} = \mathbf{W}^{(j)T}\mathbf{a}^{(j-1)} + \mathbf{b}^{(j)}} \qquad \boxed{\mathbf{a}^{(j)} = \mathbf{f}^{(j)}(\mathbf{n}^{(j)})} \qquad \mathbf{a}^{(0)} = \mathbf{x}$$

- $\mathbf{n}^{(j)} \in \mathbb{R}^{S^{(j)}}$: the **net input** to layer j: raw weighted sums before activation, one entry per neuron in that layer. $S^{(j)}$ is the width (number of neurons) of layer j.
- $\mathbf{a}^{(j)} \in \mathbb{R}^{S^{(j)}}$: the **activation output** of layer j: the net input passed through the activation function $\mathbf{f}^{(j)}$. This is what gets fed as input to layer $j+1$.
- $\mathbf{W}^{(j)} \in \mathbb{R}^{S^{(j-1)} \times S^{(j)}}$: weight matrix of layer j, shaped so $\mathbf{W}^{(j)T}\mathbf{a}^{(j-1)}$ maps the $S^{(j-1)}$-dimensional input to an $S^{(j)}$-dimensional output.
- $\mathbf{b}^{(j)} \in \mathbb{R}^{S^{(j)}}$: bias vector of layer j, one bias per neuron.
- **Layer width $S^{(m)} \in \mathbb{N}$.** is the **number of neurons** in layer m, controlling the dimensionality of all vectors in that layer: $\mathbf{n}^{(m)}, \mathbf{a}^{(m)}, \mathbf{b}^{(m)} \in \mathbb{R}^{S^{(m)}}$ and $\mathbf{W}^{(m)} \in \mathbb{R}^{S^{(m-1)} \times S^{(m)}}$.
- **Gradient $\nabla_{\mathbf{W}^{(m)}} J = \frac{\partial J}{\partial \mathbf{W}^{(m)}}$** is the matrix of partial derivatives of J with respect to every weight in layer m, each entry $\frac{\partial J}{\partial w_{ij}^{(m)}}$ tells you how much the total error J increases per unit increase in that one weight, pointing in the direction of steepest ascent of J in weight space.

⋆ **Objective function J.** The **expected sum of squared errors** over all output neurons and all training points:

$$\boxed{J = E\left\{\sum_{i=1}^{S^{(M)}} e_i^2\right\} = E\left\{\sum_{i=1}^{S^{(M)}} (y_i - a_i^{(M)})^2\right\} = E\{\mathbf{e}^T\mathbf{e}\}}$$

- $e_i = y_i - a_i^{(M)}$: scalar error for output neuron i: the difference between the **true label** y_i and the **network's prediction** $a_i^{(M)}$
- $\sum_{i=1}^{S^{(M)}} e_i^2 = \mathbf{e}^T\mathbf{e}$: total squared error across all $S^{(M)}$ output neurons for one training point, a single scalar measuring how wrong the prediction is
- $E\{\cdot\}$: expectation over the training distribution: the average of this squared error over all N training pairs

Expanding $\mathbf{e}^T\mathbf{e}$ (Note). The compact form $J = E\{\mathbf{e}^T\mathbf{e}\}$ expands explicitly as:

$\mathbf{e}^T\mathbf{e} = \|\mathbf{e}\|_2^2 = (e_1 - e_{S^{(M)}}) \begin{bmatrix} e_1 \\ \vdots \\ e_{S^{(M)}} \end{bmatrix}^2 = \sum_{i=1}^{S^{(M)}} e_i^2$ where the error vector is $\mathbf{e} = \mathbf{y} - \mathbf{a}^{(M)} \in \mathbb{R}^{S^{(M)}}$, with:

$$\mathbf{a}^{(M)} = \begin{bmatrix} a_1^{(M)} \\ \vdots \\ a_{S^{(M)}}^{(M)} \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_{S^{(M)}} \end{bmatrix} \quad \text{so } \mathbf{e}^T\mathbf{e} = \sum_{i=1}^{S^{(M)}} e_i^2 = \sum_{i=1}^{S^{(M)}} (y_i - a_i^{(M)})^2.$$

$J = E\{\mathbf{e}^T\mathbf{e}\} = E\{\|\mathbf{e}\|_2^2\}$ sums the squared error of every output neuron into one scalar: squaring makes all errors positive so they cannot cancel, and penalizes large errors more heavily than small ones, pushing the network to fix its worst mistakes first. Each term $(y_i - a_i^{(M)})^2$ measures how wrong output neuron i is, and J averages this total wrongness over the training distribution : this single number is what backpropagation minimizes.

⋆ **Approximating $J(\mathbf{W}, \mathbf{B})$.** The true expectation $E\{\mathbf{e}^T\mathbf{e}\}$ is unknown; approximate it as a sample average over the N training pairs:

$$\boxed{J(\mathbf{W}, \mathbf{B}) = E\{\mathbf{e}^T\mathbf{e}\} \approx \frac{1}{N}\sum_{k=1}^{N} \mathbf{e}(k)^T\mathbf{e}(k)}$$

where $\mathbf{W} = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(M)}\}$ and $\mathbf{B} = \{\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(M)}\}$ are all weights and biases across all layers, and the per-sample error vector is:

$$\mathbf{e}(k) = \mathbf{y}(k) - \mathbf{a}^{(M)} = \mathbf{y}(k) - g^{(M)}\Big(g^{(M-1)}\Big(\cdots g^{(1)}(\mathbf{x}(k))\cdots\Big)\Big) \in \mathbb{R}^{S^{(M)}}$$

Since each $g^{(j)}$ contains $\mathbf{W}^{(j)}$ and $\mathbf{b}^{(j)}$ inside it, J depends on **every** weight and bias simultaneously through this nested composition. To minimize J we therefore need $\frac{\partial J}{\partial \mathbf{W}^{(j)}}$ and $\frac{\partial J}{\partial \mathbf{b}^{(j)}}$ for every layer j; computing all these gradients efficiently through the composition is exactly what **backpropagation** does.

$J(\mathbf{W}, \mathbf{B})$ is one scalar that encodes the total prediction error of the entire network. The difficulty is that this single number depends on millions of weights at once through deeply nested functions; there is no closed-form solution, only iterative gradient descent guided by backpropagation.

⋆ **Stochastic approximation = Stochastic Gradient Descent .** Computing the full average over all N points before each weight update is expensive. Instead, approximate J using just **one training pair** at a time: $\boxed{J \approx \mathbf{e}(k)^T\mathbf{e}(k)}$

This is **stochastic gradient descent (SGD)**: update the weights after seeing each single sample, using its individual error as a proxy for the true objective. A compromise: use a **mini-batch** of L pairs ($L \ll N$), averaging their errors, more stable than single-sample SGD but cheaper than full-batch.

Why does using one sample work? By law of large numbers, the sample average over all N points converges to the true expectation. SGD makes noisy but cheap gradient estimates: each step is wrong on average but the noise averages out over many steps, and the algorithm converges to approximately the same place as full-batch gradient descent, but much faster per epoch.

Stochastic Gradient Descent (SGD). randomly pick one sample, compute which way is uphill on J, take a small step α downhill, repeat until convergence.

Stochastic: instead of computing $\frac{\partial J}{\partial w_{ij}^{(m)}}$ using all N training pairs (expensive), randomly pick **one sample** k and approximate $J \approx \mathbf{e}(k)^T\mathbf{e}(k)$. The gradient is noisy but cheap.

Gradient: $\frac{\partial J}{\partial w_{ij}^{(m)}}$ tells you how much J increases per unit increase in weight $w_{ij}^{(m)}$, the direction of steepest **ascent** on the loss surface.

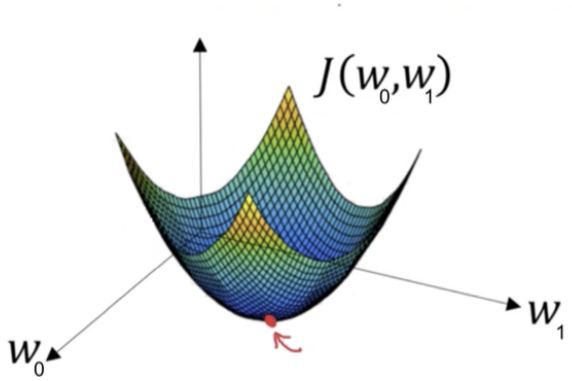
Descent : the minus sign in $w_{ij}^{(m)}(k+1) = w_{ij}^{(m)}(k) - \alpha \frac{\partial J}{\partial w_{ij}^{(m)}}$ moves **opposite** to the gradient (downhill on J) reducing the error at each step.

⋆ **Approximate Gradient Descent.** Once we have the approximate $J \approx \mathbf{e}(k)^T\mathbf{e}(k)$, update every weight and bias by stepping in the direction of **steepest descent** of J:

$$\boxed{w_{ij}^{(m)}(k+1) = w_{ij}^{(m)}(k) - \alpha \frac{\partial J}{\partial w_{ij}^{(m)}}} \qquad \boxed{b_i^{(m)}(k+1) = b_i^{(m)}(k) - \alpha \frac{\partial J}{\partial b_i^{(m)}}}$$

- $w_{ij}^{(m)}$: the weight connecting neuron j in layer $m-1$ to neuron i in layer m, a single scalar from the matrix $\mathbf{W}^{(m)}$
- $b_i^{(m)}$: the bias of neuron i in layer m
- $\frac{\partial J}{\partial w_{ij}^{(m)}}$: the **gradient**: how much J changes if we nudge this one weight slightly. **Positive gradient** means increasing $w_{ij}^{(m)}$ increases J, so we decrease it. **Negative gradient** means the opposite.
- $\alpha > 0$: the **learning rate**: controls the step size. The minus sign ensures we go downhill.
- k: the current training iteration (one sample or mini-batch presented)

Think of J as a hilly landscape over all the weights. The current weights place you at some point on this landscape. The gradient tells you which direction is uphill; you take a small step in the opposite direction (downhill) and repeat until you reach a valley (minimum). The challenge: for real neural networks the landscape is not a smooth bowl: it has many local minima, saddle points, and flat plateaus. Gradient descent only finds a local minimum, not necessarily the global one.



Why the gradient is hard to compute. $\frac{\partial J}{\partial w_{ij}^{(m)}}$ requires differentiating $J = \mathbf{e}^T\mathbf{e}$ through the full composition of nested functions:

$$J = \|\mathbf{y}(k) - \mathbf{f}^{(M)}(\mathbf{W}^{(M)T}\mathbf{f}^{(M-1)}(\cdots \mathbf{f}^{(1)}(\mathbf{W}^{(1)T}\mathbf{x}(k) + \mathbf{b}^{(1)})\cdots) + \mathbf{b}^{(M)})\|^2$$

For a weight in layer 1, you must differentiate through M nested functions using the chain rule M times. This is what backpropagation automates, it efficiently computes all $\frac{\partial J}{\partial w_{ij}^{(m)}}$ for every layer m simultaneously in a single backward pass.

8.4.1 Backpropagation Algorithm

Backpropagation takes the error $\mathbf{e} = \mathbf{y} - \mathbf{a}^{(M)}$ at the output layer and propagates it backwards through the network layer by layer using the chain rule, computing $\frac{\partial J}{\partial \mathbf{W}^{(m)}}$ for every layer m simultaneously in one backward pass, telling each weight exactly how much it contributed to the total error so gradient descent can correct it.

⋆ **Forward Propagation.** Before computing any gradients, run the input $\mathbf{x}(k)$ **forward** through all M layers to compute every activation vector; these will be needed during the backward pass:

$$\boxed{\mathbf{a}^{(0)} = \mathbf{x}(k)} \qquad \boxed{\mathbf{a}^{(m+1)} = \mathbf{f}^{(m+1)}\left(\mathbf{W}^{(m+1)T}\mathbf{a}^{(m)} + \mathbf{b}^{(m+1)}\right), \quad m = 0, 1, \ldots, M-1}$$

The final output is $\mathbf{a}^{(M)} = \mathbf{a}$, the network's prediction. The activation function $\mathbf{f}^{(m+1)}$ acts **element-wise** on each entry of the net input vector independently.

- $\mathbf{a}^{(0)} = \mathbf{x}(k)$: the raw input, the starting point, no computation done
- $\mathbf{W}^{(m+1)T}\mathbf{a}^{(m)}$: linear transformation: multiply the previous layer's output by the weight matrix to get the weighted sums
- $+\mathbf{b}^{(m+1)}$: add the bias vector: shifts each neuron's score independently
- $\mathbf{f}^{(m+1)}(\cdot)$: apply activation element-wise: introduces nonlinearity
- $\mathbf{a}^{(m+1)}$: the result: the output of layer $m+1$, becomes the input to layer $m+2$

Forward propagation is just the MLP evaluation formula applied layer by layer from left to right. You are not learning anything yet: you are simply running the current weights to get a prediction. Every intermediate $\mathbf{a}^{(m)}$ *must be stored because backpropagation will need them when computing gradients.*

⋆ **Sensitivity matrix $\mathbf{F}'^{(m)}(\mathbf{n}^{(m)})$.** To propagate gradients backward, we need the derivative of the activation function at each layer. Define the **sensitivity matrix** $\mathbf{F}'^{(m)}(\mathbf{n}^{(m)})$ as the **diagonal matrix** of activation function derivatives evaluated at the net inputs $\mathbf{n}^{(m)}$:

$$\boxed{\mathbf{F}'^{(m)}(\mathbf{n}^{(m)}) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \cdots & 0 \\ 0 & \dot{f}^m(n_2^m) & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \dot{f}^m(n_{S^m}^m) \end{bmatrix} \in \mathbb{R}^{S^{(m)} \times S^{(m)}}} \qquad \dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m} = \frac{\partial a_j^{(m)}}{\partial n_j^m}$$

where each diagonal entry is the scalar derivative of the activation function applied to neuron j's net input:

Why diagonal? Because $\mathbf{f}^{(m)}$ acts **element-wise**: neuron j's activation only depends on its own net input n_j^m, not on any other neuron's net input. So the cross-derivatives are all zero, leaving only the diagonal. The matrix is $S^{(m)} \times S^{(m)}$ because layer m has $S^{(m)}$ neurons, each contributing one derivative.

For ReLU: $\dot{f}(n) = 1$ if $n > 0$, else 0: the diagonal is all 1s and 0s. For sigmoid: $\dot{f}(n) = \sigma(n)(1-\sigma(n))$: the diagonal holds values between 0 and 0.25. This matrix is the chain rule ingredient for one layer: it tells you how much each neuron's net input affects its own output.

⋆ **Sensitivity vector $\mathbf{s}^{(m)}$ and Backward Pass** The **sensitivity** $\mathbf{s}^{(m)} \in \mathbb{R}^{S^{(m)}}$ of layer m measures how much J changes with respect to the net input $\mathbf{n}^{(m)}$ of that layer:

$$\mathbf{s}^{(m)} = \frac{\partial J}{\partial \mathbf{n}^{(m)}}$$

Computed starting at the output layer and propagating backwards:

Output layer $(m = M)$: $\boxed{\mathbf{s}^{(M)} = -2\,\mathbf{F}'^{(M)}(\mathbf{n}^{(M)})\,(\mathbf{y} - \mathbf{a})}$

Hidden layers $(m = M-1, M-2, \ldots, 1)$: $\boxed{\mathbf{s}^{(m)} = \mathbf{F}'^{(m)}(\mathbf{n}^{(m)})\,\mathbf{W}^{(m+1)}\,\mathbf{s}^{(m+1)}}$

Propagation flows: $\mathbf{s}^{(M)} \to \mathbf{s}^{(M-1)} \to \cdots \to \mathbf{s}^{(2)} \to \mathbf{s}^{(1)}$

The sensitivity vector $\mathbf{s}^{(m)} = \frac{\partial J}{\partial \mathbf{n}^{(m)}} \in \mathbb{R}^{S^{(m)}}$ tells you how much the total error J would change if you nudged the net input $\mathbf{n}^{(m)}$ of each neuron in layer m: it is the "blame score" for each neuron, large $|s_j^{(m)}|$ means neuron j in layer m is highly responsible for the current error.

Breaking down the output layer sensitivity $\mathbf{s}^{(M)} = -2\,\mathbf{F}'^{(M)}(\mathbf{n}^{(M)})\,(\mathbf{y} - \mathbf{a})$:

- $(\mathbf{y} - \mathbf{a}) \in \mathbb{R}^{S^{(M)}}$: the **error vector**: how wrong each output neuron is. This is the starting signal: large error = large sensitivity.
- -2: comes from differentiating $J = \mathbf{e}^T\mathbf{e} = \|\mathbf{y} - \mathbf{a}\|^2$ with respect to $\mathbf{a}$, giving $\nabla_{\mathbf{a}} J = -2(\mathbf{y} - \mathbf{a})$
- $\mathbf{F}'^{(M)}(\mathbf{n}^{(M)})$: diagonal matrix of activation derivatives at the output layer: applies the chain rule through the final activation function

Breaking down the hidden layer sensitivity $\mathbf{s}^{(m)} = \mathbf{F}'^{(m)}(\mathbf{n}^{(m)})\,\mathbf{W}^{(m+1)}\,\mathbf{s}^{(m+1)}$:

- $\mathbf{s}^{(m+1)} \in \mathbb{R}^{S^{(m+1)}}$: the sensitivity already computed for layer $m+1$, carried backward one step
- $\mathbf{W}^{(m+1)} \in \mathbb{R}^{S^{(m)} \times S^{(m+1)}}$: the weight matrix of layer $m+1$: routes the error signal back through the connections (transposition of the forward pass direction)
- $\mathbf{F}'^{(m)}(\mathbf{n}^{(m)})$: diagonal activation derivative matrix for layer m: applies the chain rule through layer m's activation

The hidden layer formula is the chain rule made explicit. To find how J depends on layer m's net input, you must first know how J depends on layer $m+1$'s net input ($\mathbf{s}^{(m+1)}$), then multiply by the weights connecting the two layers ($\mathbf{W}^{(m+1)}$) to route the gradient back, then multiply by the activation derivative ($\mathbf{F}'^{(m)}$) to pass through layer m's nonlinearity. This is repeated until you reach layer 1.

Generalization to other loss functions. The output sensitivity $\mathbf{s}^{(M)} = -2\mathbf{F}'^{(M)}(\mathbf{n}^{(M)})(\mathbf{y} - \mathbf{a})$ is specific to $J = \mathbf{e}^T\mathbf{e}$. For any other objective function (e.g. cross-entropy), simply replace $-2(\mathbf{y} - \mathbf{a})$ with $\nabla_{\mathbf{a}} J$, the gradient of J with respect to the network output $\mathbf{a}$:

$$\mathbf{s}^{(M)} = \mathbf{F}'^{(M)}(\mathbf{n}^{(M)})\,\nabla_{\mathbf{a}} J$$

The hidden layer formula $\mathbf{s}^{(m)} = \mathbf{F}'^{(m)}(\mathbf{n}^{(m)})\mathbf{W}^{(m+1)}\mathbf{s}^{(m+1)}$ stays exactly the same regardless of the loss.

This is the beauty of backprop's modular design: only the very first backward step (at the output layer) depends on the loss function. Everything else (the chain rule through hidden layers) is identical for any loss. Swapping $J = \mathbf{e}^T\mathbf{e}$ for cross-entropy only changes the starting vector $\nabla_{\mathbf{a}} J$.

8.4.2 Weight Update Rules

⋆ **Matrix weight update.** Once the sensitivities $\mathbf{s}^{(m)}$ are computed for all layers, update the weights and biases of every layer simultaneously:

$$\boxed{\mathbf{W}^{(m)}(k+1) = \mathbf{W}^{(m)}(k) - \alpha\, \mathbf{a}^{(m-1)}\, \mathbf{s}^{(m)T}} \qquad \boxed{\mathbf{b}^{(m)T}(k+1) = \mathbf{b}^{(m)T}(k) - \alpha\, \mathbf{s}^{(m)T}}$$

- $\mathbf{W}^{(m)}(k) \in \mathbb{R}^{S^{(m-1)} \times S^{(m)}}$: current weight matrix of layer m at iteration k
- $\mathbf{a}^{(m-1)} \in \mathbb{R}^{S^{(m-1)}}$: activation output of the **previous** layer, computed during the forward pass and stored. This is the input that arrived at layer m.
- $\mathbf{s}^{(m)T} \in \mathbb{R}^{1 \times S^{(m)}}$: sensitivity of layer m, computed during the backward pass. This is how much J cares about each neuron in layer m.
- $\mathbf{a}^{(m-1)}\mathbf{s}^{(m)T} \in \mathbb{R}^{S^{(m-1)} \times S^{(m)}}$: the **outer product**: a matrix of the same shape as $\mathbf{W}^{(m)}$, giving the gradient $\frac{\partial J}{\partial \mathbf{W}^{(m)}}$. Entry (i, j) = how much weight $w_{ij}^{(m)}$ contributed to the error.
- α: learning rate: scales the step size downhill
- **Bias update $\mathbf{b}^{(m)T}(k+1) = \mathbf{b}^{(m)T}(k) - \alpha\, \mathbf{s}^{(m)T}$:** The gradient of J with respect to the bias is simply $\mathbf{s}^{(m)}$ itself: the bias has no incoming activation to multiply by (it is always multiplied by the constant 1), so the update is just the sensitivity directly.

Compare with the perceptron update: $\mathbf{W}(i+1) = \mathbf{W}(i) + \alpha\, \mathbf{x}(i)\, \mathbf{e}^T(i)$:

- Perceptron: $+\alpha\, \mathbf{x}\, \mathbf{e}^T$: add input times error
- MLP backprop: $-\alpha\, \mathbf{a}^{(m-1)}\, \mathbf{s}^{(m)T}$: subtract input times sensitivity

Same structure: **input to the layer** $\times$ **error signal at the layer**. The perceptron is just a single-layer special case of backprop.

The outer product $\mathbf{a}^{(m-1)}\mathbf{s}^{(m)T}$ has shape $S^{(m-1)} \times S^{(m)}$, exactly matching $\mathbf{W}^{(m)}$. Entry (i, j) encodes: how much did the i-th input activation contribute to the j-th neuron's sensitivity? Weights that connected large activations to sensitive neurons get updated the most.

⋆ **Full training loop:**

One iteration:

1. Randomly select one training sample $(\mathbf{x}(k), \mathbf{y}(k))$
2. **Forward pass**: compute $\mathbf{a}^{(0)}, \mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(M)}$ and store all intermediate activations
3. **Error estimation**: compute $\mathbf{e}(k) = \mathbf{y}(k) - \mathbf{a}^{(M)}$ and approximate $J \approx \mathbf{e}^T\mathbf{e}$
4. **Backward pass**: compute $\mathbf{s}^{(M)}, \mathbf{s}^{(M-1)}, \ldots, \mathbf{s}^{(1)}$ backwards through the network
5. **Weight update**: update all $\mathbf{W}^{(m)}$ and $\mathbf{b}^{(m)}$ using the sensitivity vectors

One epoch: Complete one full pass through all N training samples: N iterations. One epoch includes N weight updates.

Training: Requires **multiple epochs**, the network sees the full training set many times, progressively reducing J with each pass until convergence or a stopping criterion is met.

The training loop is a cycle of three steps repeated thousands of times: forward (get prediction), error (measure mistake), backward (figure out who is responsible and by how much), update (fix it a little). Each full cycle is one iteration. After N cycles (one epoch) you have used every training point once. Training for 100 epochs means the network has seen each point 100 times, each time slightly adjusting all weights based on that point's error.

8.5 Regularization Methods

⋆ **Regularization** Any method that **prevents overfitting** or **helps the optimization** is called **regularization**. Two broad categories exist for neural networks:

- **Analytical regularization**: add a penalty term directly to the objective function J: e.g. L1 or L2 penalties on the weights. This modifies the backpropagation gradients directly.
- **Empirical regularization**: practical tricks applied during training that implicitly constrain the network: no explicit penalty term added to J. Very popular in modern deep learning. Act as implicit regularizers without modifying the objective function directly. These include **dropout**, **early stopping**, **data augmentation**, and **adding noise**

The core problem regularization solves: a large MLP has millions of parameters and can memorize the training data perfectly ($J \approx 0$) while failing completely on unseen data. Regularization forces the network to learn generalizable patterns rather than memorizing noise.

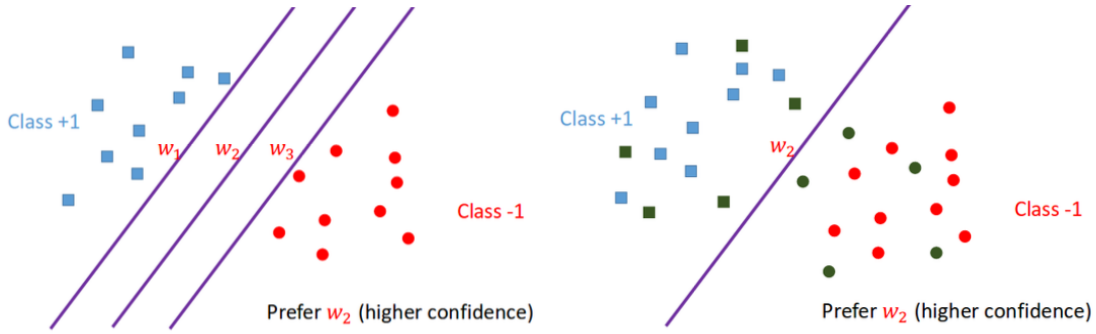
⋆ **L2 Regularization = Weight Decay.** Adding an L2 penalty $\frac{\eta}{2}\|\mathbf{W}^{(m)}\|_F^2$ to J for each layer is equivalent to **weight decay**, multiplying every weight by a shrinkage factor $(1-\eta\alpha) < 1$ at every update step:

$$\boxed{\mathbf{W}^{(m)}(k+1) = (1-\eta\alpha)\,\mathbf{W}^{(m)}(k) - \alpha\,\mathbf{a}^{(m-1)}\mathbf{s}^{(m)T}} \qquad \boxed{\mathbf{b}^{(m)T}(k+1) = (1-\eta\alpha)\,\mathbf{b}^{(m)T}(k) - \alpha\,\mathbf{s}^{(m)T}}$$

- $(1-\eta\alpha)\,\mathbf{W}^{(m)}(k)$: **decay term**: multiplies every weight by a factor slightly less than 1 at every step, continuously shrinking weights toward zero regardless of the gradient. This is the regularization effect.
- $-\alpha\,\mathbf{a}^{(m-1)}\mathbf{s}^{(m)T}$: **gradient term**: same as the standard backprop update, pushing weights in the direction that reduces J
- $\eta > 0$: decay rate: controls how aggressively weights are shrunk. Too large and all weights collapse to zero; too small and overfitting is not prevented.
- $\eta\alpha$: the combined shrinkage per step: must satisfy $\eta\alpha < 1$ to keep weights from exploding

8.5.1 Adding Noise and Data Augmentation

⋆ **Adding Noise to the Input.** At each training iteration, replace $\mathbf{x}(k)$ with $\tilde{\mathbf{x}}(k) = \mathbf{x}(k) + \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon}$ is small random noise, then train on the corrupted input. This forces the network to learn a decision boundary **robust to small perturbations**: only a boundary sitting far from both classes (high confidence margin, w_2 in the figure) remains correct after noise. Boundaries too close to the data (w_1, w_3) will misclassify noisy points.



Drawback. Too much noise causes perturbed points to **cross the boundary** into the wrong class region, corrupting labels and sending contradictory signals to the network. Keep noise magnitude small.

⋆ **Adding Noise to Weights.** Add random noise directly to $\mathbf{W}^{(m)}$ at each iteration before the forward pass. This is mathematically **equivalent to adding a regularization penalty to** J: only weights that are small and robust survive, producing the same effect as L2 regularization.

⋆ **Data Augmentation.** Artificially expand the training set by adding **label-preserving transformed copies** of existing examples:

$$(\mathbf{x}(k), \mathbf{y}(k)) \longrightarrow (\mathbf{x}(k), \mathbf{y}(k)),\ (T_1(\mathbf{x}(k)), \mathbf{y}(k)),\ (T_2(\mathbf{x}(k)), \mathbf{y}(k)),\ \ldots$$

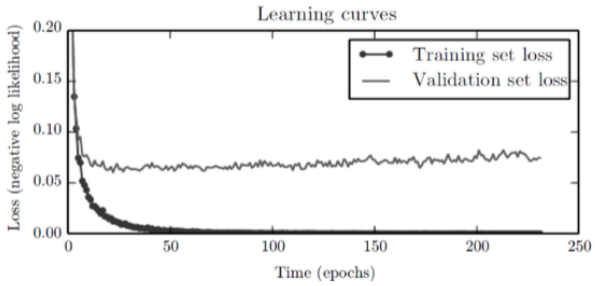
Common image transformations: horizontal flip, random crop, rotation. The network learns features **invariant** to the applied transformations.

Warning: transformations must preserve the label. If the transformation changes the true class, augmentation corrupts your training data:

8.5.2 Early Stopping

⋆ **Early Stopping.** Instead of training until $J_{\text{train}} \approx 0$ (which overfits), monitor the **validation error** during training and **stop when it starts increasing**; that is the point where the network transitions from learning to memorizing.

Training loss always decreases: the network can always memorize the training data better with more epochs. Validation loss tells you when generalization stops improving. The two curves diverging is the signature of overfitting: the network is now fitting noise in the training set rather than the true underlying pattern.



1. Monitor validation error every epoch during training
2. Every time validation error **improves**, save a checkpoint copy of $\mathbf{W}^{(m)}, \mathbf{b}^{(m)}$ for all m
3. If validation error has not improved for some number of epochs, **stop training**
4. Return the **stored checkpoint**, not the final weights, but the best ones seen

Validation error. J_{val} is J evaluated on the held-out validation set with weights **frozen** no updates, just a forward pass: $J_{\text{val}} = \frac{1}{N_{\text{val}}} \sum_{k=1}^{N_{\text{val}}} \mathbf{e}(k)^T \mathbf{e}(k)$

where N_{val} is the number of validation points. It measures how well the current weights generalize to unseen data.

The final weights after overfitting are worse than the checkpoint weights from earlier. By storing the best weights seen so far, you always recover the model at its peak generalization performance regardless of when you stop.

Reusing validation data. After the first run identifies the optimal number of epochs T^*, run a second training that uses **both training and validation data**. Two strategies:

- **Strategy 1**: start fresh, train on training + validation combined for exactly T^* epochs. Uses the stopping point discovered in run 1 but now trains on all available data.
- **Strategy 2**: start from the weights found in run 1, continue training on training + validation data until the training loss drops below the validation loss observed at the early stopping point T^*.

Both strategies recover the wasted validation data after using it for its intended purpose: finding T^*. Strategy 1 is simpler; strategy 2 exploits the already-trained weights and typically requires fewer additional epochs.

8.5.3 Dropout

⋆ **Dropout.** At each update step, sample a **binary mask** $\mathbf{m} \in \{0,1\}^{S^{(m)}}$ for every input and hidden layer, multiply it element-wise with the activations, then run forward and backward pass as usual, effectively **randomly switching off neurons** for that iteration:

$$\tilde{\mathbf{a}}^{(m)} = \mathbf{m}^{(m)} \odot \mathbf{a}^{(m)}, \quad m_j^{(m)} \sim \text{Bernoulli}(1-p)$$

where p is the **dropout probability**: the chance a unit is zeroed out.
Typical values: $p = 0.2$ for input units, $p = 0.5$ for hidden units. A new mask is sampled independently at **every iteration**.

Dropout forces the network to never rely on any single neuron being present, every neuron must learn to be useful on its own without counting on its neighbours. This prevents **co-adaptation**: groups of neurons that only work together to memorize specific training examples. The result is a more robust, distributed representation that generalizes better. At test time, all neurons are active and weights are scaled by $(1-p)$ to account for the fact that more units are present than during training.

8.5.4 Adversarial Training

Adversarial Training. A regularization technique in which the training set is augmented with **adversarial examples**, inputs deliberately perturbed by a small, carefully computed noise that maximally increases the model's loss, forcing the network to learn decision boundaries that are robust to worst-case input perturbations, not just random noise.

8.6 Convolutional Neural Networks (CNNs)

⋆ **The Challenge of Dealing with Images.** Standard image classification benchmarks — **MNIST** (handwritten digits, 10 classes) and **CIFAR-10** (10 natural image classes, $32 \times 32 \times 3$ pixels) — reveal a structural problem: raw images are spatially structured inputs that are fundamentally different from tabular feature vectors.

⋆ **Image Classification and MLPs: Weight Proliferation.** Although a two-layer MLP can reach 98.2% accuracy on MNIST, **fully connected MLPs are not the model of choice for image tasks**. The reason: treating each pixel channel as an independent input means every neuron in the first hidden layer requires one weight per input. For a $32 \times 32 \times 3$ image:

$$\#\,\text{params per neuron} = 32 \times 32 \times 3 = 3{,}072$$

The total weight count grows unmanageably as image resolution increases. This is called **weight proliferation**.

The fully connected layer has no notion of spatial proximity: it treats a pixel next to its neighbor the same as one on the opposite side of the image. This is the wrong inductive bias for images.

⋆ **Exploiting Spatial Structure.** Since adjacent pixels carry correlated local information, that information can be **summarized** rather than discarded. The operator that performs this summarization is the **convolution operator**.

⋆ **Kernel / Filter.** The **kernel** (or **filter**) $\mathbf{K} \in \mathbb{R}^{k \times k}$ is a small matrix whose entries encode a way of extracting a specific spatial feature from an image.

⋆ **Feature Map.** Sliding the kernel across image I and computing at each position (i, j) the sum of element-wise products with the local patch produces the **feature map** (also: **convolved feature** or **activation map**):

$$\boxed{\text{FM}(i,j) = \sum_{m=0}^{k-1}\sum_{n=0}^{k-1} K(m,n)\cdot I(i+m,\, j+n)}$$

Each entry of FM is one scalar summary of what the kernel found at that spatial location.

The same kernel is reused at every spatial location; this weight sharing *is what makes convolution vastly more parameter-efficient than a fully connected layer.*

⋆ **Different Kernels Produce Different Feature Maps.** The weight pattern in **K** fully determines which spatial structure the convolution responds to. Different kernels applied to the same input produce completely different feature maps.

Each kernel is a feature detector: the pattern of weights encodes what contrast structure maximally activates it. The feature map is the spatial record of where that pattern fired across the image.

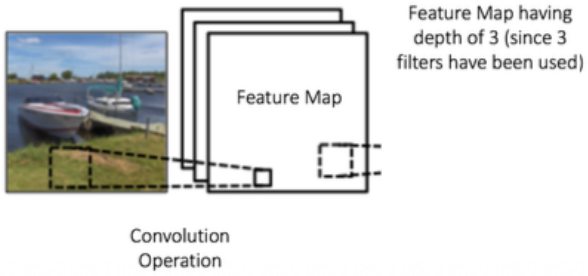
⋆ **CNNs Learn the Kernel Values.** Choosing kernels by hand does not scale. A **Convolutional Neural Network (CNN)** treats every entry of **K** as a **learnable parameter** and discovers optimal filter values via backpropagation, the network learns which features minimize the training objective.

Convolution solves weight proliferation via spatial weight sharing; learning solves feature engineering by finding the best kernels from data. CNNs combine both.

⋆ **Convolutional Neural Network (CNN).** A **CNN** is a neural network that replaces fully connected layers with a repeated pipeline of three operations:
convolution (detect local features via a learned kernel)
ReLU (introduce nonlinearity)
max pooling (compress spatial size while retaining what was detected) stacked many times so that early layers detect simple patterns (edges, textures) and deeper layers combine them into increasingly abstract concepts (shapes, objects), with a final **fully connected layer** that classifies from the learned representation.

8.6.1 Structure of a CNN

⋆ **Structure of a CNN: Feature Map Size Parameters.** The size of the **feature map** produced by a convolutional layer is controlled by three parameters: **depth**, **stride**, and **zero-padding**.



Depth. The number of distinct filters applied to the input. Each filter produces one feature map, so D filters produce D feature maps, viewed as a 3D volume of depth D, one slice per filter.
Each filter learns to detect a different pattern; depth controls how many parallel feature detectors operate on the same input.

2. Stride s. The number of pixels the kernel moves per step. Larger stride skips more positions and yields a smaller feature map: $\boxed{\text{FM size} = \left\lfloor \frac{H-k}{s} + 1 \right\rfloor \times \left\lfloor \frac{W-k}{s} + 1 \right\rfloor}$ where $H \times W$ is the input size, k is the kernel size, and s is the stride.

Stride $s = 1$ gives a dense, overlapping scan; stride $s = 2$ halves the output size in each dimension.

3. Zero-Padding. The input image is bordered with zeros before convolution. This controls whether border pixels are treated equally to interior ones and directly controls the output feature map size:

- **Wide convolution**: zero-padding added, output size $\geq$ input size.
- **Narrow convolution**: no padding, output strictly smaller than input.

⋆ **Introducing Nonlinearity: ReLU on Feature Maps.** Convolution is a linear operation. After each convolution, **ReLU** is applied **element-wise per pixel** to the feature map, replacing all negative values with zero:

$$\text{Rectified FM}(i,j) = \max(0,\ \text{FM}(i,j))$$

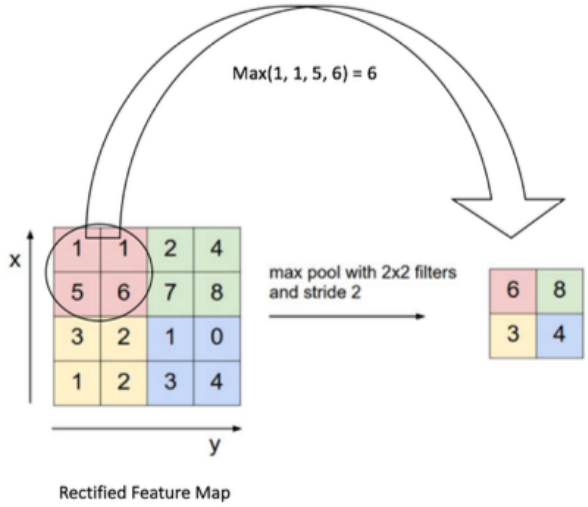
The output is called a **rectified feature map**. This introduces the nonlinearity needed to model complex, non-linear classification boundaries. ReLU outperforms tanh and sigmoid for this purpose in practice.

Without ReLU, stacking convolution layers would still be a linear operation and no depth would add expressive power.

⋆ **Pooling = Downsampling.** **Spatial pooling** (subsampling) reduces the spatial dimensions of each rectified feature map while retaining the most important information. Types: **Max**, **Average**, **Sum**; Max pooling is standard in practice. For a $p \times p$ window with stride s:

$$\boxed{\text{Pool}(i,j) = \max_{0 \leq m,n < p} \text{FM}(i \cdot s + m,\ j \cdot s + n)}$$

Pooling is applied **independently to each feature map**.



- Reduces spatial dimensions $\rightarrow$ fewer parameters $\rightarrow$ controls overfitting.
- **Invariant** to small translations, distortions, and transformations: if a feature shifts slightly within a pooling window, the maximum is unchanged.
- Yields a nearly **scale-invariant** representation: objects can be detected regardless of exact location.

Core division of labor: convolutions extract features from the image; pooling makes those features invariant to transformations.

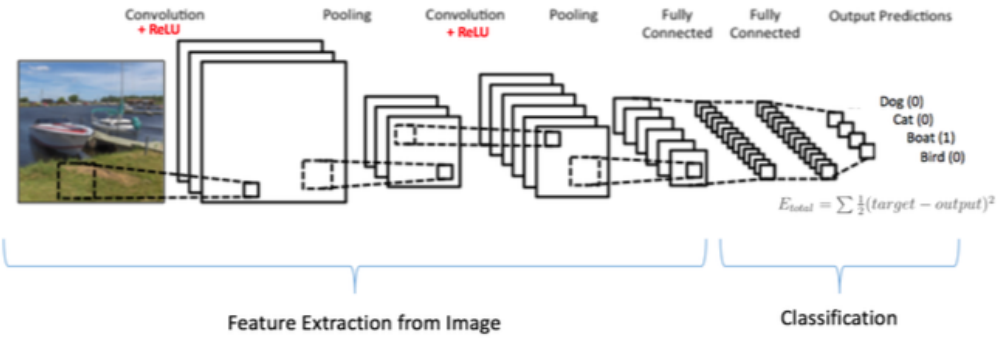
⋆ **Convolution–ReLU–Pooling Repeated.** The triplet **Conv** $\rightarrow$ **ReLU** $\rightarrow$ **Pool** forms one **convolutional layer** and can be stacked many times, each with its own depth and stride. Early layers detect simple local patterns; deeper layers combine them into increasingly abstract representations.

⋆ **Fully Connected Layer: Feature Combinations.** Individual features extracted by convolutional and pooling layers are already useful, but **non-linear combinations** of those features can be even better. The **fully connected layer** learns these combinations cheaply: it is just a standard MLP applied to the flattened pooling output.

Convolutions find what is present locally; the fully connected layer learns how to combine those local findings into a global decision.

⋆ **Training a CNN: Backpropagation.** A CNN is trained end-to-end using the same backpropagation + gradient descent framework as an MLP.

The total error across all output neurons is: $\boxed{E_{\text{total}} = \sum_i \frac{1}{2}(\text{target}_i - \text{output}_i)^2}$



1. Initialize all filter values and weights randomly.
2. **Forward pass**: run each image through the full network and read the output predictions.
3. **Compute error**: $E_{\text{total}} = \sum_i \frac{1}{2}(\text{target}_i - \text{output}_i)^2$.
4. **Backward pass**: use backpropagation to compute gradients of E_{total} with respect to every weight and filter value in the network.
5. **Update**: apply gradient descent: adjust each weight in proportion to its contribution to the total error.
6. Repeat for many epochs.

What is fixed vs. learned. The **architecture** (number of filters, filter sizes, depth, stride, network structure) is a fixed hyperparameter set before training and does not change. Only the **filter matrix values** and **fully connected weights** are updated by backpropagation.

This is the same backprop algorithm from the MLP section; nothing changes conceptually. The filter entries are simply additional learnable parameters, treated identically to connection weights during gradient descent.

8.6.2 Softmax and Cross-Entropy

⋆ **Two Problems with Squared Error $J = \mathbf{e}^T\mathbf{e}$.**

Problem 1: When the output layer uses sigmoid and the true label is 1 but the network output is ≈ 0, the sigmoid is deep in saturation: $\dot{f}(n) \approx 0$, so the gradient of J with respect to the net input is nearly zero: the network cannot correct the error regardless of how wrong it is.

Problem 2 : For mutually exclusive class labels, outputs should form a valid probability distribution summing to 1. Squared error treats each output independently and does not enforce this constraint: the network has no knowledge that its outputs should be a distribution.

Solution. Force the output layer to represent a proper **probability distribution across discrete classes**: use **softmax** as the output activation and **cross-entropy** as the loss.

⋆ **Softmax.** Turns any vector of real-valued scores into class probabilities: it exponentiates each score to make it positive, then divides by the total so all outputs sum to 1.
AKA A **non-local** output activation that converts a vector of raw scores (**logits**) into a valid probability distribution by exponentiating each score and normalizing by the sum across all classes guaranteeing outputs are positive and sum to 1.

The output units in the final layer use a **non-local** nonlinearity: each output a'_i depends on *all* logits a_j in the group, ensuring outputs are positive and sum to 1:

$$\boxed{a'_i = \frac{e^{a_i}}{\sum_{j \in \text{group}} e^{a_j}}} \qquad \frac{\partial a'_i}{\partial a_i} = a'_i(1 - a'_i)$$

The pre-softmax input a_i is called the **logit**. Softmax converts a vector of arbitrary real-valued scores into a valid probability distribution.

Unlike sigmoid or ReLU which act on one neuron independently, softmax couples all output neurons: increasing one probability necessarily decreases the others. This enforces the constraint that class probabilities sum to 1.

⋆ **Relationship with Logistic Regression.** In **logistic regression** the logit is a **linear** function of the features: $a = \boldsymbol{\beta}^T \mathbf{x}(j) + \beta_0$, with 0 as the logit for the second class ($z(j) \in \{0, 1\}$). Applying softmax recovers the logistic sigmoid:

$$p = \frac{e^a}{e^0 + e^a} = \frac{e^a}{1 + e^a}, \qquad 1 - p = \frac{1}{1 + e^a}$$

The negative log-likelihood for one point is: $-z(j) \log p - (1 - z(j)) \log(1 - p)$. Renaming $y_1 = 1 - z(j)$, $y_2 = z(j)$ and softmax outputs $a'_1 = 1 - p$, $a'_2 = p$, this becomes the **cross-entropy**:

$$J' = -\sum_{i=1}^{S^{(M)}} y_i \log a'_i$$

In a neural network, the logits a_i are **nonlinear** functions of the input learned through multiple layers, making a network with softmax output a **nonlinear generalization of logistic regression**.

Logistic regression is just a one-layer network with a softmax output and cross-entropy loss. A deep network with softmax output learns the same probabilistic model but with logits computed by highly nonlinear feature extractors rather than a fixed linear function.

⋆ **Cross-Entropy: The Right Loss for Softmax.** The cross-entropy loss for $S^{(M)}$ output classes over one training point is:

$$\boxed{J' = -\sum_{i=1}^{S^{(M)}} y_i \log a'_i}$$

Its gradient with respect to the logit a_i (applying the chain rule through softmax) simplifies cleanly to:

$$\boxed{\frac{\partial J'}{\partial a_i} = \sum_j \frac{\partial J'}{\partial a'_j} \frac{\partial a'_j}{\partial a_i} = (a'_i - y_i)}$$

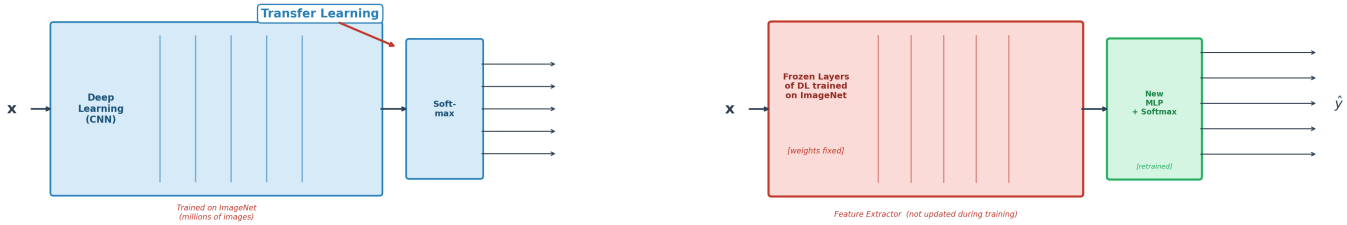
When the true label is $y_i = 1$ and the output $a'_i \approx 0$, this gradient is $(0 - 1) = -1$: **large**, regardless of saturation. This directly fixes the vanishing gradient problem of squared error.

Cross-entropy punishes confident wrong predictions exponentially via $-\log a'_i \to \infty$ *as* $a'_i \to 0$*, and provides a strong gradient signal exactly where squared error provides almost none.*

⋆ **Sigmoids in Hidden Layers.** Softmax + cross-entropy fixes the saturation problem at the **output layer**. In **hidden layers**, no such fix exists: sigmoid activations still saturate for large net inputs, producing near-zero gradients that block learning in earlier layers, especially as depth increases. **ReLU is the modern standard for hidden layers** precisely because it has no saturation region for positive inputs and passes gradients through unchanged.

8.7 Transfer Learning

⋆ **Transfer Learning.** Training a deep CNN from scratch requires millions of labeled images and days of compute. **Transfer learning** reuses a network already trained on a large dataset (e.g. **ImageNet** — 1.2M images, 1000 classes) for a new task with far less data. The key insight: the convolutional layers learn **general visual features** (edges, textures, shapes) that are useful across many tasks, not just the original one.



How it works.

- **Step 1**: Take a deep CNN (e.g. VGG, ResNet) pre-trained on ImageNet. Its convolutional layers are already excellent **feature extractors**.
- **Step 2**: **Freeze** the convolutional layers: fix their weights, do not update them during training.
- **Step 3**: Replace the final classifier (softmax layer) with a new MLP tailored to your task (different number of output classes).
- **Step 4**: Train **only the new MLP** on your (smaller) dataset; the frozen layers just extract features.

The frozen convolutional layers act as a fixed feature extractor: they transform any input image into a rich feature vector, and the new MLP learns to classify from that representation. You get the benefit of a network trained on millions of images without needing millions of images yourself.

CNN Architecture. Pooling does not have to follow every convolutional layer; multiple Conv+ReLU operations can be stacked in succession before a single pooling step. Some of the best performing CNNs have tens of convolutional and pooling layers.

⋆ **Visualizing CNN Features: Hierarchical Representation.** CNNs learn a **hierarchy of features**: each layer builds on the previous:

- **Layer 1**: detects low-level patterns directly from raw pixels: edges, orientations, color blobs.
- **Layer 2**: combines edges into simple shapes and textures — corners, curves, facial parts (eyes, mouths).
- **Layer 3+**: combines shapes into high-level semantic structures: full faces, object parts.

This hierarchy is not designed; it emerges *from training. The network discovers that edges are useful building blocks for shapes, and shapes are useful building blocks for objects. Real-world learned filters may detect patterns with no human-interpretable meaning at all.*

8.8 Recurrent Neural Network (RNN)

⋆ **Recurrent Neural Network (RNN).** A family of neural networks introduced in Rumelhart et al. (1986) for handling **sequential data**, where inputs and outputs can have **variable length**. Especially well suited for **natural language processing (NLP)**, but also speech, time series and any data with an order.

A standard MLP or CNN expects a fixed size input and outputs once. An RNN is built to process inputs that arrive one after another, where the length is not known in advance and can differ across examples.

⋆ **Sequential Data.** A single **data point** is a sequence of vectors: $\boxed{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(\tau)}}$ $1 \leq t \leq \tau$

- $\mathbf{x}^{(t)}$: the entry of the sequence at time step t (e.g. a word vector, an audio frame).
- τ : the **length** of the sequence, can vary from one data point to another.
- **Batch data** : a set of many sequences, each with its own τ.
- **Label** : can be a **scalar** (one prediction per sequence), a **vector**, or another **sequence**.

Typical tasks: **sentiment analysis** (sequence in, scalar out), **machine translation** (sequence in, sequence out).

The label structure determines the RNN architecture. A scalar label means we only read out at the end; a sequence label means we read out at every step or run an encoder/decoder pair.

⋆ **More Complicated Sequential Data.** The **input** need not be a 1D sequence of vectors :

- Data point can be a **2D input** like an image.
- Label can be a **different type** of sequence, e.g. a text sentence.

Image captioning is the canonical example : input is one image, output is a variable length sentence describing it.

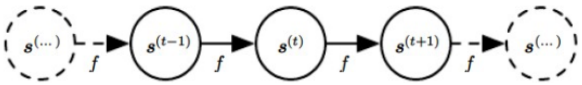
This shows RNNs are not restricted to one-dimensional inputs. Anything can be the source as long as the output has sequential structure ; in practice a CNN feature extractor is plugged into an RNN that generates the output sequence.

⋆ **Dynamical System View.** An RNN is mathematically a **discrete time dynamical system**, the state at the next step is a function of the state at the current step:

$$\boxed{\mathbf{s}^{(t+1)} = f(\mathbf{s}^{(t)}; \boldsymbol{\theta})}$$

- $\mathbf{s}^{(t)}$: **state** at time t, summarizes everything the system has computed so far.
- f : the **transition function**, fixed and reused at every step.
- $\boldsymbol{\theta}$: parameters of f, also fixed across time (same $\boldsymbol{\theta}$ at every step).

This is the abstract form of any recurrent computation : a single update rule applied repeatedly. The same f and $\boldsymbol{\theta}$ at every step is what makes the system "recurrent" rather than just deep.



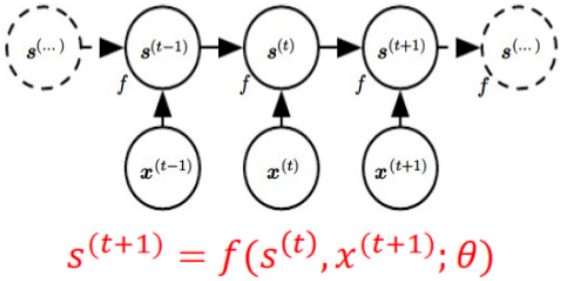
$s^{(t+1)} = f(s^{(t)}; \theta)$

⋆ **System Driven by External Input.** Real RNNs are not closed dynamical systems , each step receives a fresh input $\mathbf{x}^{(t+1)}$ from the outside:

$$\boxed{\mathbf{s}^{(t+1)} = f\left(\mathbf{s}^{(t)}, \mathbf{x}^{(t+1)}; \boldsymbol{\theta}\right)}$$

The transition function now takes **two arguments**: the previous state and the current external input. $\boldsymbol{\theta}$ still shared across all t.

This is the form actually used in RNNs : the hidden state evolves under both its own dynamics (through $\mathbf{s}^{(t)}$) and the influence of the input stream (through $\mathbf{x}^{(t+1)}$). The state acts as a memory of the past while continuously absorbing new information.



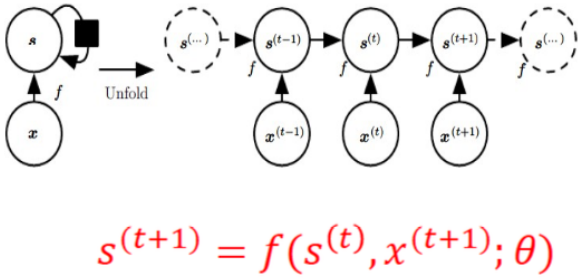
Finance intuition. Predicting tomorrow's market move : $\mathbf{x}^{(t)}$ is the new information arriving on day t (price, volume, news sentiment), and $\mathbf{s}^{(t)}$ is the model's running summary of the market built from all past days (e.g. "trend bullish, volatility rising"). The update $\mathbf{s}^{(t+1)} = f(\mathbf{s}^{(t)}, \mathbf{x}^{(t+1)}; \boldsymbol{\theta})$ just says : take yesterday's belief, mix in today's new data, get today's updated belief, and read the prediction off from $\mathbf{s}^{(t)}$. **GARCH(1,1)** is the textbook scalar version of this : conditional variance evolves as $\sigma_t^2 = \omega + \alpha\,\varepsilon_{t-1}^2 + \beta\,\sigma_{t-1}^2$, which fits the form exactly with state $\mathbf{s}^{(t)} = \sigma_t^2$ (running volatility), input $\mathbf{x}^{(t)} = \varepsilon_{t-1}^2$ (new squared return shock), and fixed parameters $\boldsymbol{\theta} = (\omega, \alpha, \beta)$. An RNN generalizes this : same recurrent structure, but f is a learned nonlinear function and $\mathbf{s}^{(t)}$ is a high dimensional vector instead of a single scalar.

⋆ **Folded vs. Unfolded (Compact) View.** The same RNN can be drawn in two equivalent ways :

- **Folded form** : a single state node $\mathbf{s}$ with a self loop and an input $\mathbf{x}$, the self loop carries a **one step time delay** (drawn as a small black square).
- **Unfolded form** : the same circuit "rolled out" in time, one copy of the state per step, $\mathbf{s}^{(t-1)} \to \mathbf{s}^{(t)} \to \mathbf{s}^{(t+1)}$, with inputs $\mathbf{x}^{(t-1)}, \mathbf{x}^{(t)}, \mathbf{x}^{(t+1)}$ feeding each step.

Key property. The same function f and the same parameters $\boldsymbol{\theta}$ are used at **every time step** of the unfolded graph. The folded picture is a compact way of expressing this weight sharing.

The folded form is what the model is conceptually : one cell with a memory loop. The unfolded form is what training operates on : a deep feedforward graph with tied weights, where depth equals sequence length. The diagram shows the same RNN drawn both ways : on the left, a single cell $\mathbf{s}$ with a self loop and input $\mathbf{x}$; on the right, the loop is "unfolded" in time into a chain $\mathbf{s}^{(t-1)} \to \mathbf{s}^{(t)} \to \mathbf{s}^{(t+1)}$ where each step takes its own input $\mathbf{x}^{(t)}$ but reuses the same f and $\boldsymbol{\theta}$.



Finance intuition. The folded form is the model you write down ("one volatility update rule"). The unfolded form is what you actually run on a price series : at every trading day the same rule fires, taking yesterday's market summary $\mathbf{s}^{(t-1)}$ and today's data $\mathbf{x}^{(t)}$ to produce today's summary $\mathbf{s}^{(t)}$. Training the RNN on a year of daily data is identical to training a 252 layer deep network whose layers all share the same weights.

⋆ **RNN Core Properties.**

- Use the **same computational function and parameters** across all time steps of the sequence.
- At each step, the cell takes the **current input entry** *and* the **previous hidden state** to compute the output entry.
- **Loss** is typically computed at **every time step** and summed over the sequence.

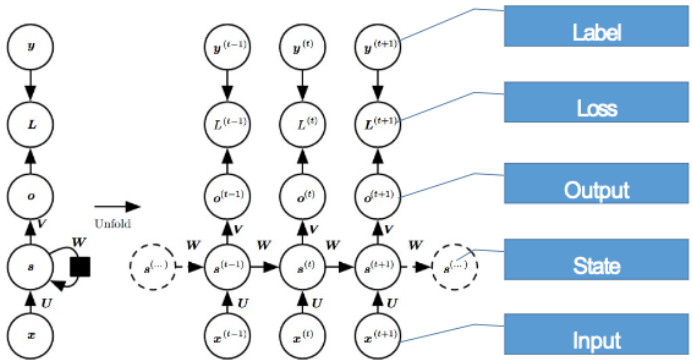
This is what makes the RNN handle sequences of any length with a fixed number of parameters : the previous hidden state acts as memory, while the shared function keeps the parameter count independent of sequence length.

⋆ **RNN Computational Graph.** The unfolded RNN has five components per time step:

- **Input** $\mathbf{x}^{(t)}$: sequence entry at time t, fed in via weight matrix $\mathbf{U}$.
- **State** $\mathbf{s}^{(t)}$: hidden state at time t, propagated forward via weight matrix $\mathbf{W}$.
- **Output** $\mathbf{o}^{(t)}$: raw output, produced from the state via weight matrix $\mathbf{V}$.
- **Loss** $L^{(t)}$: per step loss comparing $\mathbf{o}^{(t)}$ to the true label $\mathbf{y}^{(t)}$.
- **Label** $\mathbf{y}^{(t)}$: ground truth target at time t.

Weight matrices $\mathbf{U}, \mathbf{W}, \mathbf{V}$ are shared across all time steps.

At each time t, the input $\mathbf{x}^{(t)}$ is multiplied by $\mathbf{U}$ and fed into the state cell, which also receives the previous state $\mathbf{s}^{(t-1)}$ multiplied by $\mathbf{W}$. These two contributions combine inside the cell to form the new hidden state $\mathbf{s}^{(t)}$, the running memory of the sequence. The state is then projected through $\mathbf{V}$ to produce the output $\mathbf{o}^{(t)}$, which is compared against the true label $\mathbf{y}^{(t)}$ to compute the per step loss $L^{(t)}$. The same $\mathbf{U}, \mathbf{W}, \mathbf{V}$ are reused at every step. The total training loss is the sum of all $L^{(t)}$ across the sequence, and gradients flow backward not only through each step but also along the horizontal $\mathbf{W}$ arrows linking states across time.



Finance intuition. $\mathbf{U}$ decides how much weight to give today's market data, $\mathbf{W}$ decides how much yesterday's market summary should persist into today, and $\mathbf{V}$ decides how to read a prediction (e.g. tomorrow's return direction) out of the current summary. Errors observed at any future day propagate back through $\mathbf{W}$ to correct how the model handled past days.

⋆ **RNN Forward Equations.** At every time step t:

$$\boxed{\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{s}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}} \qquad \boxed{\mathbf{s}^{(t)} = \tanh\left(\mathbf{a}^{(t)}\right)}$$

$$\boxed{\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{s}^{(t)}} \qquad \boxed{\hat{\mathbf{y}}^{(t)} = \mathrm{softmax}\left(\mathbf{o}^{(t)}\right)}$$

- $\mathbf{a}^{(t)}$: pre activation of the hidden layer, blends the previous state with the current input plus a bias $\mathbf{b}$.
- $\mathbf{W}\mathbf{s}^{(t-1)}$: recurrent contribution, carries memory forward.
- $\mathbf{U}\mathbf{x}^{(t)}$: input contribution, injects the new entry of the sequence.
- $\mathbf{s}^{(t)} = \tanh(\mathbf{a}^{(t)})$: updated hidden state, tanh keeps values bounded in $(-1, 1)$.
- $\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{s}^{(t)}$: raw **logits**, linear readout from the state with output bias $\mathbf{c}$.
- $\hat{\mathbf{y}}^{(t)} = \mathrm{softmax}(\mathbf{o}^{(t)})$: output as a probability distribution (used for next step classification, e.g. next word prediction).

The state update is essentially a small one layer MLP with two inputs : the previous state and the current input. Everything the network "remembers" must be packed into the fixed size vector $\mathbf{s}^{(t)}$, which

is then handed forward to the next step.

tanh. Smooth nonlinearity that squashes each entry of its input into $(-1, 1)$, applied **element wise** : $\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}} \in (-1, 1)$.

softmax. Non local output activation that turns a vector of real valued scores (logits) into a valid probability distribution that sums to 1 : $\text{softmax}(\mathbf{o})_i = \dfrac{e^{o_i}}{\sum_j e^{o_j}}$.

⋆ **Advantages of RNNs.**

- **Hidden state as memory** : $\mathbf{s}^{(t)}$ is a **lossy summary** of all past inputs $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(t)}$, compressed into a fixed size vector regardless of sequence length.
- **Parameter sharing** : because $\mathbf{U}, \mathbf{W}, \mathbf{V}$ are reused at every step, the model has far fewer parameters than an unrolled feedforward network of the same depth, lowering **capacity** and improving **generalization**.
- **Prior knowledge built in** : weight sharing across time encodes the assumption that the same processing rule applies at every position in the sequence, which is the right inductive bias for language, audio, and most natural sequences.

Compression into a fixed size state is both the strength and the weakness of RNNs : it allows arbitrary length sequences to be handled with finite memory, but also causes information from the distant past to be progressively overwritten, motivating the gated architectures (LSTM, GRU) introduced later.

⋆ **Universality.** RNNs are **universal** : any function computable by a **Turing machine** can be computed by a finite size recurrent network (Siegelmann & Sontag, 1995).

Despite the heavy parameter sharing, RNNs lose no theoretical expressive power : any algorithmic computation is in principle within reach. The practical limits come from optimization and finite training data, not from the model class itself.

8.8.1 Training RNNs

⋆ **Training RNNs : Backpropagation Through Time (BPTT).** The RNN is trained by **unfolding** its computational graph in time, treating it as one large feedforward graph with **tied weights**, and applying standard backpropagation. The resulting algorithm is called **Backpropagation Through Time (BPTT)**. Once gradients are computed, any general purpose gradient based optimizer (SGD, Adam, etc.) can be used.

Two conceptual stages.

- First : compute the gradients of the loss with respect to the **internal nodes** (states $\mathbf{s}^{(t)}$ and outputs $\mathbf{o}^{(t)}$) at every time step.
- Then : compute the gradients with respect to the **parameters** $\mathbf{U}, \mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}$ by summing contributions across all time steps (since the same parameters appear at every step).

Because the same weight matrix is reused at every time step, its total gradient is the sum of the gradients arriving from each step. This is identical to backprop in a feedforward net with tied weights, but the chain of multiplications by $\mathbf{W}$ *along the temporal axis is exactly what causes the vanishing/exploding gradient problem discussed at the end.*

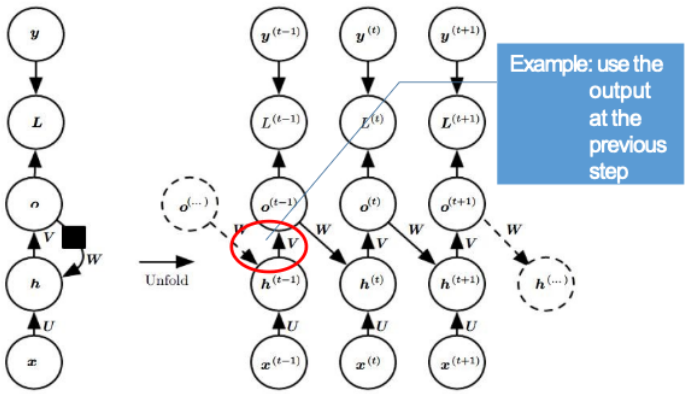
8.8.2 RNN Variants: Bidirectional and Encoder-Decoder RNNs

⋆ **RNN Variants.** The basic RNN is only one of many possible architectures. Two main directions of variation:

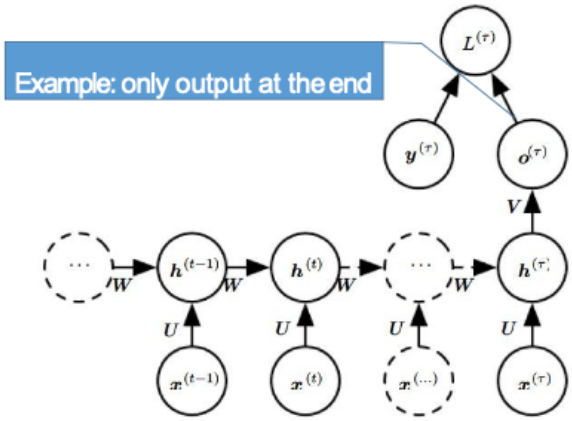
- **Different forms of "memory of the past"** : information from previous steps need not be carried only by the hidden state $\mathbf{s}^{(t)}$. It can also be carried by the previous output $\mathbf{o}^{(t-1)}$, by skip connections, or by attention.
- **Different output schedules** : instead of producing an output at every step, the network may emit only **one output at the end** of the sequence.

RNN variants are architectural choices, not separate models. The **RNN** is a **family of architectures**, not one fixed model. All variants are different ways of **wiring the same recurrent cell** : same BPTT training algorithm, same kinds of weight matrices $\mathbf{U}, \mathbf{W}, \mathbf{V}$, same forward equations at the cell level. What changes from one variant to another is only **where the recurrence flows** (through the hidden state vs. through the output) and **when outputs are read out** (every step vs. only at the end). The choice depends on the task : sequence labeling needs an output at every step, classification needs only one output at the end, teacher forcing favors output recurrence. None of these is a new theory, just a different connection pattern of the same building blocks.

⋆ **Variant 1 : output recurrence.** The recurrence runs through the **previous output** $\mathbf{o}^{(t-1)}$ instead of the previous hidden state. At each step, $\mathbf{h}^{(t)}$ is computed from the current input $\mathbf{x}^{(t)}$ and $\mathbf{o}^{(t-1)}$. *Reading the diagram : the red circle highlights how the previous output* $\mathbf{o}^{(t-1)}$ *(instead of* $\mathbf{h}^{(t-1)}$*) is fed forward into the next step's hidden state* $\mathbf{h}^{(t)}$ *via* $\mathbf{V}$*. The horizontal* $\mathbf{W}$ *arrows that normally connect states across time are replaced by output to state arrows.* This is a strictly less powerful variant : $\mathbf{o}^{(t)}$ is typically lower dimensional than $\mathbf{h}^{(t)}$, so passing only $\mathbf{o}^{(t-1)}$ forward loses information. Its advantage is that training can be parallelized across time steps when the true outputs $\mathbf{y}^{(t-1)}$ are available (**teacher forcing**) : each step's input is known in advance, so all hidden states can be computed independently.



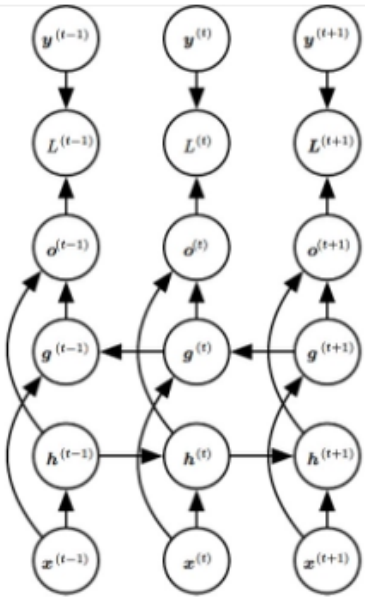
⋆ **Variant 2 : output only at the end.** A single output $\mathbf{o}^{(\tau)}$ is produced after the entire sequence has been read, with one loss $L^{(\tau)}$ computed against one target $\mathbf{y}^{(\tau)}$. *This is the standard architecture for* ***sequence classification*** *tasks (sentiment analysis, document level labeling) where the goal is to summarize the whole input into a single decision. The final hidden state* $\mathbf{h}^{(\tau)}$ *acts as a learned fixed length summary of the entire sequence.*



⋆ **Bidirectional RNNs (BiRNN).** A standard RNN at time t only sees inputs from the past $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(t)}$. In many applications the output at time t depends on the **whole input sequence**, including the future. **Bidirectional RNNs** solve this by running **two RNNs in parallel** :

- A **forward RNN** with hidden state $\mathbf{h}^{(t)}$, processing the sequence left to right.
- A **backward RNN** with hidden state $\mathbf{g}^{(t)}$, processing the sequence right to left.

The output at step t depends on **both** states : $\mathbf{o}^{(t)} = f(\mathbf{h}^{(t)}, \mathbf{g}^{(t)})$.



Classic motivation : in speech recognition, the correct interpretation of the current sound often depends on the next few phonemes or even the next few words. The forward state captures everything before t, the backward state captures everything after t, and combining them gives a representation of the full context surrounding step t.

*Reading the diagram : two parallel state chains run through the sequence. The bottom chain $\mathbf{h}^{(t)}$ flows left to right (forward RNN, sees only past inputs). The middle chain $\mathbf{g}^{(t)}$ flows right to left (backward RNN, sees only future inputs). At each time t, the output $\mathbf{o}^{(t)}$ is built from **both** $\mathbf{h}^{(t)}$ and $\mathbf{g}^{(t)}$, so it depends on the entire input sequence on both sides of t. The per step loss $L^{(t)}$ is then computed against the label $\mathbf{y}^{(t)}$ exactly as in a standard RNN.*

Limitation. Bidirectional RNNs cannot be used in **online / real time** settings, since they need access to the entire input sequence (including future entries) before producing any output. They are appropriate when the full sequence is available at inference time (offline transcription, document tagging, etc.).

Alternative : RTRL. Besides BPTT, RNNs can also be trained with **Real Time Recurrent Learning (RTRL)**, which propagates gradients **forward** in time alongside the activations instead of backward at the end.

Useful for online / streaming settings where the full sequence is never available, but rarely used in practice because it is far more expensive per step than BPTT.

⋆ **Encoder Decoder RNNs (Sequence to Sequence).** A plain RNN can map a sequence to either **one vector** (output at the end) or to **a sequence of the same length** (output at every step). It cannot directly map a sequence of length n_x to a sequence of **different** length n_y. The **encoder decoder** architecture (also called **seq2seq**) solves this problem.

Typical applications : **machine translation**, **speech recognition**, **question answering**, **summarization**, basically any task where input and output sequences have unrelated lengths.

Setup. Learn to generate an output sequence given an input sequence:

$$\boxed{\left(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n_x)}\right) \longrightarrow \left(\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(n_y)}\right)}$$

where n_x and n_y may differ and may both vary across examples.

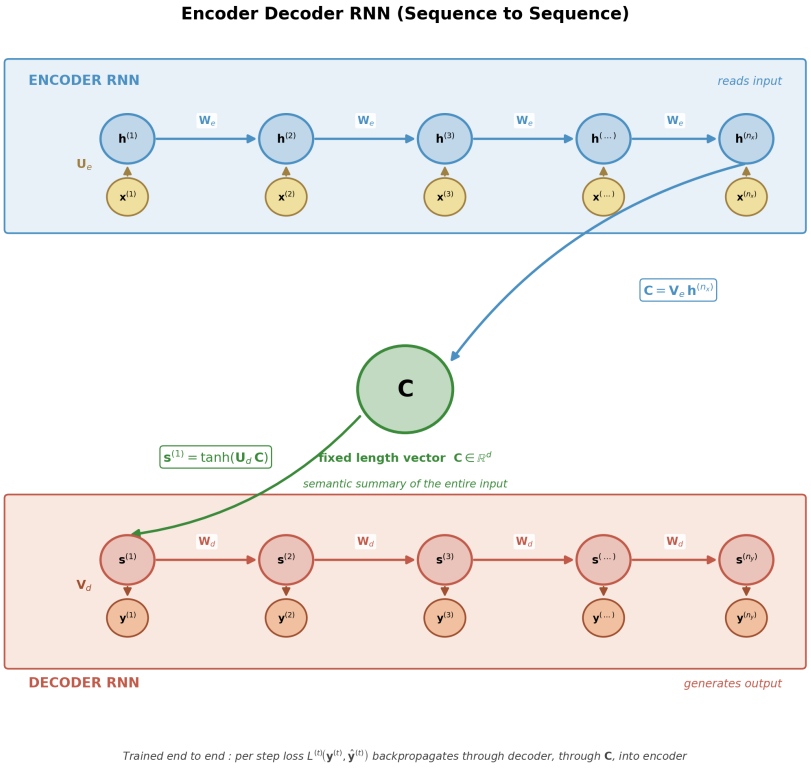
Encoder Decoder Architecture. Two RNNs are chained together via a **context vector**:

- **Encoder RNN** : reads the input sequence $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n_x)}$ one entry at a time. Its **final hidden state** is used to compute a fixed size **context vector C** that semantically summarizes the entire input.

- **Decoder RNN** : conditioned on $\mathbf{C}$, generates the output sequence $\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n_y)}$ one entry at a time. Equivalently, it can compute the probability of any candidate output sequence given $\mathbf{C}$.

Matrix	Transformation
$\mathbf{U}_e$	input → encoder hidden state
$\mathbf{W}_e$	encoder state → next encoder
$\mathbf{V}_e$	final encoder state → context
$\mathbf{U}_d$	context $\mathbf{C}$ → initial decoder s
$\mathbf{W}_d$	decoder state → next decoder
$\mathbf{V}_d$	decoder state → output $\mathbf{y}^{(t)}$

$\mathbf{C} \in \mathbb{R}^d$ *is a fixed length vector regardless of input length* n_x. *The encoder compresses the entire input into* $\mathbf{C}$, *and the decoder uses* $\mathbf{C}$ *to seed and condition its own recurrence as it generates the output sequence.*



Role of C. $\mathbf{C}$ is a **fixed size bottleneck** : the decoder must reconstruct or generate the entire output using only this single vector and its own recurrence. The encoder is forced to compress all relevant input information into $\mathbf{C}$.

This is the classical Sutskever et al. (2014) seq2seq design. Encoder and decoder are trained ***jointly****, end to end, by backpropagating the decoder's per step loss all the way back through* $\mathbf{C}$ *into the encoder. The fixed size of* $\mathbf{C}$ *is also its main weakness : long input sequences are hard to compress losslessly into one vector, which is the limitation that motivated the later* ***attention mechanism****.*

⋆ **The Challenge of Long Term Dependencies.** Gradients propagated over many time steps tend to either **vanish** (most of the time) or **explode** (rarely, but with severe damage to optimization).

Why this happens. BPTT requires multiplying gradients by the recurrent weight matrix $\mathbf{W}$ once for every time step gradients pass through. Over τ steps this is roughly $\mathbf{W}^\tau$: if the dominant eigenvalue of $\mathbf{W}$ is $|\lambda| < 1$ the product shrinks to zero (**vanishing**), if $|\lambda| > 1$ it grows without bound (**exploding**). Activation derivatives like $\tanh'$ further compress gradients toward zero.

Consequences.

- **Vanishing gradients** : the network cannot learn dependencies between events separated by many time steps, since the gradient signal from a distant target is too weak to update earlier parameters.
- **Exploding gradients** : weight updates become huge, training diverges. Often handled in practice by **gradient clipping**.

Vanishing gradients are the deeper and more common problem : the model is structurally fine, but optimization simply cannot push information across long temporal gaps. This is the central motivation

*for the gated architectures (**LSTM**, **GRU**) covered next, which provide a direct path for gradients to flow through time without being repeatedly multiplied by* $\mathbf{W}$.

8.9 Gated RNNs: LSTM, GRU

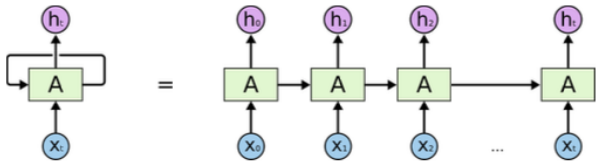
⋆ **Gated RNNs.** The most effective sequence models used in practice are **gated RNNs**, recurrent networks whose cells contain learned **gates** that explicitly control what information is kept, forgotten, or read out at each step. The two dominant gated architectures are:

- **Long Short Term Memory (LSTM)**, Hochreiter & Schmidhuber (1997).
- **Gated Recurrent Unit (GRU)**, Cho et al. (2014), a simpler variant of the LSTM.

Gating is the central trick that solves the long term dependency problem : instead of forcing every state update to pass through a tanh *and a multiplication by* $\mathbf{W}$*, gates allow the network to leave information unchanged across many steps when desired.*

⋆ **Traditional RNN as a Repeating Module.** A traditional RNN is a **chain of repeating modules**. A single module A takes an input $\mathbf{x}_t$ and produces a hidden output $\mathbf{h}_t$, with a self loop carrying information from one step to the next. **Unrolling** the loop in time gives a chain of identical copies $A \to A \to A \to \dots$, each receiving its own input $\mathbf{x}_t$ and emitting $\mathbf{h}_t$.

Internal structure. In a vanilla RNN, the module A contains a **single, very simple** computation, typically just one tanh layer applied to a linear combination of $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$.



This single layer cell is what makes the vanilla RNN both elegant and limited : every piece of information that must persist across time has to survive being squashed by tanh *and rotated by* $\mathbf{W}$ *at every single step. Long range memory has to "fight its way through" the same nonlinearity over and over.*

⋆ **Short Range Dependencies : RNNs Work.** When the gap between **relevant past information** and the place it is needed is small, vanilla RNNs handle it well. The relevant signal only needs to survive a few state updates before it is consumed.

Predicting the next word in "the sky is ____" needs only the immediately preceding context. A few time steps of memory are enough, and gradients flowing back over a few steps remain strong.

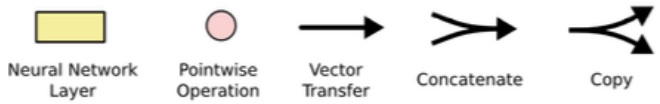
⋆ **Long Range Dependencies : RNNs Fail.** As the gap between the relevant information and the prediction grows, vanilla RNNs become **unable** to connect them. Information from many steps ago is overwritten by intervening updates, and gradients flowing back over many steps vanish.

Classic example : "I grew up in Iran and studied there and in the United States, so I can speak . . . (Persian and English)". The cue word "Iran" is many tokens away from the prediction site, and a vanilla RNN cannot reliably preserve it long enough to influence the answer. This is the long term dependency problem in concrete form.

8.9.1 Long Short Term Memory (LSTM)

Understanding LSTM Networks: colah.github.io/posts/2015-08-Understanding-LSTMs

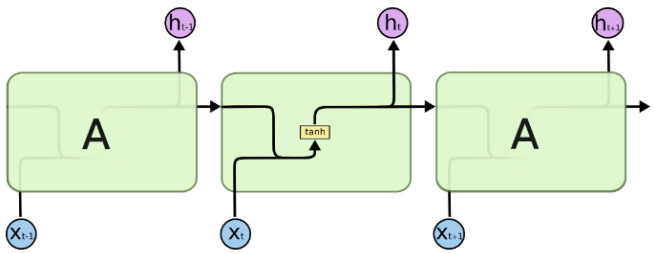
Notation:



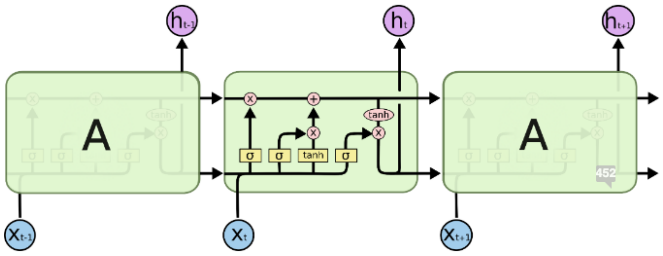
- **learned neural network layer** (linear map followed by an activation, with trainable weights).
- **pointwise operation** applied element wise (e.g. $\times$ for multiplication, $+$ for addition).
- **vector transfer** from one node to another.
- **concatenation** of two vectors into one.
- **copy** of a vector sent to two destinations.

⋆ **Long Short Term Memory (LSTM).** A class of RNNs **capable of learning long term dependencies**, introduced by **Hochreiter & Schmidhuber (1997)** and refined by many later authors. LSTMs work very well across a wide range of sequence problems and are now standard.

⋆ **LSTM Design Goal.** LSTMs are **explicitly designed** to avoid the long term dependency problem. They keep the chain of repeating modules from a vanilla RNN, but replace the simple inner structure (one tanh layer) with a more complex one that lets information flow across many time steps without being repeatedly squashed.



The repeating module in a standard RNN contains a single layer.



The repeating module in an LSTM contains four interacting layers.

⋆ **The Memory Cell.** The LSTM keeps the same chain like structure as a vanilla RNN, but the **repeating module** has a different internal structure. Instead of a **single neural network layer**, the LSTM module contains **four layers interacting in a specific way**. This module is called a **memory cell**.

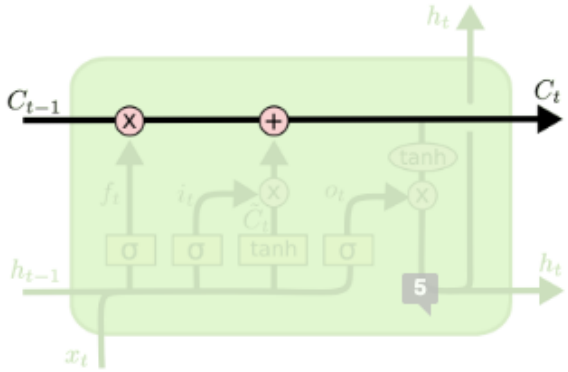
The four internal layers (previewed).

- Three **sigmoid** layers acting as **gates**, each producing values in $(0, 1)$ that decide **how much** of a signal to let through.
- One **tanh** layer producing **candidate content** in $(-1, 1)$ to potentially write into the memory.

Gates are the key innovation. A sigmoid output near 0 means "block this", near 1 means "let this pass". By learning when to open and close these gates, the LSTM can carry information across hundreds of time steps when needed and discard it when not. The detailed mechanics of each gate are covered in the next slides.

⋆ **The Cell State.** The key innovation of the LSTM is the **cell state** $\mathbf{C}_t$, drawn as the horizontal line running across the top of the cell. It acts like a **conveyor belt** : it flows straight through the entire chain of cells with only **minor linear interactions** (one multiplication and one addition per step).

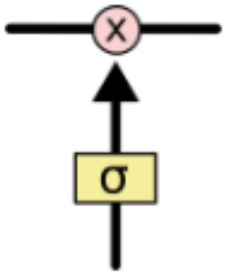
This is what fixes the long term dependency problem. In a vanilla RNN, every time step squashes the state through tanh *and multiplies by* $\mathbf{W}$*, so information from the distant past degrades quickly. In an LSTM, information can travel along the cell state largely* ***unchanged*** *for many steps, which keeps the gradient strong on the long range path.*



⋆ **Gates.** The LSTM modifies the cell state through structures called **gates**, which optionally let information through. A gate is the combination of a **sigmoid layer** and a **pointwise multiplication**:

$$\boxed{\text{gate output} = \sigma(\mathbf{W} \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}) \ \odot \ (\text{signal})}$$

- Sigmoid output is in $(0, 1)$, applied element wise : it describes **how much of each component** should pass through.
- Value near $\mathbf{0}$: "let nothing through".
- Value near $\mathbf{1}$: "let everything through".
- $\odot$: pointwise (element wise) multiplication with the signal being filtered.



The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”

An LSTM has three gates : forget gate $\mathbf{f}_t$, input gate $\mathbf{i}_t$, and **output gate $\mathbf{o}_t$**, each protecting and controlling a different aspect of the cell state.

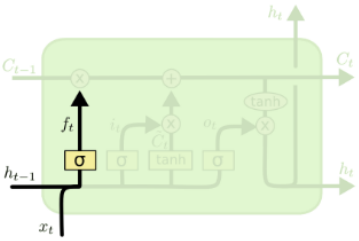
Gates are learned : the network discovers when to open and close each gate based on the current input and the previous hidden state. They make the cell state act like a writable, erasable memory rather than a fixed nonlinearity.

⋆ **Step 1 : Forget Gate.** Decide **what information to throw away** from the previous cell state $\mathbf{C}_{t-1}$. A sigmoid layer (the **forget gate layer**) looks at $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$ and outputs a number in $(0, 1)$ for each entry of $\mathbf{C}_{t-1}$:

$$\boxed{\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)}$$

- $[\mathbf{h}_{t-1}, \mathbf{x}_t]$: concatenation of previous hidden state and current input.
- $\mathbf{W}_f, \mathbf{b}_f$: learned forget gate weights and bias.
- $\mathbf{f}_t$: same shape as $\mathbf{C}_{t-1}$, each entry decides per coordinate : $\mathbf{1}$ = "completely keep", $\mathbf{0}$ = "completely erase".

The forget gate gives the LSTM the ability to actively delete information that is no longer relevant. In a language model, when a new subject appears in a sentence the gender of the previous subject can be erased so the wrong pronoun is not used later.



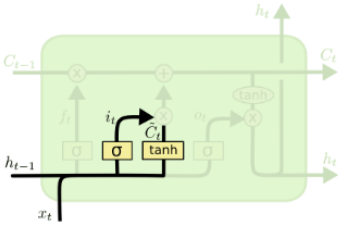
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

⋆ **Step 2 : Input Gate and Candidate Values.** Decide **what new information to write** into the cell state. This has two parallel pieces:

$$\boxed{\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)} \qquad \boxed{\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C)}$$

- **Input gate** $\mathbf{i}_t \in (0, 1)$: sigmoid that decides **which entries** of the cell state to update.
- **Candidate values** $\tilde{\mathbf{C}}_t \in (-1, 1)$: tanh layer that proposes **new content** that could be added to the cell state.

Splitting the update into two layers separates the questions "where to write?" (input gate) and "what to write?" (candidate values). The two are combined in the next step via $\mathbf{i}_t \odot \tilde{\mathbf{C}}_t$*, which writes the new content only into the slots the gate has opened.*



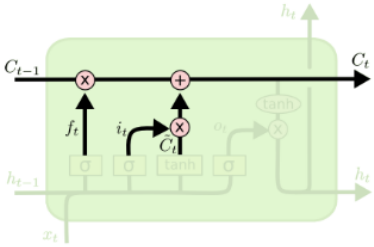
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

⋆ **Step 3 : Update the Cell State.** Combine the previous decisions to produce the new cell state $\mathbf{C}_t$:

$$\boxed{\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t}$$

- $\mathbf{f}_t \odot \mathbf{C}_{t-1}$: the old cell state with selected entries erased by the forget gate.
- $\mathbf{i}_t \odot \tilde{\mathbf{C}}_t$: the new candidate values, scaled by how much the input gate decided to update each entry.
- $\odot$: pointwise (element wise) multiplication.

This is the only step that touches $\mathbf{C}_t$ *directly. The previous two steps just decided what to do : here the LSTM actually performs the update. Notice that this is purely linear (multiply and add), no* tanh *is applied to* $\mathbf{C}$ *on its main path : that is the structural reason gradients can flow back across many time steps without vanishing.*



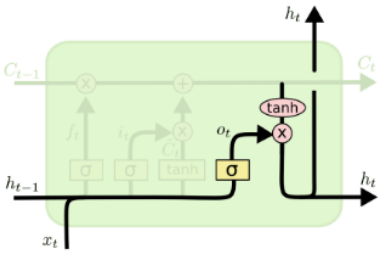
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

⋆ **Step 4 : Cell Output.** The hidden state $\mathbf{h}_t$ that leaves the cell is a **filtered version of the cell state**. Two operations:

$$\boxed{\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)} \qquad \boxed{\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t)}$$

- **Output gate** $\mathbf{o}_t \in (0, 1)$: sigmoid that decides **what parts** of the cell state to expose.
- $\tanh(\mathbf{C}_t)$: squashes the cell state into $(-1, 1)$ for the output (the cell state itself can grow outside this range).
- $\mathbf{h}_t$: the gated, squashed view of $\mathbf{C}_t$ that becomes both the cell's output and the input to the next time step.

The cell state $\mathbf{C}_t$ stores everything the LSTM remembers, but not all of it is relevant at every step. The output gate selects only the part of memory that is useful right now. In a language model, after seeing a subject the cell may store many facts about it (number, gender, animacy) but only output the ones needed to predict the next word (e.g. singular vs. plural for verb conjugation).



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Pipeline LSTM: At every time step, the LSTM walks through four steps in order : **(1) decide what to forget**, **(2) decide what to write**, **(3) actually update the memory**, **(4) decide what to expose**. The "long" memory lives in the cell state $\mathbf{C}_t$, the "short" memory lives in the hidden output $\mathbf{h}_t$, hence *Long Short Term Memory.*

Step 1 : Forget. Look at the evidence available right now ($\mathbf{h}_{t-1}$, $\mathbf{x}_t$) and decide, per memory slot, how much old content to erase.

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$$

The concatenation $[\mathbf{h}_{t-1}, \mathbf{x}_t]$ packages the past and the present into one vector of evidence. $\mathbf{W}_f$ projects that evidence into the shape of the memory, and $\sigma(\cdot)$ squashes every entry into $(0, 1)$, turning the result into a soft mask : entries near 1 mean *keep*, near 0 mean *erase*.

Step 2 : Write. From the same evidence, decide *which* slots to update and *what* new content to put there.

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i), \qquad \tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C)$$

$\mathbf{i}_t$ is another sigmoid mask in $(0, 1)$ that selects *where* to write. $\tilde{\mathbf{C}}_t$ uses tanh instead of σ, producing values in $(-1, 1)$ that act as the actual *content* to write : signed because content can push memory entries up or down. So $\mathbf{i}_t$ answers "where" and $\tilde{\mathbf{C}}_t$ answers "what".

Step 3 : Update. Carry out the decisions from steps 1 and 2 on the actual memory.

$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t$$

$\mathbf{f}_t \odot \mathbf{C}_{t-1}$ keeps a fraction of every old memory entry (forgetting selectively). $\mathbf{i}_t \odot \tilde{\mathbf{C}}_t$ pushes the new content only into the slots the input gate has opened. The two are added pointwise to give the new long term memory. Crucially this update is **purely linear** (multiply and add, no tanh on the main path), which is what lets gradients flow back across many steps without vanishing.

Step 4 : Expose. The full memory $\mathbf{C}_t$ may contain many things, but only some are useful right now. Pick what to send out as this step's output.

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o), \qquad \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t)$$

$\tanh(\mathbf{C}_t)$ rescales the memory into $(-1, 1)$ for safe output (the raw $\mathbf{C}_t$ can grow outside this range). $\mathbf{o}_t$ is a final sigmoid mask that picks which dimensions of that rescaled memory are relevant at the moment. The result $\mathbf{h}_t$ is both the cell's output for time t and the input that goes back into the gates at time $t + 1$.

All four weight matrices $\{\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_C, \mathbf{W}_o\}$ and their biases are shared across all time steps : the cell learns one universal update rule that fires at every step.

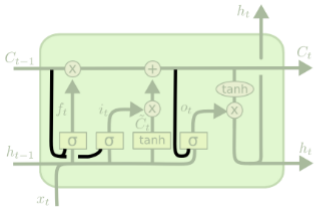
8.9.2 Variants on Long Short Term Memory

⋆ **Variant : Peephole Connections.** Introduced by **Gers & Schmidhuber (2000)**. The standard LSTM gates only see $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$. Peephole connections **also let the gates look directly at the cell state**, giving them more information when deciding what to forget, write, or expose:

$$\boxed{\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot [\mathbf{C}_{t-1}, \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)} \quad \boxed{\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{C}_{t-1}, \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)} \quad \boxed{\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{C}_t, \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)}$$

The forget and input gates peek at the **previous** cell state $\mathbf{C}_{t-1}$, the output gate at the **current** cell state $\mathbf{C}_t$ (since by then it is already updated).

Compared to the standard LSTM cell, extra arrows now run from the cell state line down into each sigmoid gate, that is the "peephole". The gate's input is no longer just $[\mathbf{h}_{t-1}, \mathbf{x}_t]$ but also includes the cell state, so the gate decision is informed by what is currently stored in memory.



Useful for tasks requiring precise timing (e.g. counting), where the gate needs to know the current memory content to fire at the right moment. In most applications the improvement over standard LSTM is marginal.

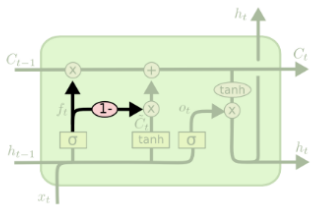
⋆ **Variant : Coupled Forget and Input Gates.** Tie the forget and input decisions together so they cannot disagree. **Only forget when about to write something new in the same place, only write when forgetting something old.** The cell state update becomes:

$$\boxed{\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + (1 - \mathbf{f}_t) \odot \tilde{\mathbf{C}}_t}$$

- $\mathbf{f}_t$: forget gate, controls how much of the old memory to keep at each entry.
- $1 - \mathbf{f}_t$: automatically determines how much new content to write, replacing the separate input gate $\mathbf{i}_t$.

This removes one gate (and its parameters) without losing much expressiveness. Each entry is a convex combination of "keep old" and "write new", so every memory slot is a fixed budget split between memory and update. Slightly cheaper than a full LSTM and often comparable in performance.

Input gate $\mathbf{i}_t$ is gone. The forget gate's output $\mathbf{f}_t$ is also routed (via $1-\mathbf{f}_t$) to the candidate $\tilde{\mathbf{C}}_t$, so a single sigmoid controls both sides of the update at once. Whatever fraction the forget gate decides to keep is exactly the fraction the cell refuses to overwrite, and vice versa.



⋆ **Gated Recurrent Unit (GRU).** Introduced by **Cho et al. (2014)** as a simpler alternative to the LSTM. Two main differences:

- Combines forget and input gates into a single **update gate** $\mathbf{z}_t$.
- **Merges the cell state and hidden state** : there is no separate $\mathbf{C}_t$, only $\mathbf{h}_t$.

The full GRU at time t:

$$\boxed{\mathbf{z}_t = \sigma(\mathbf{W}_z \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t])} \quad \boxed{\mathbf{r}_t = \sigma(\mathbf{W}_r \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t])}$$

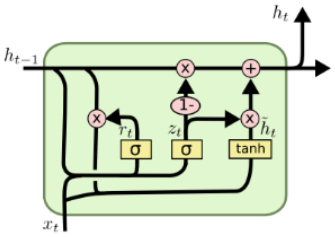
$$\boxed{\tilde{\mathbf{h}}_t = \tanh(\mathbf{W} \cdot [\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t])} \quad \boxed{\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t}$$

- **Update gate** $\mathbf{z}_t$: how much of the previous state to carry forward vs. replace with new content. Plays the role of LSTM's forget + input combined.

- **Reset gate** $\mathbf{r}_t$: how much of the previous state to use when computing the new candidate. When $\mathbf{r}_t \approx 0$, the candidate is computed almost as if there were no past, letting the unit "reset".
- $\tilde{\mathbf{h}}_t$: candidate values for the new hidden state, conditioned on a reset version of the past.
- Final $\mathbf{h}_t$: convex combination of the old state and the candidate, controlled by $\mathbf{z}_t$.

The GRU has fewer parameters than an LSTM (3 weight matrices instead of 4, no separate cell state) and trains faster, with comparable performance on many tasks. It is increasingly the default choice when the LSTM's full expressivity is not needed.

There is no top cell state line, $\mathbf{h}_t$ carries everything. The reset gate $\mathbf{r}_t$ acts on $\mathbf{h}_{t-1}$ before it enters the tanh candidate $\tilde{\mathbf{h}}_t$, controlling how much past to use when proposing new content. The update gate $\mathbf{z}_t$ then mixes the old hidden state with the candidate via $(1 - \mathbf{z}_t)$ and $\mathbf{z}_t$, so a single sigmoid decides both how much to keep and how much to overwrite.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

⋆ **Clockwork RNN (CW-RNN).** A different approach to long term dependencies : split the hidden layer into **several modules running at different clock speeds**. Module k is updated only every T_k time steps, with T_k chosen as a power of 2 (e.g. $1, 2, 4, 8, 16, \ldots$).

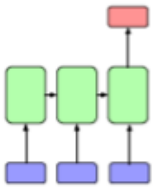
- Slower modules see longer time scales and capture long range dependencies.
- Faster modules see fine grained detail.
- **Slower modules are not connected to faster ones** (faster → slower only), giving fewer weights than a full RNN.
- Trains and evaluates faster because slow modules skip most steps.

The architecture imposes an explicit multi resolution prior on time : information that is supposed to persist a long time is naturally placed in the slow modules and updated rarely, so it cannot be overwritten at every step. This is a structural alternative to gating.

8.10 RNN Architecture Taxonomy

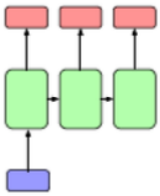
⋆ **RNN Architecture Taxonomy.** RNNs can be wired into many shapes depending on whether the input and output are single vectors or sequences. Convention : blue = inputs, red = outputs, green = hidden state.

Sequence to vector (many to one). A sequence of inputs is read step by step, and **a single output** is produced at the end. The final hidden state acts as a fixed length summary of the whole sequence.



Tasks : text classification, sentiment analysis, video classification. Anything that takes a variable length sequence and produces one decision.

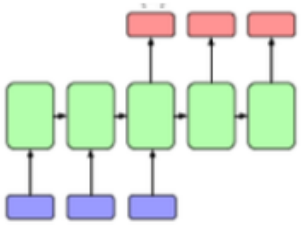
Vector to sequence (one to many). A **single input** initializes a chain of hidden states that emit a sequence of outputs.



Tasks : image captioning. The input image is encoded once (e.g. by a CNN) and the RNN generates a variable length caption from it.

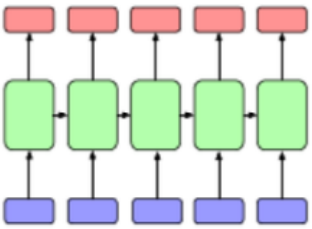
Sequence to sequence, different lengths (many to many, delayed). A sequence of inputs is read first, then a sequence of outputs is generated, with input and output lengths possibly different. This is the **encoder decoder** architecture from earlier.

Tasks : machine translation, speech recognition, question answering. The encoder reads the whole input before the decoder starts producing.



Sequence to sequence, same length (many to many, synchronized). An output is produced **at every time step**, in lockstep with the input.

Tasks : character level language modeling (predict next character at each step), part of speech tagging, frame level video labeling. Useful whenever each input position has its own target.



All of these are wired from the same recurrent cell (vanilla RNN, LSTM, or GRU). What changes is only **where inputs enter** and **where outputs are read off**, not the cell itself.

8.11 Appendix

8.11.1 RNN Idea

⋆ **RNN, the fundamental model.** An RNN has three things : an **input**, a **hidden state**, and an **output**, all evolving over a sequence of steps. The whole model is built from one recurrence equation applied repeatedly:

$$\boxed{\mathbf{s}^{(t)} = f\left(\mathbf{W}\mathbf{s}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} + \mathbf{b}\right)}$$

Read it as : *"the new hidden state is a learned mix of the old hidden state and the new input, squashed by a nonlinearity."* Every other component of the network is a wrapper around this single line.

⋆ **The Three Roles Inside the Equation.**

- **Input $\mathbf{x}^{(t)}$, what arrives now.** The new observation at step t. The matrix $\mathbf{U}$ controls how the model **reads** this input : "of all the features in $\mathbf{x}^{(t)}$, which ones matter ?"
- **Hidden state $\mathbf{s}^{(t)}$, the model's memory.** A vector the model updates and carries forward at every step. The matrix $\mathbf{W}$ controls how the past **propagates** : "of the information I was holding in $\mathbf{s}^{(t-1)}$, what should I keep, update, or fade ?"
- **Output $\mathbf{o}^{(t)}$, what the model says.** A linear readout from the state, $\mathbf{o}^{(t)} = \mathbf{V}\mathbf{s}^{(t)} + \mathbf{c}$, optionally followed by softmax for classification. The matrix $\mathbf{V}$ asks : "given my current memory, what is the best prediction right now ?"

The hidden state is not part of the data, only $\mathbf{x}^{(t)}$ is. The model invents the hidden state itself to compress the past into a fixed size summary. Each entry of $\mathbf{s}^{(t)}$ is a learned feature whose meaning is discovered during training.

⋆ **Why "Hidden Layer".** In an MLP, a hidden layer is a vector of neurons sitting between input and output. The RNN's $\mathbf{s}^{(t)}$ is exactly that, **a hidden layer**, with one extra ingredient : it also receives its own previous value $\mathbf{s}^{(t-1)}$ via $\mathbf{W}$. This recurrence is what turns an ordinary hidden layer into a recurrent one. Unfolding the RNN over τ steps gives a τ layer deep MLP whose layers all share the same weights and each layer also takes a fresh input.

⋆ **Finance Example : Predicting Next Day Return Direction.** At every trading day t, predict whether tomorrow's return will be positive or negative.

Input $\mathbf{x}^{(t)}$. Today's market data, e.g. a 5 dimensional vector: $\mathbf{x}^{(t)} = \begin{bmatrix} \text{log return today} \\ \text{trading volume} \\ \text{VIX level} \\ \text{news sentiment} \\ \text{10y yield change} \end{bmatrix}$

Hidden state $\mathbf{s}^{(t)}$. The model's running internal summary of the market, say a 64 dimensional vector. Training discovers what each entry represents. After training, the entries might end up encoding things like : *strength of the recent uptrend, recent volatility level, risk on vs risk off regime, momentum decay,* etc. The model invented these features because they were useful for predicting direction.

Recurrence on a Tuesday.

- $\mathbf{s}^{(\text{Mon})}$: Monday's summary, "trend bullish, volatility low, momentum strong".
- $\mathbf{x}^{(\text{Tue})}$: Tuesday's data, log return -2%, volume doubled, VIX spiked.
- $\mathbf{W}\mathbf{s}^{(\text{Mon})}$: how much of Monday's bullish belief to keep.
- $\mathbf{U}\mathbf{x}^{(\text{Tue})}$: how alarming Tuesday's spike is.
- Add them, apply tanh, obtain $\mathbf{s}^{(\text{Tue})}$: updated summary, "trend weakening, volatility rising, caution".

Output. $\mathbf{o}^{(\text{Tue})} = \mathbf{V}\mathbf{s}^{(\text{Tue})} + \mathbf{c}$, then softmax gives the probability that Wednesday's return is positive.
What each matrix is learning.

- $\mathbf{U}$: how to read raw market data into "what kind of day was today".
- $\mathbf{W}$: how persistent each market regime (bull vs bear, calm vs volatile) is from one day to the next.
- $\mathbf{V}$: how to translate the current market summary into a directional bet.

This is exactly why ***GARCH(1,1) is a hand designed scalar RNN****, with a 1D hidden state (variance), a scalar* $\mathbf{W} = \beta$*, a scalar* $\mathbf{U} = \alpha$*, no nonlinearity, and a fixed output rule. An RNN replaces every part of that with a high dimensional, nonlinear, learned version.*

⋆ **When Does a Problem Suit an RNN ?** A problem is RNN shaped when the data has **intrinsic order with dependency**, meaning the meaning of each entry depends on what came before it (and possibly after). Time is the most common source of order, but not the only one (text, DNA, code, pen strokes, etc.).

Quick rule of thumb.

- If **shuffling the entries** of the input changes the meaning, the data is sequential, an RNN is appropriate.
- If shuffling makes no difference, the data is not sequential, an RNN is the wrong tool (use MLP, CNN, or a tree based method).

Defining a finance problem solvable with an RNN. Three checks:

1. **Order matters** : the prediction at t depends on **when** things happened, not just **what** happened (returns, order book updates, news streams).
2. **Variable history length** : the relevant past may be 5 days for one regime and 5 months for another, you cannot fix it in advance with a sliding window.
3. **Path dependence** : two paths reaching the same current price can imply different futures (volatility clustering, momentum, regime persistence).

If all three hold, an RNN (or LSTM, GRU) is a natural model. Examples : *intraday return forecasting, volatility forecasting, regime detection, credit event prediction from transaction streams, limit order book modeling.*

8.11.2 Macro Overview

⋆ **What is a Neural Network ?** A **neural network** is a parameterized function built by stacking simple computational units (**neurons**) into layers. Each neuron computes a weighted sum of its inputs plus a bias, then applies a nonlinear **activation function**. Stacking many layers produces a function flexible enough to approximate almost any input output mapping, with the parameters learned from data by minimizing a loss via **gradient descent** guided by **backpropagation**.

Every model in this lecture is one specific way of choosing : (i) what a neuron computes, (ii) how neurons are connected, (iii) how parameters are shared. Everything else, training, regularization, evaluation, follows the same recipe.

⋆ **What is Deep Learning ?** **Deep learning** is the use of neural networks with **many layers** (typically more than 10), trained **end to end** on large datasets. "Deep" refers to architectural depth, "learning" refers to the fact that nothing is hand engineered : feature extraction, representation, and decision are all learned jointly from data. The defining property of deep models is that earlier layers learn **low level features** (edges, syllables, short term trends) and later layers compose them into **high level abstractions** (faces, sentences, regimes).

Classical machine learning relies on hand crafted features fed into a shallow model. Deep learning replaces the human feature engineer with a stack of learnable transformations. The shift from "design features, fit a model" to "design an architecture, learn features and model jointly" is the conceptual core of the field.

⋆ **Three Big Problem Families.** Almost every deep learning model exists to handle one of three input structures :

- **Tabular / vector data** : fixed length feature vectors, no spatial or temporal structure (e.g. tabular finance data, classical regression / classification). Solved by **MLPs**.
- **Spatial / grid data** : data with local structure where neighbors matter, mainly images and video frames. Solved by **CNNs**.
- **Sequential / temporal data** : ordered data where past influences future, language, audio, time series. Solved by **RNNs, LSTMs, GRUs**.

The three architectures are not competitors, they are the right tool for three different shapes of input. The choice of model is dictated by the geometry of the data, not by performance benchmarks.

⋆ **Model Hierarchy.** Each model in this lecture is a refinement of the one before it, motivated by a specific limitation:

- **Perceptron** : single neuron with sign activation. Learns one hyperplane. **Limit** : only linearly separable data.
- **Multi class Perceptron** : S perceptrons in parallel, partitions space into up to 2^S regions. **Limit** : still single layer, still linear.
- **MLP (Multi Layer Perceptron)** : stack of perceptron layers with smooth nonlinearities. **Universal approximator**. **Limit** : weight proliferation on images, no notion of locality, no memory.
- **CNN (Convolutional Neural Network)** : MLP for grid data. Replaces full connections with **convolutions** (local receptive fields, weight sharing) and **pooling** (translation invariance). **Limit** : assumes input is a grid, no temporal memory.
- **RNN (Recurrent Neural Network)** : MLP for sequences. Adds a **hidden state** that loops back into itself, so parameters are shared across time. **Limit** : vanishing / exploding gradients, fails on long term dependencies.
- **LSTM, GRU** : RNNs with **gates** that explicitly control what to keep, write, and expose. Solve the long term dependency problem.
- **Encoder Decoder (seq2seq)** : two RNNs / LSTMs chained through a **context vector** so input and output sequences can have different lengths. Standard for translation, summarization, speech.
- **Bidirectional RNN, Clockwork RNN** : variants for tasks where future context is available, or where multiple time scales must be handled.

Each model in this list inherits the one above and adds exactly one structural idea : depth (MLP), spatial weight sharing (CNN), temporal weight sharing (RNN), gating (LSTM / GRU), encoder decoder coupling (seq2seq).

⋆ **Cross Cutting Themes.** Five ideas appear in every architecture, regardless of input type:

- **Backpropagation** : the universal training algorithm. Computes gradients through any composition of differentiable layers via the chain rule. **Used by every model.**
- **Gradient descent** : the optimization algorithm. Stochastic and mini batch variants make it scalable. **Used by every model.**
- **Activation functions** : the nonlinearity that gives the network its expressive power. Sigmoid and tanh historically, ReLU in modern networks. Softmax at the output for classification.

- **Regularization** : the family of techniques that prevent overfitting (L2 / weight decay, dropout, early stopping, data augmentation, adversarial training). **Architecture independent.**
- **Loss functions** : the scalar objective being minimized. Mean squared error for regression, cross entropy for classification. The choice of loss controls only the very first step of backprop.

Once these five are understood, learning a new architecture is mostly about understanding its specific connectivity pattern. The training machinery underneath is always the same.

⋆ **What Defines a Deep Learning Problem ?** A problem is well suited to deep learning when:

1. There is **enough labeled data** to learn millions of parameters (or pretrained models exist for transfer learning).
2. The mapping from input to output is **complex and nonlinear**, so classical models underfit.
3. The input has **structure** (spatial, temporal, hierarchical) that an architecture can exploit through weight sharing.
4. Prediction quality matters more than interpretability of individual features (because deep models are largely black boxes).

If any of these is missing, simpler models often win. Deep learning shines exactly where hand crafted features hit a ceiling and the structure of the data can be encoded into the architecture itself.