كلية الحاسبات والذكاء الاصطناعى
FACULTY OF COMPUTERS AND ARTIFICIAL INTELLIGENCE
مؤسسة تعليمية معتمدة

جامعة الفيوم

كلية الحاسبات والذكاء الاصطناعى

مؤسسة تعليمية معتمدة

الكتاب الجامعى

2025 - 2026

Lecture notes on Machine Learning
for Computer Science Department

First Edition

Contents

Chapter 1

Introduction

Machine learning has emerged as one of the most transformative technologies of the 21st century, fundamentally changing how we interact with computers and process information. Unlike traditional programming, where explicit instructions dictate every action, machine learning enables computers to learn patterns from data and make intelligent decisions without being explicitly programmed for every scenario.

At its core, machine learning is about teaching computers to recognize patterns, make predictions, and improve their performance through experience. This paradigm shift has revolutionized countless industries, from healthcare diagnostics to autonomous vehicles, from financial fraud detection to personalized content recommendations. Understanding machine learning is no longer optional for technology professionals—it has become essential knowledge in our increasingly data-driven world.

This chapter provides a comprehensive introduction to machine learning, exploring its fundamental concepts, evolution, and primary methodologies. We will examine the three main paradigms—supervised learning, unsupervised learning, and reinforcement learning—and delve into their specific applications. Special attention will be given to understanding the critical differences between key techniques such as classification, regression, and clustering, illustrated with practical examples. Finally, we will extend our exploration into the realm of deep learning and neural networks, which have driven the recent explosion in machine learning capabilities.

1.1 What is Machine Learning?

1.1.1 Definition and Core Concepts

Formal Definition

Machine Learning is a subset of artificial intelligence that focuses on developing algorithms and statistical models that enable computer systems to improve their performance on a specific task through experience. Rather than following explicitly programmed instructions, machine learning systems learn from data, identifying patterns and making data-driven predictions or decisions.

The formal definition, articulated by computer scientist Tom Mitchell, states that a computer program is said to learn from experience E with respect to some task T and performance measure P if its performance on T, as measured by P, improves with experience E. This elegant definition captures the essence of machine learning: continuous improvement through exposure to data.

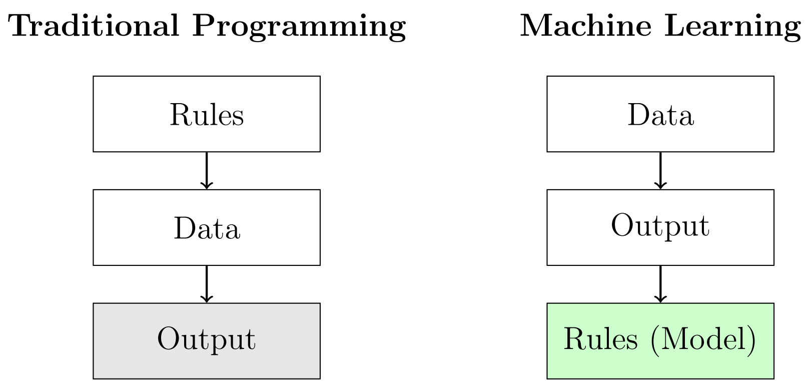
Key Characteristics

Machine learning encompasses several fundamental characteristics:

- **Performance Improvement:** The broad range of ML techniques enables software applications to improve their performance over time.
- **Pattern Discovery:** Machine learning algorithms are trained to find relationships and patterns in data. They use historical data as input to:
  - Make predictions
  - Classify information
  - Cluster data points
  - Reduce dimensionality
- **Content Generation:** Machine learning algorithms may even help in generating new content like ChatGPT.

1.1.2 Traditional Programming vs. Machine Learning

To appreciate the power of machine learning, consider the fundamental difference between traditional programming and machine learning approaches:



Traditional Programming: We provide the computer with explicit rules and data as input, and it produces output based on those rules. For example, to identify spam emails, a programmer might write rules like "if the email contains certain keywords" or "if the sender address matches a blacklist." Every scenario must be anticipated and coded.
This can be summarized as: Rules + Data → Output
Machine Learning: We provide the computer with data and the corresponding outputs (labels), and the algorithm learns the rules automatically. For spam detection, we would feed thousands of emails labeled as "spam" or "not spam," and the algorithm discovers patterns that distinguish spam from legitimate messages without us explicitly defining those patterns.
This can be summarized as: Data + Output → Rules (Model)
Goal: Build systems that automatically improve performance through experience.
This fundamental shift enables machines to handle complex problems where defining explicit rules would be impractical or impossible, such as recognizing faces in photos, understanding natural language, or predicting customer behavior.

Real-World Analogy

Just like humans learn to recognize faces by seeing many examples, ML algorithms learn patterns from data to make predictions about new, unseen information.

1.2 The Evolution of Machine Learning

Machine learning has evolved through several distinct eras, each characterized by different approaches, computational capabilities, and theoretical breakthroughs.

1.2.1 Early Foundations (1950s-1960s)

The conceptual groundwork for machine learning was laid in the 1950s with pioneers like Arthur Samuel, who developed a checkers-playing program that improved through self-play. The term "machine learning" itself was coined by Samuel in 1959. During this period, the perceptron algorithm was introduced by Frank Rosenblatt, marking one of the first neural network models. However, computational limitations and theoretical challenges constrained practical applications.

1.2.2 Knowledge-Based Systems (1970s-1980s)

The focus shifted toward expert systems and symbolic approaches, where human knowledge was explicitly encoded into computer systems. While not strictly machine learning by modern standards, these systems demonstrated the potential for computers to perform intelligent tasks. Decision tree algorithms and other symbolic learning methods gained prominence during this era.

1.2.3 Statistical Learning Era (1990s)

The 1990s witnessed a paradigm shift toward statistical and probabilistic approaches. Support Vector Machines, ensemble methods like Random Forests, and the refinement of neural network training through backpropagation marked this period. Machine learning began moving from theoretical research to practical applications, with increasing computational power enabling more sophisticated algorithms.

1.2.4 The Big Data Revolution (2000s-2010s)

The explosion of internet data and advances in computing power catalyzed unprecedented progress. The rise of deep learning, facilitated by Graphics Processing Units that could handle massive parallel computations, enabled breakthroughs in computer vision, natural language processing, and speech recognition. Companies like Google, Facebook, and Amazon began deploying machine learning at scale, demonstrating its commercial viability.

1.2.5 Modern Era (2010s-Present)

Today, machine learning permeates virtually every aspect of technology. Deep learning models with billions of parameters achieve human-level or superhuman performance on many tasks. Transfer learning allows models trained on one task to be adapted for others. Reinforcement learning has produced systems that master complex games and control real-world robotics. The field continues to evolve rapidly, with emerging areas like federated learning, explainable AI, and quantum machine learning promising new frontiers.

1.3 Core Paradigms of Machine Learning

Machine learning encompasses three fundamental paradigms, each addressing different types of problems and learning scenarios. Understanding these paradigms is crucial for selecting the appropriate approach for any given task.

1.3.1 Supervised Learning

Supervised Learning

Supervised learning is learning from labeled examples, where training data includes both input features and corresponding correct outputs.

Supervised learning is the most common and intuitive paradigm in machine learning. In this approach, the algorithm learns from labeled training data, where each example consists of input features and a corresponding correct output (label or target value). The goal is to learn a mapping function from inputs to outputs that can generalize to new, unseen data.

Think of supervised learning as learning with a teacher. Just as a student learns mathematics by studying worked examples with solutions, a supervised learning algorithm learns by examining numerous examples where the correct answer is provided. Once trained, the algorithm can apply this learned knowledge to predict outcomes for new inputs.

Key Characteristics

- Training data includes both input features and correct output labels
- The algorithm learns to predict outputs for new inputs based on training examples
- Supervision: The training data (observations, measurements, etc.) are accompanied by labels indicating the class of the observations
- New data is classified based on the training set
- Performance is evaluated by comparing predictions to known correct answers
- Requires substantial labeled data, which can be expensive to obtain

Main Tasks in Supervised Learning

Supervised learning divides into two primary categories based on the nature of the output: **classification** and **regression**.

Common Applications

- Email spam detection
- Medical diagnosis
- Image classification
- Customer retention
- Diagnostics
- Identity fraud detection
- Credit/loan approval
- Weather forecasting

1.3.2 Unsupervised Learning

Unsupervised Learning

Unsupervised learning finds hidden patterns in unlabeled data, discovering structure without guidance about what to look for.

Unsupervised learning addresses scenarios where we have input data but no corresponding labels or target outputs. The algorithm must discover hidden patterns, structures, or relationships within the data without guidance about what to look for. This paradigm is analogous to exploratory learning, where the algorithm acts as a detective identifying meaningful patterns in complex data.
While supervised learning answers specific questions (Is this email spam? What will the stock price be tomorrow?), unsupervised learning explores the data to reveal its inherent structure (What natural groups exist in my customer base? What are the main themes in these documents?).

Key Characteristics

- Training data consists only of input features with no labels
- The class labels of training data are unknown
- The algorithm identifies patterns, structures, or relationships autonomously
- Given a set of measurements, observations, etc. with the aim of establishing the existence of classes or clusters in the data
- No single correct answer exists; results depend on the algorithm and parameters
- Useful for data exploration and preprocessing for supervised tasks

Main Tasks in Unsupervised Learning

The primary techniques in unsupervised learning are **clustering** and **dimensionality reduction**.

Common Applications

- Customer segmentation
- Data compression
- Recommender systems
- Targeted marketing
- Big data visualization
- Structure discovery
- Meaningful compression
- Feature elicitation
- Social network analysis
- Libraries and document organization

1.3.3 Reinforcement Learning

Reinforcement Learning

Reinforcement learning involves learning through trial and error with rewards, where an agent learns to make decisions by interacting with an environment.

Reinforcement learning represents a fundamentally different paradigm where an agent learns to make decisions by interacting with an environment. Unlike supervised learning, there are no explicit correct answers provided. Instead, the agent receives feedback in the form of rewards or penalties based on its actions, learning through trial and error to maximize cumulative reward over time.

This paradigm mirrors how humans and animals learn through experience. A child learning to ride a bicycle doesn't receive step-by-step instructions for every possible situation; instead, they try different actions, experience the consequences (maintaining balance or falling), and gradually develop a policy for successful riding. Similarly, reinforcement learning agents develop strategies through exploration and feedback.

Key Characteristics

- Agent interacts with an environment through actions

- Receives rewards or penalties based on action consequences
- Goal is to learn a policy maximizing long-term cumulative reward
- Must balance exploration (trying new actions) with exploitation (using known good actions)

Common Applications

- Game playing (Chess, Go, video games)
- Robotics and robot navigation
- Autonomous driving
- Real-time decisions
- Skill acquisition
- Learning tasks
- Game AI

Reinforcement learning excels in sequential decision-making problems where actions have long-term consequences, such as game playing, robotics, resource allocation, and autonomous navigation.

Course Focus

We'll primarily focus on **supervised** and **unsupervised learning**, which form the foundation of most practical ML applications.

1.4 Supervised Learning Tasks: Classification and Regression

Supervised learning encompasses two fundamental task types distinguished by the nature of their output variables: classification and regression. Understanding the difference between these tasks is crucial for selecting appropriate algorithms and evaluation metrics.

1.4.1 Classification: Predicting Discrete Categories

Classification

Classification involves predicting a discrete category or class label from a predefined set of possibilities. The output variable is categorical, meaning it represents membership in one of several distinct groups.

Classification problems can be binary (two classes) or multiclass (three or more classes).

Defining Characteristics

- Predicts categorical class labels (discrete or nominal)
- Output is a discrete class label (e.g., "spam" or "not spam")
- Decision boundaries separate different classes in feature space
- Evaluation uses accuracy, precision, recall, and F1-score
- Classes are mutually exclusive (each instance belongs to exactly one class)
- Class label(s) is predicted based on a trained model that was trained on labeled data

Real-World Classification Examples

Email Spam Detection

Given the content, sender, and metadata of an email, classify it as "spam" or "legitimate." Features might include word frequencies, sender reputation, presence of links, and time sent. The output is a binary decision.

Medical Diagnosis

Based on patient symptoms, test results, and medical history, classify a condition as "benign," "malignant," or "inconclusive." This multiclass classification helps doctors make informed treatment decisions.
Example question: Is a tumor cancerous or benign?

Image Recognition

Identify the object in a photograph from categories like "cat," "dog," "bird," or "car." Modern deep learning systems can classify images across thousands of categories with remarkable accuracy.

Additional Classification Examples

- **Credit/loan approval:** Is a customer trusted to get a loan?
- **Customer churn prediction:** Will a customer continue their subscription?
- **Fraud detection:** Is a transaction fraudulent?
- **Weather forecast:** Is it going to rain?
- **Customer retention:** Will the customer stay or leave?
- **Identity fraud detection:** Is this identity legitimate?

Common classification algorithms include Logistic Regression, Decision Trees, Random Forests, Support Vector Machines, Naive Bayes, and Neural Networks. The choice depends on factors like dataset size, feature dimensionality, and interpretability requirements.

1.4.2 Regression: Predicting Continuous Values

Regression (Numeric Prediction)

Regression addresses problems where the output variable is continuous and numerical rather than categorical. The goal is to predict a quantity that can take any value within a range, establishing relationships between input features and the target variable.

Defining Characteristics

- Models continuous-valued functions
- Output is a continuous numerical value (e.g., price, temperature, or probability)
- Models the relationship between features and the target variable
- Evaluation uses Mean Squared Error, Mean Absolute Error, and R-squared
- Predictions can theoretically take any value in a continuous range
- Predicts unknown or missing numeric values

Real-World Regression Examples

House Price Prediction

Estimate the selling price of a house based on features like square footage, number of bedrooms, location, age, and local amenities. The output is a specific dollar amount, not a category.

Stock Market Forecasting

Predict tomorrow's closing price of a stock based on historical prices, trading volume, market indicators, and news sentiment. The prediction is a specific price value.

Energy Consumption Prediction

Forecast the kilowatt-hours of electricity a building will consume based on weather conditions, occupancy patterns, time of day, and historical usage. This enables efficient resource allocation.

Additional Regression Examples

- **Temperature forecasting:** Predict the exact temperature
- **Advertising popularity prediction:** Expected number of clicks or views
- **Weather forecasting:** Specific temperature values
- **Population growth prediction:** Future population numbers
- **Market forecasting:** Stock prices, sales revenue
- **Estimating life expectancy:** Predicted lifespan in years

Popular regression algorithms include Linear Regression, Polynomial Regression, Ridge and Lasso Regression, Support Vector Regression, Decision Tree Regression, and Neural Networks. The selection depends on the relationship complexity between features and targets.

1.4.3 Classification vs. Regression: Critical Distinctions

While both classification and regression are supervised learning tasks, they differ fundamentally in their objectives, outputs, and evaluation approaches. Understanding these distinctions is essential for problem formulation and algorithm selection.

Output Type

Classification: Produces discrete, categorical labels from a finite set. Examples include "yes/no," "low/medium/high," or named categories like "dog/cat/bird." The output represents class membership.
Regression: Produces continuous numerical values from an infinite range. Examples include specific prices like $347,529.50, temperatures like 23.7 C, or probabilities like 0.846. The output represents a quantity.

Evaluation Metrics

Classification: Measured using accuracy (percentage of correct predictions), precision (proportion of positive predictions that were correct), recall (proportion of actual positives identified), and F1-score (harmonic mean of precision and recall).
Regression: Measured using Mean Squared Error (average squared difference between predictions and actual values), Mean Absolute Error (average absolute difference), R-squared (proportion of variance explained), and Root Mean Squared Error.

Practical Example Comparison

Consider a credit card application scenario:

Classification approach: Predict whether the application will be "approved" or "rejected" based on income, credit score, and employment history. The output is a binary decision.

Regression approach: Predict the specific credit limit amount (e.g., \$5,000 or \$25,000) that should be granted based on the same features. The output is a continuous dollar amount.

This example illustrates how the same problem domain can be formulated as either classification or regression depending on what information is most valuable for the application.

1.5 Unsupervised Learning Tasks: Clustering and Dimensionality Reduction

1.5.1 Clustering: Discovering Natural Groupings

Clustering

Clustering is the task of grouping a set of objects in such a way that objects in the same group are more similar to each other than to those in other groups.

Clustering is the primary technique in unsupervised learning, focused on discovering natural groupings or structures within unlabeled data. Unlike classification, where predefined categories exist, clustering creates these categories by identifying similarities among data points without prior knowledge of group membership.

Understanding Clustering

Clustering algorithms group similar instances together based on feature similarity, using distance or similarity metrics to determine which points belong together. The underlying assumption is that instances within the same cluster share common characteristics that distinguish them from instances in other clusters.

Defining Characteristics

- No predefined labels or categories—groups emerge from data patterns
- Groups data points based on similarity or proximity measures
- Number of clusters may be predetermined or discovered automatically
- Evaluation uses internal metrics like silhouette score and within-cluster variance

Real-World Clustering Examples

Customer Segmentation

A retail company analyzes customer purchase behavior, demographics, and browsing patterns to identify natural customer segments. Clustering might reveal groups like “budget-conscious families,” “luxury shoppers,” “frequent small purchasers,” and “seasonal buyers.” Each segment can then receive targeted marketing.

Document Organization

A news aggregator processes thousands of articles without predefined topics. Clustering groups similar articles together, automatically creating categories like “technology news,” “sports updates,” “political coverage,” and “entertainment stories” based on content similarity.

Anomaly Detection

In cybersecurity, network traffic patterns are clustered. Normal behavior forms dense clusters, while unusual activity (potential attacks) appears as outliers or forms tiny, isolated clusters, enabling early threat detection.

Additional Clustering Applications

- Gene expression analysis
- Social network analysis
- Libraries and document organization
- Recommender systems
- Targeted marketing

Common clustering algorithms include K-Means, Hierarchical Clustering, DBSCAN, Gaussian Mixture Models, and Mean Shift. Algorithm selection depends on cluster shape assumptions, dataset size, and whether the number of clusters is known.

1.5.2 Dimensionality Reduction

Dimensionality Reduction

Dimensionality reduction is the transformation of data from a high-dimensional space into a low-dimensional space so that the low-dimensional representation retains some meaningful properties of the original data.

Key Concepts

Dimensionality reduction techniques transform data from a high-dimensional space into a lower-dimensional space while preserving important information. This is crucial for:

- Visualization of high-dimensional data
- Reducing computational complexity
- Removing noise and redundant features
- Improving model performance
- Addressing the curse of dimensionality

Main Approaches

Feature Selection: Selecting a subset of relevant features from the original feature set.
Feature Extraction (Projection): Creating new features by combining or transforming existing ones, such as Principal Component Analysis (PCA).

Applications

- Big data visualization
- Meaningful compression
- Structure discovery
- Feature elicitation
- Data preprocessing for machine learning

1.5.3 Clustering vs. Classification: Fundamental Differences

Clustering and classification are often confused because both group data instances, but they represent fundamentally different approaches with distinct objectives and requirements. The key distinction lies in whether labels exist during training.

Label Availability

Classification: Requires labeled training data where each instance's correct category is known. The algorithm learns to predict these predefined categories for new instances. Labels must be assigned by humans or another external process before training.
Clustering: Works with unlabeled data, discovering groups without prior knowledge of categories. The algorithm determines both the number of groups and their composition based solely on data patterns. No human labeling is needed.

Learning Paradigm

Classification: Supervised learning—learns from examples with known correct answers. The algorithm is trained to replicate human judgment about category membership.
Clustering: Unsupervised learning—discovers structure without guidance. The algorithm defines its own notion of what makes a meaningful group based on mathematical similarity.

Objective

Classification: Predict the correct category for new instances based on learned patterns. The categories have predefined meanings (e.g., “fraudulent transaction” vs. “legitimate transaction”).
Clustering: Discover natural groupings in data and understand data structure. The groups have no inherent meaning until humans interpret what characteristics define each cluster.

Practical Example Comparison

Consider analyzing customer data at an e-commerce company:
Classification scenario: The company has historical data on customers who eventually became high-value buyers (labeled “high-value”) and those who didn’t (labeled “regular”). You train a classifier on this labeled data to predict which new customers are likely to become high-value buyers, enabling proactive engagement strategies.
Clustering scenario: The company wants to understand its customer base without preconceptions. You apply clustering to customer features (purchase frequency, average order value, product categories, time since last purchase). The algorithm might discover four natural groups: occasional large purchasers, frequent small purchasers, seasonal shoppers, and inactive customers. These insights inform targeted strategies for each discovered segment.
This example demonstrates how classification answers a specific predefined question (Will this customer become high-value?), while clustering explores the data to reveal its inherent structure (What natural customer types exist?).

1.6 Comparative Summary: Classification, Regression, and Clustering

Understanding the distinctions among classification, regression, and clustering is fundamental to selecting the appropriate machine learning approach. While these techniques may seem similar superficially, they address different problem types and operate under different assumptions.

1.6.1 Comparative Overview

Aspect	**Classification**	**Regression**	**Clustering**
Learning Type	Supervised	Supervised	Unsupervised
Output Type	Discrete categories	Continuous values	Group assignments
Training Data	Labeled examples	Labeled examples	Unlabeled data
Goal	Predict category	Predict quantity	Discover groups
Example	Email spam detection	House price prediction	Customer segmentation

Table 1.1: Comparison of Classification, Regression, and Clustering

1.6.2 Decision Framework for Task Selection

Choosing the appropriate technique requires careful consideration of your problem characteristics:

Choose Classification when: Your goal is to assign instances to predefined categories, you have labeled training data, and the output should be a discrete class.

Example: Determining whether a tumor is malignant or benign based on medical imaging.

Choose Regression when: Your goal is to predict a numerical value, you have labeled training data with continuous target values, and precise quantity matters.

Example: Forecasting next month's sales revenue based on historical data and marketing spend.

Choose Clustering when: You want to discover natural groupings without predefined categories, labels don't exist or are expensive to obtain, and exploratory analysis is the primary goal.

Example: Analyzing gene expression data to identify groups of genes with similar behavior.

1.7 Deep Learning

Deep Learning Definition

Deep Learning is a set of Machine Learning algorithms that uses neural networks to accomplish supervised learning (and other tasks). It requires a huge amount of training data.

Deep learning represents a specialized subset of machine learning that has achieved remarkable success in recent years. It uses multi-layered neural networks to automatically learn hierarchical representations of data.

1.7.1 From Classical Machine Learning to Deep Learning

While classical machine learning algorithms have achieved remarkable success across numerous domains, they face limitations when confronting highly complex patterns or massive datasets. Deep learning emerged as a paradigm shift that addresses these limitations through neural networks with multiple layers, enabling automatic feature learning and unprecedented performance on challenging tasks.

The Limitations of Classical Approaches

Traditional machine learning algorithms like Support Vector Machines, Decision Trees, and Naive Bayes require careful feature engineering—the manual process of selecting and transforming input variables. Data scientists must identify relevant features, understand domain-specific patterns, and encode that knowledge explicitly. This process is labor-intensive, requires deep expertise, and may miss subtle patterns.
For instance, in image recognition, classical approaches required researchers to manually design features like edge detectors, color histograms, and texture descriptors. Different problems demanded entirely new feature engineering efforts. Moreover, these approaches struggled with high-dimensional data where patterns exist across complex interactions of many variables.

The Deep Learning Revolution

Deep learning transforms this paradigm by learning features automatically from raw data through hierarchical representations. Instead of manually designing features, deep neural networks discover progressively abstract features at each layer. In image recognition, early layers might detect edges and textures, middle layers recognize shapes and parts, and deep layers identify complete objects.
This approach, inspired by the hierarchical organization of the human visual cortex, enables end-to-end learning where the same architecture can be applied to diverse problems with minimal task-specific modification. The combination of deep architectures, massive datasets, and powerful computational resources (especially GPUs) has produced breakthroughs once thought impossible.

1.7.2 Deep Learning Architectures

Different problem domains have inspired specialized neural network architectures optimized for particular data types and tasks. Understanding these architectures helps practitioners select appropriate models for their applications.

Convolutional Neural Networks (CNNs)

CNNs revolutionized computer vision by exploiting the spatial structure of images. Unlike fully connected networks that treat all inputs independently, CNNs use convolutional layers that apply learnable filters across the image, detecting local patterns like edges, textures, and shapes. Pooling layers reduce spatial dimensions while retaining important features, enabling the network to recognize objects regardless of their position in the image. Applications extend beyond image classification to object detection (identifying and locating multiple objects), semantic segmentation (assigning a class to every pixel), facial recognition, medical image analysis, and autonomous vehicle perception. CNNs have achieved superhuman performance on many visual tasks.

Recurrent Neural Networks (RNNs)

RNNs address sequential data where the order of inputs matters, such as text, speech, or time series. Unlike feedforward networks that process each input independently, RNNs maintain hidden states that capture information from previous time steps, enabling them to model temporal dependencies and context.
Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) enhanced basic RNNs by solving the vanishing gradient problem, enabling them to learn long-range dependencies. These architectures power applications like machine translation, speech recognition, sentiment analysis, and predictive text generation.

Transformer Architecture

Transformers represent the latest breakthrough in sequence modeling, introducing the attention mechanism that allows the network to focus on relevant parts of the input when making predictions. Unlike RNNs that process sequences sequentially, transformers process all positions in parallel, enabling much faster training on modern hardware.
The transformer architecture underlies recent language models like BERT, GPT, and their successors, achieving state-of-the-art performance across natural language processing tasks including translation, summarization, question answering, and text generation. Transformers have also been adapted for computer vision tasks with models like Vision Transformers.

Generative Models

Generative deep learning models learn to create new data similar to their training data. Generative Adversarial Networks (GANs) pit a generator network against a discriminator in an adversarial game, producing remarkably realistic synthetic images, videos, and other content. Variational Autoencoders (VAEs) learn compressed representations of data that can be sampled to generate new instances. Diffusion models, the latest generation of generative models, have produced stunning results in text-to-image generation.

1.7.3 Real-World Applications and Impact

Deep learning has transcended academic research to become a transformative force across industries, powering applications that seemed like science fiction just a decade ago.

Healthcare: Deep learning assists radiologists in detecting tumors in medical images, often matching or exceeding human expert performance. Models predict patient deterioration, personalize treatment recommendations, accelerate drug discovery, and analyze genetic sequences for disease markers.
Autonomous Vehicles: Self-driving cars rely on deep learning for perception (identifying pedestrians, vehicles, lane markings), prediction (forecasting other agents' behavior), and planning (determining safe trajectories). Multiple neural networks work together to process camera, lidar, and radar data in real-time.
Natural Language Processing: Virtual assistants understand spoken commands, machine translation enables real-time multilingual communication, sentiment analysis gauges public opinion, and large language models generate human-quality text for various applications.
Finance: Banks deploy deep learning for fraud detection, algorithmic trading, credit scoring, and risk assessment. Models analyze transaction patterns, market sentiment, and economic indicators to make sophisticated predictions.
Entertainment: Recommendation systems powered by deep learning personalize content on streaming platforms. Generative models create art, music, and video effects. Game AI exhibits realistic behavior through reinforcement learning.

Chapter 2

Unsupervised Learning

Unsupervised learning is a paradigm in machine learning where algorithms learn patterns, structures, and relationships from unlabeled data without explicit guidance or target outputs. Unlike supervised learning, where each training example is paired with a corresponding label or target value, unsupervised learning algorithms must discover the inherent structure in the data autonomously. This approach mirrors how humans naturally learn about the world through observation and pattern recognition, without explicit instruction for every concept.
The fundamental challenge in unsupervised learning is to extract meaningful information from data when no ground truth labels are available. This requires algorithms to identify similarities, differences, and organizational principles that govern the data distribution. Unsupervised learning has become increasingly important in the era of big data, where labeled datasets are expensive and time-consuming to create, while unlabeled data is abundant.

2.0.1 Motivation and Applications

Unsupervised learning addresses several critical needs in modern machine learning and data science:

- **Data exploration and understanding**: Before applying complex models, understanding the structure and characteristics of data is essential. Unsupervised methods help reveal hidden patterns, detect anomalies, and provide insights into data organization.

- **Dimensionality reduction**: High-dimensional data is difficult to visualize and process. Unsupervised techniques can reduce dimensionality while preserving essential information, making data more manageable and interpretable.

- **Feature learning**: Unsupervised methods can automatically discover useful representations and features from raw data, which can then be used for downstream supervised tasks.

- **Preprocessing for supervised learning**: Clustering and dimensionality reduction can serve as preprocessing steps that improve the performance of supervised algorithms.
- **Handling unlabeled data**: In many domains, unlabeled data is plentiful while labeled data is scarce or expensive to obtain. Unsupervised learning leverages this abundant resource.

Real-world applications of unsupervised learning span diverse domains including customer segmentation in marketing, document organization in information retrieval, anomaly detection in cybersecurity, gene expression analysis in bioinformatics, and recommendation systems in e-commerce.

2.1 Problem Formulation

In the unsupervised learning setting, we are given a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_n\}$ consisting of n observations, where each observation $x_i \in \mathbb{R}^d$ is a d-dimensional feature vector. Crucially, unlike supervised learning, there are no corresponding labels or target values. The goal is to learn a function or model that captures the underlying structure of the data. This can take various forms:

- A clustering function $f : \mathbb{R}^d \to \{1, 2, \ldots, k\}$ that assigns each data point to one of k clusters
- A dimensionality reduction mapping $g : \mathbb{R}^d \to \mathbb{R}^m$ where $m \ll d$
- A probability distribution $p(x)$ that models the data generation process
- A representation function $h : \mathbb{R}^d \to \mathbb{R}^l$ that maps data to a learned feature space

The absence of labels means we cannot directly evaluate performance using traditional metrics like accuracy or mean squared error. Instead, evaluation relies on intrinsic measures of data organization, domain expertise, or performance on downstream tasks.

2.2 Main Categories of Unsupervised Learning

Unsupervised learning encompasses several distinct problem types, each addressing different aspects of data structure and organization.

2.2.1 Clustering

Clustering aims to partition data into groups (clusters) such that observations within the same group are more similar to each other than to those in other groups. Formally, we seek a partition $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ of the dataset $\mathcal{D}$ that optimizes some criterion of cluster quality.

Types of Clustering

Partitional Clustering divides the dataset into non-overlapping subsets, where each data point belongs to exactly one cluster. The most common approach optimizes an objective function:

$$\min_{\mathcal{C}} \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu_i) \tag{2.1}$$

where μ_i is the representative (centroid or medoid) of cluster C_i, and $d(\cdot, \cdot)$ is a distance or dissimilarity measure.
Popular partitional clustering algorithms include:

- **K-means**: Uses Euclidean distance and represents clusters by their centroids (arithmetic means)
- **K-medoids**: Uses arbitrary distance metrics and represents clusters by actual data points (medoids)
- **K-modes and K-prototypes**: Extensions for categorical and mixed data types

Hierarchical Clustering builds a tree-like structure (dendrogram) of nested clusters. It comes in two varieties:

- **Agglomerative (bottom-up)**: Starts with each point as its own cluster and iteratively merges the closest pairs
- **Divisive (top-down)**: Starts with all points in one cluster and recursively splits clusters

The merge or split decisions depend on linkage criteria such as single linkage (minimum distance), complete linkage (maximum distance), or average linkage.

Density-Based Clustering identifies clusters as dense regions separated by sparse regions. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is the prototypical algorithm, which groups points that are closely packed together while marking isolated points as outliers.
A point x is a core point if at least minPts points lie within distance ϵ:

$$|\{x' \in \mathcal{D} : d(x, x') \leq \epsilon\}| \geq \text{minPts} \tag{2.2}$$

Points reachable from core points form clusters, while unreachable points are classified as noise.

Model-Based Clustering assumes data is generated from a mixture of probability distributions. Gaussian Mixture Models (GMMs) are the most common approach, where data is modeled as:

$$p(x) = \sum_{i=1}^{k} \pi_i \mathcal{N}(x|\mu_i, \Sigma_i) \tag{2.3}$$

where π_i are mixing coefficients, and $\mathcal{N}(x|\mu_i, \Sigma_i)$ are Gaussian components. The Expectation-Maximization (EM) algorithm is typically used to estimate parameters.

2.2.2 Dimensionality Reduction

Dimensionality reduction transforms high-dimensional data into a lower-dimensional representation while preserving important structural properties. Given data $x \in \mathbb{R}^d$, we seek a mapping to $z \in \mathbb{R}^m$ where $m \ll d$.

Linear Methods

Principal Component Analysis (PCA) finds orthogonal directions of maximum variance in the data. The first principal component is:

$$w_1 = \arg\max_{\|w\|=1} \mathrm{Var}(w^T x) = \arg\max_{\|w\|=1} w^T \Sigma w \tag{2.4}$$

where Σ is the covariance matrix. Subsequent components are found iteratively, subject to orthogonality constraints. PCA can be solved via eigendecomposition of the covariance matrix:

$$\Sigma = W \Lambda W^T \tag{2.5}$$

where columns of W are eigenvectors (principal components) and Λ contains eigenvalues (explained variances).
The dimensionality-reduced representation is:

$$z = W_m^T x \tag{2.6}$$

where W_m contains the top m eigenvectors.

Linear Discriminant Analysis (LDA) maximizes the ratio of between-class to within-class variance. While typically used in supervised settings, variants exist for unsupervised scenarios.

Independent Component Analysis (ICA) seeks statistically independent components rather than uncorrelated ones (as in PCA). It models data as:

$$x = As \tag{2.7}$$

where s contains independent source signals and A is a mixing matrix. ICA finds the unmixing matrix W such that $z = Wx$ are maximally independent.

Nonlinear Methods

Linear methods may fail to capture complex nonlinear structures in data. Nonlinear dimensionality reduction techniques address this limitation.

Autoencoders are neural networks that learn compressed representations through reconstruction:

$$z = f_{\text{encoder}}(x; \theta_e) \tag{2.8}$$
$$\hat{x} = f_{\text{decoder}}(z; \theta_d) \tag{2.9}$$

Training minimizes reconstruction error:

$$\min_{\theta_e, \theta_d} \sum_{i=1}^{n} \|x_i - \hat{x}_i\|^2 \tag{2.10}$$

Variants include denoising autoencoders, variational autoencoders (VAEs), and sparse autoencoders.

t-SNE (t-Distributed Stochastic Neighbor Embedding) preserves local structure by converting distances to probability distributions. For high-dimensional data, pairwise similarities are:

$$p_{ij} = \frac{\exp(-\|x_i - x_j\|^2/2\sigma^2)}{\sum_{k \neq l} \exp(-\|x_k - x_l\|^2/2\sigma^2)} \tag{2.11}$$

In low-dimensional space, using Student's t-distribution:

$$q_{ij} = \frac{(1 + \|z_i - z_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|z_k - z_l\|^2)^{-1}} \tag{2.12}$$

The algorithm minimizes the Kullback-Leibler divergence between P and Q:

$$\text{KL}(P\|Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{2.13}$$

UMAP (Uniform Manifold Approximation and Projection) constructs a fuzzy topological representation of data and optimizes a low-dimensional layout. It often preserves both local and global structure better than t-SNE while being computationally more efficient.

Manifold Learning Methods assume data lies on a low-dimensional manifold embedded in high-dimensional space:

- **Isomap**: Preserves geodesic distances along the manifold using shortest paths in a neighborhood graph

- **Locally Linear Embedding (LLE)**: Preserves local linear reconstructions of points from their neighbors
- **Laplacian Eigenmaps**: Preserves local neighborhood relationships using graph Laplacian

2.2.3 Density Estimation

Density estimation aims to learn the underlying probability distribution $p(x)$ from which data is sampled. This enables generating new samples, computing likelihoods, and detecting anomalies.

Parametric Methods

Parametric density estimation assumes a specific functional form with parameters θ:

$$p(x;\theta) = f(x;\theta) \tag{2.14}$$

Maximum likelihood estimation finds:

$$\hat{\theta} = \arg\max_{\theta} \prod_{i=1}^{n} p(x_i;\theta) = \arg\max_{\theta} \sum_{i=1}^{n} \log p(x_i;\theta) \tag{2.15}$$

Gaussian Mixture Models are a flexible parametric approach that can approximate complex distributions.

Non-Parametric Methods

Non-parametric methods make minimal assumptions about the distribution's form.

Kernel Density Estimation (KDE) estimates the density as:

$$\hat{p}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right) \tag{2.16}$$

where $K(\cdot)$ is a kernel function (typically Gaussian) and h is the bandwidth parameter controlling smoothness.

Histogram-Based Methods partition the space into bins and estimate density within each bin. Adaptive histograms adjust bin sizes based on local data density.

2.2.4 Association Rule Learning

Association rule learning discovers interesting relationships and patterns in large datasets, particularly in transactional data. An association rule has the form $X \Rightarrow Y$, meaning "if X then Y," where X and Y are itemsets.
Key metrics include:

- **Support**: Frequency of itemset occurrence

$$\text{support}(X) = \frac{\text{count}(X)}{n} \tag{2.17}$$

- **Confidence**: Conditional probability

$$\text{confidence}(X \Rightarrow Y) = \frac{\text{support}(X \cup Y)}{\text{support}(X)} \tag{2.18}$$

- **Lift**: Measure of dependence

$$\text{lift}(X \Rightarrow Y) = \frac{\text{support}(X \cup Y)}{\text{support}(X) \times \text{support}(Y)} \tag{2.19}$$

The Apriori algorithm efficiently finds frequent itemsets using the principle that any subset of a frequent itemset must also be frequent.

2.3 Evaluation Metrics

Evaluating unsupervised learning presents unique challenges due to the absence of ground truth labels. Several approaches exist:

2.3.1 Internal Validation Measures

Internal measures evaluate clustering quality using only the data itself:

Silhouette Coefficient measures how similar an object is to its own cluster compared to other clusters. For point i:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \tag{2.20}$$

where $a(i)$ is the average distance to points in the same cluster, and $b(i)$ is the minimum average distance to points in other clusters. Values range from -1 to 1, with higher values indicating better clustering.

Davies-Bouldin Index measures the average similarity between each cluster and its most similar cluster:

$$\text{DB} = \frac{1}{k} \sum_{i=1}^{k} \max_{j \neq i} \frac{\sigma_i + \sigma_j}{d(c_i, c_j)} \tag{2.21}$$

where σ_i is the average distance of points in cluster i to the cluster center c_i. Lower values indicate better clustering.

Calinski-Harabasz Index (Variance Ratio Criterion) measures the ratio of between-cluster to within-cluster variance:

$$\mathrm{CH} = \frac{\mathrm{SSB}/(k-1)}{\mathrm{SSW}/(n-k)} \tag{2.22}$$

where SSB is the between-cluster sum of squares and SSW is the within-cluster sum of squares. Higher values indicate better-defined clusters.

Dunn Index measures the ratio of minimum inter-cluster distance to maximum intra-cluster distance:

$$\mathrm{Dunn} = \frac{\min_{i \neq j} d(C_i, C_j)}{\max_k \Delta(C_k)} \tag{2.23}$$

Higher values indicate better clustering with compact, well-separated clusters.

2.3.2 External Validation Measures

When ground truth labels are available (for evaluation purposes), external measures compare the discovered structure to known labels:

Adjusted Rand Index (ARI) measures agreement between two partitions, adjusted for chance:

$$\mathrm{ARI} = \frac{\mathrm{RI} - \mathbb{E}[\mathrm{RI}]}{\max(\mathrm{RI}) - \mathbb{E}[\mathrm{RI}]} \tag{2.24}$$

Values range from -1 to 1, with 1 indicating perfect agreement and 0 indicating random clustering.

Normalized Mutual Information (NMI) measures the mutual information between cluster assignments and true labels, normalized to $[0, 1]$:

$$\mathrm{NMI} = \frac{2 \cdot I(C;T)}{H(C) + H(T)} \tag{2.25}$$

where I is mutual information and H is entropy.

Fowlkes-Mallows Index is the geometric mean of precision and recall at the pair level:

$$\mathrm{FM} = \sqrt{\frac{TP}{TP+FP} \cdot \frac{TP}{TP+FN}} \tag{2.26}$$

2.3.3 Task-Specific Evaluation

Often, unsupervised learning serves as a preprocessing step for downstream tasks. Evaluation can be based on the performance improvement in these tasks, such as:

- Classification accuracy after dimensionality reduction
- Reconstruction error in autoencoders
- Log-likelihood on held-out data for density estimation
- Perplexity for language models
- Anomaly detection performance (AUC-ROC)

2.3.4 Applications of unsupervised learning

Unsupervised learning techniques find applications across diverse domains, addressing problems where labeled data is scarce, expensive, or impossible to obtain:

- **Image segmentation and computer vision**: Clustering algorithms partition images into meaningful regions based on color, texture, or spatial features. Dimensionality reduction techniques enable efficient representation of visual data, while autoencoders learn compact encodings for image compression and denoising.
- **Document organization and text mining**: Unsupervised methods group documents by topic, discover latent themes through topic modeling, and reduce high-dimensional text representations to interpretable lower-dimensional spaces. Association rule learning identifies frequent patterns in text corpora.
- **Customer segmentation and marketing**: Clustering algorithms identify distinct customer groups based on purchasing behavior, demographics, or engagement patterns. These segments enable targeted marketing strategies, personalized recommendations, and improved customer relationship management without requiring predefined customer categories.
- **Bioinformatics and genomics**: Unsupervised learning reveals population structure in genetic data, identifies co-expressed genes in transcriptomic studies, and discovers functional modules in protein interaction networks. Dimensionality reduction visualizes complex biological datasets and identifies latent factors driving biological processes.
- **Anomaly and outlier detection**: Density estimation and clustering-based approaches identify unusual patterns in network traffic for cybersecurity, detect fraudulent transactions in financial systems, and flag equipment failures in predictive maintenance applications by identifying observations that deviate significantly from learned normal patterns.

- **Recommendation systems**: Collaborative filtering uses unsupervised learning to discover latent user preferences and item characteristics, enabling personalized recommendations without explicit ratings or labels for all user-item pairs.
- **Natural language processing**: Unsupervised methods learn word embeddings that capture semantic relationships, perform language modeling, and extract features for downstream tasks. Clustering identifies semantic similarity among words or documents.
- **Financial analysis**: Dimensionality reduction identifies common risk factors in asset returns, while clustering groups securities with similar behavior. Anomaly detection flags unusual market activities or potential fraudulent transactions.
- **Social network analysis**: Community detection algorithms partition networks into cohesive groups, revealing organizational structure without predefined community labels. Embedding techniques position nodes in low-dimensional spaces that preserve network topology.
- **Exploratory data analysis**: Unsupervised learning serves as a preliminary step to understand data structure, identify patterns, detect quality issues, and generate hypotheses before applying more complex supervised methods or domain-specific analyses.

2.4 Challenges and Considerations

Unsupervised learning faces several fundamental challenges:

2.4.1 Model Selection

Determining the appropriate number of clusters, dimensionality, or model complexity is often difficult. Common approaches include:

- **Elbow method**: Plot objective function values against different model complexities and select the "elbow" point
- **Gap statistic**: Compare within-cluster dispersion to that expected under a null reference distribution
- **Information criteria**: Use AIC (Akaike Information Criterion) or BIC (Bayesian Information Criterion) to balance fit and complexity
- **Cross-validation**: Evaluate stability and consistency across different data subsets

2.4.2 Scalability

Many unsupervised algorithms have high computational complexity, making them impractical for large datasets. Solutions include:

- Sampling-based approaches (e.g., CLARA for clustering)
- Mini-batch processing (e.g., mini-batch K-means)
- Approximate methods (e.g., randomized SVD for PCA)
- Distributed and parallel implementations

2.4.3 High Dimensionality

The curse of dimensionality affects distance-based methods as distances become less meaningful in high dimensions. Strategies include:

- Dimensionality reduction as preprocessing
- Feature selection to identify relevant features
- Subspace clustering methods that find clusters in subspaces
- Ensemble methods that combine multiple views

2.4.4 Interpretability

Understanding and explaining discovered patterns can be challenging, especially with complex models. Techniques for improving interpretability include:

- Visualization of low-dimensional embeddings
- Cluster profiling using statistical summaries
- Feature importance analysis
- Domain expert validation

2.4.5 Stability and Reproducibility

Many unsupervised algorithms are sensitive to initialization and hyperparameters. Best practices include:

- Multiple runs with different initializations
- Stability analysis across parameter settings
- Consensus clustering to combine multiple solutions
- Careful hyperparameter tuning using validation procedures

2.5 Practical Guidelines

When applying unsupervised learning in practice:

1. **Understand your objectives**: Clarify what patterns or structures you seek to discover and how results will be used.
2. **Preprocess carefully**: Normalize features, handle missing values, and remove irrelevant attributes based on domain knowledge.
3. **Start simple**: Begin with straightforward methods (K-means, PCA) before moving to complex approaches.
4. **Validate extensively**: Use multiple evaluation metrics and approaches since no single measure captures all aspects of quality.
5. **Visualize results**: Use dimensionality reduction for visualization and sanity checks, even if your main analysis uses other methods.
6. **Involve domain experts**: Interpretation and validation benefit greatly from domain expertise.
7. **Consider computational constraints**: Choose algorithms appropriate for your data size and computational resources.
8. **Document thoroughly**: Record preprocessing steps, hyperparameters, and evaluation procedures for reproducibility.
9. **Iterate and refine**: Unsupervised learning is often exploratory; be prepared to adjust approaches based on initial findings.
10. **Combine methods**: Ensemble approaches and pipelines that combine multiple techniques often yield better results than individual methods.

2.6 K-Means

All the algorithms we discussed so far are supervised, that is, they assume that labeled training data is available. In many applications this is too much to hope for; labeling may be expensive, error prone, or sometimes impossible. For instance, it is very easy to crawl and collect every page within the `www.purdue.edu` domain, but rather time consuming to assign a topic to each page based on its contents. In such cases, one has to resort to unsupervised learning. A prototypical unsupervised learning algorithm is K-means, which is clustering algorithm. Given $X = \{x_1, \ldots, x_m\}$ the goal of K-means is to partition it into k clusters such that each point in a cluster is similar to points from its own cluster than with points from some other cluster.

Towards this end, define prototype vectors $\mu_1, \ldots, \mu_k$ and an indicator vector r_{ij} which is 1 if, and only if, x_i is assigned to cluster j. To cluster our dataset we will minimize the following distortion measure, which minimizes the distance of each point from the prototype vector:

$$J(r, \mu) := \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{k} r_{ij} \|x_i - \mu_j\|^2, \tag{2.27}$$

where $r = \{r_{ij}\}$, $\mu = \{\mu_j\}$, and $\| \cdot \|^2$ denotes the usual Euclidean square norm.
Our goal is to find r and μ, but since it is not easy to jointly minimize J with respect to both r and μ, we will adapt a two stage strategy:

Stage 1 Keep the μ fixed and determine r. In this case, it is easy to see that the minimization decomposes into m independent problems. The solution for the i-th data point x_i can be found by setting:

$$r_{ij} = 1 \text{ if } j = \underset{j'}{\arg\min} \|x_i - \mu_{j'}\|^2, \tag{2.28}$$

and 0 otherwise.

Stage 2 Keep the r fixed and determine μ. Since the r's are fixed, J is an quadratic function of μ. It can be minimized by setting the derivative with respect to μ_j to be 0:

$$\sum_{i=1}^{m} r_{ij}(x_i - \mu_j) = 0 \text{ for all } j. \tag{2.29}$$

Rearranging obtains

$$\mu_j = \frac{\sum_i r_{ij} x_i}{\sum_i r_{ij}}. \tag{2.30}$$

Since $\sum_i r_{ij}$ counts the number of points assigned to cluster j, we are essentially setting μ_j to be the sample mean of the points assigned to cluster j.
The algorithm stops when the cluster assignments do not change significantly. Detailed pseudo-code can be found in Algorithm 1.
Two issues with K-Means are worth noting. First, it is sensitive to the choice of the initial cluster centers μ. A number of practical heuristics have been developed. For instance, one could randomly choose k points from the given dataset as cluster centers. Other methods try to pick k points from X which are farthest away from each other. Second, it makes a hard assignment of every point to a cluster center. Variants which we will encounter later in
the book will relax this. Instead of letting $r_{ij} \in \{0, 1\}$ these soft variants will replace it with the probability that a given x_i belongs to cluster j.
The K-Means algorithm concludes our discussion of a set of basic machine learning methods for classification and regression. They provide a useful starting point for an aspiring machine learning researcher. In this book we will see many more such algorithms as well as connections between these basic algorithms and their more advanced counterparts.

Algorithm 1 K-Means

```
1: Cluster(X) {Cluster dataset X}
2: Initialize cluster centers $\mu_j$ for $j = 1, \ldots, k$ randomly
3: repeat
4:     for $i = 1$ to $m$ do
5:         Compute $j' = \arg\min_{j=1,\ldots,k} d(x_i, \mu_j)$
6:         Set $r_{ij'} = 1$ and $r_{ij} = 0$ for all $j' \neq j$
7:     end for
8:     for $j = 1$ to $k$ do
9:         Compute $\mu_j = \frac{\sum_i r_{ij} x_i}{\sum_i r_{ij}}$
10:    end for
11: until Cluster assignments $r_{ij}$ are unchanged
12: return $\{\mu_1, \ldots, \mu_k\}$ and $r_{ij}$
```

2.6.1 Dissimilarity Measures

The choice of dissimilarity measure significantly impacts clustering results. Common measures include:

- **Euclidean distance** (for continuous attributes):

$$d(x_i, x_j) = \sqrt{\sum_{l=1}^{p} (x_{il} - x_{jl})^2} \tag{2.31}$$

- **Manhattan distance**:

$$d(x_i, x_j) = \sum_{l=1}^{p} |x_{il} - x_{jl}| \tag{2.32}$$

- **Gower's distance** (for mixed data types):

$$d(x_i, x_j) = \frac{1}{p} \sum_{l=1}^{p} \delta_{ijl} \cdot d_l(x_{il}, x_{jl}) \tag{2.33}$$

  where δ_{ijl} indicates if the l-th attribute is valid for comparison, and d_l is an attribute-specific dissimilarity.

2.7 K-Medoids Clustering Algorithm

K-medoids is a partitioning-based clustering algorithm that serves as a robust alternative to the popular K-means algorithm. While K-means represents clusters by their centroids (geometric mean of points), K-medoids represents each cluster by one of its actual data points, called a medoid. This fundamental difference makes K-medoids more resistant to noise and outliers, and applicable to datasets where computing means is not meaningful or possible.

The algorithm is particularly valuable when working with arbitrary distance metrics or when the actual representative objects (rather than abstract centroids) are needed for interpretation. K-medoids is also known by its most common implementation, the Partitioning Around Medoids (PAM) algorithm.

2.7.1 Problem Formulation

Given a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_n\}$ of n objects and a dissimilarity measure $d(x_i, x_j)$ between any two objects, the K-medoids algorithm aims to partition the data into k clusters by selecting k representative objects (medoids) from the dataset.
Let $M = \{m_1, m_2, \ldots, m_k\}$ denote the set of medoids, where each $m_i \in \mathcal{D}$. The objective is to minimize the total dissimilarity of objects to their nearest medoid:

$$J(M) = \sum_{i=1}^{n} \min_{m_j \in M} d(x_i, m_j) \tag{2.34}$$

This objective function can also be expressed as:

$$J(M, C) = \sum_{j=1}^{k} \sum_{x_i \in C_j} d(x_i, m_j) \tag{2.35}$$

where $C = \{C_1, C_2, \ldots, C_k\}$ is a partition of $\mathcal{D}$ such that each data point is assigned to the cluster of its nearest medoid.

2.7.2 BUILD Phase

The BUILD phase selects k initial medoids greedily. The algorithm proceeds as follows:

1. Select the object that minimizes the sum of dissimilarities to all other objects as the first medoid:

$$m_1 = \arg\min_{x_i \in \mathcal{D}} \sum_{j=1}^{n} d(x_i, x_j) \tag{2.36}$$

2. For each subsequent medoid selection ($l = 2, \ldots, k$), compute the gain g_{ih} of selecting object x_h as a new medoid, given that x_i is currently assigned to its nearest medoid in the existing set. The gain represents the reduction in dissimilarity if x_h becomes a medoid:

$$g_{ih} = \max\{d(x_i, \text{nearest medoid}) - d(x_i, x_h), 0\} \tag{2.37}$$

3. Select the object that maximizes the total gain:

$$m_l = \arg\max_{x_h \notin M} \sum_{i=1}^{n} g_{ih} \tag{2.38}$$

2.7.3 SWAP Phase

After initialization, the SWAP phase iteratively improves the medoid selection:

1. For each medoid $m_i \in M$ and each non-medoid object $x_h \notin M$, calculate the total cost change ΔC_{ih} that would result from swapping m_i with x_h.
2. For each data point x_j, compute its contribution to the cost change:
$$\delta_{jih} = d(x_j, \text{second nearest medoid}) - d(x_j, x_h) \tag{2.39}$$
if x_j is currently assigned to m_i, or
$$\delta_{jih} = \min\{d(x_j, x_h) - d(x_j, m_i), 0\} \tag{2.40}$$
if x_j is not currently assigned to m_i.
3. The total cost change is:
$$\Delta C_{ih} = \sum_{j=1}^{n} \delta_{jih} \tag{2.41}$$
4. Select the swap that produces the greatest reduction in cost:
$$(i^*, h^*) = \underset{m_i \in M, x_h \notin M}{\arg\min} \Delta C_{ih} \tag{2.42}$$
5. If $\Delta C_{i^*h^*} < 0$, perform the swap and repeat. Otherwise, terminate.

2.7.4 Algorithm Complexity Analysis

The computational complexity of PAM consists of:

- **BUILD phase**: $O(k(n-k)^2)$, where each of the k medoid selections requires evaluating $(n-k)$ candidates across n objects.
- **SWAP phase**: $O(k(n-k)^2 \cdot t)$ per iteration, where t is the number of iterations until convergence. Each iteration considers $k(n-k)$ possible swaps, and each swap evaluation requires computing dissimilarities for all n objects.

The overall complexity is $O(k(n-k)^2 \cdot t)$, which can be substantial for large datasets. This has motivated the development of more efficient variants.

2.7.5 Algorithmic Variants

Several variants of K-medoids have been developed to address computational limitations and improve performance:

CLARA (Clustering LARge Applications)

CLARA applies PAM to multiple small samples of the dataset rather than the entire dataset. For each sample:

1. Draw a random sample of size s from the dataset (typically $s \ll n$).
2. Apply PAM to the sample to obtain k medoids.
3. Assign all objects in the full dataset to their nearest medoid.
4. Compute the total dissimilarity.

The process is repeated multiple times, and the set of medoids with the lowest total dissimilarity is selected. CLARA reduces complexity to $O(ks^2 + k(n-k))$, making it suitable for large datasets.

CLARANS (Clustering Large Applications based on RANdomized Search)

CLARANS improves upon CLARA by viewing the clustering process as searching through a graph where each node represents a set of k medoids, and edges connect nodes that differ by a single medoid. Rather than examining all neighbors exhaustively like PAM, CLARANS randomly samples a limited number of neighbors at each step:

1. Start with a random set of k medoids.
2. Randomly sample at most *maxneighbor* swaps.
3. If a better configuration is found, move to it and repeat.
4. If no improvement is found after *maxneighbor* samples, declare the current configuration a local minimum.
5. Restart from a random configuration up to *numlocal* times.

CLARANS provides a good balance between clustering quality and computational efficiency.

2.7.6 Advantages and Limitations

Advantages

- **Robustness**: More resistant to noise and outliers than K-means, as medoids are actual data points rather than computed means that can be influenced by extreme values.
- **Interpretability**: Medoids are actual objects from the dataset, making them easier to interpret and explain than abstract centroids.

- **Flexibility**: Works with any dissimilarity measure, not limited to Euclidean distance. This allows K-medoids to handle categorical data, mixed data types, or custom distance metrics.

- **Deterministic assignment**: Once medoids are fixed, cluster assignments are deterministic based on the dissimilarity measure.

Limitations

- **Computational cost**: The $O(k(n-k)^2 \cdot t)$ complexity makes PAM impractical for large datasets, though variants like CLARA and CLARANS address this issue.

- **Number of clusters**: Like K-means, the number of clusters k must be specified in advance, which may require domain knowledge or auxiliary methods for selection.

- **Local optima**: The algorithm can converge to local optima depending on initialization, though this is less problematic than with K-means.

- **Memory requirements**: Storing the full dissimilarity matrix requires $O(n^2)$ space, which can be prohibitive for very large datasets.

2.7.7 Practical Considerations

When implementing K-medoids clustering:

1. **Preprocessing**: Normalize or standardize features if using distance metrics sensitive to scale.

2. **Initialization**: While PAM's BUILD phase provides a systematic initialization, multiple random initializations with the best result selected can improve outcomes.

3. **Distance computation**: Pre-compute and store the dissimilarity matrix if memory allows, especially for repeated clustering tasks.

4. **Validation**: Use both internal metrics (silhouette coefficient, Davies-Bouldin index) and external validation when ground truth is available.

5. **Scalability**: For datasets with more than a few thousand objects, consider CLARA or CLARANS rather than PAM.

2.7.8 Comparison with K-Means

Table 2.1 summarizes the key differences between K-means and K-medoids clustering.

Table 2.1: Comparison of K-means and K-medoids clustering algorithms

Aspect	**K-means**	**K-medoids**
Cluster representative	Centroid (mean)	Medoid (actual point)
Robustness to outliers	Low	High
Distance metric	Euclidean (typically)	Any dissimilarity
Computational complexity	$O(nkd \cdot t)$	$O(k(n-k)^2 \cdot t)$
Interpretability	Lower	Higher
Data type flexibility	Continuous	Any
Memory requirements	$O(nkd)$	$O(n^2)$ (with matrix)

2.7.9 Implementation Pseudocode

Algorithm 2 presents the complete PAM algorithm in pseudocode.

Algorithm 2 PAM (Partitioning Around Medoids)

```
Input: Dataset $\mathcal{D}$, number of clusters $k$, dissimilarity matrix $D$
Output: Set of medoids $M$, cluster assignments $C$

// BUILD Phase
$M \leftarrow \emptyset$
for $l = 1$ to $k$ do
    if $l = 1$ then
        $m_1 \leftarrow \arg\min_{x_i \in \mathcal{D}} \sum_{j=1}^{n} d(x_i, x_j)$
        $M \leftarrow M \cup \{m_1\}$
    else
        maxGain $\leftarrow -\infty$
        for all $x_h \in \mathcal{D} \setminus M$ do
            totalGain $\leftarrow 0$
            for all $x_i \in \mathcal{D}$ do
                $d_{\text{nearest}} \leftarrow \min_{m_j \in M} d(x_i, m_j)$
                totalGain $\leftarrow$ totalGain $+ \max\{d_{\text{nearest}} - d(x_i, x_h), 0\}$
            end for
            if totalGain $>$ maxGain then
                maxGain $\leftarrow$ totalGain
                bestMedoid $\leftarrow x_h$
            end if
        end for
        $M \leftarrow M \cup \{\text{bestMedoid}\}$
    end if
end for

// SWAP Phase
improved $\leftarrow$ True
while improved do
    improved $\leftarrow$ False
    minCostChange $\leftarrow 0$
    for all $m_i \in M$ do
        for all $x_h \in \mathcal{D} \setminus M$ do
            $\Delta C \leftarrow 0$
            for all $x_j \in \mathcal{D}$ do
                Compute cost change $\delta_{jih}$ for swapping $m_i$ with $x_h$
                $\Delta C \leftarrow \Delta C + \delta_{jih}$
            end for
            if $\Delta C <$ minCostChange then
                minCostChange $\leftarrow \Delta C$
                bestSwap $\leftarrow (m_i, x_h)$
                improved $\leftarrow$ True
            end if
        end for
    end for
    if improved then
        Perform swap: replace bestSwap[0] with bestSwap[1] in $M$
    end if
end while
```

2.8 Principal Component Analysis

Principal Component Analysis (PCA) is one of the most widely used techniques for dimensionality reduction and data analysis. Introduced by Karl Pearson in 1901 and later developed by Harold Hotelling in the 1930s, PCA remains a cornerstone of modern data science and machine learning. The fundamental idea behind PCA is elegantly simple yet powerful: transform the original variables into a new set of uncorrelated variables called principal components, ordered such that the first few retain most of the variation present in all of the original variables.

PCA addresses several critical challenges in data analysis. High-dimensional data is difficult to visualize, computationally expensive to process, and often contains redundant information due to correlations among features. By identifying directions of maximum variance in the data, PCA provides a lower-dimensional representation that preserves the most important information while discarding noise and redundancy. This makes PCA invaluable for data compression, visualization, noise reduction, feature extraction, and as a preprocessing step for other machine learning algorithms.

The versatility of PCA is reflected in its applications across diverse domains: facial recognition systems use PCA to identify eigenfaces, financial analysts employ it to identify common factors driving stock price movements, geneticists use it to visualize population structure, and neuroscientists apply it to analyze brain imaging data. Understanding PCA is essential for anyone working with high-dimensional data.

2.8.1 Problem Formulation

Consider a dataset consisting of n observations, where each observation is a d-dimensional vector. We can represent this data as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, where each row corresponds to an observation and each column to a feature:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{bmatrix} \tag{2.43}$$

The goal of PCA is to find a linear transformation that projects this data onto a lower-dimensional subspace of dimension $m < d$ while maximizing the variance of the projected data. Equivalently, we seek to minimize the reconstruction error when projecting back to the original space.

More formally, we want to find an orthonormal basis $\{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m\}$ where each $\mathbf{w}_i \in \mathbb{R}^d$ and $\mathbf{w}_i^T \mathbf{w}_j = \delta_{ij}$, such that projecting the data onto this basis preserves maximum variance or equivalently minimizes reconstruction error.

2.8.2 Mathematical Foundations

Data Centering

The first step in PCA is to center the data by subtracting the mean of each feature. Let $\bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i$ be the mean vector. The centered data matrix is:

$$\tilde{\mathbf{X}} = \mathbf{X} - \mathbf{1}_n\bar{\mathbf{x}}^T \tag{2.44}$$

where $\mathbf{1}_n$ is an n-dimensional vector of ones. Centering ensures that the principal components pass through the origin of the data cloud and simplifies subsequent calculations. Each row of $\tilde{\mathbf{X}}$ represents a centered observation: $\tilde{\mathbf{x}}_i = \mathbf{x}_i - \bar{\mathbf{x}}$.

Covariance Matrix

The sample covariance matrix captures the linear relationships between features:

$$\mathbf{\Sigma} = \frac{1}{n-1}\tilde{\mathbf{X}}^T\tilde{\mathbf{X}} = \frac{1}{n-1}\sum_{i=1}^{n}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T \tag{2.45}$$

This $d \times d$ symmetric positive semi-definite matrix has elements:

$$\Sigma_{jk} = \frac{1}{n-1}\sum_{i=1}^{n}(x_{ij} - \bar{x}_j)(x_{ik} - \bar{x}_k) \tag{2.46}$$

The diagonal elements Σ_{jj} are the variances of individual features, while off-diagonal elements Σ_{jk} (for $j \neq k$) are covariances representing linear dependencies between features.

Variance Maximization Perspective

The first principal component is the direction along which the projected data has maximum variance. For a unit vector $\mathbf{w} \in \mathbb{R}^d$ with $\|\mathbf{w}\| = 1$, the projection of data point $\tilde{\mathbf{x}}_i$ onto $\mathbf{w}$ is the scalar:

$$z_i = \mathbf{w}^T\tilde{\mathbf{x}}_i \tag{2.47}$$

The variance of the projected data is:

$$\begin{aligned}
\mathrm{Var}(\mathbf{w}^T\tilde{\mathbf{X}}) &= \frac{1}{n-1}\sum_{i=1}^{n}(\mathbf{w}^T\tilde{\mathbf{x}}_i)^2 && (2.48)\\
&= \frac{1}{n-1}\sum_{i=1}^{n}\mathbf{w}^T\tilde{\mathbf{x}}_i\tilde{\mathbf{x}}_i^T\mathbf{w} && (2.49)\\
&= \mathbf{w}^T\left(\frac{1}{n-1}\sum_{i=1}^{n}\tilde{\mathbf{x}}_i\tilde{\mathbf{x}}_i^T\right)\mathbf{w} && (2.50)\\
&= \mathbf{w}^T\mathbf{\Sigma}\mathbf{w} && (2.51)
\end{aligned}$$

The first principal component is found by solving the constrained optimization problem:

$$\mathbf{w}_1 = \arg\max_{\mathbf{w}} \mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w} \quad \text{subject to} \quad \mathbf{w}^T \mathbf{w} = 1 \tag{2.52}$$

Using the method of Lagrange multipliers, we form the Lagrangian:

$$\mathcal{L}(\mathbf{w}, \lambda) = \mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w} - \lambda(\mathbf{w}^T \mathbf{w} - 1) \tag{2.53}$$

Taking the derivative with respect to $\mathbf{w}$ and setting it to zero:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 2\boldsymbol{\Sigma}\mathbf{w} - 2\lambda\mathbf{w} = 0 \tag{2.54}$$

This yields the eigenvalue equation:

$$\boldsymbol{\Sigma}\mathbf{w} = \lambda\mathbf{w} \tag{2.55}$$

Thus, $\mathbf{w}$ must be an eigenvector of $\boldsymbol{\Sigma}$, and the variance along this direction is:

$$\mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w} = \mathbf{w}^T \lambda \mathbf{w} = \lambda \tag{2.56}$$

To maximize variance, we choose the eigenvector corresponding to the largest eigenvalue. Subsequent principal components are eigenvectors corresponding to successively smaller eigenvalues, with the orthogonality constraint ensuring they are uncorrelated.

Reconstruction Error Minimization Perspective

An alternative derivation views PCA as minimizing reconstruction error. When we project data onto a lower-dimensional subspace and then reconstruct it, we incur an error. For a subspace spanned by orthonormal vectors $\{\mathbf{w}_1, \dots, \mathbf{w}_m\}$, the projection of $\tilde{\mathbf{x}}_i$ is:

$$\tilde{\mathbf{x}}_i^{\text{proj}} = \sum_{j=1}^{m} (\mathbf{w}_j^T \tilde{\mathbf{x}}_i)\mathbf{w}_j = \mathbf{W}\mathbf{W}^T \tilde{\mathbf{x}}_i \tag{2.57}$$

where $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_m]$ is the $d \times m$ matrix of principal components.
The reconstruction error for the entire dataset is:

$$E = \sum_{i=1}^{n} \|\tilde{\mathbf{x}}_i - \tilde{\mathbf{x}}_i^{\text{proj}}\|^2 = \sum_{i=1}^{n} \|\tilde{\mathbf{x}}_i - \mathbf{W}\mathbf{W}^T \tilde{\mathbf{x}}_i\|^2 \tag{2.58}$$

Expanding this expression:

$$E = \sum_{i=1}^{n} \|\tilde{\mathbf{x}}_i\|^2 - 2\sum_{i=1}^{n} \tilde{\mathbf{x}}_i^T \mathbf{W}\mathbf{W}^T \tilde{\mathbf{x}}_i + \sum_{i=1}^{n} \|\mathbf{W}^T \tilde{\mathbf{x}}_i\|^2 \tag{2.59}$$

$$= \sum_{i=1}^{n} \|\tilde{\mathbf{x}}_i\|^2 - \sum_{i=1}^{n} \|\mathbf{W}^T \tilde{\mathbf{x}}_i\|^2 \tag{2.60}$$

Since the first term is constant, minimizing reconstruction error is equivalent to maximizing the second term, which is the variance of the projected data. This establishes the equivalence between the variance maximization and reconstruction error minimization formulations.

2.8.3 Eigendecomposition Solution

The solution to PCA involves computing the eigendecomposition of the covariance matrix $\boldsymbol{\Sigma}$:

$$\boldsymbol{\Sigma} = \mathbf{W}\boldsymbol{\Lambda}\mathbf{W}^T \tag{2.61}$$

where:

- $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_d]$ is a $d \times d$ orthogonal matrix whose columns are the eigenvectors
- $\boldsymbol{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_d)$ is a diagonal matrix of eigenvalues
- Eigenvalues are ordered: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$

The eigenvectors $\{\mathbf{w}_1, \dots, \mathbf{w}_d\}$ form an orthonormal basis:

$$\mathbf{w}_i^T \mathbf{w}_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{2.62}$$

Each eigenvalue λ_i represents the variance explained by the corresponding principal component $\mathbf{w}_i$.

Total Variance

The total variance in the data is the sum of variances along all original dimensions, which equals the trace of the covariance matrix:

$$\text{Var}_{\text{total}} = \text{tr}(\boldsymbol{\Sigma}) = \sum_{i=1}^{d} \Sigma_{ii} = \sum_{i=1}^{d} \lambda_i \tag{2.63}$$

This equality follows from the invariance of trace under similarity transformations:

$$\text{tr}(\boldsymbol{\Sigma}) = \text{tr}(\mathbf{W}\boldsymbol{\Lambda}\mathbf{W}^T) = \text{tr}(\boldsymbol{\Lambda}\mathbf{W}^T\mathbf{W}) = \text{tr}(\boldsymbol{\Lambda}) = \sum_{i=1}^{d} \lambda_i \tag{2.64}$$

Proportion of Variance Explained

The proportion of total variance explained by the first m principal components is:

$$\rho_m = \frac{\sum_{i=1}^{m} \lambda_i}{\sum_{i=1}^{d} \lambda_i} \tag{2.65}$$

This quantity is crucial for selecting the number of components to retain. Typically, we choose m such that ρ_m exceeds a threshold (e.g., 0.90 or 0.95), meaning we preserve 90% or 95% of the total variance.

2.8.4 Dimensionality Reduction via PCA

Once we have computed the principal components, dimensionality reduction involves projecting the data onto the first m components:

$$\mathbf{Z} = \tilde{\mathbf{X}}\mathbf{W}_m \tag{2.66}$$

where $\mathbf{W}_m = [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m]$ is the $d \times m$ matrix containing the first m eigenvectors, and $\mathbf{Z} \in \mathbb{R}^{n\times m}$ is the reduced representation.
For a single data point $\mathbf{x}$, the projection is:

$$\mathbf{z} = \mathbf{W}_m^T(\mathbf{x} - \bar{\mathbf{x}}) \tag{2.67}$$

The components of $\mathbf{z}$ are called the principal component scores:

$$z_j = \mathbf{w}_j^T(\mathbf{x} - \bar{\mathbf{x}}) = \sum_{k=1}^{d} w_{jk}(x_k - \bar{x}_k) \tag{2.68}$$

Reconstruction

We can reconstruct an approximation of the original data from the reduced representation:

$$\hat{\mathbf{x}} = \bar{\mathbf{x}} + \mathbf{W}_m\mathbf{z} = \bar{\mathbf{x}} + \sum_{j=1}^{m} z_j\mathbf{w}_j \tag{2.69}$$

The reconstruction error for a single point is:

$$\|\mathbf{x} - \hat{\mathbf{x}}\|^2 = \sum_{j=m+1}^{d} z_j^2 = \sum_{j=m+1}^{d} (\mathbf{w}_j^T(\mathbf{x} - \bar{\mathbf{x}}))^2 \tag{2.70}$$

The total reconstruction error across all data points is:

$$E_m = \sum_{i=1}^{n}\sum_{j=m+1}^{d} (\mathbf{w}_j^T\tilde{\mathbf{x}}_i)^2 = (n-1)\sum_{j=m+1}^{d} \lambda_j \tag{2.71}$$

2.8.5 Singular Value Decomposition (SVD) Approach

An alternative and often more numerically stable method for computing PCA uses Singular Value Decomposition (SVD) directly on the data matrix without explicitly forming the covariance matrix.
The SVD of the centered data matrix $\tilde{\mathbf{X}} \in \mathbb{R}^{n\times d}$ is:

$$\tilde{\mathbf{X}} = \mathbf{U}\mathbf{S}\mathbf{V}^T \tag{2.72}$$

where:

- $\mathbf{U} \in \mathbb{R}^{n\times n}$ is an orthogonal matrix of left singular vectors
- $\mathbf{S} \in \mathbb{R}^{n\times d}$ is a rectangular diagonal matrix of singular values $s_1 \geq s_2 \geq \cdots \geq s_{\min(n,d)} \geq 0$
- $\mathbf{V} \in \mathbb{R}^{d\times d}$ is an orthogonal matrix of right singular vectors

The connection to PCA comes from the relationship between SVD and eigendecomposition:

$$\begin{align}
\boldsymbol{\Sigma} &= \frac{1}{n-1}\tilde{\mathbf{X}}^T\tilde{\mathbf{X}} \tag{2.73}\\
&= \frac{1}{n-1}(\mathbf{U}\mathbf{S}\mathbf{V}^T)^T(\mathbf{U}\mathbf{S}\mathbf{V}^T) \tag{2.74}\\
&= \frac{1}{n-1}\mathbf{V}\mathbf{S}^T\mathbf{U}^T\mathbf{U}\mathbf{S}\mathbf{V}^T \tag{2.75}\\
&= \frac{1}{n-1}\mathbf{V}\mathbf{S}^T\mathbf{S}\mathbf{V}^T \tag{2.76}\\
&= \mathbf{V}\left(\frac{\mathbf{S}^T\mathbf{S}}{n-1}\right)\mathbf{V}^T \tag{2.77}
\end{align}$$

Comparing with the eigendecomposition $\boldsymbol{\Sigma} = \mathbf{W}\boldsymbol{\Lambda}\mathbf{W}^T$, we identify:

- Principal components: $\mathbf{W} = \mathbf{V}$ (right singular vectors)
- Eigenvalues: $\lambda_i = \frac{s_i^2}{n-1}$ (related to squared singular values)

The principal component scores can be computed directly as:

$$\mathbf{Z} = \tilde{\mathbf{X}}\mathbf{V}_m = \mathbf{U}_m\mathbf{S}_m \tag{2.78}$$

where $\mathbf{V}_m$ contains the first m columns of $\mathbf{V}$, $\mathbf{U}_m$ contains the first m columns of $\mathbf{U}$, and $\mathbf{S}_m$ is the $m \times m$ diagonal matrix of the first m singular values.

Advantages of SVD Approach

Using SVD for PCA offers several advantages:

1. **Numerical stability**: SVD algorithms are generally more numerically stable than eigendecomposition, especially when the covariance matrix is ill-conditioned.
2. **Computational efficiency**: For $n \ll d$ (many features, few observations), computing SVD of $\tilde{\mathbf{X}}$ is more efficient than forming and decomposing $\boldsymbol{\Sigma}$.

3. **Avoiding explicit covariance**: When d is very large, forming the $d \times d$ covariance matrix may be memory-prohibitive.

4. **Direct computation**: Principal component scores are obtained directly without additional matrix multiplication.

2.8.6 Algorithm

Algorithm 3 presents the complete PCA procedure.

Algorithm 3 Principal Component Analysis (PCA)

```
Input: Data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, number of components $m$
Output: Principal components $\mathbf{W}_m$, transformed data $\mathbf{Z}$, eigenvalues $\{\lambda_1, \ldots, \lambda_m\}$

// Step 1: Center the data
Compute mean vector: $\bar{\mathbf{x}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$
Center data: $\tilde{\mathbf{X}} = \mathbf{X} - \mathbf{1}_n \bar{\mathbf{x}}^T$

// Step 2: Compute principal components (Method A: Eigendecomposition)
Compute covariance matrix: $\boldsymbol{\Sigma} = \frac{1}{n-1} \tilde{\mathbf{X}}^T \tilde{\mathbf{X}}$
Compute eigendecomposition: $\boldsymbol{\Sigma} = \mathbf{W} \boldsymbol{\Lambda} \mathbf{W}^T$
Sort eigenvalues in descending order: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$
Reorder eigenvectors accordingly
Select first $m$ eigenvectors: $\mathbf{W}_m = [\mathbf{w}_1, \ldots, \mathbf{w}_m]$

// Alternative Step 2: Compute principal components (Method B: SVD)
Compute SVD: $\tilde{\mathbf{X}} = \mathbf{U} \mathbf{S} \mathbf{V}^T$
Select first $m$ right singular vectors: $\mathbf{W}_m = \mathbf{V}_m$
Compute eigenvalues: $\lambda_i = \frac{s_i^2}{n-1}$ for $i = 1, \ldots, m$

// Step 3: Project data
Compute principal component scores: $\mathbf{Z} = \tilde{\mathbf{X}} \mathbf{W}_m$

// Step 4: Compute explained variance
for $i = 1$ to $m$ do
    explained$_i = \frac{\lambda_i}{\sum_{j=1}^{d} \lambda_j}$
end for
cumulative$_m = \frac{\sum_{i=1}^{m} \lambda_i}{\sum_{j=1}^{d} \lambda_j}$

return $\mathbf{W}_m, \mathbf{Z}, \{\lambda_1, \ldots, \lambda_m\}, \bar{\mathbf{x}}$
```

2.8.7 Selecting the Number of Components

Determining the appropriate number of principal components to retain is a critical decision in applying PCA. Several methods exist:

Scree Plot

A scree plot displays eigenvalues (or explained variance) in descending order. The "elbow" or point where the curve begins to level off suggests a natural cutoff. Components before the elbow capture most of the signal, while those after primarily represent noise.

Cumulative Explained Variance

Choose m such that the cumulative explained variance exceeds a predetermined threshold:

$$m = \min \left\{ k : \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{j=1}^{d} \lambda_j} \geq \tau \right\} \tag{2.79}$$

Common thresholds are $\tau = 0.90$ (90% variance) or $\tau = 0.95$ (95% variance).

Kaiser Criterion

Retain components with eigenvalues greater than 1 (when using the correlation matrix) or greater than the average eigenvalue:

$$m = |\{i : \lambda_i > \bar{\lambda}\}| \quad \text{where} \quad \bar{\lambda} = \frac{1}{d} \sum_{j=1}^{d} \lambda_j \tag{2.80}$$

This criterion is based on the idea that a component should explain more variance than a single original variable on average.

Cross-Validation

For supervised learning tasks, use cross-validation to evaluate how different values of m affect downstream performance.

Parallel Analysis

Compare eigenvalues from the actual data to those from random data with the same dimensions. Retain components whose eigenvalues exceed those from random data.

2.8.8 Computational Complexity

The computational complexity of PCA depends on the chosen method:

Eigendecomposition Method

1. Computing covariance matrix: $O(nd^2)$
2. Eigendecomposition: $O(d^3)$
3. Projection: $O(ndm)$

Total complexity: $O(nd^2 + d^3)$, dominated by eigendecomposition when d is large.

SVD Method

1. SVD computation: $O(nd \min(n, d))$
2. Projection (if needed separately): $O(ndm)$

Total complexity: $O(nd \min(n, d))$
The SVD method is more efficient when $n \ll d$ (many features, few samples) or $d \ll n$ (many samples, few features).

Incremental and Randomized Methods

For very large datasets, approximate methods offer significant speedups:

- **Incremental PCA**: Processes data in mini-batches, updating components iteratively. Complexity: $O(ndm)$ with $O(dm)$ memory.
- **Randomized PCA**: Uses random projections to approximate the top eigenvectors. Complexity: $O(ndm)$ with high probability of accurate results.
- **Sparse PCA**: Introduces sparsity constraints for interpretability. Typically solved via iterative algorithms.

2.8.9 Extensions and Variants

Kernel PCA

Kernel PCA extends PCA to capture nonlinear structures by implicitly mapping data to a high-dimensional feature space using kernel functions. The kernel matrix is:

$$\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \tag{2.81}$$

where ϕ is an implicit feature map and k is a kernel function (e.g., RBF, polynomial).
Kernel PCA performs eigendecomposition on the centered kernel matrix:

$$\tilde{\mathbf{K}} = \mathbf{K} - \mathbf{1}_n\mathbf{K}/n - \mathbf{K}\mathbf{1}_n/n + \mathbf{1}_n\mathbf{K}\mathbf{1}_n/n^2 \tag{2.82}$$

The principal components in the feature space are obtained from eigenvectors of $\tilde{\mathbf{K}}$.

Sparse PCA

Sparse PCA introduces sparsity constraints to make principal components more interpretable by having many zero coefficients. The optimization problem becomes:

$$\max_{\mathbf{w}} \mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w} \quad \text{subject to} \quad \|\mathbf{w}\|_2 = 1, \|\mathbf{w}\|_0 \leq s \tag{2.83}$$

where $\|\mathbf{w}\|_0$ counts non-zero elements and s controls sparsity. This non-convex problem is typically relaxed using L_1 regularization:

$$\max_{\mathbf{w}} \mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w} - \lambda \|\mathbf{w}\|_1 \quad \text{subject to} \quad \|\mathbf{w}\|_2 = 1 \tag{2.84}$$

Robust PCA

Robust PCA decomposes a data matrix into low-rank and sparse components:

$$\mathbf{X} = \mathbf{L} + \mathbf{S} \tag{2.85}$$

where $\mathbf{L}$ is a low-rank matrix (capturing the main structure) and $\mathbf{S}$ is a sparse matrix (capturing outliers and noise). This is solved via convex optimization:

$$\min_{\mathbf{L},\mathbf{S}} \|\mathbf{L}\|_* + \lambda \|\mathbf{S}\|_1 \quad \text{subject to} \quad \mathbf{X} = \mathbf{L} + \mathbf{S} \tag{2.86}$$

where $\|\mathbf{L}\|_*$ is the nuclear norm (sum of singular values).

Probabilistic PCA

Probabilistic PCA provides a probabilistic interpretation where data is generated from a latent variable model:

$$\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_m) \tag{2.87}$$

$$\mathbf{x}|\mathbf{z} \sim \mathcal{N}(\mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \sigma^2 \mathbf{I}_d) \tag{2.88}$$

This model can be fit using maximum likelihood estimation or EM algorithm, and it naturally handles missing data.

Independent Component Analysis (ICA)

While PCA finds uncorrelated components, ICA finds statistically independent components. ICA assumes:

$$\mathbf{x} = \mathbf{A}\mathbf{s} \tag{2.89}$$

where $\mathbf{s}$ contains independent source signals. ICA maximizes non-Gaussianity of components rather than variance.

2.8.10 Applications

PCA finds applications across numerous domains:

Image Compression and Processing

In image processing, PCA can compress images by retaining only the most significant components. For face recognition, eigenfaces (principal components of face images) provide a compact representation:

- Each face image is vectorized into high-dimensional space
- PCA identifies principal components (eigenfaces)
- Faces are represented by projections onto eigenfaces
- Recognition compares these compact representations

Data Visualization

High-dimensional data can be projected onto 2 or 3 principal components for visualization:

$$\mathbf{z}_{2D} = [\mathbf{w}_1, \mathbf{w}_2]^T(\mathbf{x} - \bar{\mathbf{x}}) \tag{2.90}$$

This reveals clusters, outliers, and overall data structure in an interpretable low-dimensional space.

Feature Extraction and Preprocessing

PCA serves as a preprocessing step for machine learning:

- Reduces dimensionality to avoid curse of dimensionality
- Removes multicollinearity among features
- Reduces computational cost of subsequent algorithms
- Can improve generalization by removing noise

Anomaly Detection

Points with large reconstruction errors are potential anomalies:

$$\text{score}(\mathbf{x}) = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 = \sum_{j=m+1}^{d} (\mathbf{w}_j^T(\mathbf{x} - \bar{\mathbf{x}}))^2 \tag{2.91}$$

High scores indicate unusual patterns not well-represented by principal components.

Signal Processing

In signal processing, PCA separates signal from noise:

- Principal components with large eigenvalues capture signal
- Components with small eigenvalues represent noise
- Reconstruction using top components filters noise

Genomics and Bioinformatics

In genomics, PCA reveals population structure:

- Each individual is a data point in SNP (genetic variant) space
- Principal components capture genetic ancestry
- Visualization reveals population substructure
- Used as covariates in genome-wide association studies

Finance

In quantitative finance, PCA identifies common factors:

- Apply PCA to correlation matrix of asset returns
- Principal components represent systematic risk factors
- Used for portfolio construction and risk management
- Factor models based on principal components

2.8.11 Practical Considerations

When to Use PCA

PCA is most effective when:

- Data has linear structure
- Features are correlated or redundant
- Dimensionality reduction is needed
- Noise reduction is beneficial
- Computational efficiency is important

When to Avoid PCA

PCA may be inappropriate when:

- Data has strongly nonlinear structure (consider kernel PCA or manifold learning)
- Feature interpretability is critical (principal components are linear combinations)
- Data contains categorical variables (consider correspondence analysis)
- Supervised learning with specific targets (consider supervised dimensionality reduction)

Preprocessing Steps

Before applying PCA:

1. **Handle missing values**: Impute or remove observations with missing data
2. **Remove outliers**: Extreme outliers can distort principal components
3. **Standardize features**: Unless all features have the same scale and units
4. **Check for nonlinearity**: Visualize relationships to assess linearity

Interpretation

Interpreting principal components involves:

- **Loading plots**: Visualize coefficients $\mathbf{w}_j$ to understand which original features contribute most
- **Biplot**: Combines score plot (data in PC space) and loading plot
- **Explained variance**: Quantify importance of each component
- **Physical meaning**: Relate components to domain knowledge when possible

The loading of feature k on component j is the coefficient w_{jk}. Large absolute values indicate strong contribution.

Chapter 3

Naive Bayes and Logistic Regression

This chapter explores the fundamental connection between supervised learning and probabilistic reasoning through Bayesian principles. We examine two prominent classification algorithms that approach the learning problem from complementary perspectives: Naive Bayes, which models the generative process of data creation, and Logistic Regression, which directly models the decision boundary. Understanding both approaches provides crucial insights into the bias-variance tradeoff and the practical considerations that guide algorithm selection in real-world applications.

3.1 Bayesian Foundations for Classification

3.1.1 Problem Formulation

In supervised classification, our objective is to learn an unknown mapping $f : \mathcal{X} \rightarrow \mathcal{Y}$, which can equivalently be expressed as estimating the conditional probability distribution $P(Y|X)$. Throughout this chapter, we focus on binary classification problems where Y represents a boolean-valued target variable, and X denotes a feature vector composed of n attributes: $X = \langle X_1, X_2, \ldots, X_n \rangle$, with each X_i representing the i-th feature.

3.1.2 Notational Conventions

To maintain clarity and consistency throughout our discussion, we adopt the following notational standards:

- **Random Variables:** Uppercase letters (e.g., X, Y) denote random variables, whether scalar or vector-valued.
- **Features:** For vector-valued variables, subscripts identify individual components (e.g., X_i represents the i-th feature of X).
- **Values:** Lowercase letters represent specific realizations of random variables (e.g., $X_i = x_{ij}$ indicates that feature X_i takes its j-th possible value).

- **Abbreviated Notation:** When context permits, we simplify expressions by omitting variable names (e.g., writing $P(x_{ij}|y_k)$ instead of $P(X_i = x_{ij}|Y = y_k)$).
- **Expected Value:** $E[X]$ denotes the expectation of random variable X.
- **Training Examples:** Superscripts index training instances (e.g., X_i^j refers to feature X_i in the j-th example).
- **Indicator Function:** $\delta(x)$ equals 1 when logical statement x is true, and 0 otherwise.
- **Counting Operator:** $\#D\{x\}$ counts the number of elements in set D satisfying property x.
- **Estimates:** A circumflex accent indicates estimated values (e.g., $\hat{\theta}$ represents an estimate of parameter θ).

3.1.3 Bayes Rule for Classification

The foundation of Bayesian classification lies in Bayes theorem, which allows us to express the posterior probability $P(Y|X)$ in terms of more tractable components:

$$P(Y = y_i|X = x_k) = \frac{P(X = x_k|Y = y_i)P(Y = y_i)}{\sum_j P(X = x_k|Y = y_j)P(Y = y_j)} \tag{3.1}$$

In this formulation, y_m represents the m-th possible value of the target variable Y, x_k denotes the k-th possible feature vector, and the summation in the denominator ranges over all possible values of Y.
This decomposition suggests a learning strategy: estimate $P(X|Y)$ (the likelihood) and $P(Y)$ (the prior) from training data, then apply Bayes rule to compute $P(Y|X = x_k)$ for novel instances x_k. This approach characterizes *generative classifiers*, which model how data is generated from each class.

3.1.4 The Curse of Dimensionality in Bayesian Classification

While the Bayesian approach appears elegant, implementing it without simplifying assumptions proves computationally and statistically infeasible for most practical problems. Consider the data requirements for reliable parameter estimation.
For a boolean target variable Y, estimating the prior distribution $P(Y)$ requires relatively modest amounts of data. With approximately one hundred independent training examples, maximum likelihood estimation typically yields estimates within a few percentage points of the true value [**?**].
However, accurately estimating the likelihood $P(X|Y)$ presents a fundamentally different challenge. When Y is boolean and X consists of n boolean features, we must estimate parameters:

$$\theta_{ij} \equiv P(X = x_i|Y = y_j) \tag{3.2}$$

Here, index i ranges over all 2^n possible feature vector configurations, while j takes on 2 values (corresponding to the two classes). This yields approximately 2^{n+1} parameters.
To determine the exact number of independent parameters, observe that for any fixed class y_j, the probabilities must satisfy the constraint $\sum_i \theta_{ij} = 1$. Therefore, for each class, we need estimate only $2^n - 1$ independent parameters, giving a total of $2(2^n - 1)$ parameters across both classes.
This exponential growth creates two critical problems:

1. **Parameter Explosion:** We require distinct parameters for every possible instance in the feature space. For example, with merely 30 boolean features, we must estimate over 3 billion parameters.

2. **Data Requirements:** Obtaining reliable estimates demands observing each distinct feature vector configuration multiple times, requiring sample sizes that scale exponentially with dimensionality.

This computational and statistical bottleneck necessitates introducing simplifying assumptions to make Bayesian classification tractable.

3.2 The Naive Bayes Classifier

To overcome the intractable sample complexity of full Bayesian classification, the Naive Bayes algorithm introduces a strong but often effective independence assumption. This assumption reduces the number of parameters from $2(2^n-1)$ to merely $2n$, enabling practical learning even in high-dimensional spaces.

3.2.1 Conditional Independence: Concept and Intuition

Before developing the Naive Bayes algorithm, we must understand conditional independence, a fundamental concept in probability theory.
Definition: Given three sets of random variables X, Y, and Z, we say X is **conditionally independent** of Y given Z if and only if:

$$(\forall i, j, k) \quad P(X = x_i|Y = y_j, Z = z_k) = P(X = x_i|Z = z_k) \tag{3.3}$$

This definition states that once we know the value of Z, learning the value of Y provides no additional information about X.
Illustrative Example: Consider three boolean random variables describing atmospheric conditions: *Rain*, *Thunder*, and *Lightning*. We can reasonably assert that *Thunder* is conditionally independent of *Rain* given *Lightning*.
The reasoning is causal: since *Lightning* directly causes *Thunder*, once we observe whether *Lightning* has occurred, the presence or absence of *Rain* provides no additional predictive power for *Thunder*. Note that *Thunder* and *Rain* are certainly dependent in general (both being more likely during storms), but this dependence vanishes when we condition on *Lightning*.

While this example involves single random variables, the definition extends naturally to sets of variables. For instance, we might assert that variables $\{A, B\}$ are conditionally independent of $\{C, D\}$ given $\{E, F\}$.

3.2.2 The Naive Bayes Assumption

The Naive Bayes classifier makes a particularly strong conditional independence assumption: given the class label Y, all features X_i are mutually independent. Formally, each feature X_i is conditionally independent of every other feature X_k (where $k \neq i$) and every subset of other features, given Y.
Despite being called "naive" because this assumption rarely holds perfectly in real-world data, it often works remarkably well in practice. The assumption's value lies in its dramatic simplification of the likelihood model.

3.2.3 Mathematical Derivation

To understand how conditional independence simplifies our model, consider the case where $X = \langle X_1, X_2 \rangle$. Using probability theory and our independence assumption:

$$\begin{aligned} P(X|Y) &= P(X_1, X_2|Y) && (3.4) \\ &= P(X_1|X_2, Y)P(X_2|Y) && (3.5) \\ &= P(X_1|Y)P(X_2|Y) && (3.6) \end{aligned}$$

The transition from equation (3.4) to (3.5) applies the chain rule of probability, a general property that holds universally. The key step is from (3.5) to (3.6), where we apply our conditional independence assumption: given Y, X_1 does not depend on X_2, so $P(X_1|X_2, Y) = P(X_1|Y)$.
Generalizing to n features under the conditional independence assumption:

$$P(X_1, \ldots, X_n|Y) = \prod_{i=1}^{n} P(X_i|Y) \tag{3.7}$$

This factorization achieves our goal of tractability. For boolean variables Y and X_i, we need only $2n$ parameters to define $P(X_i = x_{ik}|Y = y_j)$ across all necessary combinations of i, j, and k. This represents an exponential reduction from the $2(2^n - 1)$ parameters required without independence assumptions.

3.2.4 Classification Rule and Algorithm

To classify a new instance $X = x$, we seek the class y that maximizes the posterior probability $P(Y = y|X = x)$. Using Bayes rule and our independence assumption:

$$
\begin{aligned}
Y_{\text{NB}} &= \arg\max_{y\in\{0,1\}} P(Y=y|X=x) && (3.8)\\
&= \arg\max_{y\in\{0,1\}} \frac{P(X=x|Y=y)P(Y=y)}{P(X=x)} && (3.9)\\
&= \arg\max_{y\in\{0,1\}} P(X=x|Y=y)P(Y=y) && (3.10)\\
&= \arg\max_{y\in\{0,1\}} P(Y=y)\prod_{i=1}^{n} P(X_i=x_i|Y=y) && (3.11)
\end{aligned}
$$

The transition from (3.9) to (3.10) follows because $P(X = x)$ is constant across classes. The final step (3.11) applies our factorization from equation (3.7).

3.2.5 Learning Algorithm

The Naive Bayes learning algorithm estimates the required probabilities from training data $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$:
Step 1: Estimate Class Priors

$$\hat{P}(Y = y_k) = \frac{\#D\{Y = y_k\}}{|D|} \tag{3.12}$$

Step 2: Estimate Feature Likelihoods
For discrete features:

$$\hat{P}(X_i = x_{ij}|Y = y_k) = \frac{\#D\{X_i = x_{ij} \wedge Y = y_k\}}{\#D\{Y = y_k\}} \tag{3.13}$$

Step 3: Classify New Instances
For a test instance x, compute:

$$Y_{\text{NB}}(x) = \arg\max_{y_k} \hat{P}(Y = y_k)\prod_{i=1}^{n} \hat{P}(X_i = x_i|Y = y_k) \tag{3.14}$$

3.2.6 Smoothing for Sparse Data

A practical challenge arises when a feature value x_i never appears with class y_k in the training data, yielding $\hat{P}(X_i = x_i|Y = y_k) = 0$. This zero probability causes the entire product in equation (3.11) to vanish, regardless of other evidence.
Laplace Smoothing (also called additive smoothing) addresses this issue by adding pseudocounts to all frequency estimates:

$$\hat{P}(X_i = x_{ij}|Y = y_k) = \frac{\#D\{X_i = x_{ij} \wedge Y = y_k\} + \alpha}{\#D\{Y = y_k\} + \alpha|V_i|} \tag{3.15}$$

where $|V_i|$ denotes the number of possible values for feature X_i, and $\alpha > 0$ is the smoothing parameter (commonly $\alpha = 1$). This ensures all probabilities remain positive while minimally distorting the estimates when sufficient data exists.

3.2.7 Naive Bayes as a Linear Classifier

An important theoretical result reveals that Naive Bayes, despite its probabilistic formulation, implements a linear decision boundary in the feature space.
For discrete boolean features, taking the logarithm of the classification rule:

$$\log P(Y=1|X) - \log P(Y=0|X) = \log \frac{P(Y=1)}{P(Y=0)} + \sum_{i=1}^{n} \log \frac{P(X_i|Y=1)}{P(X_i|Y=0)} \tag{3.16}$$

This expression is linear in the features X_i, meaning the decision boundary (where the difference equals zero) forms a hyperplane.
For Gaussian Naive Bayes with class-independent variances (i.e., $\sigma_{i1} = \sigma_{i0} = \sigma_i$ for all i), similar algebra shows:

$$\log \frac{P(Y=1|X)}{P(Y=0|X)} = w_0 + \sum_{i=1}^{n} w_i X_i \tag{3.17}$$

where the weights w_i depend on the differences in class means. This linear structure has profound implications for understanding Naive Bayes's relationship to other algorithms, particularly Logistic Regression.

3.3 Logistic Regression

While Naive Bayes takes a generative approach by modeling $P(X|Y)$, Logistic Regression directly models the conditional distribution $P(Y|X)$, earning its classification as a *discriminative* algorithm. This distinction leads to different learning characteristics and performance profiles.

3.3.1 The Logistic Function

Logistic Regression models the posterior probability using the logistic (sigmoid) function:

$$P(Y=1|X) = \frac{1}{1+\exp(-w_0 - \sum_{i=1}^{n} w_i X_i)} = \frac{1}{1+\exp(-w^T x)} \tag{3.18}$$

where $w = [w_0, w_1, \ldots, w_n]^T$ represents the weight vector (including the intercept term w_0), and we augment the feature vector as $x = [1, X_1, \ldots, X_n]^T$ to incorporate the intercept.
The logistic function has several advantageous properties:

- **Bounded Output:** Maps any real-valued input to the interval $(0, 1)$, naturally interpreted as a probability.
- **Smooth and Differentiable:** Enables gradient-based optimization.
- **S-shaped Curve:** Provides a smooth transition between classes, with steepness controlled by weight magnitudes.

The complementary probability follows:

$$P(Y = 0|X) = \frac{\exp(-w^T x)}{1 + \exp(-w^T x)} = \frac{1}{1 + \exp(w^T x)} \tag{3.19}$$

3.3.2 Maximum Likelihood Estimation

To learn the weight vector w, we employ maximum likelihood estimation. Given training data $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, the likelihood function is:

$$L(w) = \prod_{j=1}^{m} P(Y = y^{(j)}|X = x^{(j)}, w) \tag{3.20}$$

Taking the negative log-likelihood (which we aim to minimize) yields the cross-entropy loss:

$$\begin{aligned} \ell(w) &= -\log L(w) && (3.21) \\ &= -\sum_{j=1}^{m} \left[y^{(j)} \log P(Y = 1|x^{(j)}, w) + (1 - y^{(j)}) \log P(Y = 0|x^{(j)}, w) \right] && (3.22) \end{aligned}$$

Substituting the logistic function:

$$\ell(w) = \sum_{j=1}^{m} \left[y^{(j)}(w^T x^{(j)}) - \log(1 + \exp(w^T x^{(j)})) \right] \tag{3.23}$$

Unlike linear regression, this objective function lacks a closed-form solution, necessitating iterative optimization methods.

3.3.3 Gradient Descent Optimization

The gradient of the log-likelihood with respect to w is:

$$\nabla_w \ell(w) = \sum_{j=1}^{m} (P(Y = 1|x^{(j)}, w) - y^{(j)}) x^{(j)} \tag{3.24}$$

This elegant form reveals that the gradient equals the sum of prediction errors weighted by features, analogous to linear regression's gradient.
Batch Gradient Descent: Update weights using all training examples:

$$w \leftarrow w - \eta \nabla_w \ell(w) \tag{3.25}$$

where $\eta > 0$ is the learning rate.
Stochastic Gradient Descent (SGD): Update weights using one randomly selected example $(x^{(j)}, y^{(j)})$ at a time:

$$w \leftarrow w - \eta (P(Y = 1|x^{(j)}, w) - y^{(j)}) x^{(j)} \tag{3.26}$$

SGD often converges faster in practice, especially for large datasets, though it produces noisier weight trajectories.

3.3.4 Regularization

To prevent overfitting, especially in high-dimensional spaces or with limited data, we commonly add a regularization term to the objective:

$$\ell_{\text{reg}}(w) = \ell(w) + \lambda\|w\|^2 \tag{3.27}$$

where $\lambda > 0$ controls the regularization strength. This L2 regularization (also called ridge regularization) penalizes large weights, encouraging simpler models that generalize better. The modified gradient becomes:

$$\nabla_w \ell_{\text{reg}}(w) = \nabla_w \ell(w) + 2\lambda w \tag{3.28}$$

Alternatively, L1 regularization ($\lambda \sum_i |w_i|$) promotes sparse weight vectors, effectively performing feature selection.

3.3.5 Relationship to Generalized Linear Models

Logistic Regression belongs to the family of Generalized Linear Models (GLMs), which extend linear regression to non-Gaussian response distributions. GLMs consist of three components:

1. **Random Component:** Specifies the probability distribution of the response (here, Bernoulli for binary classification).

2. **Systematic Component:** A linear predictor $\eta = w^T x$.

3. **Link Function:** Relates the expected value of the response to the linear predictor. For Logistic Regression, the logit link function is $\eta = \log \frac{P(Y=1|X)}{P(Y=0|X)}$.

This framework unifies various regression models and provides theoretical foundations for understanding Logistic Regression's properties.

3.4 Comparing Naive Bayes and Logistic Regression

Though Naive Bayes and Logistic Regression can both produce linear decision boundaries, they differ fundamentally in their learning approaches, assumptions, and practical characteristics. Understanding these differences guides algorithm selection for specific applications.

3.4.1 Generative versus Discriminative Learning

Naive Bayes (Generative):

- Models the joint distribution $P(X, Y) = P(X|Y)P(Y)$
- Learns how data is generated from each class
- Makes explicit assumptions about feature distributions
- Can generate synthetic data samples from learned distributions
- Requires estimating more parameters than strictly necessary for classification

Logistic Regression (Discriminative):

- Directly models the conditional distribution $P(Y|X)$
- Focuses exclusively on the decision boundary
- Makes no assumptions about feature distributions
- Cannot generate synthetic samples
- Estimates only parameters directly relevant to classification

As **?** demonstrate, this fundamental difference affects both asymptotic performance and learning curves.

3.4.2 Connection and Asymptotic Equivalence

A remarkable theoretical result establishes that Gaussian Naive Bayes (with class-independent variances) and Logistic Regression make identical predictions in the infinite data limit, provided the Naive Bayes assumptions hold [**?**].
Specifically, both models produce decision boundaries of the form:

$$P(Y = 1|X) = \frac{1}{1 + \exp(w_0 + \sum_{i=1}^{n} w_i X_i)} \tag{3.29}$$

However, the weights are learned differently:

- **Naive Bayes:** Derives weights analytically from class-conditional means and variances
- **Logistic Regression:** Learns weights by directly maximizing conditional likelihood

When Naive Bayes assumptions are valid, both approaches converge to the same optimal decision boundary as training data increases. When these assumptions are violated, Logistic Regression typically achieves superior asymptotic accuracy.

3.4.3 Bias-Variance Tradeoff

The performance difference between these algorithms reflects a fundamental tradeoff between bias and variance:
Naive Bayes:

- **Higher Bias:** Strong independence assumptions may not reflect reality
- **Lower Variance:** Fewer parameters reduce sensitivity to training sample variation
- **Implication:** Performs well with limited data when assumptions are approximately correct

Logistic Regression:

- **Lower Bias:** Makes minimal assumptions about feature relationships
- **Higher Variance:** More flexible model requires more data to stabilize
- **Implication:** Performs well with abundant data, especially when Naive Bayes assumptions fail

3.4.4 Convergence Rates

? prove that the two algorithms converge to their asymptotic accuracies at different rates:

- **Naive Bayes:** Parameter estimates converge in $O(\log n)$ examples, where n is the feature dimensionality. This logarithmic rate makes Naive Bayes highly sample-efficient.
- **Logistic Regression:** Parameter estimates converge in $O(n)$ examples, requiring linearly more data as dimensionality increases.

This theoretical analysis suggests a practical guideline: Naive Bayes often outperforms Logistic Regression in small-sample regimes, while Logistic Regression tends to excel given sufficient training data.

3.4.5 Practical Considerations

Several practical factors influence algorithm selection:
When to Prefer Naive Bayes:

- Limited training data available
- Features are genuinely close to conditionally independent
- Computational resources are constrained (simpler training)
- Need for probabilistic interpretation and uncertainty quantification

- Real-time prediction requirements (faster evaluation)

When to Prefer Logistic Regression:

- Abundant training data available
- Features exhibit significant correlations
- Maximum accuracy is paramount
- Feature importance interpretation is desired (through weight magnitudes)
- Regularization is needed to control overfitting

Computational Complexity:

- **Training:** Naive Bayes requires $O(mn)$ time (single pass through data), while Logistic Regression requires $O(kmn)$ for k gradient descent iterations.
- **Prediction:** Both algorithms classify in $O(n)$ time per instance.

3.4.6 Empirical Performance

Experimental studies across diverse datasets reveal nuanced performance patterns:

- In text classification (with high-dimensional sparse features), Naive Bayes often performs surprisingly well despite feature correlations.
- In image classification and other domains with strong feature dependencies, Logistic Regression typically dominates.
- Ensemble methods sometimes combine both approaches to leverage their complementary strengths.

3.5 Statistical Perspectives on Learning

Understanding supervised learning through a statistical lens provides deeper insights into algorithm behavior and performance characteristics.

3.5.1 Learning as Statistical Estimation

Function approximation algorithms can be viewed as statistical estimators of $P(Y|X)$. This perspective enables us to:

- Quantify uncertainty in predictions
- Derive confidence intervals for parameters
- Understand how performance scales with sample size
- Compare algorithms using established statistical theory

3.5.2 Bias and Variance Decomposition

For any learning algorithm, we can decompose the expected prediction error into three components:

$$\text{Expected Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error} \tag{3.30}$$

where:

- **Bias:** Error from incorrect modeling assumptions
- **Variance:** Error from sensitivity to training data fluctuations
- **Irreducible Error:** Fundamental noise in the data generation process

This decomposition explains why no single algorithm dominates all problems—the optimal choice depends on which error component dominates for the specific task.

3.5.3 Sample Complexity Considerations

The number of training examples required for reliable learning depends on several factors:

- **Model Complexity:** More parameters require more data
- **Feature Dimensionality:** Higher dimensions increase sample requirements
- **Noise Level:** More noise demands more data for robust estimation
- **Desired Accuracy:** Achieving higher accuracy requires exponentially more data

For Naive Bayes with n features and k classes, we need approximately $O(nk)$ examples. For full Bayesian classification without independence assumptions, we need $O(k \cdot 2^n)$ examples—an exponential difference that underscores the value of simplifying assumptions.

3.6 Advanced Topics and Extensions

3.6.1 Semi-Naive Bayes Models

Recognizing that full independence is unrealistic while full dependence is intractable, researchers have developed intermediate models:

- **Tree Augmented Naive Bayes (TAN):** Allows each feature to depend on at most one other feature besides the class
- **Bayesian Networks:** General graphical models encoding arbitrary independence structures
- **Feature Selection:** Remove redundant or dependent features before applying Naive Bayes

These approaches often achieve better accuracy than pure Naive Bayes while maintaining computational tractability.

3.6.2 Multi-Class Extensions

Both Naive Bayes and Logistic Regression extend naturally to multi-class problems:
Multi-Class Naive Bayes: Simply compute $P(Y = y_k|X)$ for all classes k and select the maximum.
Multi-Class Logistic Regression (Softmax Regression): Generalize the logistic function to:

$$P(Y = k|X) = \frac{\exp(w_k^T x)}{\sum_{j=1}^{K} \exp(w_j^T x)} \tag{3.31}$$

where K is the number of classes and each class has its own weight vector w_k.

3.6.3 Handling Missing Data

Both algorithms can handle missing features during prediction:

- **Naive Bayes:** Simply omit the likelihood term for missing features (implicitly assuming they provide no information)
- **Logistic Regression:** Requires imputation strategies (mean substitution, model-based imputation, or multiple imputation)

3.6.4 Online Learning

Both algorithms support online learning, where the model updates incrementally as new data arrives:

- **Naive Bayes:** Update sufficient statistics (counts) incrementally
- **Logistic Regression:** Use stochastic gradient descent with decaying learning rates

This capability is valuable for applications with streaming data or concept drift.

3.7 Summary and Key Insights

This chapter has explored two fundamental approaches to supervised classification, revealing deep connections between probabilistic modeling and practical algorithm design.

3.7.1 Core Principles

- **Bayesian Foundation:** Bayes rule provides a principled framework for designing learning algorithms. Given the goal of learning $P(Y|X)$, we can estimate $P(X|Y)$ and $P(Y)$ from training data, then apply Bayes rule for classification. This approach characterizes *generative classifiers*.

- **Necessity of Simplifying Assumptions:** Without prior assumptions, learning Bayesian classifiers requires impractical amounts of training data (exponential in feature dimensionality). The Naive Bayes conditional independence assumption reduces parameter complexity from exponential to linear, enabling practical learning while often maintaining good performance.
- **Linear Decision Boundaries:** Despite their probabilistic formulations, both Naive Bayes (for discrete features or Gaussian features with class-independent variances) and Logistic Regression implement linear classifiers, defining hyperplane decision surfaces in feature space.
- **Discriminative Alternative:** Logistic Regression directly estimates $P(Y|X)$ rather than modeling $P(X|Y)$, making it a *discriminative* classifier. This approach avoids assumptions about feature distributions and focuses computational resources entirely on the classification boundary.
- **Asymptotic Equivalence with Practical Differences:** Under the Naive Bayes assumptions, Gaussian Naive Bayes and Logistic Regression converge to identical classifiers given infinite data. However, their different biases mean that if the assumptions fail, Logistic Regression achieves superior asymptotic performance.
- **Bias-Variance Tradeoff:** Naive Bayes exhibits higher bias but lower variance than Logistic Regression. When the conditional independence assumption is approximately correct, Naive Bayes's bias is appropriate and it outperforms Logistic Regression, especially with limited data. When the assumption is violated, Logistic Regression's flexibility becomes advantageous.
- **Statistical Perspective:** Viewing learning algorithms as statistical estimators provides valuable insights into their behavior, convergence properties, and sample complexity requirements. This perspective enables principled algorithm selection and performance analysis.

3.7.2 Practical Guidelines

- Choose Naive Bayes for rapid prototyping, limited data scenarios, or when computational efficiency is critical
- Choose Logistic Regression when accuracy is paramount and sufficient training data is available
- Consider regularization for both algorithms when overfitting is a concern
- Validate independence assumptions empirically before trusting Naive Bayes on critical applications
- Use cross-validation to compare algorithms on your specific dataset rather than relying solely on theoretical considerations

Chapter 4

Linear Regression with Gradient Descent Optimization

Linear regression is one of the most fundamental and widely used techniques in statistical modeling and machine learning. It provides a simple yet powerful framework for understanding and predicting relationships between variables. This chapter explores linear regression from both statistical and optimization perspectives, with particular emphasis on gradient descent as an efficient method for finding optimal model parameters.
The beauty of linear regression lies in its interpretability and mathematical tractability. Despite its simplicity, it serves as the foundation for many advanced machine learning algorithms and provides crucial insights into the principles of supervised learning. The linear relationship assumption, while restrictive, is often a reasonable approximation for many real-world phenomena, especially when the goal is to understand the average behavior of a system rather than capture every nuance.
Understanding how to optimize linear regression models using gradient descent is essential for anyone working in data science, machine learning, or statistical modeling. Gradient descent is not only applicable to linear regression but forms the backbone of training deep neural networks and other complex models. By mastering these concepts in the context of linear regression, you build intuition that transfers directly to more sophisticated machine learning techniques.
Linear regression addresses the fundamental problem of supervised learning: given a set of input-output pairs, can we learn a function that maps inputs to outputs? When the assumed function is linear, the problem becomes tractable both analytically and computationally. The method has applications spanning economics, biology, engineering, social sciences, and virtually every domain where quantitative analysis is performed.

4.0.1 Applications in Modern Data Science

Linear regression finds applications in numerous domains:

- **Economics and Finance**: Modeling relationships between economic indicators (GDP, inflation, unemployment), predicting stock prices, and analyzing market trends

- **Healthcare and Medicine**: Predicting patient outcomes based on clinical measurements, analyzing dose-response relationships, and studying disease progression
- **Marketing and Business**: Forecasting sales based on advertising expenditure, customer segmentation, and market analysis
- **Engineering**: Calibration of instruments, quality control, and predictive maintenance
- **Social Sciences**: Analyzing survey data, studying demographic trends, and understanding social phenomena
- **Environmental Science**: Climate modeling, pollution analysis, and resource management

The versatility of linear regression makes it an indispensable tool in the modern data scientist's toolkit. Moreover, many complex non-linear relationships can be approximated or transformed into linear ones through feature engineering, making linear regression applicable to an even broader range of problems.

4.1 Prerequisites: Correlation and Variable Dependency

4.1.1 Understanding Variable Relationships

Before delving into linear regression, it is crucial to understand the relationships between variables. In correlation analysis, two variables are treated equally to measure the strength and direction of their relationship. This symmetric treatment helps us understand whether two variables move together, but it doesn't imply causation or directionality.
In regression, however, we make an explicit distinction: one variable is considered the independent (predictor) variable, denoted as x, and the other is the dependent (outcome) variable, denoted as y. This asymmetric treatment reflects our interest in understanding how changes in x affect y, which is essential for prediction and decision-making.

4.1.2 Covariance: Measuring Joint Variability

Covariance quantifies how two variables vary together. It is calculated as:

$$\text{Cov}(x, y) = \frac{\sum_{i=1}^{n}(x_i - \bar{X})(y_i - \bar{Y})}{n-1} \tag{4.1}$$

where $\bar{X}$ and $\bar{Y}$ are the sample means of x and y respectively. The numerator represents the sum of products of deviations from the mean. When both variables deviate from their means in the same direction (both above or both below), the product is positive; when they deviate in opposite directions, the product is negative.
The standard deviation, which measures the spread of a single variable, is defined as:

$$\sigma_x = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{X})^2} \tag{4.2}$$

4.1.3 Pearson's Correlation Coefficient

While covariance indicates the direction of a linear relationship, its magnitude depends on the units of measurement, making it difficult to interpret. The Pearson correlation coefficient addresses this limitation by standardizing the covariance:

$$r = \frac{\mathrm{Cov}(x, y)}{\sigma_x \sigma_y} \tag{4.3}$$

This standardization yields a dimensionless quantity with several important properties:

- Values always range between -1 and $+1$
- $r = 1$ indicates perfect positive linear relationship
- $r = -1$ indicates perfect negative linear relationship
- $r = 0$ indicates no linear relationship
- Values closer to ± 1 indicate stronger linear relationships
- The correlation coefficient is unit-free, allowing comparison across different measurement scales

4.1.4 Interpreting Covariance and Correlation

The sign and magnitude of covariance and correlation provide insights into variable relationships:

$$\mathrm{Cov}(X, Y) > 0 \quad \Rightarrow \quad X \text{ and } Y \text{ tend to increase together (positive association)} \tag{4.4}$$
$$\mathrm{Cov}(X, Y) < 0 \quad \Rightarrow \quad X \text{ and } Y \text{ tend to move in opposite directions (negative association)} \tag{4.5}$$
$$\mathrm{Cov}(X, Y) = 0 \quad \Rightarrow \quad \text{No linear association between } X \text{ and } Y \tag{4.6}$$

It's important to note that zero covariance (or correlation) does not necessarily imply independence in general, though for normally distributed variables, zero correlation does imply independence. Moreover, correlation measures only linear relationships; variables may have strong non-linear relationships while having near-zero correlation.

4.1.5 Importance for Linear Regression

Understanding correlation and covariance is essential for linear regression because:

1. They help identify which variables might be good predictors
2. Strong correlation between predictor and outcome suggests a predictable linear relationship
3. Correlation between predictors (multicollinearity) can cause problems in multiple regression
4. They provide a baseline for comparing model performance

4.2 Linear Regression Fundamentals

4.2.1 What is Linear Regression?

Linear regression is a statistical method that models the relationship between variables by fitting a linear equation to observed data. The fundamental assumption is that there exists a linear relationship between the independent variable(s) and the dependent variable. The goal is to find the best-fitting straight line (or hyperplane in higher dimensions) through the data points.

The method seeks the optimal line that minimizes the sum of squared differences between predicted and actual values. This criterion, known as the least squares criterion, has several desirable statistical properties and yields a unique solution under standard conditions.

4.2.2 Distinction from Correlation

While both correlation and regression deal with relationships between variables, they serve different purposes:

- **Correlation**: Measures the strength and direction of association between two variables, treating them symmetrically. It answers: "How strongly are these variables related?"
- **Regression**: Models how one variable depends on another, with asymmetric treatment. It answers: "How does the outcome change when the predictor changes?" and "What value should I predict for a new observation?"

Regression provides not just a measure of association but a predictive equation that can be used for forecasting and understanding the nature of the relationship.

4.2.3 The Linear Model Equation

Recall from basic algebra the equation of a straight line:

$$y = mx + c \tag{4.7}$$

where:

- y is the dependent variable (vertical axis)
- x is the independent variable (horizontal axis)
- m is the slope (rate of change)
- c is the y-intercept (value when $x = 0$)

In statistical notation, we write this as the linear regression model.

4.3 Simple Linear Regression

4.3.1 Model Definition

In simple linear regression, there is one independent variable and one dependent variable. The model estimates the slope and y-intercept of the line of best fit, which represents the relationship between the variables.
The simple linear regression model is expressed as:

$$y = h_\theta(x) = \theta_0 + \theta_1 x \tag{4.8}$$

where:

- y is the predicted value of the dependent variable
- x is the independent variable (feature)
- θ_0 is the y-intercept (bias term)
- θ_1 is the slope coefficient (weight)
- $h_\theta(x)$ denotes the hypothesis function parameterized by $\boldsymbol{\theta} = [\theta_0, \theta_1]^T$

4.3.2 Interpretation of Parameters

The parameters have intuitive interpretations:

- θ_0 **(Intercept)**: The expected value of y when $x = 0$. It represents the baseline value before considering the effect of the predictor.
- θ_1 **(Slope)**: The expected change in y for a one-unit increase in x. It quantifies the strength and direction of the relationship.

For example, in a model predicting house prices from area: if $\theta_0 = 50000$ and $\theta_1 = 100$, this means a house with zero area would theoretically cost \$50,000 (the intercept may not always have practical meaning), and each additional square foot adds \$100 to the price.

4.3.3 Residuals and Prediction Errors

In practice, observed data points rarely fall exactly on a straight line. The difference between the observed value and the predicted value is called the residual or error:

$$\epsilon_i = y_i - \hat{y}_i = y_i - (\theta_0 + \theta_1 x_i) \tag{4.9}$$

where:

- y_i is the observed (actual) value
- $\hat{y}_i$ or $y_{\text{predicted}}$ is the predicted value
- ϵ_i is the residual for observation i

Residuals capture the deviation of individual observations from the fitted line. They contain information about:

- How well the model fits the data
- Whether the linear assumption is appropriate
- Presence of outliers or influential observations
- Violations of model assumptions

4.3.4 The Least Squares Criterion

The best-fitting line is determined by the least squares criterion, which minimizes the sum of squared residuals. Why squared residuals? Several reasons:

1. Squaring makes all errors positive, preventing positive and negative errors from canceling out
2. Squaring penalizes larger errors more heavily than smaller ones, making the method sensitive to outliers
3. Squared error functions have nice mathematical properties (differentiable, convex)
4. Under certain statistical assumptions, least squares estimates have optimal properties (they are unbiased and have minimum variance among linear unbiased estimators)

4.3.5 The Cost Function

The cost function (also called loss function or objective function) quantifies how well a given set of parameters fits the data. For linear regression, we use the Mean Squared Error (MSE):

$$J(\theta_0, \theta_1) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i)^2 \tag{4.10}$$

where:

- J is the cost function (a scalar value)
- n is the number of training examples
- The summation runs over all observations in the dataset

The MSE has several interpretations:

- It's the average squared distance from points to the line
- It's related to the variance of residuals
- Lower MSE indicates better fit to the training data

Sometimes a factor of $\frac{1}{2}$ is added to simplify derivatives: $J = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$, but this doesn't affect the optimization since we're looking for the minimum.

4.3.6 Optimization Objective

The fundamental goal in linear regression is to find parameter values that minimize the cost function:

$$\boldsymbol{\theta}^* = \arg\min_{\theta_0, \theta_1} J(\theta_0, \theta_1) \tag{4.11}$$

This is an optimization problem. For linear regression with MSE cost, the problem is convex, meaning:

- There is a unique global minimum (no local minima)
- Any local minimum is also the global minimum
- Gradient descent is guaranteed to converge to the optimal solution (with appropriate learning rate)

There are two main approaches to solving this optimization problem:

1. **Analytical solution (Normal Equation)**: Directly solve for the parameters using calculus and linear algebra. This gives the exact solution in one step but becomes computationally expensive for large datasets.

2. **Iterative solution (Gradient Descent)**: Start with initial parameter values and iteratively improve them. This is more scalable and generalizes to more complex models.

This chapter focuses on the gradient descent approach, as it provides insights applicable to a much broader class of machine learning problems.

4.4 Gradient Descent Optimization

Gradient Descent is one of the most fundamental optimization algorithms in machine learning. The key idea is beautifully simple: to find the minimum of a function, we repeatedly take steps in the direction of steepest descent (negative gradient).
Imagine standing on a mountainside in thick fog, trying to reach the valley below. Without being able to see far ahead, a reasonable strategy is to:

1. Look around your immediate vicinity
2. Determine which direction slopes downward most steeply
3. Take a step in that direction
4. Repeat until you reach relatively flat ground

This is precisely what gradient descent does mathematically. The "direction of steepest descent" is given by the negative gradient of the cost function.

4.4.1 Why Gradient Descent?

While linear regression has a closed-form solution (the normal equation), gradient descent is preferred in many situations:

- **Scalability**: Works efficiently with large datasets where computing matrix inverses is expensive
- **Online learning**: Can update parameters as new data arrives
- **Generalizability**: The same algorithm applies to non-convex problems and complex models (neural networks)
- **Memory efficiency**: Doesn't require storing large matrices
- **Flexibility**: Easily handles regularization and constraints

4.4.2 Computing Gradients

For a function $J(\theta_0, \theta_1)$ of two variables, the gradient is a vector of partial derivatives:

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \end{bmatrix} \tag{4.12}$$

The gradient points in the direction of greatest increase of the function. Therefore, the negative gradient points in the direction of greatest decrease, which is where we want to go to minimize the cost function.

To update θ_0 and θ_1, we compute the partial derivatives (gradients) of the cost function with respect to each parameter. The gradient tells us the direction and rate of steepest increase of the cost function. By moving in the opposite direction (negative gradient), we can reduce the cost.

Given the cost function:

$$J = \frac{1}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i)^2 \tag{4.13}$$

Let us derive the partial derivative with respect to θ_0. Using the chain rule:

$$\begin{align}
\frac{\partial J}{\partial \theta_0} &= \frac{\partial}{\partial \theta_0}\left[\frac{1}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i)^2\right] \tag{4.14}\\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial \theta_0}\left[(\theta_0 + \theta_1 x_i - y_i)^2\right] \tag{4.15}\\
&= \frac{1}{n}\sum_{i=1}^{n}2(\theta_0 + \theta_1 x_i - y_i)\cdot\frac{\partial}{\partial \theta_0}[\theta_0 + \theta_1 x_i - y_i] \tag{4.16}\\
&= \frac{1}{n}\sum_{i=1}^{n}2(\theta_0 + \theta_1 x_i - y_i)\cdot 1 \tag{4.17}\\
&= \frac{2}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i) \tag{4.18}
\end{align}$$

Similarly, for θ_1:

$$\frac{\partial J}{\partial \theta_1} = \frac{\partial}{\partial \theta_1}\left[\frac{1}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i)^2\right] \tag{4.19}$$

$$= \frac{1}{n}\sum_{i=1}^{n} 2(\theta_0 + \theta_1 x_i - y_i) \cdot \frac{\partial}{\partial \theta_1}[\theta_0 + \theta_1 x_i - y_i] \tag{4.20}$$

$$= \frac{1}{n}\sum_{i=1}^{n} 2(\theta_0 + \theta_1 x_i - y_i) \cdot x_i \tag{4.21}$$

$$= \frac{2}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i) \cdot x_i \tag{4.22}$$

The key difference is that the gradient with respect to θ_1 includes the x_i term because θ_1 is multiplied by x_i in the prediction function.
The partial derivatives can be summarized as:

$$\frac{\partial J}{\partial \theta_0} = \frac{2}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i) \tag{4.23}$$

$$\frac{\partial J}{\partial \theta_1} = \frac{2}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i) \cdot x_i \tag{4.24}$$

These gradients tell us how the cost function changes with respect to each parameter. Positive gradient means the cost increases as the parameter increases; negative gradient means the cost decreases as the parameter increases.

4.4.3 Parameter Update Rules

The basic gradient descent update rule moves parameters in the opposite direction of the gradient:

$$\theta_j := \theta_j - \frac{\partial J}{\partial \theta_j} \tag{4.25}$$

For our linear regression problem:

$$\theta_0 := \theta_0 - \frac{2}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i) \tag{4.26}$$

$$\theta_1 := \theta_1 - \frac{2}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i) \cdot x_i \tag{4.27}$$

These updates are performed simultaneously for all parameters in each iteration.

4.4.4 The Learning Rate

In practice, the gradient magnitude can be too large, causing the parameters to overshoot the minimum and possibly diverge. To control this, we introduce a learning rate $\alpha > 0$:

$$\theta_0 := \theta_0 - \alpha\frac{2}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i) \tag{4.28}$$

$$\theta_1 := \theta_1 - \alpha\frac{2}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i) \cdot x_i \tag{4.29}$$

The learning rate α is a hyperparameter that controls the step size:

- **Too large**: The algorithm may overshoot the minimum, oscillate, or even diverge
- **Too small**: The algorithm will converge very slowly, requiring many iterations
- **Just right**: Fast convergence to the optimal solution

Choosing the Learning Rate

Common strategies for selecting the learning rate include:

1. **Fixed learning rate**: Use a constant value (e.g., 0.01, 0.001). The most commonly used rates are: $0.001, 0.003, 0.01, 0.03, 0.1, 0.3$
2. **Learning rate schedule**: Start with a larger value and gradually decrease it
3. **Adaptive methods**: Algorithms like Adam, RMSprop adjust the learning rate automatically
4. **Grid search**: Try multiple values and select based on performance on validation data

With a large learning rate, we can cover more ground quickly, but we risk overshooting the minimum since the slope of the cost surface is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we recalculate it frequently, but the computational cost increases significantly.

4.4.5 The Gradient Descent Algorithm

The complete batch gradient descent algorithm can be summarized as follows:

Algorithm 4 Batch Gradient Descent for Linear Regression

```
Initialize $\theta_0, \theta_1$ randomly or to zero
Set learning rate $\alpha$
Set convergence threshold $\epsilon$ and maximum iterations max_iter
iter $\leftarrow 0$
repeat
    $\text{grad}_0 \leftarrow \frac{2}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i)$
    $\text{grad}_1 \leftarrow \frac{2}{n}\sum_{i=1}^{n}(\theta_0 + \theta_1 x_i - y_i) \cdot x_i$
    $\theta_0 \leftarrow \theta_0 - \alpha \cdot \text{grad}_0$
    $\theta_1 \leftarrow \theta_1 - \alpha \cdot \text{grad}_1$
    Compute $J(\theta_0, \theta_1)$
    iter $\leftarrow$ iter + 1
until $|J^{(\text{iter})} - J^{(\text{iter}-1)}| < \epsilon$ or iter $\geq$ max_iter
return $\theta_0, \theta_1$
```

4.4.6 Convergence Properties

For linear regression with MSE cost function, gradient descent has strong convergence guarantees:

- The cost function is convex (bowl-shaped)
- There exists a unique global minimum
- With appropriate learning rate, gradient descent will converge to this minimum
- The convergence rate is linear (error decreases proportionally to number of iterations)

Monitoring the cost function over iterations helps diagnose convergence:

- **Healthy convergence**: Cost decreases monotonically and plateaus
- **Oscillation**: Cost bounces up and down (learning rate too large)
- **Divergence**: Cost increases (learning rate too large)
- **Slow convergence**: Cost decreases very slowly (learning rate too small or poor feature scaling)

4.5 Detailed Worked Example with Three Iterations

Let us work through a complete example to illustrate the gradient descent process in detail. We will perform three full iterations to see how the parameters evolve and the cost function decreases.

4.5.1 The Dataset

Consider the following dataset with 10 observations representing the relationship between an independent variable x and a dependent variable y:

x	y
5	11
3	7
4	9
2	5
6	13
1	3
8	17
7	9
9	19
10	21

Table 4.1: Training dataset for linear regression example

Looking at this data, we can observe that y generally increases as x increases, suggesting a positive linear relationship. Our goal is to find the best-fitting line $y = \theta_0 + \theta_1 x$ using gradient descent.

4.5.2 Initial Setup

Hyperparameters:

- Initial parameters: $\theta_0^{(0)} = 1$ and $\theta_1^{(0)} = 0.5$
- Learning rate: $\alpha = 0.01$ (we'll use a small value for stability)
- Number of observations: $n = 10$

4.5.3 Iteration 1

Step 1.1: Compute Predictions and Errors

With $\theta_0 = 1$ and $\theta_1 = 0.5$, we compute predictions for each data point:

Step 1.2: Compute Cost Function

$$J^{(0)}(\theta_0, \theta_1) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{10}(776.25) = 77.625 \tag{4.30}$$

This high cost indicates that our initial parameter values result in poor predictions.

i	x_i	y_i	$\hat{y}_i$	$(\hat{y}_i - y_i)$	$(\hat{y}_i - y_i)^2$	$(\hat{y}_i - y_i) \cdot x_i$
1	5	11	3.5	-7.5	56.25	-37.5
2	3	7	2.5	-4.5	20.25	-13.5
3	4	9	3.0	-6.0	36.00	-24.0
4	2	5	2.0	-3.0	9.00	-6.0
5	6	13	4.0	-9.0	81.00	-54.0
6	1	3	1.5	-1.5	2.25	-1.5
7	8	17	5.0	-12.0	144.00	-96.0
8	7	9	4.5	-4.5	20.25	-31.5
9	9	19	5.5	-13.5	182.25	-121.5
10	10	21	6.0	-15.0	225.00	-150.0
Sums:				-76.5	776.25	-535.5

Table 4.2: Calculations for Iteration 1

Step 1.3: Compute Gradients

$$\frac{\partial J}{\partial \theta_0} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) = \frac{2}{10}(-76.5) = -15.3 \tag{4.31}$$

$$\frac{\partial J}{\partial \theta_1} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) \cdot x_i = \frac{2}{10}(-535.5) = -107.1 \tag{4.32}$$

Both gradients are negative, indicating that increasing both θ_0 and θ_1 will decrease the cost.

Step 1.4: Update Parameters

$$\theta_0^{(1)} = \theta_0^{(0)} - \alpha \cdot \frac{\partial J}{\partial \theta_0} = 1 - 0.01 \times (-15.3) = 1 + 0.153 = 1.153 \tag{4.33}$$

$$\theta_1^{(1)} = \theta_1^{(0)} - \alpha \cdot \frac{\partial J}{\partial \theta_1} = 0.5 - 0.01 \times (-107.1) = 0.5 + 1.071 = 1.571 \tag{4.34}$$

Summary of Iteration 1:

- Starting: $\theta_0 = 1.000$, $\theta_1 = 0.500$, $J = 77.625$
- Ending: $\theta_0 = 1.153$, $\theta_1 = 1.571$

4.5.4 Iteration 2

Step 2.1: Compute Predictions and Errors

With $\theta_0 = 1.153$ and $\theta_1 = 1.571$, we compute new predictions:

i	x_i	y_i	$\hat{y}_i$	$(\hat{y}_i - y_i)$	$(\hat{y}_i - y_i)^2$	$(\hat{y}_i - y_i) \cdot x_i$
1	5	11	9.008	-1.992	3.968	-9.960
2	3	7	5.866	-1.134	1.286	-3.402
3	4	9	7.437	-1.563	2.443	-6.252
4	2	5	4.295	-0.705	0.497	-1.410
5	6	13	10.579	-2.421	5.861	-14.526
6	1	3	2.724	-0.276	0.076	-0.276
7	8	17	13.721	-3.279	10.752	-26.232
8	7	9	12.150	3.150	9.923	22.050
9	9	19	15.292	-3.708	13.749	-33.372
10	10	21	16.863	-4.137	17.114	-41.370
Sums:				-15.065	65.669	-114.750

Table 4.3: Calculations for Iteration 2

Step 2.2: Compute Cost Function

$$J^{(1)}(\theta_0, \theta_1) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{10}(65.669) = 6.567 \tag{4.35}$$

Notice that the cost has decreased dramatically from 77.625 to 6.567! This indicates that our parameter updates moved us significantly closer to the optimal solution.

Step 2.3: Compute Gradients

$$\frac{\partial J}{\partial \theta_0} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) = \frac{2}{10}(-15.065) = -3.013 \tag{4.36}$$

$$\frac{\partial J}{\partial \theta_1} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) \cdot x_i = \frac{2}{10}(-114.750) = -22.950 \tag{4.37}$$

The gradients have decreased in magnitude (closer to zero), which is expected as we approach the minimum.

Step 2.4: Update Parameters

$$\theta_0^{(2)} = \theta_0^{(1)} - \alpha \cdot \frac{\partial J}{\partial \theta_0} = 1.153 - 0.01 \times (-3.013) = 1.153 + 0.030 = 1.183 \tag{4.38}$$

$$\theta_1^{(2)} = \theta_1^{(1)} - \alpha \cdot \frac{\partial J}{\partial \theta_1} = 1.571 - 0.01 \times (-22.950) = 1.571 + 0.230 = 1.801 \tag{4.39}$$

Summary of Iteration 2:

- Starting: $\theta_0 = 1.153$, $\theta_1 = 1.571$, $J = 6.567$
- Ending: $\theta_0 = 1.183$, $\theta_1 = 1.801$

4.5.5 Iteration 3

Step 3.1: Compute Predictions and Errors

With $\theta_0 = 1.183$ and $\theta_1 = 1.801$, we compute new predictions:

i	x_i	y_i	$\hat{y}_i$	$(\hat{y}_i - y_i)$	$(\hat{y}_i - y_i)^2$	$(\hat{y}_i - y_i) \cdot x_i$
1	5	11	10.188	-0.812	0.659	-4.060
2	3	7	6.586	-0.414	0.171	-1.242
3	4	9	8.387	-0.613	0.376	-2.452
4	2	5	4.785	-0.215	0.046	-0.430
5	6	13	11.989	-1.011	1.022	-6.066
6	1	3	2.984	-0.016	0.000	-0.016
7	8	17	15.591	-1.409	1.985	-11.272
8	7	9	13.790	4.790	22.944	33.530
9	9	19	17.392	-1.608	2.586	-14.472
10	10	21	19.193	-1.807	3.265	-18.070
Sums:				-3.115	33.054	-24.550

Table 4.4: Calculations for Iteration 3

Step 3.2: Compute Cost Function

$$J^{(2)}(\theta_0, \theta_1) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{10}(33.054) = 3.305 \tag{4.40}$$

The cost continues to decrease (from 6.567 to 3.305), confirming that gradient descent is working correctly.

Step 3.3: Compute Gradients

$$\frac{\partial J}{\partial \theta_0} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) = \frac{2}{10}(-3.115) = -0.623 \tag{4.41}$$

$$\frac{\partial J}{\partial \theta_1} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) \cdot x_i = \frac{2}{10}(-24.550) = -4.910 \tag{4.42}$$

The gradients continue to decrease in magnitude, indicating convergence toward the optimum.

Step 3.4: Update Parameters

$$\theta_0^{(3)} = \theta_0^{(2)} - \alpha \cdot \frac{\partial J}{\partial \theta_0} = 1.183 - 0.01 \times (-0.623) = 1.183 + 0.006 = 1.189 \tag{4.43}$$

$$\theta_1^{(3)} = \theta_1^{(2)} - \alpha \cdot \frac{\partial J}{\partial \theta_1} = 1.801 - 0.01 \times (-4.910) = 1.801 + 0.049 = 1.850 \tag{4.44}$$

Summary of Iteration 3:

- Starting: $\theta_0 = 1.183$, $\theta_1 = 1.801$, $J = 3.305$
- Ending: $\theta_0 = 1.189$, $\theta_1 = 1.850$

4.5.6 Progress Summary

Let's summarize the progress made over these three iterations:

Iteration	$\boldsymbol{\theta_0}$	$\boldsymbol{\theta_1}$	Cost $\boldsymbol{J}$
0 (Initial)	1.000	0.500	77.625
1	1.153	1.571	6.567
2	1.183	1.801	3.305
3	1.189	1.850	–

Table 4.5: Parameter evolution across iterations

4.5.7 Key Observations

From this detailed example, we can make several important observations:

1. **Cost Reduction**: The cost function decreased dramatically: $77.625 \rightarrow 6.567 \rightarrow 3.305$. The largest improvement occurred in the first iteration.
2. **Parameter Evolution**: Both parameters increased from their initial values, moving toward better fit to the data:
   - θ_0: $1.000 \rightarrow 1.153 \rightarrow 1.183 \rightarrow 1.189$
   - θ_1: $0.500 \rightarrow 1.571 \rightarrow 1.801 \rightarrow 1.850$
3. **Decreasing Updates**: The size of parameter updates decreases with each iteration, indicating convergence:
   - Iteration 1: $\Delta\theta_0 = 0.153$, $\Delta\theta_1 = 1.071$
   - Iteration 2: $\Delta\theta_0 = 0.030$, $\Delta\theta_1 = 0.230$
   - Iteration 3: $\Delta\theta_0 = 0.006$, $\Delta\theta_1 = 0.049$

4. **Gradient Magnitude**: The gradients approach zero as we near the minimum:
   - Iteration 1: $|\nabla_{\theta_0} J| = 15.3$, $|\nabla_{\theta_1} J| = 107.1$
   - Iteration 2: $|\nabla_{\theta_0} J| = 3.013$, $|\nabla_{\theta_1} J| = 22.950$
   - Iteration 3: $|\nabla_{\theta_0} J| = 0.623$, $|\nabla_{\theta_1} J| = 4.910$
5. **Prediction Improvement**: Looking at the errors in the tables, predictions become increasingly accurate with each iteration.

4.5.8 Convergence Behavior

If we were to continue this process for many more iterations (say, 1000), the parameters would converge to their optimal values. For this dataset, the optimal solution (which can be computed analytically) is approximately:

$$\theta_0^* \approx 1.2, \quad \theta_1^* \approx 1.9 \tag{4.45}$$

Our gradient descent algorithm is clearly moving in the right direction, and with continued iterations, it would eventually reach these optimal values within numerical precision.
The rate of convergence depends heavily on the learning rate. Our choice of $\alpha = 0.01$ represents a conservative value that ensures stability. A larger learning rate would converge faster but might risk instability; a smaller one would be more stable but require many more iterations.

4.6 Variations of Gradient Descent

While we've focused on batch gradient descent, there are important variations that offer different trade-offs between computational efficiency and convergence behavior.

4.6.1 Batch Gradient Descent (BGD)

Batch gradient descent, which we've been using, calculates the error for each example in the training dataset but only updates the model parameters after all training examples have been evaluated.
Update equations:

$$\theta_0 := \theta_0 - \alpha \frac{2}{n} \sum_{i=1}^{n} (\theta_0 + \theta_1 x_i - y_i) \tag{4.46}$$

$$\theta_1 := \theta_1 - \alpha \frac{2}{n} \sum_{i=1}^{n} (\theta_0 + \theta_1 x_i - y_i) \cdot x_i \tag{4.47}$$

Advantages:

- Stable error gradient and convergence

- Efficient use of vectorized operations
- Guaranteed convergence for convex problems with appropriate learning rate
- Theoretically optimal gradient direction

Disadvantages:

- Requires entire dataset in memory
- Slow for very large datasets
- Cannot adapt to streaming data
- May be slow to escape saddle points

4.6.2 Stochastic Gradient Descent (SGD)

Stochastic gradient descent calculates the error and updates the model for *each individual example* in the training dataset, without averaging.
Update equations (for each example i):

$$\theta_0 := \theta_0 - \alpha \cdot 2(\theta_0 + \theta_1 x_i - y_i) \tag{4.48}$$

$$\theta_1 := \theta_1 - \alpha \cdot 2(\theta_0 + \theta_1 x_i - y_i) \cdot x_i \tag{4.49}$$

Advantages:

- Much faster updates (one per example rather than one per epoch)
- Can work with streaming data
- Memory efficient (process one example at a time)
- Noise in gradient can help escape local minima (for non-convex problems)
- Can start making progress before seeing all data

Disadvantages:

- Noisy gradient estimates lead to erratic convergence
- May never truly converge (oscillates around minimum)
- Loss of vectorization efficiency
- Sensitive to learning rate choice
- Typically requires learning rate scheduling

4.6.3 Mini-Batch Gradient Descent

Mini-batch gradient descent represents a compromise, splitting the training dataset into small batches and updating parameters based on each batch.
Update equations (for batch B):

$$\theta_0 := \theta_0 - \alpha \frac{2}{|B|} \sum_{i \in B} (\theta_0 + \theta_1 x_i - y_i) \tag{4.50}$$

$$\theta_1 := \theta_1 - \alpha \frac{2}{|B|} \sum_{i \in B} (\theta_0 + \theta_1 x_i - y_i) \cdot x_i \tag{4.51}$$

where $|B|$ is the batch size (commonly 32, 64, 128, or 256).
Advantages:

- Balance between BGD and SGD
- Efficient use of modern hardware (GPUs, vectorization)
- More stable gradient estimates than SGD
- Faster than BGD for large datasets
- Can parallelize batch processing
- Moderate memory requirements

Disadvantages:

- Introduces batch size as an additional hyperparameter
- Still has some noise in gradient estimates
- Requires careful tuning of batch size

4.7 Multiple Linear Regression

4.7.1 Generalization to Multiple Variables

Real-world problems often involve multiple predictor variables. Multiple linear regression extends simple linear regression to handle n independent variables (features):

$$h_\theta(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n \tag{4.52}$$

This can be written more compactly using vector notation:

$$h_\theta(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x} \tag{4.53}$$

where:

- $\mathbf{x} = [1, x_1, x_2, \ldots, x_n]^T$ is the feature vector (we add 1 for the intercept)
- $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \ldots, \theta_n]^T$ is the parameter vector
- T denotes transpose

4.7.2 Cost Function for Multiple Regression

The cost function generalizes naturally:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{4.54}$$

where:

- m is the number of training examples
- $\mathbf{x}^{(i)}$ is the feature vector for the i-th example
- $y^{(i)}$ is the target value for the i-th example
- The notation i (in the dataset) should not be confused with i used elsewhere as an index

4.7.3 Gradient Computation for Multiple Features

The partial derivative with respect to each parameter θ_j is:

$$\frac{\partial J}{\partial \theta_j} = \frac{2}{m}\sum_{i=1}^{m}(h_\theta(\mathbf{x}^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \tag{4.55}$$

for $j = 0, 1, 2, \ldots, n$, where by convention $x_0^{(i)} = 1$ for all i (to handle the intercept term).

4.7.4 Update Rules for Multiple Regression

The gradient descent update rule becomes:

$$\theta_j := \theta_j - \alpha\frac{2}{m}\sum_{i=1}^{m}(h_\theta(\mathbf{x}^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \tag{4.56}$$

for all $j = 0, 1, 2, \ldots, n$ simultaneously.
In vectorized form:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha\nabla J(\boldsymbol{\theta}) \tag{4.57}$$

4.7.5 The normal equations

While gradient descent provides one approach to minimizing J, we can also explore an alternative method that finds the minimum directly without using an iterative process. This technique involves computing the partial derivatives of J with respect to each parameter θ_j and equating them to zero. To accomplish this efficiently—avoiding extensive algebraic manipulations and lengthy matrix derivative calculations—we'll first establish some matrix calculus notation.

Matrix derivatives

For a function $f : \mathbb{R}^{n\times d} \mapsto \mathbb{R}$ mapping from n-by-d matrices to the real numbers, we define the derivative of f with respect to A to be:

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1d}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{n1}} & \cdots & \frac{\partial f}{\partial A_{nd}} \end{bmatrix}$$

Thus, the gradient $\nabla_A f(A)$ is itself an n-by-d matrix, whose (i, j)-element is $\partial f/\partial A_{ij}$. For example, suppose $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ is a 2-by-2 matrix, and the function $f : \mathbb{R}^{2\times 2} \mapsto \mathbb{R}$ is given by

$$f(A) = \frac{3}{2}A_{11} + 5A_{12}^2 + A_{21}A_{22}.$$

Here, A_{ij} denotes the (i, j) entry of the matrix A. We then have

$$\nabla_A f(A) = \begin{bmatrix} \frac{3}{2} & 10A_{12} \\ A_{22} & A_{21} \end{bmatrix}.$$

Let us now proceed to find in closed-form the value of θ that minimizes $J(\theta)$. We begin by re-writing J in matrix-vectorial notation.
Given a training set, define the **design matrix** X to be the n-by-d matrix (actually n-by-$d+1$, if we include the intercept term) that contains the training examples' input values in its rows:

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(n)})^T — \end{bmatrix}.$$

Also, let $\vec{y}$ be the n-dimensional vector containing all the target values from the training set:

$$\vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Now, since $h_\theta(x^{(i)}) = (x^{(i)})^T\theta$, we can easily verify that

$$\begin{aligned} X\theta - \vec{y} &= \begin{bmatrix} (x^{(1)})^T\theta \\ \vdots \\ (x^{(n)})^T\theta \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix} \\ &= \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ \vdots \\ h_\theta(x^{(n)}) - y^{(n)} \end{bmatrix}. \end{aligned}$$

Thus, using the fact that for a vector z, we have that $z^T z = \sum_i z_i^2$:

$$\begin{aligned}\frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) &= \frac{1}{2}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2 \\ &= J(\theta)\end{aligned}$$

Finally, to minimize J, let's find its derivatives with respect to θ. Hence,

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) \\ &= \frac{1}{2}\nabla_\theta \left((X\theta)^T X\theta - (X\theta)^T\vec{y} - \vec{y}^T(X\theta) + \vec{y}^T\vec{y}\right) \\ &= \frac{1}{2}\nabla_\theta \left(\theta^T(X^TX)\theta - \vec{y}^T(X\theta) - \vec{y}^T(X\theta)\right) \\ &= \frac{1}{2}\nabla_\theta \left(\theta^T(X^TX)\theta - 2(X^T\vec{y})^T\theta\right) \\ &= \frac{1}{2}\left(2X^TX\theta - 2X^T\vec{y}\right) \\ &= X^TX\theta - X^T\vec{y}\end{aligned}$$

In the third step, we used the fact that $a^Tb = b^Ta$, and in the fifth step used the facts $\nabla_x b^Tx = b$ and $\nabla_x x^TAx = 2Ax$ for symmetric matrix A (for more details, see Section 4.3 of "Linear Algebra Review and Reference"). To minimize J, we set its derivatives to zero, and obtain the **normal equations**:

$$X^TX\theta = X^T\vec{y}$$

Thus, the value of θ that minimizes $J(\theta)$ is given in closed form by the equation

$$\theta = (X^TX)^{-1}X^T\vec{y}.$$[1]

4.7.6 Feature Scaling

When features have different scales (e.g., house size in thousands of square feet vs. number of bedrooms), gradient descent can converge very slowly. Feature scaling addresses this issue.

Normalization (Min-Max Scaling):

$$x'_j = \frac{x_j - \min(x_j)}{\max(x_j) - \min(x_j)} \tag{4.58}$$

This scales features to the range $[0, 1]$.

[1]Note that in the above step, we are implicitly assuming that X^TX is an invertible matrix. This can be checked before calculating the inverse. If either the number of linearly independent examples is fewer than the number of features, or if the features are not linearly independent, then X^TX will not be invertible. Even in such cases, it is possible to "fix" the situation with additional techniques, which we skip here for the sake of simplicty.

Standardization (Z-score Normalization):

$$x'_j = \frac{x_j - \mu_j}{\sigma_j} \tag{4.59}$$

where μ_j is the mean and σ_j is the standard deviation of feature j. This centers features at zero with unit variance.
Feature scaling dramatically improves gradient descent convergence because:

- The cost function becomes more spherical (less elongated)
- The gradient points more directly toward the minimum
- A single learning rate works well for all parameters

4.8 Model Evaluation

4.8.1 Importance of Evaluation Metrics

The strength of any linear regression model must be assessed using appropriate evaluation metrics. These metrics provide quantitative measures of how well the observed outputs are being predicted by the model. Different metrics emphasize different aspects of prediction quality.

4.8.2 Root Mean Squared Error (RMSE)

The Root Mean Squared Error is one of the most frequently used metrics:

$$\text{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h_\theta(\mathbf{x}^{(i)}) - y^{(i)})^2} \tag{4.60}$$

Properties:

- Has the same units as the target variable, making it interpretable
- Penalizes large errors more heavily due to squaring
- Always non-negative; zero indicates perfect predictions
- More sensitive to outliers than MAE

4.8.3 Mean Absolute Error (MAE)

The Mean Absolute Error measures the average magnitude of errors:

$$\text{MAE} = \frac{1}{m}\sum_{i=1}^{m}|h_\theta(\mathbf{x}^{(i)}) - y^{(i)})| \tag{4.61}$$

Properties:

- Same units as target variable
- Less sensitive to outliers than RMSE
- All errors weighted equally
- More robust metric when dataset contains anomalies

4.8.4 R-Squared (R^2) Coefficient of Determination

The R^2 metric measures the proportion of variance in the dependent variable explained by the model:

$$R^2 = 1 - \frac{\sum_{i=1}^{m}(y^{(i)} - h_\theta(\mathbf{x}^{(i)}))^2}{\sum_{i=1}^{m}(y^{(i)} - \bar{y})^2} \tag{4.62}$$

where $\bar{y} = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}$ is the mean of observed values.
Interpretation:

- $R^2 = 1$: Perfect predictions
- $R^2 = 0$: Model no better than predicting the mean
- $R^2 < 0$: Model worse than predicting the mean (rare, indicates serious problems)
- Typically ranges from 0 to 1
- Unitless, allowing comparison across different problems

4.8.5 Practical Considerations for Model Evaluation

- **Training vs. Test Error**: Always evaluate on held-out test data to assess generalization
- **Cross-Validation**: Use k-fold cross-validation for more robust performance estimates
- **Multiple Metrics**: Consider multiple metrics; no single metric tells the whole story
- **Domain Context**: Interpret metrics in the context of the problem domain

4.9 Practical Considerations and Best Practices

4.9.1 Convergence Criteria

Several strategies exist for determining when to stop gradient descent:

1. **Fixed iterations**: Stop after a predetermined number of iterations

2. **Cost threshold**: Stop when $J < \epsilon$ for some small ϵ
3. **Cost change**: Stop when $|J^{(t)} - J^{(t-1)}| < \epsilon$
4. **Gradient magnitude**: Stop when $\|\nabla J\| < \epsilon$
5. **Parameter change**: Stop when $\|\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\| < \epsilon$

In practice, a combination of these criteria often works best, with a maximum iteration limit to prevent infinite loops.

4.9.2 Diagnosing Convergence Issues

Common problems and their solutions:

- **Cost increasing**: Learning rate too large; reduce α by factors of 10
- **Slow convergence**: Learning rate too small; increase α, or check feature scaling
- **Oscillating cost**: Learning rate too large; reduce α
- **Plateauing at high cost**: May be stuck in local minimum (rare for linear regression) or may need better features

4.9.3 Assumptions of Linear Regression

Linear regression models make several key assumptions that should be verified:

1. **Linearity**: The relationship between features and target is linear
2. **Independence**: Observations are independent of each other
3. **Homoscedasticity**: Constant variance of residuals across all levels of predictors
4. **Normality**: Residuals are approximately normally distributed
5. **No multicollinearity**: Features are not highly correlated (for multiple regression)

Violations of these assumptions may lead to:

- Biased parameter estimates
- Invalid confidence intervals and hypothesis tests
- Unreliable predictions
- Poor generalization to new data

Residual analysis is the primary tool for checking these assumptions.

4.10 Summary and Conclusions

This chapter has provided a comprehensive treatment of linear regression with gradient descent optimization. Let us summarize the key concepts:

4.10.1 Main Takeaways

1. **Linear Regression Fundamentals**:
    - Models relationships between variables using linear functions
    - Parameters $(\theta_0, \theta_1, \ldots, \theta_n)$ define the relationship
    - Goal is to find parameters that best fit the observed data
2. **Cost Function**:
    - Mean Squared Error measures prediction quality
    - Convex function guarantees unique global minimum
    - Lower cost indicates better fit to training data
3. **Gradient Descent**:
    - Iterative optimization algorithm
    - Updates parameters in direction of negative gradient
    - Learning rate controls step size
    - Converges to optimal solution for convex problems
4. **Algorithm Variants**:
    - Batch: Stable but slow for large datasets
    - Stochastic: Fast but noisy
    - Mini-batch: Best of both worlds for most applications
5. **Practical Considerations**:
    - Feature scaling dramatically improves convergence
    - Multiple evaluation metrics provide different insights
    - Model assumptions should be verified
    - Hyperparameter tuning (learning rate, batch size) is essential

4.10.2 Connections to Advanced Topics

The concepts learned in this chapter form the foundation for understanding more sophisticated machine learning algorithms:

- **Neural Networks**: Use gradient descent with backpropagation, a generalization of what we've learned
- **Logistic Regression**: Similar structure but with different cost function and activation
- **Regularization**: Add penalty terms to cost function to prevent overfitting
- **Optimization Methods**: Adam, RMSprop, and other advanced optimizers build on gradient descent
- **Deep Learning**: Modern deep learning relies entirely on variants of gradient descent

4.10.3 Final Thoughts

Linear regression with gradient descent optimization represents a perfect introduction to machine learning because it combines mathematical rigor with practical applicability. The simplicity of linear regression allows us to understand optimization principles deeply, while gradient descent provides a powerful and general technique applicable to vastly more complex models.

Mastering these fundamentals—understanding how gradients guide parameter updates, how learning rates affect convergence, and how to evaluate model performance—prepares you for the exciting journey into more advanced machine learning techniques. The intuition developed through working with linear regression and gradient descent will serve you well throughout your career in data science and machine learning.

Chapter 5

Machine Learning Evaluation

5.1 Introduction

The evaluation of machine learning models is fundamental to understanding their performance, reliability, and suitability for real-world applications. Without proper evaluation, we cannot determine whether a model has truly learned meaningful patterns from data or merely memorized training examples. This chapter addresses two critical aspects of model evaluation:

1. **Evaluation Metrics**: Quantitative measures that assess model performance across different dimensions
2. **Data Splitting Strategies**: Techniques for partitioning data to obtain reliable and unbiased performance estimates

The choice of appropriate evaluation metrics depends heavily on the type of machine learning task (classification, regression, clustering, etc.) and the specific requirements of the application domain. Similarly, the data splitting strategy must be carefully selected to ensure that performance estimates are representative of real-world scenarios.

5.1.1 The Importance of Proper Evaluation

Consider a medical diagnosis system that predicts whether a patient has a rare disease occurring in only 1% of the population. A naive model that always predicts "no disease" would achieve 99% accuracy—an impressive number that masks its complete failure to identify actual cases. This example illustrates why we need multiple evaluation metrics and careful consideration of the problem context.

5.2 Data Splitting Strategies

Before evaluating a model, we must decide how to partition our available data. The fundamental principle is to assess model performance on data that was not used during training, thereby testing the model's ability to generalize to new, unseen examples.

5.2.1 Train-Test Split

The simplest data splitting approach divides the dataset into two disjoint subsets:

- **Training Set**: Used to fit model parameters (typically 70-80% of data)
- **Test Set**: Used to evaluate final model performance (typically 20-30% of data)

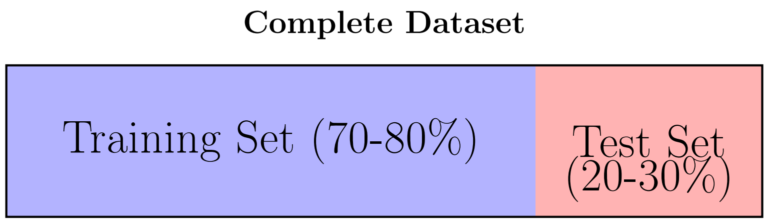


Figure 5.1: Train-test split: The dataset is divided into training and test portions

Advantages:

- Simple and computationally efficient
- Clear separation between training and evaluation data
- Suitable for large datasets

Disadvantages:

- Performance estimate can have high variance (depends on which examples end up in test set)
- Reduces available training data
- No hyperparameter tuning without further splitting

5.2.2 Train-Validation-Test Split

For more rigorous evaluation, especially when hyperparameter tuning is required, we use a three-way split:

- **Training Set**: For fitting model parameters (typically 60-70%)
- **Validation Set**: For hyperparameter tuning and model selection (typically 10-20%)
- **Test Set**: For final, unbiased performance evaluation (typically 10-20%)

The workflow proceeds as follows:

1. Train multiple models with different hyperparameters on the training set
2. Evaluate each model on the validation set
3. Select the best-performing hyperparameters based on validation performance
4. Retrain the final model on training + validation data
5. Report final performance on the test set (only once!)

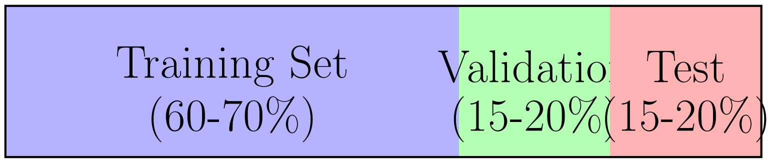


Figure 5.2: Train-validation-test split for hyperparameter tuning

5.2.3 Cross-Validation

Cross-validation provides a more robust performance estimate by training and evaluating the model multiple times on different data partitions.

K-Fold Cross-Validation

In k-fold cross-validation, the dataset is divided into k equal-sized folds. The model is trained k times, each time using k-1 folds for training and one fold for validation.

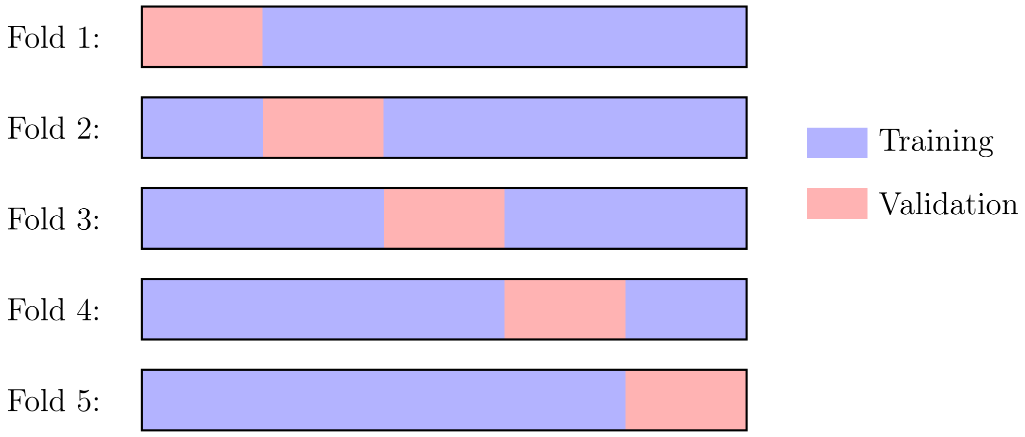


Figure 5.3: 5-fold cross-validation: Each fold serves as validation set once

The final performance is computed as the average across all k folds:

$$\text{CV Score} = \frac{1}{k}\sum_{i=1}^{k} \text{Score}_i \tag{5.1}$$

where Score_i is the performance metric on fold i.
Advantages:

- More reliable performance estimate with lower variance
- Every example is used for both training and validation
- Particularly useful for small datasets

Disadvantages:

- Computationally expensive (k times more training)

- May be infeasible for large datasets or complex models

Common k values:

- k=5 or k=10 are standard choices
- Larger k reduces bias but increases variance and computation
- Smaller k increases bias but reduces computation

Stratified K-Fold Cross-Validation

For classification problems, especially with imbalanced classes, stratified k-fold ensures that each fold maintains the same class distribution as the original dataset.
Example: Consider a binary classification dataset with 80% negative and 20% positive examples. Stratified 5-fold CV ensures each fold contains approximately 80% negative and 20% positive examples.

Leave-One-Out Cross-Validation (LOOCV)

LOOCV is an extreme case where k equals the number of examples n. Each iteration uses n-1 examples for training and 1 for validation.
Advantages:

- Maximum use of training data
- Deterministic (no randomness in splits)
- Nearly unbiased performance estimate

Disadvantages:

- Extremely computationally expensive
- High variance in performance estimate
- Training sets are very similar, leading to correlated results

5.2.4 Time Series Cross-Validation

For temporal data, traditional cross-validation violates the time-ordering assumption. Time series cross-validation uses expanding or rolling windows:
Key principles:

- Always train on past data and validate on future data
- Never shuffle time series data
- Account for temporal dependencies and seasonality

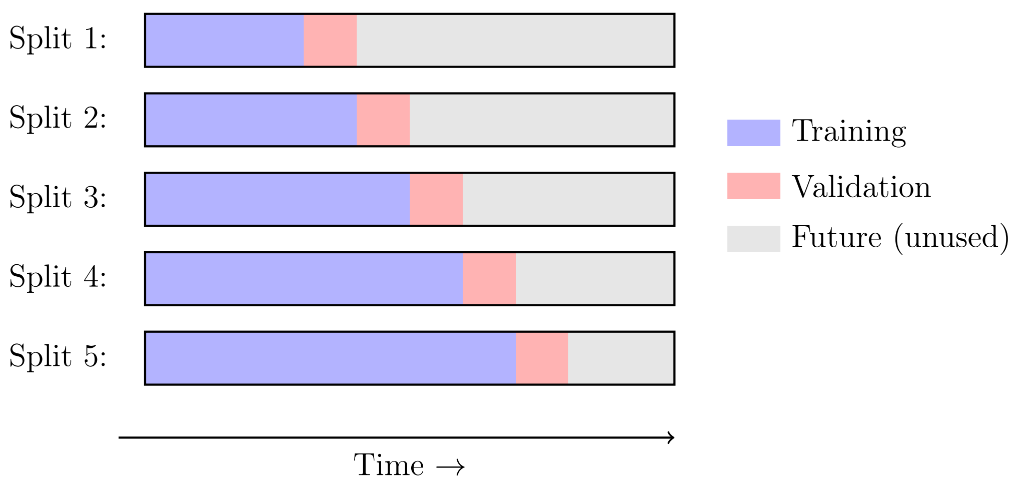


Figure 5.4: Time series cross-validation with expanding window

5.3 Classification Metrics

Classification tasks involve predicting discrete class labels. The choice of evaluation metric depends on the problem characteristics and business requirements.

5.3.1 Confusion Matrix

The confusion matrix is the foundation for most classification metrics. For binary classification:

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

Table 5.1: Confusion matrix for binary classification

Definitions:

- **True Positive (TP)**: Correctly predicted positive cases
- **True Negative (TN)**: Correctly predicted negative cases
- **False Positive (FP)**: Incorrectly predicted as positive (Type I error)
- **False Negative (FN)**: Incorrectly predicted as negative (Type II error)

5.3.2 Accuracy

Accuracy measures the proportion of correct predictions:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.2}$$

When to use:

- Balanced datasets where all classes are equally important
- When misclassification costs are equal for all classes

Limitations:

- Misleading for imbalanced datasets
- Does not distinguish between types of errors

Example: A spam filter that correctly identifies 95 legitimate emails and 85 spam messages out of 100 legitimate and 100 spam messages has an accuracy of $(95 + 85)/(100 + 100) = 0.90$ or 90%.

5.3.3 Precision

Precision measures the proportion of positive predictions that are correct:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.3}$$

Precision answers: "Of all examples predicted as positive, how many were actually positive?"
When to use:

- When false positives are costly
- Example: Medical screening where false positives lead to unnecessary procedures

5.3.4 Recall (Sensitivity, True Positive Rate)

Recall measures the proportion of actual positive examples that were correctly identified:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.4}$$

Recall answers: "Of all actual positive examples, how many did we identify?"
When to use:

- When false negatives are costly
- Example: Disease diagnosis where missing a case could be fatal

5.3.5 Specificity (True Negative Rate)

Specificity measures the proportion of actual negative examples correctly identified:

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{5.5}$$

5.3.6 F1-Score

The F1-score is the harmonic mean of precision and recall, providing a single metric that balances both:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN} \tag{5.6}$$

The harmonic mean is used (rather than arithmetic mean) because it is more sensitive to low values. If either precision or recall is low, the F1-score will be low.
F$_\beta$-Score: A generalization that allows weighting precision and recall differently:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}} \tag{5.7}$$

- $\beta < 1$: Emphasizes precision
- $\beta > 1$: Emphasizes recall
- $\beta = 1$: Equal weight (standard F1-score)

5.3.7 Precision-Recall Trade-off

Most classifiers produce a probability or score that is thresholded to make predictions. Adjusting this threshold creates a trade-off between precision and recall:
Example interpretation:

- Low threshold: Classify more examples as positive $\rightarrow$ higher recall, lower precision
- High threshold: Classify fewer examples as positive $\rightarrow$ higher precision, lower recall
- Optimal threshold depends on relative cost of FP vs. FN

5.3.8 ROC Curve and AUC

The Receiver Operating Characteristic (ROC) curve plots the True Positive Rate (Recall) against the False Positive Rate at various threshold settings.

$$\text{False Positive Rate (FPR)} = \frac{FP}{FP + TN} = 1 - \text{Specificity} \tag{5.8}$$

Area Under the ROC Curve (AUC-ROC):
The AUC quantifies the overall ability of the model to discriminate between positive and negative classes:

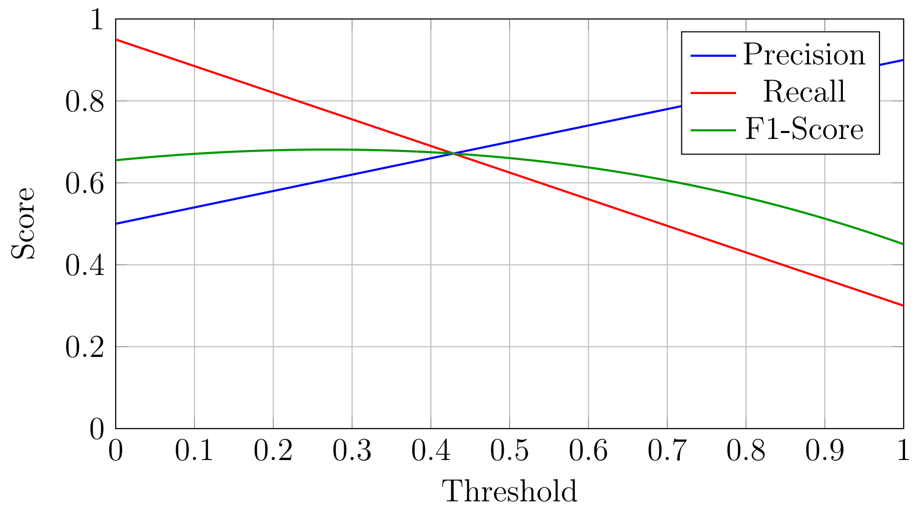


Figure 5.5: Precision-Recall trade-off as classification threshold varies

- AUC = 1.0: Perfect classifier
- AUC = 0.5: Random classifier (no discriminative power)
- AUC $<$ 0.5: Worse than random (predictions are inverted)

Interpretation: AUC represents the probability that the model ranks a random positive example higher than a random negative example.

Advantages of ROC-AUC:

- Threshold-independent evaluation
- Robust to class imbalance
- Single number summary of classifier performance

When to use:

- When you need a threshold-independent metric
- For comparing multiple models
- When false positive and false negative costs are similar

5.3.9 Precision-Recall Curve and Average Precision

For highly imbalanced datasets, the Precision-Recall (PR) curve is often more informative than ROC curve:

Average Precision (AP): The area under the precision-recall curve, computed as the weighted mean of precisions at each threshold:

$$AP = \sum_n (R_n - R_{n-1}) P_n \tag{5.9}$$

where P_n and R_n are precision and recall at the n-th threshold.

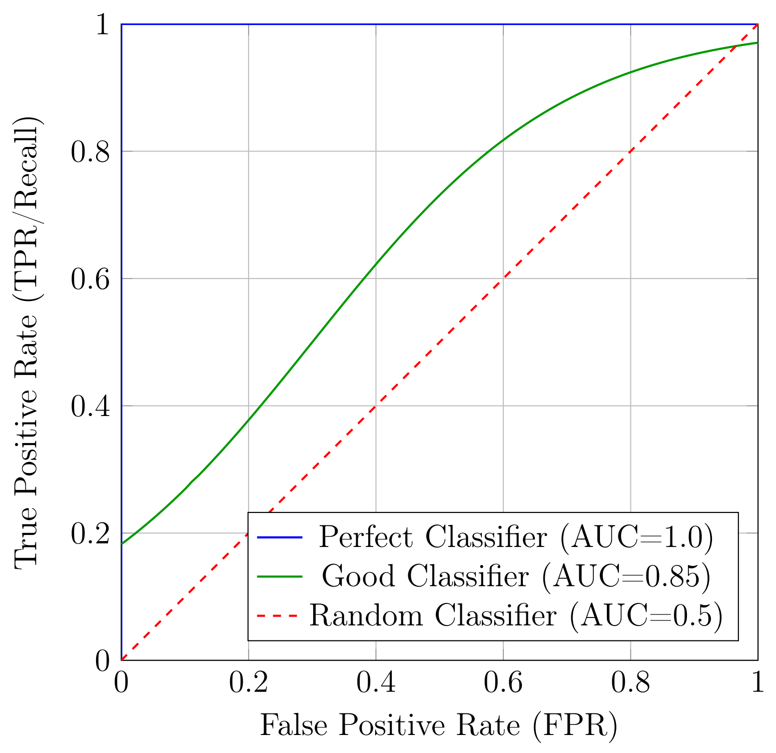


Figure 5.6: ROC curves for different classifier qualities

5.3.10 Multi-Class Classification Metrics

For problems with more than two classes, metrics can be computed using different averaging strategies:

Macro-Average

Calculate metric for each class independently, then average:

$$\text{Macro-Average} = \frac{1}{C}\sum_{i=1}^{C}\text{Metric}_i \tag{5.10}$$

where C is the number of classes. Treats all classes equally, regardless of size.

Micro-Average

Aggregate TP, FP, FN across all classes, then calculate metric:

$$\text{Micro-Average Precision} = \frac{\sum_{i=1}^{C} TP_i}{\sum_{i=1}^{C}(TP_i + FP_i)} \tag{5.11}$$

Weights classes by their support (number of samples).

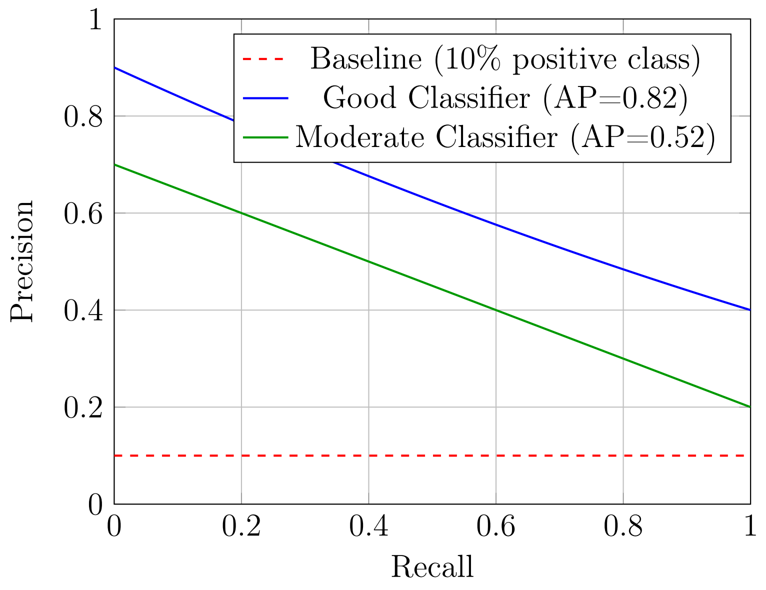


Figure 5.7: Precision-Recall curves for imbalanced dataset (10% positive class)

Weighted Average

Weight each class metric by its support:

$$\text{Weighted Average} = \frac{\sum_{i=1}^{C} n_i \cdot \text{Metric}_i}{\sum_{i=1}^{C} n_i} \tag{5.12}$$

where n_i is the number of samples in class i.
Example: Consider a 3-class problem with the following per-class F1-scores and supports:

Class	**F1-Score**	**Support**
Class A	0.90	100
Class B	0.70	50
Class C	0.50	10

- Macro F1: $(0.90 + 0.70 + 0.50)/3 = 0.70$
- Weighted F1: $(100 \times 0.90 + 50 \times 0.70 + 10 \times 0.50)/160 = 0.81$

5.3.11 Cohen's Kappa

Cohen's Kappa measures agreement between predictions and ground truth, accounting for agreement occurring by chance:

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{5.13}$$

where:

- p_o = observed agreement (accuracy)
- p_e = expected agreement by chance

Interpretation:

- $\kappa = 1$: Perfect agreement
- $\kappa = 0$: Agreement equivalent to chance
- $\kappa < 0$: Agreement worse than chance

5.3.12 Matthews Correlation Coefficient (MCC)

MCC is considered one of the best single metrics for binary classification, especially for imbalanced datasets:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5.14}$$

Properties:

- Range: [-1, 1]
- $+1$: Perfect prediction
- 0: Random prediction
- -1: Perfect inverse prediction

Advantages:

- Accounts for all four confusion matrix categories
- Robust to class imbalance
- Symmetric (no bias toward positive or negative class)

5.4 Regression Metrics

Regression tasks involve predicting continuous values. Let y_i denote the true value and $\hat{y}_i$ the predicted value for example i, with n total examples.

5.4.1 Mean Absolute Error (MAE)

MAE measures the average absolute difference between predictions and actual values:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{5.15}$$

Properties:

- Same units as the target variable
- Robust to outliers (compared to MSE)
- All errors weighted equally

When to use:

- When outliers should not dominate the metric
- When all errors have similar importance

5.4.2 Mean Squared Error (MSE)

MSE measures the average squared difference:

$$MSE = \frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{5.16}$$

Properties:

- Penalizes larger errors more heavily (quadratic)
- Units are squared (less interpretable)
- Differentiable everywhere (useful for optimization)

5.4.3 Root Mean Squared Error (RMSE)

RMSE is the square root of MSE, returning metric to original units:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2} \tag{5.17}$$

Comparison with MAE:

Property	**MAE**	**RMSE**
Outlier sensitivity	Low	High
Interpretability	High	High
Error distribution assumption	Any	Gaussian
Penalty for large errors	Linear	Quadratic

Table 5.2: Comparison of MAE and RMSE

5.4.4 R-Squared (R^2) Coefficient of Determination

R^2 measures the proportion of variance in the target variable explained by the model:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{5.18}$$

where $\bar{y}$ is the mean of actual values, SS_{res} is the residual sum of squares, and SS_{tot} is the total sum of squares.
Interpretation:

- $R^2 = 1$: Perfect predictions
- $R^2 = 0$: Model performs no better than predicting the mean
- $R^2 < 0$: Model performs worse than predicting the mean

Example: If $R^2 = 0.85$, the model explains 85% of the variance in the target variable.
Limitations:

- Always increases (or stays same) when adding features, even irrelevant ones
- Can be misleading for non-linear relationships
- Not suitable for comparing models with different numbers of features

5.4.5 Adjusted R-Squared

Adjusted R^2 penalizes model complexity:

$$R^2_{adj} = 1 - \frac{(1 - R^2)(n - 1)}{n - p - 1} \tag{5.19}$$

where n is the number of samples and p is the number of features.
Advantages:

- Accounts for number of predictors
- Only increases if new variables improve model more than expected by chance
- Better for model comparison

5.4.6 Mean Absolute Percentage Error (MAPE)

MAPE expresses error as a percentage:

$$MAPE = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right| \tag{5.20}$$

Advantages:

- Scale-independent (useful for comparing across different scales)
- Easy to interpret (percentage error)

Limitations:

- Undefined when $y_i = 0$
- Asymmetric (penalizes over-predictions more than under-predictions)
- Biased toward models that under-predict

5.4.7 Visual Evaluation

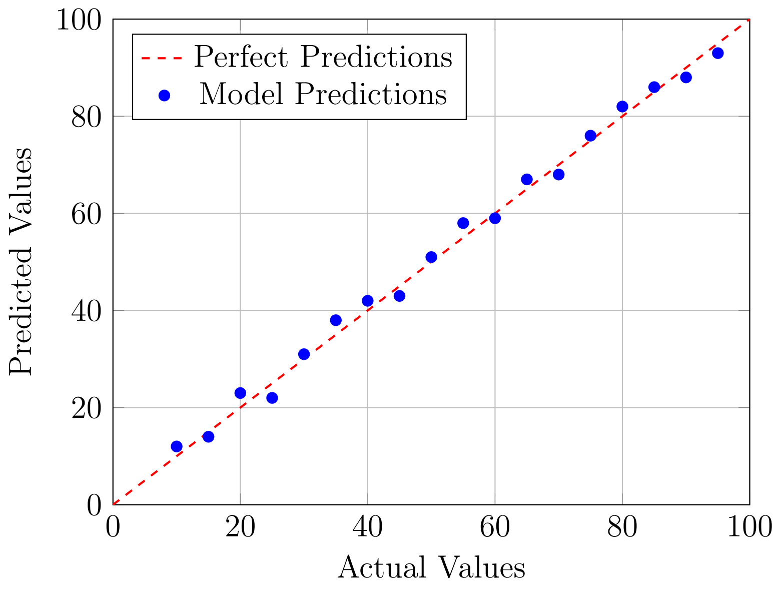


Figure 5.8: Predicted vs. Actual values plot for regression evaluation

Residual plots show prediction errors:
A good model shows randomly scattered residuals with no clear pattern.

5.5 Handling Imbalanced Datasets

Class imbalance occurs when one class significantly outnumbers others, creating challenges for both model training and evaluation.

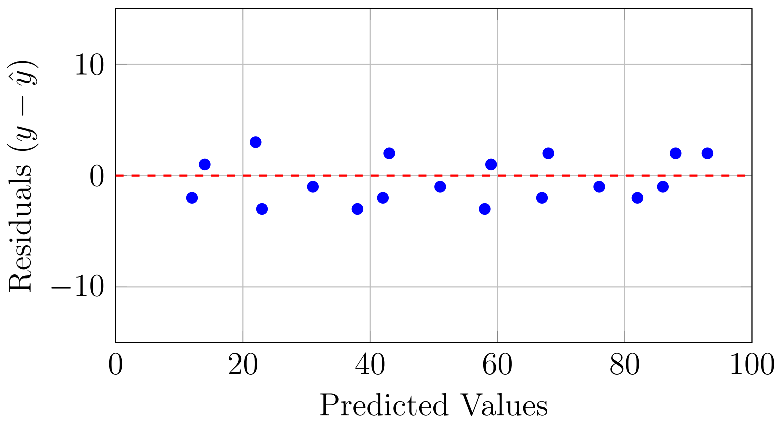


Figure 5.9: Residual plot for checking model assumptions

5.5.1 The Problem

Consider a fraud detection system where only 0.1% of transactions are fraudulent:

- A model predicting "not fraud" for all cases achieves 99.9% accuracy
- But it fails completely at the actual task (detecting fraud)

5.5.2 Evaluation Strategies

1. **Use appropriate metrics:**
   - Avoid accuracy; prefer precision, recall, F1-score
   - Use ROC-AUC or Precision-Recall AUC
   - Consider Matthews Correlation Coefficient
2. **Stratified sampling:**
   - Use stratified k-fold cross-validation
   - Ensures each fold maintains class distribution
3. **Class-specific metrics:**
   - Report precision and recall for each class
   - Use confusion matrix to identify specific weaknesses

5.5.3 Data-Level Solutions

Random Oversampling

Randomly duplicate minority class examples:
Pros: Simple, no information loss
Cons: Risk of overfitting, no new information added

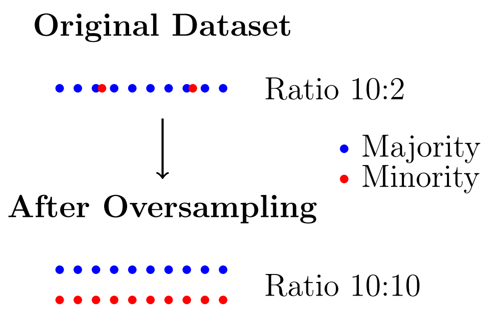


Figure 5.10: Random oversampling of minority class

Random Undersampling

Randomly remove majority class examples:
Pros: Reduces training time, balanced dataset
Cons: Loss of potentially useful information

SMOTE (Synthetic Minority Oversampling Technique)

Generate synthetic minority examples by interpolating between existing ones:

1. For each minority example, find k nearest minority neighbors
2. Select one neighbor randomly
3. Create synthetic example along line between original and neighbor:

$$x_{new} = x_i + \lambda \cdot (x_{neighbor} - x_i) \tag{5.21}$$

   where $\lambda \in [0, 1]$ is randomly chosen

Pros: Creates new, plausible examples; reduces overfitting
Cons: Can create noisy examples; computationally expensive

5.5.4 Algorithm-Level Solutions

1. Class weights: Penalize misclassification of minority class more heavily:

- Most algorithms support sample weights or class weights
- Weight inversely proportional to class frequency

2. Threshold adjustment: Lower classification threshold for minority class
3. Ensemble methods: Use balanced ensembles (e.g., BalancedRandomForest)

Scenario	**Recommended Metrics**
Balanced classification	Accuracy, F1-score, ROC-AUC
Imbalanced classification	Precision, Recall, F1-score, PR-AUC, MCC
Cost-sensitive classification	Weighted F-score, custom cost function
Multi-class classification	Macro/Weighted F1, per-class metrics
Regression (outliers rare)	RMSE, R^2
Regression (outliers present)	MAE, Median Absolute Error
Regression (multiple scales)	MAPE (if no zeros), R^2

Table 5.3: Metric selection guidelines by scenario

5.6 Best Practices and Guidelines

5.6.1 Choosing Evaluation Metrics

5.6.2 Data Splitting Best Practices

1. **Size considerations:**
    - Large datasets ($n > 100{,}000$): Simple train-test split often sufficient
    - Medium datasets ($1{,}000 < n < 100{,}000$): Use k-fold cross-validation
    - Small datasets ($n < 1{,}000$): Use stratified k-fold or LOOCV
2. **Maintain data distribution:**
    - Use stratified splitting for classification
    - Preserve temporal order for time series
    - Keep grouped data together (e.g., multiple samples from same patient)
3. **Test set integrity:**
    - Never use test set for model selection or hyperparameter tuning
    - Evaluate on test set exactly once with final model
    - If multiple test evaluations needed, create separate validation set
4. **Data leakage prevention:**
    - Apply preprocessing (scaling, imputation) after splitting
    - Fit preprocessing only on training data
    - Be cautious with feature engineering based on target variable

5.6.3 Common Pitfalls

1. **Data leakage:**
   - Fitting scalers on entire dataset before splitting
   - Using future information in time series
   - Including test samples in feature selection
2. **Inappropriate metrics:**
   - Using accuracy for highly imbalanced data
   - Optimizing for metric that doesn't align with business objective
   - Ignoring metric limitations (e.g., MAPE with zeros)
3. **Overfitting to validation set:**
   - Excessive hyperparameter tuning iterations
   - Not using separate test set
   - Model selection based on validation performance
4. **Temporal leakage:**
   - Random splitting of time series data
   - Using standard cross-validation for temporal data
   - Not accounting for concept drift

5.6.4 Reporting Results

A complete evaluation report should include:

1. **Dataset description:**
   - Size (number of examples)
   - Class distribution (for classification)
   - Feature characteristics
2. **Splitting strategy:**
   - Method used (e.g., 5-fold stratified CV)
   - Train/validation/test sizes
   - Rationale for choice
3. **Multiple metrics:**

- Primary metric (aligned with objective)
- Supporting metrics (for comprehensive view)
- Per-class metrics (for classification)

4. **Uncertainty estimates:**
   - Mean and standard deviation across CV folds
   - Confidence intervals
   - Statistical significance tests (when comparing models)

5. **Visualizations:**
   - Confusion matrix (classification)
   - ROC and/or PR curves (classification)
   - Residual plots (regression)
   - Learning curves (if relevant)

5.7 Advanced Topics

5.7.1 Learning Curves

Learning curves show model performance as a function of training set size:

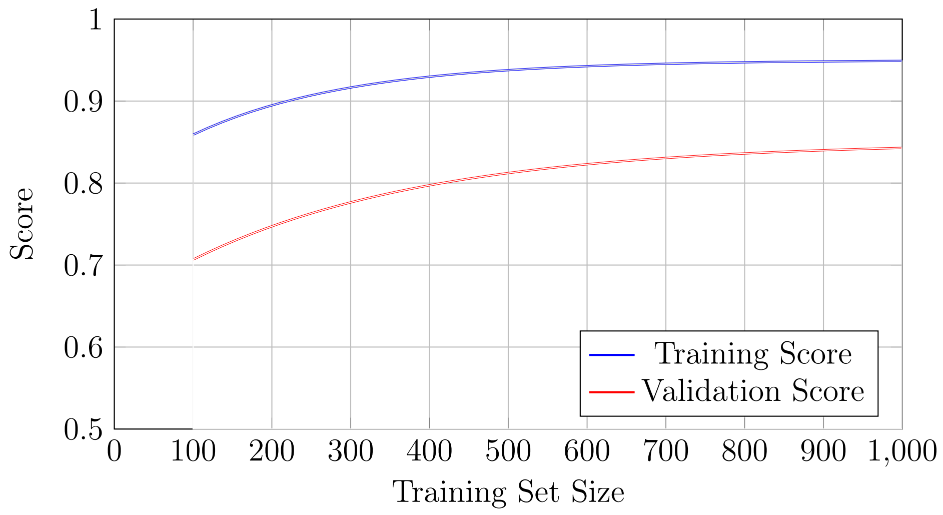


Figure 5.11: Learning curves showing high bias (both curves plateau at suboptimal performance)

Interpretation:

- Large gap between curves: High variance (overfitting) - get more data or reduce complexity

- Both curves plateau at low performance: High bias (underfitting) - increase model complexity
- Curves converge near top: Good fit - model is appropriate

5.7.2 Nested Cross-Validation

For unbiased hyperparameter tuning and model selection:

1. **Outer loop:** K-fold CV for performance estimation
2. **Inner loop:** K-fold CV for hyperparameter tuning (on outer training set)

This prevents optimistic bias from hyperparameter tuning on the same data used for performance estimation.

5.7.3 Statistical Significance Testing

When comparing models, statistical tests can determine if performance differences are significant:
Paired t-test: Compare mean performance across CV folds
McNemar's test: For binary classification, tests if two models make different errors
Considerations:

- CV folds are not independent (violates test assumptions)
- Corrected resampled t-test addresses this
- Practical significance vs. statistical significance

5.8 Summary

Proper evaluation is fundamental to machine learning success. Key takeaways:

1. **Match metrics to objectives:** Choose evaluation metrics that align with business goals and problem characteristics.
2. **Use appropriate splitting:** Select data splitting strategies that provide reliable performance estimates for your specific scenario.
3. **Consider multiple metrics:** A single metric rarely captures all aspects of model performance.
4. **Account for imbalance:** Class imbalance requires special consideration in both evaluation and data splitting.
5. **Prevent data leakage:** Maintain strict separation between training and test data throughout the pipeline.

6. **Report comprehensively:** Include dataset description, splitting strategy, multiple metrics, uncertainty estimates, and visualizations.
7. **Think critically:** Always question whether evaluation results reflect real-world performance.

The evaluation strategies and metrics presented in this chapter form the foundation for making informed decisions throughout the machine learning development lifecycle. By applying these techniques rigorously, practitioners can develop models that not only perform well on benchmarks but also deliver value in production environments.