This book introduces the fundamental principles of machine learning and intelligent modeling through practical Python implementations and real-world examples. It begins by establishing the foundations of smart systems, learning paradigms, evaluation metrics, and model validation before progressing to classical supervised and unsupervised machine learning algorithms. Readers are guided through linear regression, logistic regression, k-nearest neighbors, support vector machines, decision trees, ensemble learning, clustering techniques, and principal component analysis, with an emphasis on both the underlying mathematical concepts and their practical implementation using Python. Each chapter combines theoretical background, methodology, implementation, experimental evaluation, and discussion to provide a comprehensive learning experience. Intended for undergraduate and graduate students, researchers, and practitioners, this volume serves as both an academic textbook and a practical reference for developing reliable, interpretable, and data-driven intelligent systems using modern machine learning techniques.



Prof. Dr. Alexander I. Iliev is a professor and researcher in Artificial Intelligence, Machine Learning, Big Data Analytics, and Smart Systems. He holds three patents and has authored numerous scientific publications and books. He is affiliated with UC Berkeley, SRH University of Applied Sciences Heidelberg, and the Bulgarian Academy of Sciences.

9 786630 107265

Smart Python | Applied ML | Part 1

FOR AUTHOR USE ONLY

Alexander I. Iliev

LAP LAMBERT Academic Publishing



Alexander I. Iliev

Smart Python for Machine Learning and Intelligent Modeling

Foundations and Classical Machine Learning | Part 1

Alexander I. Iliev

Smart Python for Machine Learning and Intelligent Modeling

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

Alexander I. Iliev

Smart Python for Machine Learning and Intelligent Modeling

Foundations and Classical Machine Learning | Part 1

FOR AUTHOR USE ONLY

LAP LAMBERT Academic Publishing

Imprint
Any brand names and product names mentioned in this book are subject to trademark, brand or patent protection and are trademarks or registered trademarks of their respective holders. The use of brand names, product names, common names, trade names, product descriptions etc. even without a particular marking in this work is in no way to be construed to mean that such names may be regarded as unrestricted in respect of trademark and brand protection legislation and could thus be used by anyone.

Cover image: www.ingimage.com

Publisher:
LAP LAMBERT Academic Publishing
is a trademark of
Dodo Books Indian Ocean Ltd. and OmniScriptum S.R.L publishing group

120 High Road, East Finchley, London, N2 9ED, United Kingdom
Str. Armeneasca 28/1, office 1, Chisinau MD-2012, Republic of Moldova, Europe
Managing Directors: Ieva Konstantinova, Victoria Ursu
info@omniscriptum.com

Printed at: see last page
ISBN: 978-66-30-10726-5

Copyright © Alexander I. Iliev
Copyright © 2026 Dodo Books Indian Ocean Ltd. and OmniScriptum S.R.L publishing group

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

Smart **P**ython for **M**achine **L**earning and **I**ntelligent **M**odeling

Foundations and Classical Machine Learning

FOR AUTHOR USE ONLY

Part 1

Author:
Prof Dr Alexander Iliev Iliev

Scientific Editors:

Prof Dr Reiner Creutzburg
SRH University of Applied Sciences, Berlin

Prof Dr Daniel Arias Aranda
University of Granada, Granada

A Note from the Author:

This book, ***Smart Python for Machine Learning and Intelligent Modeling: Foundations and Classical Machine Learning | Part 1***, is the first volume in a series dedicated to machine learning and intelligent modeling. It introduces the fundamental concepts of smart systems and classical machine learning algorithms through practical Python implementations and real-world examples.

Currently, this textbook is used in the accredited ***Big Data and Artificial Intelligence*** master's degree program at the ***School of Technology and Architecture, SRH University of Applied Sciences Heidelberg***. It serves as the principal textbook for the course ***Machine Learning***.

This volume is complemented by ***Smart Python for Machine Learning and Intelligent Systems: Deep Learning, Transfer Learning, and AI Engineering | Part 2***, which extends the material presented here by covering deep learning, transfer learning, modern AI engineering, deployment, optimization, and the engineering of intelligent systems. Together, the two volumes provide a comprehensive introduction to modern machine learning using Python.

FOR AUTHOR USE ONLY

Table of Contents

PART I — Foundation of Intelligent Modeling 14
Chapter 1 - What Makes Systems Smart? 14
Defining Intelligence in Computing 14
The Three Pillars of Modern Smart Systems 14
Narrow vs General Intelligence 15
1.1 Learning Systems vs Rule-Based Systems 15
1.1.1 Rule Based Systems: Programming by Specification 15
1.1.2 Learning Systems: Programming by Example 15
1.1.3 A Taxonomy of Machine Learning Paradigms 16
Supervised Learning 16
Unsupervised Learning 16
Reinforcement Learning 16
Self-Supervised and Foundation Models 16
1.1.4 A Head-To-Head Comparison 17
1.1.5 When to Use Which 17
1.1.6 Code Example: Rule-Based vs. Learned Classifier 17
Rule-based approach (hand-crafted logic): 18
Learning-Based Approach (Trained on Historical Data): 18
1.2 Bias – Variance Tradeoff 19
1.2.1 The Core Problem: Generalization 19
1.2.2 Decomposing the Prediction Error 20
1.2.3 Bias: Systematic Error 20
1.2.4 Variance: Sensitivity to Training Data 20
1.2.5 The Tradeoff in Practice 21
1.2.6 Code Example: Visualizing the Tradeoff 21
1.2.7 Regularization as a Variance Reducer 23
1.3 Training, Validation and Testing Pipelines 23
1.3.1 Why Pipelines Matter 23

FOR AUTHOR USE ONLY

1.3.2 The 3 Data Splits 24
Training Set 24
Validation Set 24
Test Set 24
1.3.3 Cross-Validation 24
1.3.4 Stratified Splits and Temporal Splits 25
Imbalanced classes 25
Time-series data 26
1.3.5 Data Leakage: The Silent Failure 26
1.3.6 Hyperparameter Tuning Within the Pipeline 27
1.4 Evaluation Metrics Overview 28
1.4.1 Why the Right Metric Matters 28
1.4.2 The Confusion Matrix 28
1.4.3 Classification Metrics 29
Accuracy 29
Precision 29
Recall 29
F1 Score 29
ROC-AUC 30
1.4.4 Code Example: Computing and Visualizing Classification Metrics 30
1.4.5 Regression Metrics 31
1.4.6 Metrics for Imbalanced Problems 33
1.4.7 A Guide to Choose the Right Metric 33
1.4.8 All Metrics Summary 34
Chapter 2 - Baseline Models and Simple Classifiers 35
2.1 What are Baseline Models? 35
2.1.1 Defining a Baseline 35
2.1.2 Why Baseline Models Matter 36
Establishing a Performance Floor 36
Detecting Data and Evaluation Problems 36

FOR AUTHOR USE ONLY

Quantifying the Value of Complexity 36
Communicating Results Clearly 36
2.1.3 The Hierarchy of Model Complexity 36
2.1.4 Class Imbalance and Baseline Accuracy 37
2.1.5 Baseline Accuracy as a Function of Class Distribution 38
2.2 ZeroR: The Simplest Classifier 38
2.2.1 What is ZeroR? 38
2.2.2 How ZeroR Works 39
2.2.3 ZeroR Results on Three Datasets 40
2.2.4 When ZeroR 'High' Accuracy is Misleading 41
2.3 OneR: One Rule Classification 41
2.3.1 What is OneR? 41
2.3.2 How OneR Works Step by Step 42
2.3.3 Pre-processing Requirements for OneR 42
2.3.4 OneR Results on Three Datasets 43
2.3.5 Comparing ZeroR and OneR 44
2.3.6 Strengths and Limitation of OneR 45
2.3.7 ZeroR and OneR as Part of a Modelling Workflow 45
2.4. Summary: Baseline Classifiers - Summary 46
Chapter 3 – Linear Models 47
3.1. Linear Regression 47
3.1.1. What is Linear Regression? 47
3.1.3. Assumptions of Linear Regression 47
3.1.4. Polynomial Feature Expansion 48
3.1.5. Code Example: Linear Regression on Rental Data 48
3.2. Gradient Descent 50
3.2.1. Why We Need an Optimization Algorithm 50
3.2.3. The 3 Variants of Gradient Descent 50
3.2.4. The Learning Rate: Convergence and Divergence 51
3.2.5. Gradient Descent for Linear Regression 52

FOR AUTHOR USE ONLY

3.2.6. Adaptive Learning Rate Methods 53
3.3. Logistic Regression 53
3.3.1. From Regression to Classification 53
3.3.2. The Sigmoid Function 54
3.3.3. The Loss Function: Binary Cross-Entropy 54
3.3.4. Interpretability: Reading the Coefficients 54
3.3.5. Dual-Model Framework: The Rent Fairness Use Case 55
3.4. Regularization (L1 & L2) 57
3.4.1. The Problem: Overfitting and Multicollinearity 57
3.4.2. L2 Regularization: Ridge Regression 58
3.4.3. L1 Regularization: Lasso Regression 58
3.4.4. Elastic Net: Combining L1 and L2 59
3.4.5. Comparison: L1 vs L2 vs Elastic Net 59
3.4.6. Choosing the Regularization Strength (λ) 59
3.4.7. Regularization Paths: Visualizing Coefficient Shrinkage 60
3.4.8. Regularization in Logistic Regression 61
3.4.9. Summary: All the Linear Models 62
PART II — Classical Machine Learning Algorithms 64
Chapter 4 - K-Nearest Neighbors (KNN) 64
4.1 Distance Metrics: Detailed Analysis 68
4.2 Choosing K: Practical Considerations 70
4.3 Curse of Dimensionality: Impact on KNN Performance 72
4.4 Methodology 73
4.4.1 Architecture 74
4.4.2 Model 75
4.4.3 Methods 76
4.4.4 Analytical expressions 77
4.4.5 Results 78
4.4.6 Discussion 81
4.5 Conclusion 82

FOR AUTHOR USE ONLY

Chapter 5 - Efficient Nearest Neighbor Search 84
5.1 Ball Trees 85
5.1.1 Introduction 85
5.1.2 Core Concept 88
5.1.3 Methodology 89
5.1.4 Implementation 94
5.1.5 Evaluation 98
5.1.6 Discussion 102
5.1.7 Conclusion 102
5.2 KD-Trees 103
5.2.1 Introduction 103
5.2.2 Core Concept 104
5.2.3 Methodology 105
5.2.4 Implementation 106
5.2.5 Evaluation 110
5.2.6 Discussion 112
5.2.7 Conclusion 113
5.3 Tree-Based KNN Optimization 113
5.3.1 Introduction 113
5.3.2 Core Concept 114
5.3.3 Methodology 115
5.3.4 Implementation 116
5.3.5 Evaluation 122
5.3.6 Discussion 126
5.4 Conclusion 127
Chapter 6 - Support Vector Machines (SVM) 129
6.1. Margin 130
6.1.1 Types of SVM 130
6.2. Types of Margins 131
6.2.1 Hard Margin SVM 131

FOR AUTHOR USE ONLY

6.2.2 Soft Margin SVM 131
6.2.3 Theoretical Background 132
6.2.4 Statistical Learning Theory 133
6.3. Kernel Trick for Non-Linear Image Data 134
6.3.1 Methodology 135
6.3.2 Model Evaluation 138
6.4 Conclusion 139
Chapter 7 - Naïve Bayes: Text Classification 141
7.1 Bayesian Decision Theory and MAP Classification 142
7.2 Derivation of Naïve Bayes Classifier 143
7.3 Theoretical Variants 144
7.3.1 Variants of Naïve Bayes for Text Classification 144
7.3.2 Why These Variants Matter in Practice 145
7.3.3 Python Implementation: Comparing Model Variants 146
7.3.4 Laplace Smoothing and Statistical Implications 148
7.3.5 Effects of Smoothing Parameters 149
7.3.6 Implications 149
7.4 Methodology and Experimental Setup 150
7.4.1 Dataset Description 150
7.4.2 Preprocessing 150
7.4.3 Train-Test Split 150
7.4.4 Vectorization Configuration 150
7.4.5 Model Definition 150
7.4.6 Experimental Loop 151
7.5 Results 152
7.6 Python Implementation: Comparing Model Variants 154
7.7 Conclusion 156
7.8 Limitations and Future Work 156
Chapter 8 - Decision Trees 158
8.1 Decision Trees for Classification 159

FOR AUTHOR USE ONLY

8.1.1 Splitting Criteria 159
8.1.2 Gini Impurity 160
8.1.3 Entropy 160
8.1.4 Overfitting in Decision Trees 161
8.1.5 Hyperparameter Control 162
8.1.6 Pruning 162
8.2 Methodology & Dataset 163
8.3 Feature Selection and Target Definition 164
8.4 Feature Encoding 164
8.5 Train-Test Split 164
8.6 Hyperparameter Optimization 165
8.6.1 Initial Results and Performance Analysis 165
8.6.2 Optimized Performance and Comparative Results: 169
8.7 Results and Discussions 172
8.8 Conclusion 172
Chapter 9 - Ensemble Learning I 173
9.1. Use cases 173
9.1.1. Meningioma 173
9.1.2. Glioma 174
9.1.3. Pituitary Tumor 174
9.1.4. Diagnosis with Magnetic Resonance Imaging (MRI) 175
9.2. Literature Review 176
9.3. Methodology 177
9.4. Implementation 181
9.4.1. Project Environment 181
9.4.2. Dataset Structure and Data Pipeline 181
9.4.3. Image Preprocessing 182
9.4.4. Data Augmentation 184
9.4.5. CNN Training 185
9.4.6. Deep Feature Extraction 186

FOR AUTHOR USE ONLY

9.4.7. Random Forest and Hyper parameter optimization 187
9.4.8. Evaluation Metrics 188
9.4.9. Model Interpretability Using LIME 188
9.5 Results & Discussion 189
9.5.1. Validation Set Evaluation 189
9.5.2. Test Set Evaluation 190
9.5.3. Final Test Evaluation 192
9.5.4. LIME Explainability Analysis 195
9.5.5 Conclusion 196
9.6 Ensemble Learning 199
9.7 Random Forest 201
9.8 Bagging (Bootstrap Aggregating) 202
9.9 Error estimation as an out-of-bag (OOB) method 203
9.10 Advantages and Limitations 205
9.11 Conclusion 205
Chapter 10 - Ensemble Learning II 206
Foreword 206
10.1. AdaBoost 207
10.1.1. Artificial Intelligence in Healthcare 207
10.1.2. Machine Learning in Diagnosis 207
10.1.3. Ensemble Learning 207
10.1.4. AdaBoost Research Background 207
10.1.5. Objective of the Study 208
10.1.6 Dataset Description 208
10.1.7. Data preprocessing 208
10.1.8. AdaBoost algorithm and model architecture 209
10.1.9. Model implementation 211
10.1.10. Results and performance evaluation 211
10.1.11. Discussion of Part I 215
10.2. Gradient Boosting 216

FOR AUTHOR USE ONLY

10.2.1. Introduction 216
10.2.2. Gradient Boosting 216
10.2.3. Comparative analysis of Gradient Boosting methods 217
10.2.4. Comparison of Boosting and Bagging 217
10.2.5. Mathematical intuition for Gradient Boosting 218
10.2.6. Use case: predicting house prices 219
10.3. Comparative study: AdaBoost vs Gradient Boosting 228
10.3.1. Motivation 228
10.3.2. Experimental design 229
10.3.3. Baseline models 230
10.3.4. Boosting models 230
10.3.5. Cross-validation 231
10.3.6. Comparative evaluation 231
10.3.7. Explainability with SHAP 233
10.3.8. Analytical discussion 235
10.3.9. Contribution summary 236
10.4. Conclusion 236
PART III — Unsupervised Learning & Dimensionality Reduction 238
Chapter 11 - K-Means Clustering 238
11.1. Introduction 238
11.2. Methodology and Unique Contributions 239
11.2.1 Architecture and Workflow 239
11.2.2 Data Description and Preprocessing 239
11.2.3 Model Formulation and K-Means Equations 240
11.2.4 Selection Strategy and Initialization Considerations 241
11.2.5 Implementation Details 241
11.3. Original code (Tested and Working Snippets) 242
11.4. Results: Graphs, Tables, and Discussion 245
11.5. Discussion of Findings 250
11.6. Conclusion 251

FOR AUTHOR USE ONLY

Chapter 12 - DBSCAN 252
12.1. Methodology 254
12.1.1. DBSCAN Theory: Epsilon, MinPts, and Point Types 254
12.1.2. How the DBSCAN Algorithm Works Step by Step 255
12.1.3. Feature Scaling 256
12.1.4. Algorithmic Complexity 257
12.2. Data Preprocessing 258
12.2.1. Dataset Description 258
12.2.2. Feature Scaling 260
12.2.3. Choosing eps using k-Distance Plot 260
12.2.4. Applying DBSCAN 261
12.2.5. Silhouette Score and K-means Comparison 262
12.3 Results 263
12.4 Conclusion 266
Chapter 13 - Principal Component Analysis (PCA) 268
Abstract 268
Introduction 268
13.1. Understanding Dimensionality Reduction 268
13.2. Problems When There Are Too Many Data Points 269
13.2.1 Sparsity and Distance Breakdown 269
13.2.2 Redundant and Correlated Features 269
13.2.3 Noise and False Signals 269
13.3. PCA Algorithms and their Variants 269
13.3.1 Covariance-based PCA vs SVD-based PCA 270
13.3.2 Variants of PCA 271
13.3.3 Complexity and Performance Considerations 271
13.4. Case Study 273
13.5 Proposed Methodology 274
13.5.1. System Workflow 274
13.5.2. Mathematical Methodology 275

FOR AUTHOR USE ONLY

13.5.3. Observational Analysis of PCA on High- Resolution Retinal Imaging 278
13.6 Code examples for retaining image dataset 285
13.7. Applications of PCA 291
13.7.1 Image Compression 292
13.7.2 Noise Reduction 292
13.7.3 Visualization 292
13.8. Limitations 293
13.8.1. Linearity 293
13.8.2. Loss of Interpretability 293
13.8.3. Sensitivity to Scaling 293
13.8.4. Information Loss 293
13.9. Conclusion 294
References 295

FOR AUTHOR USE ONLY

PART I — Foundation of Intelligent Modeling

Chapter 1 - What Makes Systems Smart?

Defining Intelligence in Computing

In everyday conversation the word 'smart' describes actions that may be vastly different in nature: a recommendation that feels uncannily accurate, a spam filter that catches mail we ourselves would have deleted, a navigation app that reroutes around traffic before we even notice the jam. But what is happening inside these systems? Is there genuine intelligence at work, or something more mechanical?

Computer scientists and engineers have long debated the right word for it, but the working definition used throughout this module is deliberately practical: a system is smart when it can handle situations it was not explicitly programmed for. A pocket calculator is not smart as every behavior is hard-coded. A chess engine that was hand-crafted with thousands of opening rules is only marginally smarter. A system that learned to play chess by playing millions of games against itself, discovering strategies its programmers never anticipated, is smart in the sense we care about.

FOR AUTHOR USE ONLY

The Three Pillars of Modern Smart Systems

Contemporary AI and big-data systems draw their 'smartness' from three interlocking capabilities:

- ***Data***: the raw fuel. Without sufficient, representative data a system cannot learn anything reliable about the world.
- ***Algorithms***: the machinery that extracts patterns from data. From simple linear regression to deep neural networks, algorithms are the mathematical engines of learning.
- ***Computation***: the scale at which data and algorithms can operate. The same algorithms that existed decades ago now produce remarkable results simply because modern hardware can apply them to orders of magnitude more data.

Remove any one of these three and the system stops being smart. A powerful algorithm trained on too little data overfits noise. Mountains of data processed by a trivial algorithm miss subtle patterns. A theoretically sound approach that cannot be computed in reasonable time never ships.

Narrow vs General Intelligence

It is important to calibrate expectations early. Every system covered in this module is a form of narrow artificial intelligence: it excels at one well-defined task and is useless outside it. A model trained to detect tumors in chest X-rays cannot read handwriting. A language model that writes fluent prose cannot reliably do long-division without additional scaffolding. Artificial General Intelligence, a system that transfers knowledge across arbitrary domains the way humans do, remains an open research problem.

This distinction matters for practitioners. When you deploy a smart system, you are deploying a very powerful, very specialized tool. Knowing its boundaries is as important as knowing its capabilities.

1.1 Learning Systems vs Rule-Based Systems

1.1.1 Rule Based Systems: Programming by Specification

The oldest paradigm in software engineering is deceptively simple, if you can describe every situation a system will encounter, write a rule for each one. Rule-based systems, also called expert systems or knowledge-based systems, encode human expertise directly as logical conditions and actions.

A classic example is a spam filter built from rules:

- IF subject contains "Win a prize" THEN mark as spam
- IF sender domain is not in whitelist AND body contains "click here" THEN mark as spam
- IF … THEN …

FOR AUTHOR USE ONLY

Writing such a system requires a domain expert to articulate rules and a programmer to implement them. The system is fully transparent: every decision traces back to a specific rule. It is also predictable, auditable, and fast.

The fundamental limitation is coverage. The world is not finite. Spammers evolve their language; edge cases multiply; new domains arise constantly. Maintaining a large rule base becomes a full-time job, and gaps in the rules produce gaps in behavior. Rule-based systems also struggle to handle uncertainty: a rule either fires or does not.

1.1.2 Learning Systems: Programming by Example

Instead of specifying the rules, a learning system is given data, thousands or millions of labelled examples and asked to induce the rules itself. The programmer designs the learning algorithm and the data pipeline; the algorithm discovers the patterns.

Using the same spam example, a learning system receives a corpus of emails pre-labelled 'spam' or 'not spam'. It analyzes word frequencies, sender patterns, structural features, and learns a model that assigns probabilities. When a new email arrives the model outputs, for example, 0.97 (very likely spam) or 0.03 (very likely legitimate).

The key philosophical shift is this: the programmer no longer needs to know what distinguishes spam from legitimate mail. They only need enough labelled examples and a suitable learning algorithm. This shift is what enables systems to tackle problems that are too complex, too high-dimensional, or too poorly understood for manual rule-writing.

1.1.3 A Taxonomy of Machine Learning Paradigms

Machine learning is not one technique but a family of related approaches:

Supervised Learning

The algorithm is given labelled training data, input–output pairs and learns a mapping from inputs to outputs. Classification (discrete outputs) and regression (continuous outputs) are the two canonical tasks. Most industrial ML is supervised.

FOR AUTHOR USE ONLY

Unsupervised Learning

No labels are provided. The algorithm discovers structure in the data on its own: clusters, embeddings, compressed representations, or generative models of the data distribution. Useful when labelling is expensive or when the goal is exploration rather than prediction.

Reinforcement Learning

An agent interacts with an environment, receives rewards or penalties for its actions, and learns a policy that maximizes cumulative reward. Used for sequential decision-making problems such as game-playing, robotics, and recommendation systems.

Self-Supervised and Foundation Models

A modern hybrid where the supervision signal is derived automatically from the data itself (e.g., predicting the next word in a sentence). This enables training on enormous unlabeled corpora, producing general-purpose representations that can be fine-tuned for downstream tasks. Large language models belong to this category.

1.1.4 A Head-To-Head Comparison

Table 1.1. Rule-based systems versus learning systems across key dimensions.

Rule-Based Systems	Learning Systems
Knowledge source: human experts	Knowledge source: data
Rules are explicit and auditable	Model weights are often opaque
Predictable on known inputs	Generalizable to unseen inputs
Brittle: gaps in rules = gaps in behavior	Robust: interpolates between examples
Cheap to run, expensive to maintain	Expensive to train, cheap to run
Ideal when rules are known and stable	Ideal when patterns are latent or complex
No data required	Requires substantial labelled data
No training phase	Training can take hours to weeks

1.1.5 When to Use Which

FOR AUTHOR USE ONLY

The choice is rarely binary. Production systems often combine both approaches. A medical diagnosis assistant might use a rule engine to enforce hard clinical constraints (drug interactions, dosage limits) while a learning model ranks differential diagnoses by probability. This hybrid architecture captures the strengths of each: reliability and compliance from rules, flexibility, and pattern-matching from learning.

As a rule of thumb:

- If you can fully specify the logic and it will not change often, use rules.
- If the problem involves high-dimensional inputs, noisy data, or latent patterns humans cannot easily articulate, use learning.
- If both apply, use both.

1.1.6 Code Example: Rule-Based vs. Learned Classifier

The following example illustrates the conceptual difference concretely. Both implementations solve the same task — classifying whether a loan applicant is likely to default — but they go about it in fundamentally different ways.

Rule-based approach (hand-crafted logic):

```python
def rule_based_loan_decision(applicant: dict) -> str:
    """
    Hard-coded rules written by a credit-risk expert.
    Every condition is explicit and auditable.
    """
    # Hard rejection rules
    if applicant["credit_score"] < 500:
        return "reject"
    if applicant["debt_to_income_ratio"] > 0.6:
        return "reject"
    if applicant["months_employed"] < 3:
        return "reject"

    # Conditional approval
    if applicant["credit_score"] >= 750 and applicant["debt_to_income_ratio"] < 0.3:
        return "approve"

    # Grey area — escalate to human review
    return "manual_review"


applicant = {
    "credit_score": 620,
    "debt_to_income_ratio": 0.45,
    "months_employed": 12
}

print(rule_based_loan_decision(applicant))  # → 'manual_review'
```

Code Snippet 1.1. A rule-based loan-default classifier.

FOR AUTHOR USE ONLY

Learning-Based Approach (Trained on Historical Data):

```python
import numpy as np
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# ── Simulated historical loan data ──────────────────────────────
np.random.seed(42)
n = 2000

credit_scores       = np.random.randint(300, 850, n)
dti_ratios          = np.random.uniform(0.1, 0.8, n)
months_employed     = np.random.randint(0, 120, n)
loan_amounts        = np.random.randint(5_000, 100_000, n)

```

```python
# Ground-truth defaults: lower score + higher DTI → higher probability
default_prob = 1 / (1 + np.exp(
    0.01 * credit_scores - 3 * dti_ratios - 0.02 * months_employed + 1
))
defaulted = (np.random.rand(n) < default_prob).astype(int)

X = np.column_stack([credit_scores, dti_ratios, months_employed, loan_amounts])
y = defaulted

# ── Train / test split ──────────────────────────────────────────
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

scaler  = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test  = scaler.transform(X_test)

# ── Learn the decision boundary from data ───────────────────────
model = GradientBoostingClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# ── Predict on new applicant ────────────────────────────────────
new_applicant = scaler.transform([[620, 0.45, 12, 25_000]])
prob_default  = model.predict_proba(new_applicant)[0][1]
print(f'Predicted default probability: {prob_default:.2%}')

# ── Feature importance: what did the model learn? ───────────────
feature_names = ['credit_score', 'dti_ratio', 'months_employed', 'loan_amount']
for name, imp in sorted(
    zip(feature_names, model.feature_importances_), key=lambda x: -x[1]
):
    print(f'  {name:<20} {imp:.3f}')
```

FOR AUTHOR USE ONLY

Code Snippet 1.2. A learning-based loan-default classifier trained with gradient boosting.

The rule-based version requires an expert to enumerate every relevant condition. The learning-based version requires data and a choice of algorithm, but the model then discovers the relevant conditions itself, including non-linear interactions the expert might miss. The tradeoff is interpretability: the rules are immediately understandable; the gradient boosting model requires additional tools (feature importance, SHAP values) to explain.

1.2 Bias – Variance Tradeoff

1.2.1 The Core Problem: Generalization

Every machine learning model is trained on a finite sample of data. The goal, however, is not to memorize that sample, it is to learn something true about the underlying process that generated it, so that the model performs well on new, unseen data. This property is called generalization.

The bias-variance tradeoff is the formal framework for understanding why generalization fails. It decomposes the expected prediction error of a model into three additive components: bias, variance, and irreducible noise.

1.2.2 Decomposing the Prediction Error

For a regression problem with mean-squared error, we can write:

$$Expected\ MSE\ (x) = Bias^2(x) + Variance\ (x) + Irreducible\ Noise, \qquad (Eq.1.1)$$

Where:

$Bias\ (x) = E[\hat{f}(x)] - f(x)]$ # systematic error

$Variance\ (x) = E[(\hat{f}(x) - E[\hat{f}(x)])^2]$ # sensitivity to training data

$Irreducible\ Noise = Var(\varepsilon)$ # noise in the true process

The irreducible noise term is beyond our control; it reflects genuine randomness in the world. We can only affect bias and variance and reducing one tends to increase the other.

FOR AUTHOR USE ONLY

1.2.3 Bias: Systematic Error

Bias measures how far the average prediction of a model is from the true value. A high-bias model has made strong, incorrect assumptions about the data-generating process. It consistently errs in the same direction regardless of which training sample it sees.

The canonical high-bias failure mode is underfitting: the model is too simple to capture the true structure. A linear model fitted to a quadratic signal will always underestimate peaks and overestimate troughs, no matter how much data you give it.

- High bias → underfitting → poor performance on both training and test data.
- Symptoms: training error is high; test error is similarly high; adding more data does not help much.
- Remedies: use a more expressive model, add relevant features, reduce regularization.

1.2.4 Variance: Sensitivity to Training Data

Variance measures how much the model's predictions change when trained on different samples from the same distribution. A high-variance model has learned the idiosyncrasies of its training set rather than the true underlying pattern.

The canonical high-variance failure mode is overfitting: the model is so flexible that it fits noise as well as signal. A degree-15 polynomial fitted to 20 noisy data points will pass through every training point perfectly but oscillate wildly between them.

- High variance → overfitting → excellent performance on training data, poor performance on test data.
- Symptoms: training error is very low; test error is much higher (large generalization gap).
- Remedies: collect more data, simplify the model, add regularization, use ensemble methods, apply dropout (for neural networks).

1.2.5 The Tradeoff in Practice

As model complexity increases (more parameters, higher polynomial degree, deeper trees) bias decreases and variance increases. The optimal complexity sits at the minimum of the total expected error curve. This is not a fixed point: it shifts depending on the amount of training data (more data tolerates higher complexity) and the noise level in the problem.

Table 1.2. Characteristics of high-bias (underfitting) versus high-variance (overfitting) models.

FOR AUTHOR USE ONLY

Property	High Bias (Underfitting)	High Variance (Overfitting)
Model complexity	Too simple	Too complex
Training error	High	Very low
Test error	High	High
Generalization gap	Small	Large
Effect of more data	Limited improvement	Helps substantially
Typical example	Linear model on curved data	Deep tree on small dataset
Regularization effect	Makes things worse	Makes things better

1.2.6 Code Example: Visualizing the Tradeoff

The following example generates a noisy sine wave and fits polynomial models of increasing degrees. The learning curves reveal the bias-variance tradeoff directly: low-degree models have high training and test-error (bias); high-degree models have low training error but high test-error (variance).

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import cross_val_score

np.random.seed(0)

# ── Generate noisy data from y = sin(x) + ε ──────────────────────────
n_train = 30
x_train = np.sort(np.random.uniform(0, 2 * np.pi, n_train))
y_train = np.sin(x_train) + np.random.normal(0, 0.3, n_train)

x_plot = np.linspace(0, 2 * np.pi, 300)
y_true = np.sin(x_plot)

# ── Fit polynomials of increasing degree and measure errors ──────────
degrees       = [1, 3, 5, 9, 15]
train_errors  = []
cv_errors     = []

for deg in degrees:
    model = make_pipeline(PolynomialFeatures(deg), LinearRegression())
    model.fit(x_train.reshape(-1, 1), y_train)

    train_pred  = model.predict(x_train.reshape(-1, 1))
    train_mse   = np.mean((y_train - train_pred) ** 2)
    train_errors.append(train_mse)

    # 5-fold cross-validation gives an unbiased estimate of test error
    cv_mse = -cross_val_score(
        model,
        x_train.reshape(-1, 1),
        y_train,
        cv=5,
        scoring='neg_mean_squared_error'
    ).mean()
    cv_errors.append(cv_mse)

# ── Plot ─────────────────────────────────────────────────────────────
plt.figure(figsize=(8, 4))
plt.plot(degrees, train_errors, 'o-', label='Training MSE  (bias proxy)')
plt.plot(degrees, cv_errors,    's-', label='CV MSE         (total error)')
plt.xlabel('Polynomial Degree (model complexity)')
plt.ylabel('Mean Squared Error')
plt.title('Bias–Variance Tradeoff: Training vs. Cross-Validation Error')
plt.legend()
plt.tight_layout()
plt.show()

# ── Print summary ────────────────────────────────────────────────────
print(f'{'Degree':>8}  {'Train MSE':>10}  {'CV MSE':>10}')
for d, tr, cv in zip(degrees, train_errors, cv_errors):
    print(f'{d:>8}  {tr:>10.4f}  {cv:>10.4f}')

```

FOR AUTHOR USE ONLY

Code Snippet 1.3. Fitting polynomials of increasing degree to a noisy sine wave.

1.2.7 Regularization as a Variance Reducer

Regularization is the primary engineering tool for controlling variance without reducing model capacity. It adds a penalty term to the loss function that discourages large parameter values, effectively shrinking the model's effective complexity at training time.

```python
import numpy as np
from sklearn.linear_model import Ridge, Lasso
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.model_selection import cross_val_score

np.random.seed(0)

# ── Generate noisy data from y = sin(x) + ε ──────────────────────────
n_train = 30
x_train = np.sort(np.random.uniform(0, 2 * np.pi, n_train))
y_train = np.sin(x_train) + np.random.normal(0, 0.3, n_train)

# Ridge (L2): penalises sum of squared weights → shrinks all weights
# Lasso (L1): penalises sum of absolute weights → drives some weights to zero

alphas = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]

print(f'{"Alpha":>8}  {"Ridge CV MSE":>14}  {"Lasso CV MSE":>14}')
for alpha in alphas:
    ridge_mse = -cross_val_score(
        make_pipeline(PolynomialFeatures(9), StandardScaler(), Ridge(alpha=alpha)),
        x_train.reshape(-1, 1), y_train,
        cv=5, scoring='neg_mean_squared_error'
    ).mean()

    lasso_mse = -cross_val_score(
        make_pipeline(PolynomialFeatures(9), StandardScaler(), Lasso(alpha=alpha, max_iter=100000)),
        x_train.reshape(-1, 1), y_train,
        cv=5, scoring='neg_mean_squared_error'
    ).mean()

    print(f'  {alpha:<8}  {ridge_mse:>14.4f}  {lasso_mse:>14.4f}')
```

FOR AUTHOR USE ONLY

Code Snippet 1.4. Comparing Ridge and Lasso regularization strength against validation error.

1.3 Training, Validation and Testing Pipelines

1.3.1 Why Pipelines Matter

A machine learning model that performs well in a Jupyter notebook but fails in production is one of the most common and costly failures in applied AI. The gap usually does not arise from a flawed algorithm; it arises from a flawed evaluation methodology. Without a disciplined pipeline, you have no reliable way to know how your model will behave on data it has never seen.

The training–validation–test framework is the foundational methodology for building models whose performance estimates can be trusted. It is also a prerequisite for every evaluation metric discussed in Section 4.

1.3.2 The 3 Data Splits

Data is divided into three non-overlapping partitions, each serving a distinct purpose:

Training Set

The data the model learns from. Model parameters (weights, tree splits, etc.) are optimized to minimize a loss function on this set. Typical allocation: 60-80% of the total data.

Validation Set

The data used to tune hyperparameters (learning rate, regularization strength, tree depth, number of layers, etc.) and to make model selection decisions (which architecture to use, which features to include). The model never trains on this data, but the researcher does see its performance, which means repeated evaluation on the validation set gradually leaks information. Typical allocation: 10-20%.

FOR JUNIOR USE ONLY

Test Set

The data evaluated exactly once, at the very end, to produce the final unbiased performance estimate reported in papers, product launches, and audits. If you ever tune the model based on test set performance, the estimate is no longer unbiased. Typical allocation: 10-20%.

The test set must never influence any modelling decision. It is like a capsule: you seal it away at the start and only open it once you want to test the level of learning of your system. Any use of test-set information during development, however indirect, constitutes data leakage and will lead to over-optimistic reported performance.

1.3.3 Cross-Validation

When data is scarce, holding out a fixed validation set wastes too many examples. Cross-validation (CV) addresses this by rotating which portion of the training data is used for validation across multiple rounds, then averaging the results.

K-fold cross-validation is the standard form. The training data is divided into k equal folds. In each of k rounds, one-fold is held out as the validation set, and the model is trained on the remaining k−1 folds. The k performance scores are then averaged. With k=5 or k=10, cross-validation gives a much lower-variance

estimate of generalization error than a single split, at the cost of k times the computational effort.

```python
from sklearn.model_selection import KFold, cross_validate
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_breast_cancer
import numpy as np

# ── Load a real dataset ──────────────────────────────────────
data = load_breast_cancer()
X, y = data.data, data.target

# ── 5-fold CV with multiple metrics recorded ─────────────────
model = RandomForestClassifier(n_estimators=100, random_state=42)
kf    = KFold(n_splits=5, shuffle=True, random_state=42)

cv_results = cross_validate(
    model, X, y,
    cv=kf,
    scoring=['accuracy', 'roc_auc', 'f1'],
    return_train_score=True
)

for metric in ['accuracy', 'roc_auc', 'f1']:
    train_scores = cv_results[f'train_{metric}']
    val_scores   = cv_results[f'test_{metric}']
    print(
        f'{metric:<12}  '
        f'Train: {train_scores.mean():.3f} ± {train_scores.std():.3f}  '
        f'Val:   {val_scores.mean():.3f} ± {val_scores.std():.3f}'
    )
```

FOR AUTHOR USE ONLY

Code Snippet 1.5. Estimating generalization with k-fold cross-validation across several scoring metrics.

1.3.4 Stratified Splits and Temporal Splits

Standard random splits assume that the data is identically and independently distributed (i.i.d.). This assumption breaks into two common scenarios:

Imbalanced classes

If one class is rare (e.g.: fraud: 0.1% of transactions), a random split might put all fraud cases in the training set and none in the test set, or vice versa. Stratified splitting preserves the class proportions in each split.

```python
from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# skf.split(X, y) ensures each fold has ~50% of positives and ~50% of negatives
for fold, (train_idx, val_idx) in enumerate(skf.split(X, y), 1):
    print(f'Fold {fold}: train size={len(train_idx)}, val size={len(val_idx)}')
    print(f'  Positive rate — train: {y[train_idx].mean():.3f}, val: {y[val_idx].mean():.3f}')
```

Code Snippet 1.6. Preserving class proportions across folds with StratifiedKFold.

Time-series data

With temporal data, future information must never appear in training. A model that has 'seen' tomorrow's prices while predicting today's is called a leaky model. TimeSeriesSplit enforces the temporal ordering: training always precedes validation in time.

```python
from sklearn.model_selection import TimeSeriesSplit
import numpy as np

# Simulated time series data (e.g. 100 time steps)
X = np.arange(100).reshape(-1, 1)

tscv = TimeSeriesSplit(n_splits=5)

for fold, (train_idx, val_idx) in enumerate(tscv.split(X), 1):
    print(f'Fold {fold}: train [{train_idx[0]}–{train_idx[-1]}]  val [{val_idx[0]}–{val_idx[-1]}]')
```

Code Snippet 1.7. Enforcing temporal order in validation splits with TimeSeriesSplit.

1.3.5 Data Leakage: The Silent Failure

FOR AUTHOR USE ONLY

Data leakage occurs when information from outside the training distribution contaminates the model during training or evaluation. It is the single most common cause of models that look excellent in development but fail in production. Leakage can be subtle:

- Fitting a scaler or imputer on the full dataset before splitting, then transforming both train and test with the same fitted object.
- Including features derived from the target variable (e.g., 'average salary in the same zip code as this applicant' computed using the full dataset including the applicant's own record).
- Selecting features based on their correlation with the target on the full dataset before splitting.
- For time series: using any information from after the prediction timestamp.

The standard defense against leakage is to use the sklearn's Pipelines package, which guarantee that all data-dependent transformations are fitted only on training data and then applied (without refitting) to validation and test data.

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer

X, y = load_breast_cancer(return_X_y=True)

# Split FIRST — before any transformation
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# The Pipeline ensures the scaler is fitted on X_train only.
# When we call .predict(X_test), the scaler uses the training statistics.
pipe = Pipeline([
    ('impute',  SimpleImputer(strategy='median')),   # fill missing values
    ('scale',   StandardScaler()),                   # z-score normalisation
    ('model',   GradientBoostingClassifier(
                    n_estimators=100,
                    max_depth=3,
                    random_state=42
                ))
])

pipe.fit(X_train, y_train)

print(f'Train accuracy: {pipe.score(X_train, y_train):.4f}')
print(f'Test  accuracy: {pipe.score(X_test,  y_test):.4f}')
```

FOR AUTHOR USE ONLY

Code Snippet 1.8. Wrapping preprocessing and the model in a single Pipeline to prevent data leakage.

1.3.6 Hyperparameter Tuning Within the Pipeline

Hyperparameters are configuration choices that are not learned from data but set by the engineer: learning rate, tree depth, number of neurons, regularization strength. They must be tuned using only the validation set (or cross-validation on the training set). Grid search and randomized search automate this process.

```python
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint, uniform

param_dist = {
    'model__n_estimators':   randint(50, 300),
    'model__max_depth':       randint(2, 8),
    'model__learning_rate':   uniform(0.01, 0.3),
    'model__subsample':       uniform(0.6, 0.4),
}

# CV=5 inner loop: finds best hyperparameters on training data only
search = RandomizedSearchCV(
    pipe,
    param_distributions=param_dist,
    n_iter=30,
    cv=5,
    scoring='roc_auc',
    random_state=42,
    n_jobs=-1
)

search.fit(X_train, y_train)

print('Best hyperparameters:', search.best_params_)
print(f'Best CV ROC-AUC:  {search.best_score_:.4f}')
# Report final performance on the held-out test set (used ONCE)
print(f'Test ROC-AUC:     {search.score(X_test, y_test):.4f}')
```

Code Snippet 1.9. Tuning hyperparameters with RandomizedSearchCV.

FOR AUTHOR USE ONLY

1.4 Evaluation Metrics Overview

1.4.1 Why the Right Metric Matters

Choosing the wrong evaluation metric is one of the most consequential mistakes in applied machine learning. A model that achieves 99% accuracy on a fraud detection task where 99% of transactions are legitimate has learned nothing, it simply predicts 'not fraud' every time. Accuracy, the most intuitive metric, is also often the most misleading.

The choice of metric should be driven by the business or scientific objective, not by what sounds impressive. Before picking a metric, answer: what is a false positive cost? What is a false negative cost? Are all errors equally bad? Is the output a hard label or a calibrated probability?

1.4.2 The Confusion Matrix

For binary classification, all standard metrics are derived from a single 2x2 table called the confusion matrix. Every prediction falls into one of four cells:

Table 1.3. The binary-classification confusion matrix and its four outcomes.

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN) - Type II error
Actual Negative	False Positive (FP) - Type I error	True Negative (TN)

Every metric below is a function of TP, FP, TN, and FN. Understanding the matrix is therefore the prerequisite for understanding every metric.

1.4.3 Classification Metrics

Accuracy $$Accuracy = \frac{(TP+TN)}{(TP+FP+TN+FN)}, \quad (Eq.1.2)$$

The proportion of all predictions that are correct. Intuitive and easy to communicate, but unreliable when classes are imbalanced. Avoid as the primary metric whenever one class is significantly more common than the other.

Precision $$Precision = \frac{TP}{(TP+FP)}, \quad (Eq.1.3)$$

Of all the cases the model flagged as positive, what fraction were? High precision means few false alarms. This metric is particularly important in settings where false positives are costly, such as blocking legitimate financial transactions or triggering unnecessary medical interventions.

Recall $$Recall\ (Sensitivity) = \frac{TP}{(TP+FN)}, \quad (Eq.1.4)$$

Of all the actual positives, what fraction did the model catch? High recall means few missed cases. Critical for applications, where missing a positive is costly, such as failing to detect cancer or missing a security breach.
Precision and recall are in tension: increasing the decision threshold improves precision (you only flag when very confident) but reduces recall (you miss more positives). This tension is quantified by the precision–recall curve.

F1 Score

$$F1 = 2 \times \frac{(Precision \times Recall)}{(Precision+Recall)}, \quad (Eq.1.5)$$

FOR AUTHOR USE ONLY

The harmonic mean of precision and recall, giving equal weight to both. Useful as a single scalar summary when neither precision nor recall can be sacrificed. For asymmetric costs, the Fβ score generalizes this: β > 1 weights recall more; β < 1 weights precision more.

ROC-AUC

The Receiver Operating Characteristic (ROC) curve plots True Positive Rate (= Recall) against False Positive Rate (= FP / (FP + TN)) as the decision threshold varies from 0 to 1. The Area Under the Curve (AUC) summarizes the entire curve as a single number between 0 and 1. An AUC of 0.5 means the model is no better than random; an AUC of 1.0 means it perfectly ranks all positives above all negatives.

AUC is particularly valuable because it is threshold-independent: it measures how well the model ranks examples, not whether a specific threshold produces good predictions. It is robust to class imbalance at the ranking level.

1.4.4 Code Example: Computing and Visualizing Classification Metrics

FOR AUTHOR USE ONLY

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import (confusion_matrix, classification_report,
    roc_curve, roc_auc_score,
    precision_recall_curve, average_precision_score,
    ConfusionMatrixDisplay
)

# ── Data and model ──────────────────────────────────────────
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)
model = GradientBoostingClassifier(random_state=42)
model.fit(X_train, y_train)

y_pred      = model.predict(X_test)
y_prob      = model.predict_proba(X_test)[:, 1]

```

```python
# ── Confusion matrix ──────────────────────────────────────────
disp = ConfusionMatrixDisplay(
    confusion_matrix(y_test, y_pred),
    display_labels=['Malignant', 'Benign']
)
disp.plot(cmap='Blues')
plt.title('Confusion Matrix')
plt.tight_layout()
plt.show()

# ── Full classification report ────────────────────────────────
print(classification_report(y_test, y_pred,
    target_names=['Malignant', 'Benign']))

# ── ROC and Precision-Recall curves side by side ──────────────
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# ROC
fpr, tpr, _ = roc_curve(y_test, y_prob)
auc_score   = roc_auc_score(y_test, y_prob)
ax1.plot(fpr, tpr, lw=2, label=f'AUC = {auc_score:.3f}')
ax1.plot([0, 1], [0, 1], 'k--', lw=1, label='Random baseline')
ax1.set_xlabel('False Positive Rate')
ax1.set_ylabel('True Positive Rate (Recall)')
ax1.set_title('ROC Curve')
ax1.legend()

# Precision-Recall
prec, rec, _ = precision_recall_curve(y_test, y_prob)
ap_score      = average_precision_score(y_test, y_prob)
ax2.plot(rec, prec, lw=2, label=f'Avg Precision = {ap_score:.3f}')
ax2.axhline(
    y=y_test.mean(), color='k', linestyle='--', lw=1,
    label='Random baseline'
)
ax2.set_xlabel('Recall')
ax2.set_ylabel('Precision')
ax2.set_title('Precision-Recall Curve')
ax2.legend()

plt.tight_layout()
```

FOR AUTHOR USE ONLY

Code Snippet 1.10. Computing and visualizing the core classification metrics.

1.4.5 Regression Metrics

When the model output is a continuous value rather than a class label, classification metrics are no longer applicable. The most common regression metrics are:

Table 1.4. Common regression metrics with their formulas, units, and key properties.

Metric	Formula	Units	Key Property
MAE	mean(|y − ŷ|)	Same as target	Robust to outliers; intuitive
MSE	mean((y − ŷ)²)	Squared	Penalizes large errors heavily; differentiable
RMSE	√MSE	Same as target	Interpretable scale; penalizes outliers
R^2	1 − SS_res / SS_tot	Dimensionless [0,1]	Fraction of variance explained; scale-free
MAPE	mean(|y − ŷ| / |y|)	Percentage	Relative error; undefined if y = 0

The choice between these depends on the application. If large errors are disproportionately costly, use MSE or RMSE. If all errors should be treated equally regardless of magnitude, use MAE. If you need a scale-free comparison across datasets, use R^2 or MAPE.

FOR AUTHOR USE ONLY

```python
from sklearn.metrics import (mean_absolute_error, mean_squared_error,r2_score, mean_absolute_percentage_error)
from sklearn.linear_model import Ridge
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np

housing = fetch_california_housing()
X, y    = housing.data, housing.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
scaler  = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test  = scaler.transform(X_test)

model   = Ridge(alpha=1.0)
model.fit(X_train, y_train)
y_pred  = model.predict(X_test)

mae  = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2   = r2_score(y_test, y_pred)
mape = mean_absolute_percentage_error(y_test, y_pred)

print(f'MAE:   {mae:.3f}  (median prediction is off by ${mae * 100_000:,.0f})')
print(f'RMSE:  {rmse:.3f}  (large errors penalised more than MAE)')
print(f'R²:    {r2:.3f}  (model explains {r2:.1%} of variance in house prices)')
print(f'MAPE:  {mape:.3f}  (average relative error: {mape:.1%})')
```

Code Snippet 1.11. Computing the common regression metrics (MAE, MSE, RMSE, R^2, MAPE).

1.4.6 Metrics for Imbalanced Problems

When classes are heavily imbalanced (e.g., 1% positives), standard accuracy is misleading and even AUC can be optimistic. The Precision-Recall AUC (also called Average Precision) is often the more informative metric, as it focuses entirely on how well the model retrieves the rare positive class.

The Matthews Correlation Coefficient (MCC) is another strong choice: it produces a balanced measure even when one class vastly outnumbers the other, ranging from −1 (perfectly inverted predictions) to +1 (perfect predictions).

```python
from sklearn.metrics import matthews_corrcoef, average_precision_score
import numpy as np

# Simulated highly imbalanced dataset: 2% positive rate
np.random.seed(1)
n       = 10_000
y_true  = (np.random.rand(n) < 0.02).astype(int)  # 2% positives

# Naive classifier: always predicts negative
y_naive = np.zeros(n, dtype=int)


# Decent classifier: captures most positives with some false alarms
y_prob_smart   = np.where(y_true == 1,
    np.random.uniform(0.6, 1.0, n),   # high scores for positives
    np.random.uniform(0.0, 0.3, n)    # low scores for negatives
)
y_smart = (y_prob_smart > 0.5).astype(int)

print('--- Naive classifier (always predicts negative) ---')
print(f'  Accuracy:  {(y_naive == y_true).mean():.4f}  ← looks great, is useless')
print(f'  MCC:       {matthews_corrcoef(y_true, y_naive):.4f}  ← reveals the truth')
print()
print('--- Smart classifier ---')
print(f'  Accuracy:  {(y_smart == y_true).mean():.4f}')
print(f'  MCC:       {matthews_corrcoef(y_true, y_smart):.4f}')
print(f'  Avg Prec:  {average_precision_score(y_true, y_prob_smart):.4f}')
```

FOR AUTHOR USE ONLY

Code Snippet 1.12. Evaluating an imbalanced problem with the Matthews correlation coefficient.

1.4.7 A Guide to Choose the Right Metric

Table 1.5. Recommended evaluation of metrics for different modelling scenarios.

Scenario	Recommended Metric(s)	Why
Balanced binary classification	F1, ROC-AUC	Both precision and recall matter equally
High cost of false positives	Precision, Precision-Recall AUC	Minimize false alarms

Scenario	Recommended Metric(s)	Why
High cost of false negatives	Recall, ROC-AUC	Minimize missed cases
Heavily imbalanced classes	MCC, Avg Precision (PR-AUC)	Accuracy and ROC can mislead
Continuous output (regression)	RMSE if outliers matter; MAE otherwise	Choose penalty structure
Scale-free comparison across tasks	R^2, MAPE	Dimensionless; comparable across domains
Ranking quality (search, recs)	NDCG, MAP, Precision@K	Rewards correct ordering, not just labels
Probabilistic output quality	Log-loss (cross-entropy), Brier score	Measures calibration of probabilities

No single metric tells the full story. In practice, always report a primary metric that aligns with the business objective and secondary metrics that reveal different failure modes. A model with excellent AUC but poor calibration will make bad decisions in any system that uses its output as a probability. A model with high recall but poor precision will create alert fatigue in a security operations center. Know what each number means and what it does not measure.

1.4.8 All Metrics Summary

Table 1.6. A summary of every metric with its task, range, and direction of improvement.

FOR AUTHOR USE ONLY

Metric	Task	Range	Higher is better?
Accuracy	Classification	0 – 1	Yes (but misleading when imbalanced)
Precision	Classification	0 – 1	Yes (fewer false alarms)
Recall	Classification	0 - 1	Yes (fewer missed positives)
F1 Score	Classification	0 - 1	Yes
ROC-AUC	Classification	0 - 1	Yes (0.5 = random)
PR-AUC / Avg Prec	Classification	0 - 1	Yes (especially for imbalanced)
MCC	Classification	−1 -	Yes (+1 = perfect)
MAE	Regression	≥ 0	No (lower = better)
RMSE	Regression	≥ 0	No (lower = better)
R^2	Regression	≤ 1	Yes (1 = perfect fit)
MAPE	Regression	≥ 0%	No (lower = better)
Log-loss	Probabilistic	≥ 0	No (lower = better, 0 = perfect)

Chapter 2 - Baseline Models and Simple Classifiers

2.1 What are Baseline Models?

Before deploying any sophisticated machine learning model, we need a reference point: something to compare against that tells you whether all that complexity is paying off. This is the role of a baseline model.

A baseline model is the simplest possible classifier or predictor you can build for a problem. It does not use clever algorithms, feature engineering, or hyperparameter tuning. Its purpose is not to solve the problem well; it is to define the floor. Any model worth deploying must beat the baseline. If it cannot, the model has not learned anything useful, and the added cost in data, computation, and complexity is unjustified.
This module introduces the concept of baselines through two of the most fundamental classifiers in machine learning: ZeroR and OneR. By the end, you will understand not only how these classifiers work, but why starting simple is one of the most important habits a data scientist can develop.

FOR AUTHOR USE ONLY

2.1.1 Defining a Baseline

A baseline model is the minimum viable benchmark against which all other models are measured. In classification tasks, the simplest possible baseline is one that ignores all input features entirely and makes predictions using only what it knows about the distribution of target classes in the training data.
Think of it as the answer to the question: what is the best result I can achieve without any real learning? If you can score 80% accuracy just by always predicting the most common class, then a model that achieves 83% accuracy has only improved by 3 percentage points, and the question becomes whether that marginal improvement justifies the complexity.
Baselines can take many forms depending on the task:

- For classification: always predict the majority class (ZeroR) or use a single rule (OneR).
- For regression: always predict the mean or median of the training labels.
- For time series: predict that tomorrow will be the same as today (naive forecast).
- For NLP: always predict the most frequent label in the training set.

2.1.2 Why Baseline Models Matter

Baseline models serve several essential functions in a machine learning workflow:

Establishing a Performance Floor

Without a baseline, you have no frame of reference. A model achieving 75% accuracy sounds impressive until you learn that always predicting the majority class would give you 74%. The baseline exposes how little your complex model has learned.

Detecting Data and Evaluation Problems

If your sophisticated model performs worse than a baseline, something is wrong, not with the baseline, but with your pipeline. This might indicate data leakage, incorrect labels, a bug in your evaluation code, or a mismatch between training and test distributions. The baseline provides a sanity check.

Quantifying the Value of Complexity

Every additional layer of complexity in a model, more features, deeper architectures, longer training has a cost in time, computation, and maintainability. Baselines help you measure the return on that investment. If a simple rule achieves 80% accuracy and your neural network achieves 82%, you must decide whether 2 percentage points is worth the overhead.

FOR AUTHOR USE ONLY

Communicating Results Clearly

When presenting results to stakeholders, reporting performance relative to a baseline is far more meaningful than reporting an absolute number. 'Our model achieves 91% recall, compared to 61% for the majority class baseline' conveys genuine progress. '91% recall' alone does not.

2.1.3 The Hierarchy of Model Complexity

A disciplined modelling workflow moves from simple to complex, evaluating each step against the last:

- Start with ZeroR (majority class prediction): the absolute floor.
- Move to OneR (single-attribute rule): the first step toward meaningful learning.
- Try simple linear models (logistic regression, linear SVM): interpretable and fast.
- Graduate to ensemble methods (random forests, gradient boosting): strong general-purpose models.

- Add deep learning or custom architectures only when simpler approaches are insufficient.

This progression is not just pedagogical. It is practical engineering. Complex models are harder to debug, more expensive to train, and more prone to overfitting on small datasets. If a simple model is sufficient, using it is the correct decision.

The key takeaway here is that a baseline is not a weak model to be embarrassed about. It is a rigorous benchmark. The job of every more complex model is to justify its existence by outperforming it.

2.1.4 Class Imbalance and Baseline Accuracy

Consider a fraud detection dataset where 99% of transactions are legitimate and 1% are fraudulent. A classifier that always predicts 'not fraud' achieves 99% accuracy. This number looks excellent. It is, in practice, catastrophically useless, it never catches any fraud.

This is why understanding your class distribution before choosing an evaluation metric is essential. Majority class baselines reveal this trap immediately:

FOR AUTHOR USE ONLY

```python
import numpy as np
from sklearn.metrics import accuracy_score

# Simulated imbalanced dataset: 99% negative, 1% positive
np.random.seed(42)
n = 10_000
y_true = (np.random.rand(n) < 0.01).astype(int)  # 1% positives

# Majority class baseline: always predict 0 (negative)
majority_class = np.bincount(y_true).argmax()
y_pred_baseline = np.full(n, majority_class)

print(f'Class distribution: {np.bincount(y_true)}')
print(f'Majority class: {majority_class}')
print(f'Baseline accuracy: {accuracy_score(y_true, y_pred_baseline):.4f}')
# Output: Baseline accuracy: 0.9900  <-- looks great, detects no fraud
```

Code Snippet 2.1. A majority-class baseline on an imbalanced dataset.

This demonstrates why accuracy is often the wrong metric for imbalanced problems. The baseline achieves near-perfect accuracy by doing nothing. Any classifier that attempts to identify fraud will likely have lower raw accuracy but far more value. See the Evaluation Metrics module for alternatives such as precision, recall, F1, and MCC.

2.1.5 Baseline Accuracy as a Function of Class Distribution

The mathematical relationship is straightforward: if the most common class accounts for a proportion p of the dataset, the majority class baseline achieves accuracy p. The table below illustrates how this plays out across different distributions:

Table 2.1. How the majority-class baseline accuracy varies with class distribution.

Dataset Type	Class Distribution	Baseline Accuracy	Interpretation
Perfectly balanced (binary)	50% / 50%	50%	Random chance — any model must do better
Mildly imbalanced	70% / 30%	70%	Moderate floor — meaningful improvement needed
Significantly imbalanced	90% / 10%	90%	High floor — model must work hard to add value
Severely imbalanced	99% / 1%	99%	Accuracy is meaningless — use recall / MCC
Multiclass balanced (3 classes)	33% / 33% / 33%	33%	Any real model must exceed random guessing
Multiclass imbalanced (3 classes)	60% / 30% / 10%	60%	Model must beat majority class significantly

FOR AUTHOR USE ONLY

One thing to keep in mind is to always compute the majority class baseline before fitting any model. If your model's accuracy is close to or below this number, it has not learned anything meaningful from the features.

2.2 ZeroR: The Simplest Classifier

2.2.1 What is ZeroR?

ZeroR (Zero Rule) is the most basic classification method in machine learning. The name reflects its strategy: it uses zero rules derived from the input

features. Instead, it looks only at the target variable, identifies the most frequent class, and predicts that class for every instance, unconditionally.
ZeroR was formally described in the context of baseline evaluation and remains the standard starting point for any classification task. Its defining properties are:

- It ignores all predictor variables (features) entirely.
- It always predicts the same class, whichever appears most in the training data.
- It requires no training in the conventional sense, just counting class frequencies.
- It is completely deterministic and completely interpretable.

2.2.2 How ZeroR Works

The algorithm has two steps. First, during 'training', it scans the target column and identifies the majority class. Second, during prediction, it returns that class for every query, regardless of the input.

```
18  # ZeroR rule: Find the majority class
19  majority_class = y.value_counts().idxmax()
20
21  # ZeroR rule: Predict the majority class for every instance
22  y_pred = [majority_class] * len(y_test)
```

Code Snippet 2.2. Implementing the ZeroR rule.

FOR AUTHOR USE ONLY

Using sklearn's DummyClassifier, ZeroR is available out of the box with strategy = 'most_frequent'. This is the standard way to implement and evaluate it in a production pipeline:

```python
from sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import train_test_split
import numpy as np

# Example: Titanic-like imbalanced binary classification
# Majority class (Did Not Survive) = 549 out of 891 records
np.random.seed(42)
n = 891
# Simulating: 62% did not survive (class 0), 38% survived (class 1)
y = np.random.choice([0, 1], size=n, p=[0.616, 0.384])
X = np.random.rand(n, 5)  # placeholder features (ZeroR ignores them)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# ZeroR using scikit-learn
zeror = DummyClassifier(strategy='most_frequent')
zeror.fit(X_train, y_train)
y_pred = zeror.predict(X_test)

print(f'ZeroR accuracy: {accuracy_score(y_test, y_pred):.4f}')
print(classification_report(y_test, y_pred, target_names=['Did Not Survive', 'Survived']))
```

Code Snippet 2.3. Implementing ZeroR with scikit-learn's DummyClassifier.

2.2.3 ZeroR Results on Three Datasets

Applying ZeroR to three well-known datasets illustrates how its performance depends entirely on class proportions:

FOR AUTHOR USE ONLY

```
----- ZeroR with Iris dataset -----
The majority class is 0 - setosa
ZeroR Rule Accuracy:  0.3333333333333333

----- ZeroR with Titanic dataset -----
The majority class is 0 - Did Not Survive
ZeroR Rule Accuracy:  0.5865921787709497

----- ZeroR with Diabetes dataset -----
The majority class is 0 - Negative Diagnosis
ZeroR Rule Accuracy:  0.6428571428571429
```

Output 2.1. ZeroR accuracy output for the Iris, Titanic, and Pima Diabetes datasets.

Table 2.2. ZeroR accuracy on the Iris, Titanic, and Pima Diabetes datasets.

Dataset	Majority Class	Class Distribution	ZeroR Accuracy
Iris	Setosa (class 0)	Balanced: 33% each	0.3333
Titanic	Did Not Survive (0)	549 / 891 = 61.6%	0.5866
Pima Diabetes	Negative Diagnosis (0)	500 / 768 = 65.1%	0.6429

The Iris result is revealing with three balanced classes, ZeroR achieves exactly the random-chance level of 33.3%. There is no majority class to exploit, so the model is no better than guessing. The Titanic and Diabetes datasets are imbalanced, giving ZeroR a meaningful accuracy score — but one that conceals the fact that it is systematically ignoring the minority class entirely.

2.2.4 When ZeroR 'High' Accuracy is Misleading

A ZeroR accuracy of 64% on the Diabetes dataset sounds reasonable until you realize that this model will never correctly identify a single diabetic patient. It predicts 'no diabetes' for everyone. In a medical context, this is not a useful model — it is a model that guarantees every diabetic patient is missed.

This is the core lesson of ZeroR: it exposes the gap between accuracy as a number and accuracy as a measure of usefulness. Always pair your ZeroR baseline with metrics sensitive to minority class performance; recall, F1, MCC, or Precision-Recall AUC.

The biggest takeaway should be that ZeroR's value is not its performance but its role as the floor. Any model that cannot beat ZeroR has failed to extract any useful information from the features. ZeroR makes this failure visible.

FOR AUTHOR USE ONLY

2.3 OneR: One Rule Classification

2.3.1 What is OneR?

OneR (One Rule) is the next step up from ZeroR. Instead of ignoring all features, it examines each feature individually and finds the single best rule, the one attribute whose rule produces the lowest error on the training data. The model then uses only that one rule for all predictions.

OneR was introduced by Robert Holte in 1993 in a landmark paper showing that simple, single-attribute rules perform surprisingly well on many real-world datasets. The insight was not that simple rules are always sufficient, but that the gap between them and far more complex models is often smaller than expected.

The key properties of OneR are:

- It uses exactly one feature for classification.
- For each value of the chosen feature, it predicts the most common class among training instances with that value.
- It selects the feature whose rule has the lowest total error rate.
- It is fully interpretable: the entire model is a single readable if-then rule.

2.3.2 How OneR Works Step by Step

The algorithm proceeds as follows:

1. **Build a rule per feature:** For every feature in the dataset, build a rule. For each unique value of that feature, find the most common class label among training examples with that value. That class label becomes the prediction for that value.
2. **Measure error rate:** For each feature's rule, calculate the total error rate on the training data; the proportion of instances that the rule misclassifies.
3. **Select the best feature:** Choose the feature whose rule has the lowest total error rate. That feature and its rule become the classifier.
4. **Predict:** For any new instance, look up its value for the chosen feature and return the corresponding class prediction.

```python
# Function to create the OneR model
def OneR(X, y):
    best_feature = None
    best_rules = {}
    best_accuracy = 0

    for column in X.columns:
        # Create rules for each value of the column
        rules = {}
        for value in X[column].unique():
            most_common_label = y[X[column] == value].mode()[0]
            rules[value] = most_common_label

        # Apply the rules
        y_pred = X[column].map(rules)

        # Calculate accuracy
        accuracy = accuracy_score(y, y_pred)
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_feature = column
            best_rules = rules

    return best_feature, best_rules, best_accuracy
```

FOR AUTHOR USE ONLY

Code Snippet 2.4. OneR implementation that selects the single best-performing feature rule.

2.3.3 Pre-processing Requirements for OneR

OneR operates on categorical features. Continuous numerical attributes must be discretized into bins before the algorithm can build rules. The standard approach is quantile-based binning using pandas' qcut function, which ensures each bin contains roughly equal numbers of instances:

```python
import pandas as pd

# For continuous features, discretize into categorical bins
# qcut divides data into 4 equal-frequency bins (quartiles)
for column in column_names:
    iris_df[column] = pd.qcut(iris_df[column], 4)
```

Code Snippet 2.5. Discretizing continuous features into quantile bins.

For example, the Iris dataset's Petal Length feature is transformed as follows:

Table 2.3. Quantile binning of the Iris petal-length feature.

Petal Length (Before Discretization)	Petal Length (After Discretization)
1.4	(0.999, 1.6]
1.3	(0.999, 1.6]
1.7	(1.6, 4.35]
5.0	(4.35, 5.1]
5.4	(5.1, 6.9]

FOR AUTHOR USE ONLY

There is a second pre-processing concern: ID-like features. Any column with unique values per row (passenger IDs, names, ticket numbers) will allow OneR to memorize the training data perfectly, achieving 100% training accuracy but will generalize to nothing. These columns must be removed before applying OneR:

```python
# Drop ID-like features or columns with high cardinality
# to avoid overfitting (these enable perfect training accuracy
# but zero generalisation)
titanic_df = titanic_df.drop(columns=['PassengerId', 'Name', 'Ticket'])
```

Code Snippet 2.6. Removing high cardinality, before applying OneR to avoid trivial overfitting.

One important advantage: OneR handles missing values naturally. Empty values are treated as a separate attribute value during rule generation, there is no need to impute or remove them.

2.3.4 OneR Results on Three Datasets

Applying OneR to the same three datasets reveals what a single well-chosen rule can achieve:

```
----- OneR with Iris dataset -----
OneR Rule - Best Feature = petal length (cm)
  If petal length (cm) == (0.999, 1.6] → class = setosa
  If petal length (cm) == (4.35, 5.1] → class = versicolor
  If petal length (cm) == (1.6, 4.35] → class = versicolor
  If petal length (cm) == (5.1, 6.9] → class = virginica
OneR Rule Train Accuracy:  0.85
OneR Rule Test Accuracy:  0.8666666666666667

----- OneR with Titanic dataset -----
OneR Rule - Best Feature = Sex
  If Sex == male → class = Did Not Survive
  If Sex == female → class = Survived
OneR Rule Train Accuracy:  0.7879213483146067
OneR Rule Test Accuracy:  0.7821229050279329

----- OneR with Diabetes dataset -----
OneR Rule - Best Feature = Glucose
  If Glucose == (-0.001, 99.0] → class = Negative Diagnosis
  If Glucose == (99.0, 117.0] → class = Negative Diagnosis
  If Glucose == (117.0, 140.25] → class = Negative Diagnosis
  If Glucose == (140.25, 199.0] → class = Positive Diagnosis
OneR Rule Train Accuracy:  0.744299674267101
OneR Rule Test Accuracy:  0.7467532467532467
```

Output 2.2. OneR rules and accuracy output for the Iris, Titanic, and Diabetes datasets.

Iris Dataset - Best Feature: Petal Length

OneR identifies petal length as the most discriminating feature. The learned rule maps petal length bins to species:

Titanic Dataset - Best Feature: Sex

For the Titanic survival prediction task, sex emerges as the single most predictive attribute — reflecting the historical 'women and children first' evacuation policy:

Pima Diabetes Dataset - Best Feature: Glucose

For diabetes prediction, glucose concentration is identified as the best single predictor. The rule maps glucose ranges to diagnosis outcomes:

This rule has real medical intuition: elevated glucose (above 140 mg/dL fasting) is a clinical marker for diabetes. OneR has, through pure statistical frequency counting, recovered a medically meaningful threshold.

FOR AUTHOR USE ONLY

2.3.5 Comparing ZeroR and OneR

The results across all three datasets demonstrate a consistent pattern: OneR meaningfully outperforms ZeroR by leveraging a single informative feature.

Table 2.4. ZeroR versus OneR accuracy across the three datasets.

Dataset	ZeroR Accuracy	OneR Accuracy	Improvement	OneR Best Feature
Iris	0.3333	0.8667	+0.5334	Petal Length
Titanic	0.5866	0.7821	+0.1955	Sex

Pima Diabetes	0.6429	0.7468	+0.1039	Glucose

The most dramatic improvement is on the Iris dataset: accuracy jumps from 33% (random chance) to 87% using a single petal length rule. This illustrates how a good feature selection, even with the simplest possible rule can capture most of the signal in the data.

It is equally instructive to observe that even the smallest improvement (Diabetes: +10.4%) is meaningful. A single glucose threshold rule correctly identifies 74-75% of diabetic patients that ZeroR misses entirely.

2.3.6 Strengths and Limitation of OneR

Table 2.5. Strengths and limitations of the OneR classifier.

Aspect	Strengths	Limitations
Interpretability	Entire model is one readable rule	Rule may oversimplify complex patterns
Speed	Training is near-instantaneous	Must discretize continuous features first
Data requirements	Works on very small datasets	Relies on a single attribute — misses interactions
Missing values	Treated as a separate category automatically	May produce noisy rules on sparse values
Overfitting risk	Low — rule is deliberately simple	High-cardinality features can cause overfitting
Use case	Excellent baseline; interpretable production rule	Unlikely to match ensemble or deep learning models

FOR AUTHOR USE ONLY

2.3.7 ZeroR and OneR as Part of a Modelling Workflow

In practice, ZeroR and OneR are always the first steps, never the last. The workflow looks like this:

- Compute ZeroR accuracy: establish the absolute floor.
- Compute OneR accuracy and note which feature it selects this is free feature importance information.
- Fit a logistic regression or simple decision tree: if it barely beats OneR, investigate your data quality.
- Escalate to more complex models only when simpler ones have been thoroughly evaluated.

The OneR best feature is also a useful diagnostic. If OneR selects an obvious proxy for the target variable (e.g., a feature that is essentially a re-encoding of the label), it signals a data leakage problem. If it selects a feature, you did not expect, it may point to an interesting signal worth investigating more deeply.
ZeroR and OneR are best served as minimum baselines when tackling a supervised classification problem. If more complex models do not clearly outperform them, those complex models are not earning their cost. The trade-off between accuracy and simplicity is what makes ZeroR and OneR indispensable benchmarks.

2.4. Summary: Baseline Classifiers - Summary

Table 2.6. Side-by-side summary of the ZeroR and OneR classifiers.

Property	ZeroR	OneR
Features used	None	One (the best single feature)
Rule type	Always predict majority class	Lookup table: feature value → class
Training required	Count class frequencies only	Evaluate all features; select lowest error
Interpretability	Trivially interpretable	Single if-then rule; fully readable
Pre-processing needed	None	Discretize continuous features; remove ID columns
Handles missing values	Not applicable (ignored features)	Yes — treated as a separate category
Typical use	Absolute performance floor	First meaningful baseline; free feature importance
Typical accuracy range	Equals majority class proportion	Meaningfully above ZeroR on most datasets
Main limitation	Detects nothing about minority classes	Ignores feature interactions

FOR AUTHOR USE ONLY

Chapter 3 – Linear Models

3.1. Linear Regression

3.1.1. What is Linear Regression?

Linear regression is one of the oldest and most widely used algorithms in statistics and machine learning. Its goal is straightforward: given a set of input features, learn a linear mapping to a continuous output variable. Despite its simplicity, it forms the mathematical backbone of many more advanced methods and remains a strong baseline in practice.

In a regression problem we are given a dataset of n observations, each described by p features. We want to find a function that predicts a continuous target y from an input vector x. Linear regression assumes this function is a weighted sum of the input features plus a bias term:

$$\hat{Y} = w_0 + w_1x_1 + w_2x_2 + \ldots + w_px_p \,, \qquad \textit{(Eq.3.1)}$$

In vector notation: $\hat{y} = w^\top x$ (where $x_0 = 1$ absorbs the bias w_0)

Here w is the weight vector (also called coefficients or parameters) and ŷ is the predicted value. The task of training the model is to find the w that makes predictions as accurate as possible on the training data.

FOR FURTHER USE ONLY

3.1.2. The Ordinary Least Squares (OLS) Objective

The standard way to measure prediction error in regression is Mean Squared Error (MSE). Training the linear regression model means minimizing the sum of squared residuals, the differences between predicted and actual values:

$$loss(w) = \frac{\{1\}}{\{n\}}\sum_{i=1}^{n}(y_i - (w^Tx_i + b))^2 \,, \qquad \textit{(Eq.3.2)}$$

This formulation is called Ordinary Least Squares (OLS). Because the loss function is convex and quadratic, it has a unique global minimum with an exact closed-form solution:

$$w^{\cdot} = (X^TX)^{\{-1\}}X^Ty \;\text{ -> \# The Normal Equation,} \qquad \textit{(Eq.3.3)}$$

The Normal Equation gives the exact solution in one step without iteration. However, inverting an $n \times p$ matrix is computationally expensive for large datasets ($O(p^3)$ complexity), which is why gradient-based optimization (covered in Section 2) is preferred at scale.

3.1.3. Assumptions of Linear Regression

Linear regression makes several assumptions that practitioners should always check:

- Linearity: the relationship between features and target is linear.

- Independence: observations are independent of each other.
- Homoscedasticity: the variance of residuals is constant across all values of the predictors.
- Normality of residuals: residuals are approximately normally distributed (important for inference, less critical for prediction).
- No multicollinearity: predictors are not perfectly correlated with each other (addressed by regularization in Section 4).

3.1.4. Polynomial Feature Expansion

Pure linear regression can only model straight-line relationships. When the true relationship is curved or involves interactions between features, polynomial feature expansion allows linear regression to fit non-linear patterns while keeping the model mathematically linear in its parameters:

Original features: $[x_1, x_2]$

Degree-2 expansion: $[x_1, x_2, x_1^2, x_2^2, x_1 \cdot x_2]$

The model is still linear in the expanded features:

$$\hat{y} = w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1x_2, \qquad \textit{(Eq.3.4)}$$

The trade-off is that expansion dramatically increases the number of features (for p features at degree d, the count grows as $O(p^d/d!)$), making regularization essential to prevent overfitting in high-dimensional spaces.

FOR AUTHOR USE ONLY

3.1.5. Code Example: Linear Regression on Rental Data

The following pipeline mirrors the preprocessing used in a rent prediction study on German housing data: log-transforming the skewed target variable, expanding polynomial features, and fitting a linear model.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, r2_score

# Simulate rental data (mirrors immo_data structure)
np.random.seed(42)
n = 1000
rooms         = np.random.randint(1, 6, n).astype(float)
area          = rooms * 20 + np.random.normal(0, 10, n)   # sqm
year_built    = np.random.randint(1960, 2023, n).astype(float)
has_balcony   = np.random.randint(0, 2, n).astype(float)

# True rent/sqm: non-linear relationship + noise
rent_per_sqm  = (8 + 0.05 * rooms**2 + 0.003 * (year_built - 1960)
                 + 1.5 * has_balcony + np.random.normal(0, 1.2, n))
rent_per_sqm  = np.clip(rent_per_sqm, 4, 35)   # realistic range

X = np.column_stack([rooms, area, year_built, has_balcony])
y = rent_per_sqm

# Log-transform target to address right-skewness
y_log = np.log1p(y)

X_train, X_test, y_log_train, y_log_test, y_train, y_test = train_test_split(
    X, y_log, y, test_size=0.2, random_state=42
)

# Pipeline: polynomial expansion → scale → fit OLS
pipe = Pipeline([
    ('poly',   PolynomialFeatures(degree=2, include_bias=False)),
    ('scaler', StandardScaler()),
    ('model',  LinearRegression())
])

pipe.fit(X_train, y_log_train)

# Evaluate: transform predictions back to original scale
y_pred_log = pipe.predict(X_test)
y_pred     = np.expm1(y_pred_log)   # inverse of log1p

mae = mean_absolute_error(y_test, y_pred)
r2  = r2_score(y_test, y_pred)
print(f'MAE:  {mae:.3f} EUR/m²')
print(f'R²:   {r2:.4f}')

# Plot predicted vs actual
plt.figure(figsize=(6, 5))
plt.scatter(y_test, y_pred, alpha=0.4, s=15)
lims = [y_test.min(), y_test.max()]
plt.plot(lims, lims, 'r--', lw=1.5, label='Perfect prediction')
plt.xlabel('Actual Rent per m² (EUR)')
plt.ylabel('Predicted Rent per m² (EUR)')
plt.title('Linear Regression: Predicted vs. Actual')
plt.legend()
plt.tight_layout()
plt.show()
```

FOR AUTHOR USE ONLY

Code Snippet 3.1. Linear Regression on Rental Data.

3.2. Gradient Descent

3.2.1. Why We Need an Optimization Algorithm

The Normal Equation gives the exact solution to linear regression in a single step, but it requires inverting a ($p \times p$) matrix; an operation that scales as $O(p^3)$. When p reaches tens of thousands (as in polynomial-expanded rental datasets) or millions (as in NLP), direct inversion becomes computationally prohibitive.

Gradient descent is an iterative optimization algorithm that finds the minimum of a loss function by repeatedly moving in the direction of steepest descent. It is the engine behind almost every modern machine learning model, from linear regression to deep neural networks.

3.2.2. The Core Idea: Following the Slope Downhill

Imagine standing on a hilly landscape and wanting to reach the lowest valley. If you always take a step in the direction of the steepest downward slope, you will eventually reach a valley. Gradient descent does exactly this in the space of model parameters:

Repeat until convergence:

$$W \leftarrow W - \eta \cdot \nabla L(w), \qquad \textit{(Eq.3.5)}$$

Where:

w = current weight vector

η = learning rate (step size)

∇L(w) = gradient of the loss with respect to w

FOR AUTHOR USE ONLY

For MSE loss on linear regression:

$$\nabla L(w) = -(2/n)\, X^{\top}(y - Xw), \qquad \textit{(Eq.3.6)}$$

The learning rate η is the most critical hyperparameter. Too large and the updates overshoot the minimum, causing divergence. Too small and convergence are painfully slow. Finding a good learning rate or using an adaptive schedule is a central concern in practice.

3.2.3. The 3 Variants of Gradient Descent

Table 3.1. Comparison of Gradient Descent Variants.

Variant	Data per update	Pros	Cons
Batch GD	Full dataset	Stable, exact gradient	Slow for large n; needs all data in memory
Stochastic GD	1 sample	Fast updates; escapes local minima	Noisy; requires learning rate tuning

Variant	Data per update	Pros	Cons
Mini-batch GD	32–512 samples	Best of both; GPU-friendly	One more hyperparameter (batch size)

Mini-batch gradient descent is the standard in practice. Batch sizes of 32, 64, or 128 strike a good balance between gradient estimate quality and computational efficiency on modern hardware.

3.2.4. The Learning Rate: Convergence and Divergence

The learning rate is arguably the most important hyperparameter in gradient-based optimization. The concept below (reproduced conceptually through code) shows three characteristic regimes:

- η too small: loss decreases very slowly; many more iterations required.
- η just right: smooth, steady convergence to the minimum.
- η too large: loss oscillates or diverges; the algorithm never converges.

The following code illustrates all three regimes on a simple 1D quadratic loss, making the effect of the learning rate immediately visible:

FOR AUTHOR USE ONLY

```python
import numpy as np
import matplotlib.pyplot as plt

# Gradient descent on a simple 1D quadratic: L(w) = w^2
def loss(w):      return w ** 2
def gradient(w): return 2 * w

def run_gd(lr, n_steps=40, w0=8.0):
    w, history = w0, [w0]
    for _ in range(n_steps):
        w = w - lr * gradient(w)
        history.append(w)
    return np.array(history)

learning_rates = {'Too small (0.01)': 0.01,
                  'Just right (0.3)': 0.30,
                  'Too large (1.05)': 1.05}

plt.figure(figsize=(10, 4))
for label, lr in learning_rates.items():
    history = run_gd(lr)
    losses  = [loss(w) for w in history]
    plt.plot(losses[:30], label=label)

plt.xlabel('Iteration')
plt.ylabel('Loss  L(w) = w^2')
plt.title('Effect of Learning Rate on Gradient Descent Convergence')
plt.legend()
plt.yscale('log')
plt.tight_layout()
plt.show()
```

Code Snippet 3.2. Learning Rate Regimes on 1D Quadratic Loss.

3.2.5. Gradient Descent for Linear Regression

Applying mini-batch gradient descent to the OLS loss for linear regression is straightforward. The following implementation trains from scratch (no sklearn) so the update steps are fully visible:

```python
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Data
data = fetch_california_housing()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler  = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test  = scaler.transform(X_test)

# Add bias column (column of ones)
X_train_b = np.column_stack([np.ones(len(X_train)), X_train])
X_test_b  = np.column_stack([np.ones(len(X_test)),  X_test])

# Mini-batch gradient descent
np.random.seed(0)
w          = np.zeros(X_train_b.shape[1])   # initialise weights
lr         = 0.01                            # learning rate
batch_size = 64
n_epochs   = 50
n          = len(X_train_b)
loss_history = []

for epoch in range(n_epochs):
    # Shuffle training data each epoch
    idx = np.random.permutation(n)
    X_shuf, y_shuf = X_train_b[idx], y_train[idx]

    epoch_loss = 0
    for start in range(0, n, batch_size):
        Xb = X_shuf[start : start + batch_size]
        yb = y_shuf[start : start + batch_size]

        y_pred   = Xb @ w                          # forward pass
        residual = y_pred - yb
        grad     = (2 / len(Xb)) * Xb.T @ residual  # gradient of MSE
        w       -= lr * grad                       # weight update
        epoch_loss += (residual ** 2).mean()

    loss_history.append(epoch_loss / (n // batch_size))

# Evaluate
y_pred_test = X_test_b @ w
mae  = np.mean(np.abs(y_test - y_pred_test))
rmse = np.sqrt(np.mean((y_test - y_pred_test)**2))
print(f'Test MAE:  {mae:.4f}')
print(f'Test RMSE: {rmse:.4f}')
```

FOR AUTHOR USE ONLY

Code Snippet 3.3. Mini-batch Gradient Descent from Scratch.

3.2.6. Adaptive Learning Rate Methods

Manually tuning the learning rate is tedious. Modern optimizers adjust the learning rate automatically based on the history of gradients. The most widely used methods are:

- Momentum: accumulates a velocity in directions of persistent gradient, dampening oscillations.
- RMSProp: divides the learning rate by an exponential moving average of squared gradients, adapting per-parameter.
- Adam (Adaptive Moment Estimation): combines momentum and RMSProp. It is the default optimizer for most deep learning and is also available for sklearn models via SGDRegressor.

```python
from sklearn.linear_model import SGDRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error
import numpy as np

# SGDRegressor implements mini-batch gradient descent with various options
pipe = Pipeline([
    ('scale', StandardScaler()),
    ('sgd',   SGDRegressor(
                  loss='squared_error',   # MSE loss
                  penalty='none',         # no regularisation for now
                  eta0=0.01,              # initial learning rate
                  learning_rate='invscaling',  # η decays over time
                  max_iter=100,
                  random_state=42
              ))
])

pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
print(f'SGD MAE: {mean_absolute_error(y_test, y_pred):.4f}')
```

FOR AI TOR USE ONLY

Code Snippet 3.4. Adam Optimizer / Adaptive Learning Rate.

3.3. Logistic Regression

3.3.1. From Regression to Classification

Linear regression predicts a continuous value. Many practical problems, however, require a categorical decision: Is this email spam? Is this loan applicant likely to default? Is this rental listing fairly priced or overpriced? These are classification problems, and they call for a different modelling approach.

Despite its name, logistic regression is a classification algorithm. It extends the linear regression framework to predict the probability that an observation belongs to a particular class. Rather than outputting an unbounded real number, it passes the linear combination of features through a non-linear function; the sigmoid that squashes its output to the [0, 1] range.

3.3.2. The Sigmoid Function

The sigmoid (or logistic) function maps any real number to a probability:

$$\sigma(z) = 1 / (1 + e^{(-z)}) , \qquad \textit{(Eq.3.7)}$$

Where: $z = w^T x$ (the same linear combination as in linear regression)

$\sigma(z) \to 0$ as $z \to -\infty$

$\sigma(0) = 0.5$ (decision boundary)

$\sigma(z) \to 1$ as $z \to +\infty$

The predicted probability is $P(y=1 \mid x) = \sigma(w^T x)$. A threshold (usually 0.5) converts this probability to a binary label. Crucially, the threshold can be adjusted to trade precision against recall, a key advantage in imbalanced problems such as fraud detection or fairness auditing.

FOR AUTHOR USE ONLY

3.3.3. The Loss Function: Binary Cross-Entropy

Logistic regression cannot use MSE as its loss function; the combination of sigmoid and squared error creates a non-convex landscape with many local minima. Instead, it minimizes Binary Cross-Entropy (also called Log-Loss):

$$L(w) = -(1/n)\ \Sigma_i\ [\ y_i\ log(\hat{y}_i) + (1 - y_i)\ log(1 - \hat{y}_i)\ ] , \qquad \textit{(Eq.3.8)}$$

Where: $\hat{y}_i = \sigma(w^T x_i)$ is the predicted probability for observation i

Gradient (used for gradient descent):

$$\nabla L(w) = (1/n)\ X^T(\hat{y} - y), \qquad \textit{(Eq.3.9)}$$

Log-Loss heavily penalizes confident wrong predictions: if the model assigns probability 0.99 to the wrong class, the penalty approaches infinity. This pushes the model to be well-calibrated, not just directionally correct.

3.3.4. Interpretability: Reading the Coefficients

One of the key advantages of logistic regression is interpretability. Each coefficient w_j reflects the change in log-odds for a one-unit increase in feature x_j, holding all other features constant:

$$Log - odds: log(P / (1 - P)) = w^T x , \qquad \textit{(Eq.3.10)}$$

Odds ratio for feature j: $\exp(w_j)$

If $w_j > 0$: increasing x_j raises the probability of the positive class
If $w_j < 0$: increasing x_j lowers the probability of the positive class
If $w_j = 0$: feature x_j has no effect (can be pruned)

This transparency makes logistic regression well-suited to regulated decision contexts. In domains such as credit scoring, medical diagnosis, or housing fairness, decision-makers can inspect exactly which features drive a classification outcome; a requirement that black-box models cannot satisfy.

3.3.5. Dual-Model Framework: The Rent Fairness Use Case

A practical example of logistic regression as a second-stage classifier: a Ridge regression model first predicts the expected market rent per m² from structural features. A logistic regression model then classifies whether the observed rent is within 5% of that prediction (fair deal) or significantly above it (potentially overpriced).

The fairness label is constructed as follows:

FOR AUTHOR USE ONLY

```
# Stage 1 output: predicted market rent
predicted_log_price = ridge_model.predict(X_features)
predicted_price     = np.expm1(predicted_log_price)

# Fairness label: 1 = fair, 0 = potentially overpriced
df['is_fair_deal'] = (
    df['rent_per_sqm'] <= predicted_price * 1.05
).astype(int)
```

Code Snippet 3.5. Fairness label construction.

The logistic classifier is then trained to predict this label from the same structural features, learning which property characteristics are associated with fair pricing. The result is an interpretable probability score, the 'Fair Deal Probability' shown in the model output.

3.3.6. Code Example: Logistic Regression for Rent Fairness

```python
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, roc_auc_score

# Simulate rental dataset
np.random.seed(42)
n = 3000
rooms         = np.random.randint(1, 6, n).astype(float)
area          = rooms * 20 + np.random.normal(0, 8, n)
year_built    = np.random.randint(1960, 2023, n).astype(float)
has_lift      = np.random.randint(0, 2, n).astype(float)
has_balcony   = np.random.randint(0, 2, n).astype(float)
condition     = np.random.choice(['well_kept', 'normal', 'renovated'], n)

# Simulated predicted rent (Stage 1 output)
predicted_rent = 8 + 0.04*rooms**2 + 0.003*(year_built-1960) + 1.2*has_balcony

# Observed rent: some properties are overpriced
overpriced     = np.random.rand(n) < 0.4    # 40% overpriced
observed_rent  = np.where(overpriced,
    predicted_rent * np.random.uniform(1.1, 1.5, n),
    predicted_rent * np.random.uniform(0.9, 1.05, n)
)

is_fair = (observed_rent <= predicted_rent * 1.05).astype(int)

X = pd.DataFrame({
    'rooms': rooms, 'area': area, 'year_built': year_built,
    'has_lift': has_lift, 'has_balcony': has_balcony, 'condition': condition
})
y = is_fair

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# Preprocessing pipeline
numeric_features     = ['rooms', 'area', 'year_built']
binary_features      = ['has_lift', 'has_balcony']
categorical_features = ['condition']

numeric_transformer = Pipeline([
    ('poly',   PolynomialFeatures(degree=2, include_bias=False)),
    ('scaler', StandardScaler())
])

preprocessor = ColumnTransformer([
    ('num', numeric_transformer,     numeric_features),
    ('cat', OneHotEncoder(drop='first'), categorical_features),
    ('bin', 'passthrough',           binary_features)
])

```

FOR AUTHOR USE ONLY

```python
# Logistic regression with L2 regularisation
clf = Pipeline([
    ('preprocessor', preprocessor),
    ('interaction',  PolynomialFeatures(degree=2, interaction_only=True,
                                        include_bias=False)),
    ('classifier',   LogisticRegression(
                        class_weight='balanced',   # handles imbalance
                        C=0.1,                     # inverse regularisation strength
                        max_iter=2000
                    ))
])

clf.fit(X_train, y_train)

y_pred  = clf.predict(X_test)
y_prob  = clf.predict_proba(X_test)[:, 1]

print(classification_report(y_test, y_pred,
    target_names=['Overpriced', 'Fair Deal']))
print(f'ROC-AUC: {roc_auc_score(y_test, y_prob):.4f}')

# Predict for a new property
new_property = pd.DataFrame([{
    'rooms': 3, 'area': 65, 'year_built': 2011,
    'has_lift': 0, 'has_balcony': 1, 'condition': 'normal'
}])
prob_fair = clf.predict_proba(new_property)[0][1]
verdict   = 'Fair Deal' if prob_fair > 0.5 else 'Potentially Overpriced'
print(f'Fair Deal Probability: {prob_fair:.2%} → {verdict}')
```

Code Snippet 3.6. Logistic Regression for Rent Fairness.

FOR TUTOR USE ONLY

3.4. Regularization (L1 & L2)

3.4.1. The Problem: Overfitting and Multicollinearity

As models grow more complex through polynomial feature expansion, interaction terms, or simply large feature sets two related problems emerge. Overfitting occurs when the model learns the noise in the training data rather than the true underlying pattern, resulting in poor generalization. Multicollinearity occurs when predictors are highly correlated, causing the OLS estimator to produce wildly large, unstable coefficient estimates.

Both problems share the same root cause: unconstrained optimization allows the model too much freedom. Regularization is the principled solution, it adds a penalty term to the loss function that constrains the size of the model's parameters, forcing it to find a simpler solution that generalizes better.

$$Regularized\ Loss = Data\ Loss + \lambda \cdot Penalty(w)\ , \qquad (Eq.3.11)$$

λ (lambda) controls how strongly the penalty is applied:

λ = 0 → no regularization (standard OLS)

λ → ∞ → all weights driven to zero (constant prediction)

3.4.2. L2 Regularization: Ridge Regression

Ridge regression, introduced by Hoerl and Kennard (1970), adds the squared magnitude of the weight vector as a penalty. This is the L2 norm (Euclidean distance from the origin):

$$Loss_Ridge(w) = (1/n)\ ||y - Xw||^2 + \lambda\ ||w||^2\ , \qquad \textit{(Eq.3.12)}$$

Where:

$(1/n)\ ||y - Xw||^2$ = MSE (data fit)

$\lambda\ ||w||^2$ = L2 penalty

Closed-form solution: $w* = (X^TX + \lambda I)^{-1}\ X^Ty$

Note: the λI term makes X^TX invertible even when columns are collinear. This is the key advantage over OLS in high-dimensional datasets.

Ridge has several important properties:

- It shrinks all coefficients towards zero proportionally but never exactly to zero; all features are retained.
- It resolves multicollinearity by distributing weight evenly among correlated predictors rather than producing unstable large estimates.
- It has a closed-form solution (unlike Lasso), making it computationally very fast.
- In sklearn, the hyperparameter is called alpha (λ). Note that LogisticRegression uses C = 1/λ, a larger C means less regularization.

FOR AUTHOR USE ONLY

3.4.3. L1 Regularization: Lasso Regression

Lasso (Least Absolute Shrinkage and Selection Operator) replaces the squared penalty with the absolute value, the L1 norm:

$$Loss_Lasso(w) = (1/n)\ ||y - Xw||^2 + \lambda\ ||w||_1\ , \qquad \textit{(Eq.3.13)}$$

Where:

$(1/n)\ ||y - Xw||^2$ = MSE (data fit)

$\lambda\ ||w||_1$ = L1 penalty

$||w||_1 = \Sigma_j\ |w_j|$ (sum of absolute values)

The L1 penalty has a remarkable geometric property: it drives some coefficients to exactly zero, performing automatic feature selection. Features with coefficients shrunk to zero are completely excluded from the model.

- Lasso produces sparse models; useful when you have many features and suspect only a few are truly relevant.
- It does not have a closed-form solution (the L1 norm is not differentiable at zero); it requires iterative solvers such as coordinate descent.
- When predictors are highly correlated, Lasso arbitrarily picks one and zeros the rest (Ridge distributes the weight).

3.4.4. Elastic Net: Combining L1 and L2

Elastic Net blends both penalties with a mixing parameter ρ (called l1_ratio in sklearn)

$$Loss_ElasticNet(w) = MSE + \lambda \left[\rho \cdot ||w||_1 + (1 - \rho) \cdot ||w||^2 \right], \quad \textit{(Eq.3.14)}$$

ρ = 0 → pure Ridge
ρ = 1 → pure Lasso
0 < ρ < 1 → Elastic Net (recommended for correlated + sparse problems)

Elastic Net is particularly useful when you have groups of correlated features and want sparsity: it tends to select entire groups rather than arbitrarily picking one member as Lasso does.

3.4.5. Comparison: L1 vs L2 vs Elastic Net

Table 3.2. Comparison of L1, L2 and Elastic Net Regularization.

Property	Ridge (L2)	Lasso (L1)	Elastic Net
Penalty term	$\|\|w\|\|^2$	$\|\|w\|\|_1$	mix of both
Feature selection	No (shrinks all)	Yes (zeros out)	Yes (group selection)
Correlated features	Distributes weight	Selects one	Keeps groups
Closed-form solution	Yes	No	No
Sparsity of solution	Dense	Sparse	Sparse
Best use case	Multicollinearity	Many irrelevant features	Both problems present
sklearn class	Ridge / Ridge(C)	Lasso / L1 Logistic	ElasticNet

FOR AUTHOR USE ONLY

3.4.6. Choosing the Regularization Strength (λ)

The regularization hyperparameter λ (or its inverse C in logistic regression) must be tuned using cross-validation. Too little regularization and the model overfits; too much and it underfits. The code below demonstrates this search for a Ridge regression model:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.datasets import fetch_california_housing

X, y = fetch_california_housing(return_X_y=True)

# High-dimensional feature space: polynomial expansion creates ~45 features
base_pipe = make_pipeline(
    PolynomialFeatures(degree=2, include_bias=False),
    StandardScaler()
)

X_expanded = base_pipe.fit_transform(X)

alphas     = np.logspace(-3, 3, 40)   # search 10⁻³ to 10³
ridge_cvs  = []
lasso_cvs  = []

for alpha in alphas:
    ridge_cv = -cross_val_score(
        Ridge(alpha=alpha), X_expanded, y,
        cv=5, scoring='neg_root_mean_squared_error'
    ).mean()
    ridge_cvs.append(ridge_cv)

    lasso_cv = -cross_val_score(
        Lasso(alpha=alpha, max_iter=5000), X_expanded, y,
        cv=5, scoring='neg_root_mean_squared_error'
    ).mean()
    lasso_cvs.append(lasso_cv)

best_ridge_alpha = alphas[np.argmin(ridge_cvs)]
best_lasso_alpha = alphas[np.argmin(lasso_cvs)]
print(f'Best Ridge alpha: {best_ridge_alpha:.4f}  → CV RMSE: {min(ridge_cvs):.4f}')
print(f'Best Lasso alpha: {best_lasso_alpha:.4f}  → CV RMSE: {min(lasso_cvs):.4f}')

plt.figure(figsize=(8, 4))
plt.semilogx(alphas, ridge_cvs, 'b-o', markersize=4, label='Ridge')
plt.semilogx(alphas, lasso_cvs, 'r-s', markersize=4, label='Lasso')
plt.axvline(best_ridge_alpha, color='b', linestyle='--', alpha=0.5)
plt.axvline(best_lasso_alpha, color='r', linestyle='--', alpha=0.5)
plt.xlabel('Regularisation Strength (alpha, log scale)')
plt.ylabel('5-Fold CV RMSE')
plt.title('Choosing Regularisation Strength via Cross-Validation')
plt.legend()
plt.tight_layout()
plt.show()
```

FOR AUTHOR USE ONLY

Code Snippet 3.7. Ridge Regression Hyperparameter Search.

3.4.7. Regularization Paths: Visualizing Coefficient Shrinkage

A regularization path shows how each coefficient changes as λ varies from near-zero to very large. It is a powerful diagnostic tool that reveals which

features the model considers most important and how quickly each is shrunk to zero.

```python
from sklearn.linear_model import lasso_path, ridge_regression
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_diabetes
from sklearn.preprocessing import StandardScaler

X, y = load_diabetes(return_X_y=True)
X    = StandardScaler().fit_transform(X)

# ── Lasso path: which features survive as lambda decreases? ──────────
alphas_lasso, coefs_lasso, _ = lasso_path(X, y, eps=1e-3)

plt.figure(figsize=(8, 5))
for coef_path in coefs_lasso:
    plt.plot(np.log10(alphas_lasso), coef_path)

plt.xlabel('log₁₀(alpha)  [right = less regularisation]')
plt.ylabel('Coefficient Value')
plt.title('Lasso Regularisation Path (Diabetes Dataset)')
plt.axhline(0, color='k', lw=0.5, linestyle='--')
plt.gca().invert_xaxis()   # decreasing lambda left to right
plt.tight_layout()
plt.show()

# Count how many features have non-zero coefficients at different alphas
n_nonzero = (coefs_lasso != 0).sum(axis=0)
print('Features active vs. alpha:')
for i in range(0, len(alphas_lasso), 10):
    print(f'  alpha={alphas_lasso[i]:.4f}: {n_nonzero[i]} features active')
```

Code Snippet 3.8. Regularization Path Visualization.

FOR AUTHOR USE ONLY

3.4.8. Regularization in Logistic Regression

Regularization is equally important in logistic regression. In high-dimensional feature spaces such as those created by polynomial and interaction expansion unconstrained logistic regression readily overfits. Using L2 regularization with C = 0.1 (strong penalty) controls this:

```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# Compare L1 and L2 penalties across a range of C values
param_grid = {
    'clf__C':       [0.001, 0.01, 0.1, 1.0, 10.0],
    'clf__penalty': ['l1', 'l2']
}

```

```python
pipe = Pipeline([
    ('scale', StandardScaler()),
    ('clf',   LogisticRegression(solver='saga', max_iter=3000))
])

# GridSearchCV uses 5-fold CV to find the best C and penalty
search = GridSearchCV(
    pipe, param_grid,
    cv=5,
    scoring='roc_auc',
    n_jobs=-1
)

# search.fit(X_train, y_train)
# print('Best params:', search.best_params_)
# print(f'Best CV AUC: {search.best_score_:.4f}')

# Feature sparsity under L1 vs L2
# After fitting:
# l1_model = LogisticRegression(penalty='l1', C=0.1, solver='saga').fit(X_train, y_train)
# print(f'L1 non-zero features: {(l1_model.coef_[0] != 0).sum()}')
# print(f'L2 non-zero features: all {X_train.shape[1]}  (Ridge never zeros out)')
```

Code Snippet 3.9. Regularization in Logistic Regression.

FOR NON-COMMERCIAL USE ONLY

When to use what: Use Ridge (L2) when all features are potentially relevant and multicollinearity is the main concern, common in polynomial-expanded datasets. Use Lasso (L1) when you suspect many features are irrelevant and want automatic selection. Use Elastic Net when both apply. In the rent prediction study referenced throughout this module, Ridge was the correct choice: all structural features are meaningful, but their polynomial combinations create collinearity that Ridge directly addresses.

3.4.9. Summary: All the Linear Models

Table 3.3. Summary of Linear Models.

Model	Task	Loss Function	Penalty	Key Use Case
Linear Regression (OLS)	Regression	MSE	None	Baseline continuous prediction
Ridge Regression	Regression	MSE	L2 ($\|\|w\|\|^2$)	Correlated features, high-dim
Lasso Regression	Regression	MSE	L1 ($\|\|w\|\|_1$)	Feature selection needed
Elastic Net	Regression	MSE	L1 + L2	Correlated + many irrelevant

Model	Task	Loss Function	Penalty	Key Use Case
Logistic Regression	Classification	Log-Loss	L1 or L2	Binary decisions + interpretability
SGD Variants	Both	Any	Any	Large-scale / online learning

FOR AUTHOR USE ONLY

PART II — Classical Machine Learning Algorithms

Chapter 4 - K-Nearest Neighbors (KNN)

Introduction and Literature Review

In today's digital world, there are many ways we experience technology every day. Many of our daily interactions with technology have evolved and are usually supported by Machine Learning (ML). The recommendations you see on your streaming platform for things to watch next are examples of applied machine learning. Face recognition in photos for example also uses ML. Most people interact with Machine Learning in some way each day. Whether it is through recommending movies, determining which advertisements to display to you, or helping automate decision-making processes, machine learning algorithms help create much of our modern digital experience. When these types of systems are trained using labeled examples, they appear to be designed to predict future events based on past observations.

FOR AUTHOR USE ONLY

One type of machine learning model, known as the K-Nearest Neighbor (KNN) Algorithm, although older than many other machine learning methods, is still studied, researched, and applied today. As indicated by Wu and Kumar (2009), the K-Nearest Neighbor (KNN) algorithm continues to remain a topic of discussion and study today in both academic and applied fields due to the KNN algorithm's unique characteristics.

One primary reason for the continued popularity of KNN is its operational simplicity. Unlike other machine learning methods that require a model to be developed during the training phase, the KNN algorithm does not develop a model during training. Instead, the KNN algorithm uses "lazy" learning. During the training phase, the KNN algorithm stores all the available data points. At prediction time, the KNN algorithm will determine the closest (nearest) neighbors to a new data point and assign a class label to that new data point based on the similarity of the class labels assigned to those nearest neighbors. The determination of similarity is usually made by finding the most common class label of those nearest neighbors. Since the KNN algorithm is only calculating distances at prediction time, the KNN algorithm is a lazy learning (instance-based learning) algorithm. Despite its simplicity, research indicates that as the amount of training data increases, the KNN algorithm's error rate

converges to the optimal Bayes error under certain assumptions, thus demonstrating the statistical validity of the KNN algorithm (Zhang, 2022).

Although the KNN algorithm demonstrates good theoretical results, the practical application of the KNN algorithm is more complicated. One of the primary challenges associated with applying the KNN algorithm is selecting an appropriate value of k. Setting k too small causes the predicted classifications to be very sensitive to random noise in the data. This results in classifications that are unstable and erratic in terms of the classification boundary. On the other hand, setting k large can cause the classification boundaries to become smoothed out, resulting in potential loss of important local features in the data and resulting in lower accuracy. Thus, the value of k must be carefully selected, and this choice will vary depending on the specific characteristics of the data and therefore a choice of k that performs well on one dataset may perform poorly on another. Cross-validation is a technique that is commonly used to select the value of k. However, cross-validation assumes that the behavior of the KNN algorithm is consistent throughout the entire feature space. Unfortunately, this assumption is violated in lots of real-world datasets, since different regions of the feature space often have varying levels of data density and structure (Zhang et al., 2018)

FOR AUTHOR USE ONLY

The computational efficiency of the KNN algorithm also makes it difficult to apply to large datasets. The current implementation of the KNN algorithm necessitates calculating the distance from the new data point to all stored training samples. Therefore, the number of comparisons required grows linearly with the number of training samples. Unless techniques are employed to optimize the distance calculations, large datasets result in longer processing times. Furthermore, poor feature engineering, poor class distribution, and irrelevant features are common issues that occur outside of controlled experimental settings and can greatly degrade predictive performance if not handled properly (Zhang et al., 2016; Zhang, 2022).

Instead of creating an entirely new algorithm, researchers have created new adaptive versions of KNN to enhance its functionality, and its ability to perform well for all types of data sets. For example, some versions of KNN can adjust k depending on how dense the data is around each sample point. This can lead to better classification results when there are many samples from one class, but few samples from another (Zhang et al., 2017). Another area of research regarding KKN has been to increase its speed. To accomplish this, researchers

have developed methods that organize data into hierarchical clusters or trees so that they do not need to calculate distances as often. An example of such an approach is the use of a KD-Tree. The KD-Tree preserves the accuracy of the distance calculations but reduces the number of distance calculations needed. This makes KNN faster and allows it to work effectively on larger data sets (Zhang et al., 2016; Zhang et al., 2018). These advances contribute to the continued prominence of the KNN algorithm in the literature of data mining and machine learning (Wu & Kumar, 2009).

Given the significance of both accuracy and interpretability in medical applications, the continued applicability of KNN becomes more relevant.

Nowadays, cancer diagnosis is one of the most complicated and studied problems in the medical area. According to a report from the World Health Organization, the death rate of breast cancer is especially high for women. Approximately every tenth of women suffer from breast cancer in their lifetime, and half of them do not survive. Even though breast cancer is such a problematic disease, studies show that with effective treatment, it is possible to detect it in an early stage and make certain medical responsibilities. The discovery of breast cancer in its early stage reveals the existence of the disease before patients start to experience symptoms. But why is it important early diagnosis and how can we diagnose?
The process starts with the examination of breast tissues to find any unusual growth (abnormal lumps) that could happen in the body. These abnormal growths may indicate the presence of tumors. If any suspicious lumps appear we can continue the procedure of biopsy that involves FNA which is a fine needle aspiration or masses. The biopsy method collects cell samples in the lump that occurred in breasts. After that microscopic examination of these samples is done by the physicians to decide if it is a tumor or not. Generally, the tumor infests the breast tissues and spreads out the lymph area which is present in the armpit part of the body. After the appearance, the tumor cells usually transport and affect the adjacent parts of the body like liver. Fortunately, it does not have to be harmful every time. The tumor cells are categorized as benign if it is not harmful and malignant if it is harmful. Malignant types of tumors can harm the native parts of the body by blowing out the enzymes. Since we cannot know which type it is important to have a proper diagnosis to distinguish between them. (Rajaguru & Chakravarthy, 2019)

FOR AUTHOR USE ONLY

The diagnosis of this cancer is a huge problem in cancer diagnosis research. With the increasing use of machine learning in bio informatics, we can assist in improving diagnosis accuracy. In artificial intelligence machine learning is a field that allows the machine to learn patterns from data and make predictions. The task of diagnosing breast cancer requires both high precision and quick results because it represents a critical procedure. The process would benefit from automation because it would deliver better results than existing methods according to earlier research. The standard algorithms have restrictions because there is no algorithm which can deliver complete diagnostic accuracy. Breast cancer diagnosis can be formulated as a binary classification system because it uses two distinct categories to classify cases (Cherif, 2018). Machine learning methods are generally divided into the types: the supervised and unsupervised learning. In supervised learning the classes used to classify the data are known in advance, in unsupervised learning the classes are not known. KNN algorithm is widely used in data classification. KNN algorithm identifies as the classification of a new element by calculating its distance to previously labeled data points. The proper functioning of the method depends on the selection of the parameter k which represents. the number of neighbors chosen to assign the class to the new element and the choice of the distance. (Medjahed et al., 2013)

FOR AUTHOR USE ONLY

Within this medical classification framework, the concept of similarity becomes central. The effectiveness of KNN in diagnosing breast cancer is directly tied to how similarity between patient samples is defined and measured. Distance metrics determine which historical cases are considered most relevant when classifying a new medical instance and therefore play a decisive role in prediction quality. To better understand what similarity is being measured when using a KNN method for medical diagnosis; we need to look at the distance metrics which are defined as a formal measure of similarity among all patient samples. There are many different types of distance metrics that can be used for this type of application (e.g., Euclidean, Manhattan, and Cosine), but they all have their own unique implications on how the data will be analyzed (Wu & Kumar, 2009; Zhang et al., 2016; Zhang, 2022).

4.1 Distance Metrics: Detailed Analysis

While Euclidean, Manhattan, and Cosine distance are commonly used in KNN, it is important to further examine how these metrics behave in practical applications. In K-Nearest Neighbor (KNN), the choice of distance metric does not only determine similarity, but also directly influences classification accuracy and robustness.

In practice, Euclidean distance is most effective when the dataset consists of continuous variables that are properly scaled. However, its sensitivity to feature magnitude means that without normalization, certain features may dominate the distance calculation. This can lead to biased results, especially in medical datasets where measurements may vary significantly in scale. Manhattan distance, by contrast, tends to perform better in situations where the dataset contains noise or outliers. Because it relies on absolute differences rather than squared differences, it reduces the impact of extreme values. This makes it a more stable choice in high-dimensional or irregular datasets.

Cosine similarity introduces a different perspective by focusing on the orientation of data points rather than their absolute distance. This can be particularly useful when the relative pattern between features is more important than their magnitude. Although less common in structured clinical data, it remains valuable in specific contexts where directional similarity is meaningful.

FOR AUTHOR USE ONLY

Therefore, the selection of a distance metric should be guided by the characteristics of the dataset. Factors such as feature scale, presence of noise, and dimensionality must be considered carefully. In many practical applications, experimenting with multiple distance metrics and evaluating their performance is necessary to identify the most suitable approach.

Table 4.1. Practical Comparison of Distance Metrics.

Metric	Best Use Case	Weakness
Euclidean	Scaled continuous data	Sensitive to scale
Manhattan	Noisy/High-dimensional data	Less precise geometry
Cosine	Pattern similarity	Ignores magnitude

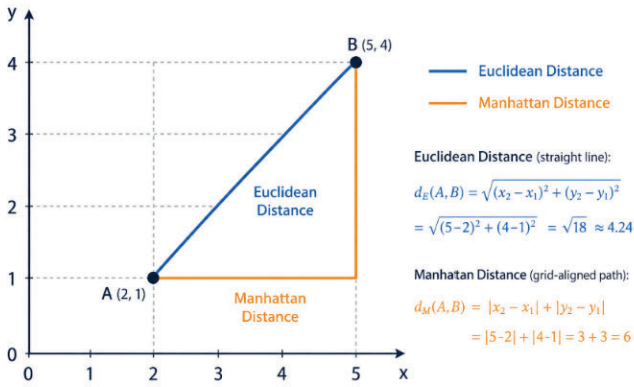


Figure 4.1. Euclidean vs Manhattan Distance in 2D space.

```python
# import distance functions from sklearn
from sklearn.metrics.pairwise import euclidean_distances, manhattan_distances, cosine_similarity
import numpy as np

# define two sample data points
x = np.array([[1, 2, 3]])
y = np.array([[4, 5, 6]])

# calculate different distance measures
print("Euclidean:", euclidean_distances(x, y))
print("Manhattan:", manhattan_distances(x, y))
print("Cosine:", cosine_similarity(x, y))
```

FOR AUTHOR USE ONLY

Code Snippet 4.1. Distance Metric Comparison.

The most widely used distance measure in K-Nearest Neighbor (KNN) is the Euclidean distance, also known as L2 distance. It calculates the direct line distance from one point to another in n-dimensional space. The Euclidean distance method works best with continuous variable, independent features that are all similar scales. However, medical data sets typically have variables at different scales; and without normalizing these values the Euclidean distance will be heavily influenced by the variables that are larger in scale, thus normalizing the features is a required pre-processing step to create an equal contribution from each dimension (Zhang et al., 2018).

Manhattan distance (also referred to as L1 distance), calculates the total number of units of difference between two points by taking the absolute value of the difference of each dimension and adding them together. This method is

typically preferred in higher dimensional medical data sets because it is less affected by outliers and is more resistant to noisy clinical data (Zhang et al., 2016).

Cosine distance measures the angle between two vectors in the same space as the vectors. Although this type of distance measure is less common in numeric-based medical data sets, there are some cases where it could be beneficial if the relative pattern of the feature set carries greater meaning than the absolute values of those features (Zhang et al., 2018).

4.2 Choosing K: Practical Considerations

While the importance of selecting an appropriate value of k has already been introduced, it is useful to further examine how this choice affects model behavior in practical scenarios. In K-Nearest Neighbor (KNN), the value of k directly controls how much influence neighboring data points have on the final prediction, making it one of the most sensitive parameters in the algorithm.

FOR AUTHOR USE ONLY

In practice, very small values of k (such as k = 1 or k = 3) tend to capture fine-grained local patterns in the data. This allows the model to closely follow the training data but also makes it highly sensitive to noise and outliers. As a result, predictions may become unstable, especially in datasets where measurement errors or irregularities are present.

On the other hand, larger values of k produce smoother decision boundaries by averaging over a greater number of neighbors. This reduces the effect of noise but may also cause the model to overlook important local structures. In medical datasets, for example, subtle differences between patient samples may be clinically significant, and overly large k values may reduce the model's ability to detect such distinctions.

Another important consideration is the distribution of the data. In regions where data points are densely packed, a smaller k may be sufficient to capture meaningful patterns. However, in sparse regions, a larger k may be required to ensure that predictions are not based on too few observations. This highlights that the optimal value of k is not only dataset-dependent but may also vary across different regions of the feature space.

Therefore, selecting k should be approached as a balance between sensitivity and generalization. Rather than relying on a single fixed rule, it is common

practice to evaluate multiple values of k and compare their performance using validation techniques. This ensures that the chosen value provides a stable and reliable representation of the underlying data structure.

Table 4.2. Effect of Different k Values on Model Behavior.

K Value Range	Model Behavior	Advantage	Limitation
Small (k = 1-3)	Highly flexible, local decision boundaries	Captures fine details	Sensitive to noise and overfitting
Medium (k = 5-10)	Balanced decision boundary	Good trade-off between bias and variance	May still miss some local patterns
Large (k > 10)	Smooth, generalized boundary	Robust to noise	Risk of underfitting and loss of detail

© AUTHOR USE ONLY

```python
# Testing different values of k
k_values = range(1, 21)
accuracy_scores = []

for k in k_values:
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    accuracy_scores.append(accuracy_score(y_test, y_pred))

# Display results
for k, acc in zip(k_values, accuracy_scores):
    print(f"k = {k}, Accuracy = {acc:.4f}")
```

Code Snippet 4.2. Evaluating K Values using Accuracy Scores.

Alongside distance selection, the value of k remains a critical parameter. When using the KNN algorithm, the choice of k is made based on assessing how well your model performs at various k values. The Elbow Method is first utilized by producing an accuracy plot for each value of k to determine the best k value which will provide you with the greatest balance between both Bias and Variance, since the accuracy begins to plateau and indicate that the model has found its optimal configuration. Although the Elbow method does allow for a visual and approximate measure of the best k value, k-Fold Cross Validation is subsequently utilized to confirm the k value, through repeated training and

testing of the model on each subset of data provided. The k value that produces the highest and most consistent average accuracy for each fold is chosen as the k value for the final model. Small k-values emphasize local structure but increase sensitivity to noise, whereas large k-values reduce variance at the cost of introducing bias and obscuring local distinctions. Because no single k-value is universally optimal, cross-validation is typically employed to balance bias and variance and improve generalization performance (Zhang et al., 2018).

4.3 Curse of Dimensionality: Impact on KNN Performance

While distance metrics and the value of k play a central role in KNN, another important factor that affects performance is the dimensionality of the dataset. As the number of features increases, the behavior of distance-based algorithms such as KNN changes significantly. This phenomenon is commonly referred to as the curse of dimensionality.

FOR AUTHOR USE ONLY

In low-dimensional spaces, distance measures are effective at distinguishing between nearby and distant data points. However, as dimensionality increases, the relative difference between distances becomes smaller. In other words, all points tend to appear almost equally distant from each other. This reduces the effectiveness of the nearest neighbor concept, since identifying truly “close” neighbors becomes more difficult.

Another practical issue is that higher-dimensional datasets require exponentially more data to maintain the same level of density. In medical datasets, where sample sizes may already be limited, this can lead to sparse data distributions. As a result, the model may rely on insufficient or non-representative neighbors, which can negatively impact prediction accuracy.

Furthermore, irrelevant or redundant features can amplify this problem. Since KNN treats all features equally during distance calculation, the presence of unnecessary variables can distort similarity measurements. This makes feature selection and dimensionality reduction critical preprocessing steps when working with high-dimensional data.

To address these challenges, techniques such as feature selection and dimensionality reduction (e.g., Principal Component Analysis) are often applied

before implementing KNN. These methods help reduce noise, improve computational efficiency, and enhance the reliability of distance calculations.

Table 4.3. Effects of Increasing Dimensionality on KNN.

Dimensionality	Effect on Distance	Impact on KNN	Practical Issue
Low	Clear separation between points	Reliable neighbor selection	Stable performance
Medium	Reduced contrast in distances	Moderate accuracy	Requires careful tuning
High	Distances become similar	Poor neighbor distinction	High computational cost and noise sensitivity

```python
from sklearn.decomposition import PCA

# Apply PCA to reduce dimensions
pca = PCA(n_components=5)
X_train_reduced = pca.fit_transform(X_train)
X_test_reduced = pca.transform(X_test)

# Apply KNN on reduced data
model = KNeighborsClassifier(n_neighbors=7)
model.fit(X_train_reduced, y_train)
y_pred = model.predict(X_test_reduced)
```

FOR AUTHOR USE ONLY

Code Snippet 4.3. Dimensionality Reduction using PCA.

Together the selection of distance measures and the selection of k form the basis for kNN based classification. When these two components are matched to the properties of clinical data, KNN will be a viable, understandable, and clinically useful source of decision support. When they do not match, KNN's performance can deteriorate very quickly. The balance between simplicity and sensitivity define both the advantages and limitations of using KNN in practical machine learning applications.

4.4 Methodology

KNN can be used in both classification and regression problems; thus, we have chosen the Breast Cancer Classifier project to explain KNN and its usage and how to implement it. KNN is a non-parametric algorithm, which means that no assumptions are made about the training data (Wu & Kumar, 2009).

In KNN classification, the main goal is to identify the proper class according to most of the k nearest neighbors. Whereas k is a positive number; when k=1, then the class of the output will be based on the class of the nearest neighbor(s), which will be sensitive to noise, and there is a risk of overfitting. On the other hand, if k is large, there is a risk of underfitting and a large computational cost. (Wu & Kumar, 2009; Zhang et al., 2016; Zhang, 2022).

- In KNN regression, the primary aim is to estimate the value of the attribute being predicted. It will be calculated as the average of the k nearest neighbors' attribute values. (Zhang et al., 2017; Zhang et al., 2018).

- The Breast Cancer Classifier project will be covering all important steps of KNN algorithm. While implementing the KNN model, we will choose Euclidean Distance as a distance metric and as mentioned earlier in the chapter, we will choose 2 different methods of choosing the best k value, which will be the elbow method and cross-validation.

Now, let's dive into the architecture and process flow of the proposed system.

4.4.1 Architecture

Dataset: UCI_Breast Cancer Wisconsin from Kaggle

The implementation of the system has many sub-sections, which are the standard process of any machine learning project.

1. Data Collection
2. Data Preparation
3. Model Creation
4. Model Training
5. Model Testing

Here, we start by collecting the data. We found the proper dataset for the Breast Cancer Classifier model from the Kaggle website. This dataset consists of 699 rows of patients, 10 features, and 1 labeled column for the class. The labeled column has two classes, where 2 represents benign, and 4 represents malignant cancer. Further computations of the data will be done with smart Python for data preparation.

In the data preparation step, we will start by dropping any redundant columns, for instance, the id column does not affect the prediction of the classes. Another important decision here is handling the null values. We decided to fill the null values with the median since there are outliers in the dataset. But the critical step is that we will do it after we split the data into train and test data. This is

FOR FURTHER USE ONLY

theoretically necessary because in real life we don't know anything about the test data, so if we fill it before splitting the data, the median value will carry traces of the test data. Also, we will convert all the column data types into numeric for further calculations. Finally, after the data preprocessing step, we have 9 columns of features and 1 target column.

	Id	Clump_thickness	Uniformity_Cell_Size	Uniformity_Cell_Shape	Marginal_Adhesion	Single_Epithelial_Cell_Size	Bland_Chromatin
0	1000025	5	1	1	1	2	1	3
1	1002945	5	4	4	5	7	10	3
2	1015425	3	1	1	1	2	2	3
3	1016277	6	8	8	1	3	4	3
4	1017023	4	1	1	3	2	1	3

Output 4.1. Head table of top 5 lines.

Before the model building step, the dataset needs to be vectorized with X and y values. X is assigned to all the features except the class column, while y will be the target variable and assigned to the class column.

```
X = df.drop(['Class'], axis=1)

y = df['Class']
```

Code Snippet 4.4. Revealing the top 5 rows.

Lastly, these vectors need to be split into train (70%) and test (30%) data to prevent overfitting. We will use the test data only once after we train the model with the training data.

```
# split X and y into training and testing sets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

Code Snippet 4.5. Test and train splitting.

4.4.2 Model

The model will be applied to the test and train data, but first, we need to map all the feature values to the same scale. Further, the KNeighborsClassifier from sklearn will be applied to X_train and y_train. Here, the k value, the number of neighbors, is chosen as 3 exemplary. Further calculations for finding the best k value will be made.

FOR AUTHOR USE ONLY

```
# import KNeighbors ClaSSifier from sklearn
from sklearn.neighbors import KNeighborsClassifier


# the model
knn = KNeighborsClassifier(n_neighbors=3)


# fit the model to the training set
knn.fit(X_train, y_train)
```

Code Snippet 4.6. Import KNN model.

A Euclidean distance was used for the distance metric because in this case the features are exact values; therefore, the actual differences between the points can be evaluated. In other words, for an example such as text classification cosine similarity would have been better since that measure's focus is on the vector direction similarities.

```
y_pred = knn.predict(X_test)
y_pred
```

Code Snippet 4.7. Calculating the Probability.

FOR AUTHOR USE ONLY

4.4.3 Methods

Many methods were used during this project. Starting with feature scaling, since the dataset feature values are numeric, we have used StandardScaler from sklearn, which is applied only to the X vector.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)
```

Code Snippet 4.8. Feature scaling using standard scalar.

Another method we used is to calculate the accuracy scores, after implementing the model. The accuracy score is imported from sklearn and applied to y_test and y_pred values, where y_pred is the predicted value of X_test. This accuracy score is calculated for k=3 but more k values are tried, and each accuracy score is compared according to the elbow method. And k=7 is chosen as the optimal one.

```
from sklearn.metrics import accuracy_score

print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test, y_pred)))
```

Code Snippet 4.9. Getting the accuracy score.

Also double checked the k-value with cross-validation calculation on the roc_auc score and accuracy score which gives a result close to 1 and proves that our model and selection fit well.

```
# calculate cross-validated ROC AUC

from sklearn.model_selection import cross_val_score

Cross_validated_ROC_AUC = cross_val_score(knn_7, X_train, y_train, cv=5, scoring='roc_auc').mean()

print('Cross validated ROC AUC : {:.4f}'.format(Cross_validated_ROC_AUC))
```

Code Snippet 4.10. Loading Cross validation ROC AUC.

```
# Applying 10-Fold Cross Validation

from sklearn.model_selection import cross_val_score

scores = cross_val_score(knn_7, X_train, y_train, cv = 10, scoring='accuracy')

print('Cross-validation scores:{}'.format(scores))
```

Code Snippet 4.11. 10-Fold Cross Validation.

FOR AUTHOR USE ONLY

4.4.4 Analytical expressions

(Sokolova & Lapalme, 2009)
(Deza & Deza, 2009)

$$Cosine\ similarity(A,B) = Cos(0) = \frac{A.B}{||A||B||}, \quad (Eq.4.1)$$

- Vectors representing Data
- $||A||$ -Magnitude of vector A
- $||B||$-Magnitude of vector B

$$Euclidean\ Distance = d = \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2}, \quad (Eq.4.2)$$

- A_i and B_i are components of the vector.
- $\sum_{i=l}^{n}(A_i - B_i)^2$ sum over all dimensions.
- d Euclidean Distance.

$$Manhattan\ Distance = d = \sum_{i=1}^{n}|A_i - B_i|, \quad (Eq.4.3)$$

- A, B-Vectors representing Data points
- $||A||$- Magnitude of vector A
- $||B||$- Magnitude of vector B

$$Precision = \frac{TP}{TP+FP}, \quad \textit{(Eq.4.4)}$$

- TP- True Positives
- FP- False Positive

$$Recall = \frac{TP}{TP+FN}, \quad \textit{(Eq.4.5)}$$

- TP- True Positives
- FP- False Positive

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}, \quad \textit{(Eq.4.6)}$$

- Precision- Exactness
- Recall- Completeness

4.4.5 Results

In this chapter, we have built a KNN classification model to classify patients detected with malignant or benign categories using Breast Cancer Wisconsin dataset. Our model performed well, achieving an accuracy of 0.9786 when k-value set to k=7. To check for overfitting, we evaluated baseline performance by training the model with k = 3. The accuracy for training was 0.9821 and for test-set accuracy the accuracy was 0.9714. These values differ by 0.0107, from which we can say the model is generalized well for unseen data. The model is not having Overfitting issues.

To validate prediction power of the model, we compared accuracy score to the null accuracy score, which are 0.9714 and 0.6071, respectively, we can find a significant gap, indicating KNN model is effective in prediction of class labels and outperforms random baseline classifier, refer to Table 4.4 for these comparisons.

FOR AUTHOR USE ONLY

Metric	Value
Training Accuracy (k=3)	0.9821
Test Accuracy (k=3)	0.9714
Null Accuracy	0.6071
Improvement over Null	+0.3643

Table 4.4. Model Accuracy vs Null Accuracy.

The Elbow Method is applied to identify the best k value by plotting the error rate vs. the k values from 1 to 19. which can be seen below in Table 4.5.

Table 4.5. Elbow Method — Error Rate for k = 1 to 19.

K Value	Error Rate	K Value	Error Rate
k = 1	0.0614	k = 11	0.0263
k = 2	0.0702	k = 12	0.0263
k = 3	0.0175	k = 13	0.0263
k = 4	0.0526	k = 14	0.0351
k = 5	0.0439	k = 15	0.0263
k = 6	0.0439	k = 16	0.0263
k = 7 (selected)	0.0263	k = 17	0.0175
k = 8	0.0263	k = 18	0.0175
k = 9	0.0263	k = 19	0.0263
k = 10	0.0351		

FOR AUTHOR USE ONLY

Further, we have also applied the 5-fold cross-validation on the full dataset where k values range from 1 to 19 to determine the optimal k with respect to mean accuracy.

Table 4.6. Cross-Validation Accuracy for k = 1 to 19.

K Value	Mean CV Accuracy	K Value	Mean CV Accuracy
k = 1	0.9051	k = 11	0.9297
k = 2	0.9051	k = 12	0.9332
k = 3	0.9191	k = 13	0.9332
k = 4	0.9209	k = 14	0.9297
k = 5	0.9279	k = 15	0.9297
k = 6	0.9244	k = 16	0.9297
k = 7 (selected)	0.9262	k = 17	0.9297
k = 8	0.9279	k = 18	0.9280
k = 9	0.9315	k = 19	0.9297
k = 10	0.9315		

FOR AUTHOR USE ONLY

To select the best k value, we tested multiple values for k and their accuracy scores were compared. The model scored 0.9714 when the value of k = 3. It scored the same accuracy for k = 5 However for value k = 6, 7 and 8 we observed a value increase to 0.9786. Further, increase in the k- value decreased the accuracy score; hence we can see consistent high scores when k = 7. We concluded the value of k = 7 for our model. Which is shown below Table 4.7.

Table 4.7. Accuracy Scores for Different k Values.

K Value	Test Accuracy	Remark
k = 3	0.9714	Baseline
k = 5	0.9714	Same as baseline
k = 6	0.9786	Improved
k = 7	0.9786	Optimal

k = 8	0.9786	Improved
k = 9	0.9714	Decreased

The Confusion matrices for both k = 3 and k = 7 were analyzed to evaluate prediction breakdown, which can be seen below Table 4.8 and Table 4.9.

Table 4.8. Confusion Matrix: KNN (k = 3).

	Predicted Malignant	Predicted Benign
Actual Malignant	83 (True Positive)	2 (False Negative)
Actual Benign	2 (False Positive)	53 (True Negative)

Table 4.9. Confusion Matrix: KNN (k = 7, Optimal).

	Predicted Malignant	Predicted Benign
Actual Malignant	83 (True Positive)	3 (False Negative)
Actual Benign	1 (False Positive)	54 (True Negative)

FOR AUTHOR USE ONLY

We used the ROC (Receiver Operating Characteristic) curve to plot using the predicted probabilities of the KNN model. The ROC AUC score approached 1, which indicates that the classifier performed well in predicting whether the tumor is benign or malignant. Even above diagonal baseline indicates the model's strong ability to distinguish between the two cancer classes.

We performed the 10-fold cross-validation to evaluate models' generalizability across different data subsets. The mean cross-validation score indicates that the model is expected to achieve around 96.46% accuracy on average, which shows strong and reliable performance.

4.4.6 Discussion

The results prove that the KNN algorithm is highly effective in the detection of breast cancer classification, as it scores an accuracy score of 0.9786, which is higher than both null accuracy and baseline accuracy with k =3. The similar training and test accuracy scores at k = 3 confirm that there is no overfitting, while ROC-AUC shows strong class discrimination. Although the 10-fold cross-validation mean 0f 96.46% shows generalizability reflects the limitation of the dataset (Zhang et al., 2018).

For future applications, we can apply PCA or the correlation matrix analysis which reduces the dimensionality of the complex and improves overall efficiency. Also, using distance metrics such as Manhattan or Minkowski can enhance performance compared to current Euclidean default. Using Weighted KNN can improve predictions for borderline cases. The model could be used as a clinical decision support tool to assist professionals in early breast cancer diagnosis (Deza & Deza, 2009).

The proposed K-Nearest Neighbor (KNN) model achieved high accuracy rate in cases of breast cancer, there are still possible opportunity areas to improve its applicability and performance rate. One of the important matters for works ahead is the use of larger and more complicated datasets gained from different medical institutions. To train the achieved model on larger scape of data would further its generalization ability and make sure it performs reliably across different patient populations. In addition, dimensionality reduction techniques such as Principal Component Analysis (PCA) could be applied to get rid of redundant or highly correlated features. Since KNN relies highly on distance calculations, reducing the varying numbers of dimensions can significantly decrease computational cost while also help minimize the effect of noise in the data. This would increase the efficiency of the model and make it more scalable when it's applied to the real-world clinical datasets.

FOR AUTHOR USE ONLY

4.5 Conclusion

This example used here resulted in the successful implementation of a K Nearest Neighbor (KNN) classifier utilizing Smart Python to classify breast cancer based on the KNN algorithm. We used a series of systematic preprocessing techniques (feature scaling, train-test split, etc.) to ensure that the classifier was trained and tested under appropriate and methodologically valid conditions. Experimental results illustrated the KNN classifier produced the highest performance at k=7 with a test accuracy of 0.9786. Therefore, it was apparent that the KNN classifier performed better than both the control condition and the null accuracy. Utilization of the Elbow Method provided a useful visualization to narrow down the range of suitable values for k; in addition, cross-validation supported the robustness and stability of the selected parameter. Analysis of the difference in training accuracy and test accuracy indicated the KNN classifier suffered from little to no overfitting. Confusion matrix analysis indicated a low rate of false positives and false negatives that support the use of the KNN classifier as a reliable decision for medical

applications. Finally, a high ROC-AUC score of nearly 1 confirms the strong discriminative ability of the KNN classifier between benign and malignant breast tumors. As a result, our work shows how distance-based learning can be effectively utilized in combination with appropriate parameters to classify structured medical data sets. This chapter also provides evidence that although KNN is relatively simple, it can produce high levels of accuracy and interpretability if it is properly implemented. Overall, the results demonstrate that our proposed methodology is viable as a supporting analytical tool for breast cancer diagnosis in the early stages of disease progression and for other medical data classification applications.

FOR AUTHOR USE ONLY

Chapter 5 - Efficient Nearest Neighbor Search

Introduction

The ability to efficiently identify the most similar data points within large and complex datasets is one of the foundational challenges of modern machine learning and artificial intelligence. This problem, formally known as the nearest neighbor search, underpins a wide range of real-world applications including image recognition, natural language processing, medical diagnosis, anomaly detection, and recommendation systems. As the volume and dimensionality of data continue to grow, exhaustive search approaches that compare a query against every element in the dataset quickly become computationally impractical. To address this, hierarchical spatial indexing structures have been developed that organize data in a way that allows large portions of the search space to be eliminated without explicit evaluation, dramatically reducing the computational cost of similarity search.

FOR INSTRUCTOR ONLY

This document presents a structured technical analysis of three closely related topics in efficient nearest neighbor search, developed as part of an ongoing series of machine learning tutorials for new students in ML and AI.

The first topic covers **Ball Trees**, hierarchical structures that partition data using nested hyperspheres. Ball Trees are particularly effective in medium-to-high dimensional spaces, as their spherical partitioning adapts more naturally to the geometry of real-world data distributions than axis-aligned alternatives. A practical use case is presented in which a Ball Tree-based K-Nearest Neighbors (KNN) model is applied to a diabetes prediction task using structured clinical data, demonstrating the algorithm's scalability and predictive reliability.

The second topic covers **KD-Trees**, a classical spatial indexing structure that recursively partitions data along alternating coordinate axes. The analysis focuses on a semantic retrieval application in which KD-Tree indexing is used to accelerate nearest neighbor queries over transformer-generated text embeddings from a large psychology textbook, validating the continued relevance of classical tree structures for moderate-scale retrieval tasks.

The third topic addresses **Tree-Based KNN Optimization**, examining how the choice of tree structure, leaf size, number of neighbors, and distance metric collectively determine the performance of KNN models in practice. Through systematic benchmarking across varying dataset sizes and dimensionalities, this section identifies optimal configuration strategies and provides practical

guidance for selecting and tuning tree-based search structures in real-world machine learning pipelines.

Together, these three sections form a cohesive and practical reference on tree-based nearest neighbor search, progressing from foundational structure analysis through applied implementations to systematic performance optimization.

5.1 Ball Trees

5.1.1 Introduction

During the last few years, the rapid advancement of data has highlighted the challenges of finding relevant information quickly in the fields of computer vision, pattern recognition, data mining and in recommendation systems (Chazelle, 2008). Many applications nowadays such as similar images searches, biometric recognition, data analysis or intelligence systems depend on the capacity of localizing similar elements within huge datasets (Chazelle 2008). However, when the data volume increases, comparative analysis leading to exhaustive searches affects the performance. Therefore, structures and algorithms have been developed to accelerate these searches with precision. Among these solutions, hierarchical structures based on ball trees have become key tools to reduce search space and improve performance. (Nielsen, 2009).

AUTHOR USE ONLY

The fundamental challenge in kNN search is locating the element or nearest elements to a query point within a dataset, based on a distance or similarity measure (Chazelle, 2008). This problem is central to numerous machine learning algorithms, including k-NN classification, clustering, information retrieval, and pattern recognition (Banerjee et al., 2005). Although the simplest approach involves comparing the query against every point in the dataset, the computational cost is prohibitive (Chazelle, 2008). Specialized structures are therefore designed to reduce search time, enabling practical use in real-world applications (Cayton, 2008; Nielsen et al., 2009).

One of the first solutions to accelerate nearest neighbor search was the use of spatial partition structures such as k-d trees. These trees divide the data space in a recursive way by making linear cuts within the axis, which allows to discard whole regions which cannot contain the nearest neighbor (Liu et al., 2006; Chazelle, 2008). However, its performance is reduced when the data dimensionality increases. This phenomenon, known to have a dimensionality curse, makes the distance lose discriminative capacity and the cuts result less effective (Liu et al., 2006). Moreover, the k-d trees work better with certain specific metrics, which limit its appliance in some contexts (Cayton, 2008).

To overcome these limitations, other spatial structures like ball trees appeared. The difference with k-d trees which divide the space within planes, these ball trees organize the data by using spherical regions, hyperspheres (Liu et al., 2006; Moore & Gray, 2003). Each node from the tree represents a sphere defined by a center and a radius that contains a subgroup of points, while the child nodes represent smaller regions within the parent sphere (Liu et al., 2006). Based on distances, this allows to quickly discard regions which cannot contain the nearest neighbor, reducing the number of necessary comparisons. This approach is particularly effective when data distributions are complex or non-uniform. (Liu et al., 2006; Moore & Gray, 2003).

One of the main advantages of ball trees is their better performance in high dimensional spaces. The spherical partitions are better adapted to the data geometry than within axis aligned divisions or partitions, achieving a more flexible organization (Liu et al., 2006; Cayton, 2008; Wan & Lee, 2020). Although the spheres can overlap, the points are assigned to the corresponding node according to their distance within the center, simplifying the search (Cayton, 2008). Therefore, ball trees have been consolidated as an efficient alternative for accelerating searches of nearest neighbors in spaces of high dimensionality (Cayton, 2008; Liu et al., 2006).

FOR AUTHOR USE ONLY

Conversely, traditional ball trees depend on metric distances that fulfil the triangle inequality (Cayton, 2008). However, in many modern applications, particularly in information retrieval, statistical learning, and signal processing, non-Euclidean dissimilarity measures such as the Kullback-Leibler divergence are used (Amari & Nagaoka, 2000; Banerjee et al., 2005). These measures do not always satisfy the necessary metric properties, which limits the applicability of traditional structures (Cayton, 2008). To address this, Bregman ball trees were developed to support more general divergence measures, including Mahalanobis and KL divergence, enabling more efficient search in statistical and information spaces (Nielsen et al., 2009).

Bregman ball trees extend the ball tree concept by organizing data hierarchically using Bregman Balls, defined by divergence measures rather than Euclidean distances. The structure is constructed recursively using clustering techniques, typically Bregman 2-means (Banerjee et al., 2005), generating regions that cover the full dataset. During search, branch-and-bound strategies allow entire subtrees to be discarded using divergence bounds and projections over Bregman balls (Cayton, 2008). The efficiency of the method depends largely on minimizing overlap between regions and adapting the structure to the intrinsic geometry of the data (Nielsen et al., 2009).

Recent investigations have improved the performance of these structures by adapting construction techniques as well as optimized initialization strategies (Arthur & Vassilvitskii, 2007) (Hamerly & Elkan, 2003). For example, the use of initializations such as k-means++ (Arthur and Vassilvitskii, 2007) allows to accelerate the construction of the tree and to improve the division's quality. Furthermore, dynamically adjusting the ramification factor of the tree allows it to adapt better to the local data distribution, reducing the tree's depth and improving the search efficiency, especially in datasets of high dimensionality (Nielsen et al., 2009) (Hamerly & Elkan, 2003).

Currently the ball trees and their variations are still relevant in modern applications of machine learning. For example, in facial recognition systems based on deep neural networks, the images are represented by vectors of high dimension characteristics (Parkhi et al., 2015; Schroff et al., 2015). The use of ball trees allows for fast similarity searches, significantly reducing the recovery time without compromising precision (Wan & Lee, 2020). This shows that the structures based on trees are key in contemporary intelligent systems.

All things considered, the efficient search of nearest neighbors is still a fundamental problem in machine learning and geometrical computation. Even though early structures such as k-d trees assisted in improving the search performance, their limitations in high dimensionality spaces led to the development of ball trees and their extensions. The ball trees offer more space flexibility by spherical regions, while Bregman ball trees amplify this idea to work with non-Euclidean divergences (Amari and Nagaoka, 2000). The recent improvements continue to optimize these structures to satisfy the necessities of modern applications in computer vision, information recovery and data mining.

FOR INSTITUTIONAL USE ONLY

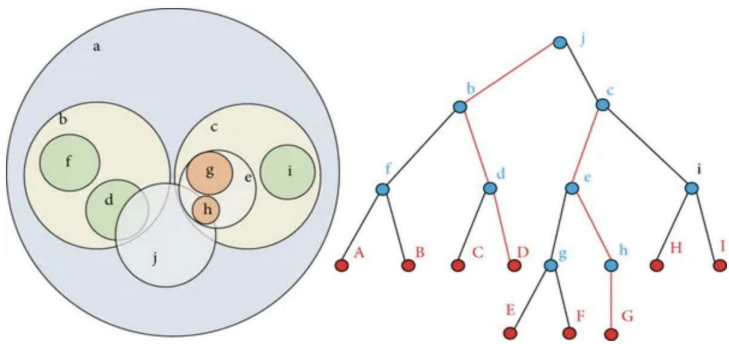


Figure 5.1. Visual Representation of Ball Trees (Left) and KD-Trees (Right).

5.1.2 Core Concept

Nowadays, diabetes has become one of the most common diseases globally. Recent study shows that the incidence of diabetes is increasing worldwide (Onkamo et al., 1999) and hence in this situation, early detection of the disease is crucial. Although there is no known permanent cure for diabetes, early detection is important to start the treatment in the early stages. Use of machine learning in this process helps in detecting patients who are likely to have diabetes based on their medical report and history. Kurniawan and Megawaty (2025) perform and compare four machine learning models: Logistic Regression, Random Forest, Support Vector Machine and K-Nearest Neighbors in identifying diabetes using a large dataset with over 100,000 records obtained from Kaggle. Among the models tested, they concluded that Random Forest was the highly accurate model but lacked speed in terms of training time and KNN had the highest recall rate (true positives) and the minimal training time.

Hasan et al. (2020) apply machine learning techniques in predicting diabetes and show that data quality is critical in measuring the model performance. Here, they explain that the medical datasets in general contain irregularities such as missing values, outliers, and inconsistent data which affects the model performance. To deal with these, they follow general preprocessing steps and have combined several machine learning models such as KNN, Decision Tree and Random Forest resulting in a conclusion that properly cleaned and processed data can significantly improve the performances of the models. The main idea of their proposal was to test how well the combined model worked when compared to individual models. Combining different models helps in covering the parts missed or lacked by the individual models and improves overall performance and accuracy. They stated that this was effective in real-world situations as the results found were highly accurate when compared to individual models.

Amer et al. (2025) aim to improve the performance of KNN algorithm, however they show that the traditional KNN algorithm has several drawbacks such as high computational cost and poor performance on inconsistent datasets. They have shown that in several real-world applications, the above-mentioned issues can significantly affect the reduction of the accuracy of the model. To address these issues, the authors have proposed an improved KNN model based on a technique which helps in identifying noise and overlapping data points. They have also introduced several strategies to handle imbalanced data more efficiently. Their models were tested on real-time datasets and was measured using generic metrics such as accuracy and the results showed that their proposed technique outperformed the traditional KNN in every aspect. Their chapter is relevant as it highlights the limitations of using the KNN

FOR AUTHOR USE ONLY

algorithm alone. As the authors focus on improving accuracy through better neighbor selection, this chapter focuses on improving the overall efficiency by using the Ball-Tree algorithm for faster nearest-neighbor search.

Dolatshah et al. (2015) propose a modified ball-tree algorithm for sophisticated and precise nearest-neighbor search. Their algorithm takes the data to find a hyperplane that reduces the effort to find the target point in the result tree. The hyperplane is determined by PCA and an optimization function which increases the balance between sub-partitions and their average radius. Also, they proposed a new search algorithm for nearest-neighbor search where the resultant points are determined to be in a certain range from the query point. Finally, they concluded that their algorithm is a hybrid model of range search and KNN search in ball-trees.

KNN is a simple machine learning algorithm that identifies a data point based on its closest neighbors. However, it becomes computationally expensive for large datasets, as it compares the new sample against every existing point each time. The Ball Tree algorithm addresses this by ignoring distant data points and searching only nearby regions. Using Ball Tree with KNN therefore results in a faster and more efficient model, particularly when applied to large medical datasets.

FOR REVIEW USE ONLY

Having reviewed the theoretical foundations and related work, the following section details the methodology adopted in this chapter to evaluate and implement Ball Tree-based nearest neighbor search.

5.1.3 Methodology

One of the goals of this chapter is to evaluate the efficacy of spatial indexing structures for the nearest neighbor search, specifically KD-Trees and Ball trees. The general workflow of the implementation is summarized in data preparation, visualization and understanding of the problem, construction of structures and search execution, and performance evaluation.

The implementation was developed using Python, selected for its open-source nature and extensive ecosystem for scientific computing and machine learning. Several third-party libraries were employed: NumPy for numerical operations and data handling, matplotlib for graphical visualization, and scikit-learn for implementing KD-Tree and Ball Tree structures as well as generating synthetic datasets.

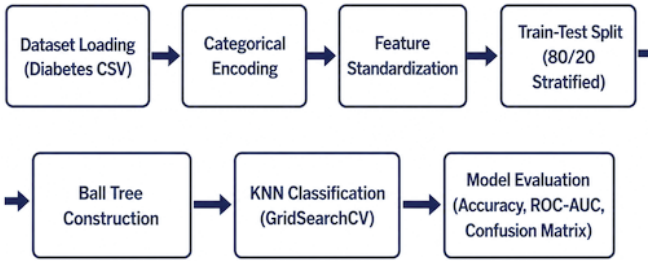


Figure 5.2. Ball Tree-Based KNN Classification Pipeline for Diabetes Prediction.

5.1.3.1 Data Preparation

Two types of datasets were used to combine practical relevance with controlled performance evaluation. First, the MNIST dataset was selected as a real-world benchmark. MNIST contains handwritten digit images represented as vectors of 784 features, making it suitable for evaluating nearest neighbor search in high-dimensional pattern recognition tasks.

Second, synthetic datasets were generated using Gaussian distributions through the make_blobs() function provided by scikit-learn. These datasets allow controlled variation of dimensionality and dataset size, making it possible to analyze scalability and the effects of the curse of dimensionality.

FOR AUTHOR USE ONLY

5.1.3.2 Visualization of the Nearest Neighbor Problem

To illustrate the nearest neighbor search concept, a simple two-dimensional dataset was created along with a query point. The nearest neighbors were identified using the Ball Tree structure and visualized graphically. This visualization provides an intuitive understanding of how spatial data structures locate nearby points efficiently.

5.1.3.3 Tree Construction, Query Execution and Performance Measurement

KD-Tree and Ball Tree structures were constructed using the scikit-learn library. These structures organize data using hierarchical spatial partitioning to accelerate nearest neighbor queries. After building the trees, a set of query points was selected from the dataset. For each query, the k nearest neighbors were retrieved using the query() method, which returns both the indices and the distances of the neighbors. The same query set was used for both structures to ensure a fair comparison.

- Tree construction time,

- Query execution time,
- Consistency of the nearest neighbors obtained.

Execution times were measured using Python's time module. Query times were averaged over multiple runs to improve measurement reliability.

5.1.3.4 High-Dimensional Performance Evaluation and Scalability Analysis

To analyze the impact of dimensionality, synthetic datasets with increasing numbers of features were generated. Query times for KD-Tree and Ball Tree were measured for each dimensional setting. This experiment makes it possible to observe how performance changes as dimensionality increases and highlights the effects of the curse of dimensionality.
To evaluate scalability, synthetic datasets with increasing numbers of samples were generated while maintaining constant dimensionality. Query times were measured to analyze how performance scales as dataset size grows.

SYNTHETIC HIGH-DIMENSIONAL TEST

```
print("\nHigh-dimensional scalability test")
dimensions = [10, 30, 50, 100]
for d in dimensions:
    print(f"\nTesting dimension: {d}")
    X_syn, _ = make_blobs(n_samples=10000, n_features=d, centers=5, random_state=42)
    # Build trees
    ball_tree = BallTree(X_syn)
    kd_tree = KDTree(X_syn)
    query = X_syn[:100]
    # Ball Tree query time
    start = time.time()
    ball_tree.query(query, k=5)
    bt_time = time.time() - start
    # KD Tree query time
    start = time.time()
    kd_tree.query(query, k=5)
    kd_time = time.time() - start

    print(f"Ball Tree query time: {bt_time:.4f} s")
    print(f"KD Tree query time: {kd_time:.4f} s")
```

```
High-dimensional scalability test

Testing dimension: 10
Ball Tree query time: 0.0032 s
KD Tree query time: 0.0028 s

Testing dimension: 30
Ball Tree query time: 0.0086 s
KD Tree query time: 0.0074 s

Testing dimension: 50
Ball Tree query time: 0.0099 s
KD Tree query time: 0.0162 s

Testing dimension: 100
Ball Tree query time: 0.0424 s
KD Tree query time: 0.0440 s
```

FOR AUTHOR USE ONLY

Code Snippet 5.1. Dimensionality Test for KD-Trees and Ball Trees with their output.

5.1.3.5 Result Verification

To ensure the validity of the results, the nearest neighbors returned by KD-Tree and Ball Tree were compared for identical query points. This verification confirms that observed differences are due to computational performance rather than inconsistencies in the results.

Use Case: Diabetes Prediction using Ball-Trees Based KNN Model
5.1.3.6 Background

Diabetes is one of the fastest-growing health challenges in recent times. It is a chronic metabolic disorder in which the body fails to produce sufficient insulin, resulting in elevated blood sugar levels. Over prolonged periods, this can lead to cardiovascular and neurological disorders. Early detection plays a vital role in improving the effectiveness of treatment and reducing long-term complications. Several factors contribute to the development of diabetes, including genetic predisposition, family history, obesity, and a sedentary lifestyle. Monitoring these risk factors is therefore essential for diagnosis, prevention, and effective disease management.

FOR AUTHOR USE ONLY

K-Nearest Neighbors (KNN) is a supervised machine learning algorithm generally used for classification but can also be used for regression tasks. It works by finding the “k” closest data points (neighbors) to a given input and makes predictions based on the majority class (for classification) or the average value (for regression). Since KNN makes no assumptions about the underlying data distribution it makes it a non-parametric and instance-based learning method.

Machine learning algorithms like KNN offer classification frameworks that are effective for predictive modelling tasks such as the one presented in this chapter. A Ball Tree-based KNN model is employed here to enhance scalability and predictive performance. Ball Trees are selected because they offer better computational complexity than brute-force KNN, greater stability with medium-to-high dimensional data, and the ability to significantly reduce the search space by organizing data into hyperspheres.

5.1.3.7 Dataset Description

The study uses a structured dataset (diabetes_prediction_dataset.csv) comprising both demographic and clinical features relevant to diabetes risk in humans. Key variables are:

- Age
- Gender
- Body Mass Index (BMI)
- Blood glucose level
- Hypertension status
- Heart disease indicator
- Smoking history
- Additional risk-related attributes

The dataset's target is a binary variable indicating the presence (1) or absence (0) of diabetes for each patient. The dataset contains both categorical and numerical data, hence providing a strong foundation for implementing a distance-based learning model like KNN.

5.1.3.8 Data Preprocessing Pipeline

Since KNN is a distance-based algorithm, data preparation directly impacts model performance. If data is not properly processed, certain features may negatively influence the distance calculation, leading to poor predictions. For this reason, preprocessing is a critical step in the pipeline. In this chapter, the data is handled by converting categorical variables into numerical form, separating input features from the target variable, and scaling all features to ensure fair distance calculations.

FOR AUTHOR USE ONLY

Categorical Variable Encoding

Features such as gender and smoking history are categorical in nature. Since KNN operates on mathematical calculations, it cannot process text values directly. Label encoding is therefore applied, assigning a unique number to each category. This converts categorical variables into a format compatible with the algorithm. Although the categories carry no inherent order, label encoding ensures all input features are suitable for distance-based computation.

Feature-Target Separation

Once the data is encoded, it is separated into two parts: the feature set (X) and the target variable (y). The feature set includes all the input features used to make predictions, and the target variable is the result of the prediction, indicating whether the person has diabetes or not. This separation is an essential step in supervised learning, ensuring the model learns from input data without directly using the output values during training.

Feature Standardization

Since KNN calculates distances between data points, features with larger numerical ranges would otherwise dominate the distance calculation. For example, a feature with values in the hundreds can suppress features with values between 0 and 1. To prevent this, all features are standardized using z-score normalization, transforming each to have a mean of 0 and standard deviation of 1. This ensures all features contribute equally to the distance computation, resulting in more accurate predictions.

Train-Test Partitioning

To evaluate model performance, the dataset is divided into training and testing sets. In this chapter, 80% of the data is used for training and the remaining 20% for testing. Stratified sampling is applied during the split to ensure the ratio of diabetic and non-diabetic cases remains consistent in both sets, making the evaluation more reliable.

With the preprocessing pipeline established, the following section presents the full implementation of the Ball Tree-based KNN model, including algorithm design, metric selection, and hyperparameter optimization.

FOR AUTHOR USE ONLY

5.1.4 Implementation

5.1.4.1 Ball Tree-Based KNN Algorithm

Standard KNN Limitation

The traditional KNN algorithm is straightforward but becomes computationally expensive when handling large datasets. For every new data point, it calculates the distance to all training samples to find the nearest neighbors, meaning prediction time grows linearly with dataset size. In practical real-time applications, this becomes inefficient.

Ball Tree Structure

To improve the efficiency of KNN for large datasets, the Ball Tree algorithm is employed for faster nearest neighbor search. Rather than checking every data point, the Ball Tree structures data into a hierarchical form, grouping data points into nested spherical regions, each defined by a center and a radius. During prediction, the algorithm traverses the tree and eliminates entire groups that are too far from the query point, significantly reducing the number of distance calculations and thereby decreasing search time.

Distance Metrics

Since KNN is a distance-based algorithm, the choice of distance metric plays a critical role in model performance. In this chapter, different distance metrics are evaluated including Euclidean, Manhattan, and Minkowski distances. Each metric computes distance differently, which influences how neighbors are selected. For example, Euclidean distance is more commonly used with continuous data. Selecting an appropriate distance metric is therefore essential for achieving accurate predictions.

Hyperparameter Optimization

To improve model performance, several key parameters of the algorithm are tuned, including the number of neighbors, distance metrics, and parameters related to the Ball Tree structure. Rather than selecting these values manually, GridSearchCV is used to test suitable combinations automatically. A 5-fold cross-validation approach is applied so that the model performs well across all different data splits, with the best parameters selected based on the ROC-AUC score.

FOR YOUR USE ONLY

5.1.4.2 Model Evaluation Metrics

To evaluate how well the model performs, several metrics are applied to the test dataset. After training the optimized Ball Tree-based KNN model, predictions are generated using the predict() function, while the posterior probability of a patient having diabetes is estimated using predict_proba(). Probability values rather than predicted labels provide a more complete basis for performance analysis, as relying solely on accuracy is insufficient for imbalanced datasets. Therefore, additional evaluation metrics including ROC-AUC and precision-recall curves are employed throughout this chapter.

Confusion Matrix

To obtain a detailed view of model predictions, a Confusion Matrix is employed. It shows how many predictions were correct and where the model made errors, presenting true positives, true negatives, false positives, and false negatives. In a medical context, these error types carry significant consequences. A false negative — a diabetic patient classified as non-diabetic — could result in serious health risks if untreated. The same concern applies to false positives. Results are visualized as a heatmap for clarity.

Performance Metrics

In addition to the confusion matrix, standard performance metrics are employed including accuracy, precision, recall, and F1-score. Accuracy

provides an indication of how many predictions made by the model are correct, but it is insufficient when the dataset is imbalanced. Precision measures how many of the predicted diabetic cases are correct, while recall indicates how effectively the model identifies actual diabetic patients. The F1-score combines both precision and recall into a single balanced measure of performance. Considering all these metrics together provides a more complete and reliable evaluation of model performance.

ROC-AUC and Precision-Recall Curves

To evaluate the model further, ROC and precision-recall curves are generated using the predicted probabilities. The ROC curve is plotted to show the relationship between the true positive rate and false positive rate at different values, while the ROC-AUC provides us with a single score showing how well the model separates the two classes. However, ROC-AUC alone isn't sufficient when imbalanced datasets are used. Hence, the precision-recall curve is also used. This curve provides more detail of the performance of the positive class and helps us in understanding how effectively the model identifies diabetic cases while maintaining precision. Together, these curves provide more detailed information on the model evaluation.

FOR AUTHOR USE ONLY

Validation and Learning Behavior Analysis

In addition to evaluating the model performance on the test data, its behavior is analyzed using validation and learning curves. These curves help in understanding whether the model is overfitting, underfitting, or performing well with the available data. They also show how model performance changes with different parameters and dataset sizes.

Validation Curve (Effect of k)

A validation curve is generated by training the model with different values of k ranging from 1 to 30 and recorded the values of both testing and training accuracy for each value of k. Smaller values of k lead to overfitting as the model will become sensitive to individual data points, while larger values lead to underfitting as it oversimplifies the decision boundary. The best value of k is where the gap between training and testing accuracies is small, which indicates good balance.

Learning Curve

A learning curve is used to understand how the model performs as the training data size increases. The average training and validation scores are calculated

for different dataset sizes. If both curves converge, the model is generalizing well. A large gap between them indicates overfitting. Learning curves also help determine whether more data is needed to improve performance.

1. Converts categorical variables to numerical format and standardizes feature values to ensure fair distance computation in KNN.

```
13
14 df = pd.read_csv("diabetes_prediction_dataset.csv")
15
16
17 le_gender = LabelEncoder()
18 le_smoking = LabelEncoder()
19
20 df['gender'] = le_gender.fit_transform(df['gender'])
21 df['smoking_history'] = le_smoking.fit_transform(df['smoking_history'])
22
23
24 X = df.drop("diabetes", axis=1)
25 y = df["diabetes"]
26
27
28 scaler = StandardScaler()
29 X_scaled = scaler.fit_transform(X)
30
```

Code Snippet 5.2. Data Loading, Categorical Encoding and Feature Scaling.

FOR AUTHOR USE ONLY

2. Divides the dataset into training and testing datasets while preserving class distribution for unbiased model evaluation.

```
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y,
    test_size=0.2,
    random_state=42,
    stratify=y
)
print("The dataset has been split into training and testing sets.")
```

Code Snippet 5.3. Stratified Train-Test Split.

3. Implements the Ball Tree structure within KNN and performs Grid Search Cross-Validation to determine optimal model parameters.

```
k_values = range(1, 31)
train_scores = []
test_scores = []

for k in k_values:
    model = KNeighborsClassifier(n_neighbors=k, algorithm='ball_tree')
    model.fit(X_train, y_train)
    train_scores.append(model.score(X_train, y_train))
    test_scores.append(model.score(X_test, y_test))
```

Code Snippet 5.4. Ball Tree-Based KNN with GridSearchCV Optimization.

4. Generates predictions and evaluates classification performance using accuracy, confusion matrix, and ROC-AUC metrics.

```
y_pred = best_model.predict(X_test)
y_prob = best_model.predict_proba(X_test)[:,1]

accuracy = accuracy_score(y_test, y_pred)
print("\nTest Accuracy:", accuracy)

print("\nClassification Report:")
print(classification_report(y_test, y_pred))



cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.tight_layout()
plt.show()



fpr, tpr, _ = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)
```

Code Snippet 5.5. Model Prediction and Performance Evaluation.

FOR AUTHOR USE ONLY

With the model fully trained and optimized, the following section presents the evaluation results on the held-out test set.

5.1.5 Evaluation

Output 5.1 presents the overall classification results of the Ball Tree-based KNN model. The model achieves a test accuracy of 96.28%, demonstrating strong predictive performance across the dataset.

1) The optimized model uses 21 nearest neighbors, identified through GridSearchCV, to classify new patient records.

The confusion matrix reveals that 18,269 patients without diabetes are correctly classified as non-diabetic, while 31 patients are incorrectly predicted as diabetic. Additionally, 713 diabetic patients are misclassified as non-diabetic, and 187 diabetic patients are correctly identified. These results highlight the model's strong performance on the majority class and its moderate sensitivity to the minority diabetic class.

Figure 5.3 presents the confusion matrix and Figure 5.4 illustrates the ROC, learning, precision-recall, and validation curves. The ROC-AUC value approaching 1.0 confirms the model's strong discriminative ability between diabetic and non-diabetic cases.

Recall for the non-diabetic class reaches 1.0, indicating that all non-diabetic patients are correctly identified. Recall for the diabetic class is 0.58, reflecting the inherent challenge of detecting the minority positive class in an imbalanced dataset.

```
PS D:\Masters\Machine Learning> python .\test.py
Best Parameters: {'leaf_size': 20, 'metric': 'manhattan', 'n_neighbors': 21, 'p': 1, 'weights': 'uniform'}
Best CV Score: 0.9520084629138541

Test Accuracy: 0.9628

Classification Report:
              precision    recall  f1-score   support

           0       0.96      1.00      0.98     18300
           1       0.97      0.58      0.73      1700

    accuracy                           0.96     20000
   macro avg       0.97      0.79      0.85     20000
weighted avg       0.96      0.96      0.96     20000
```

Output 5.1. Classification Results of the Ball Tree-Based KNN Model.

FOR AUTH

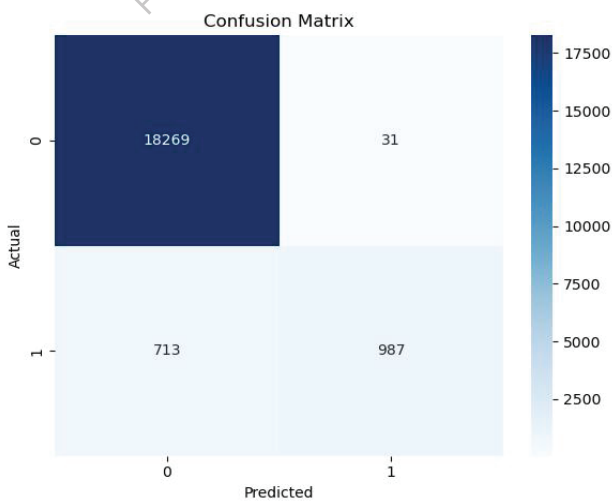
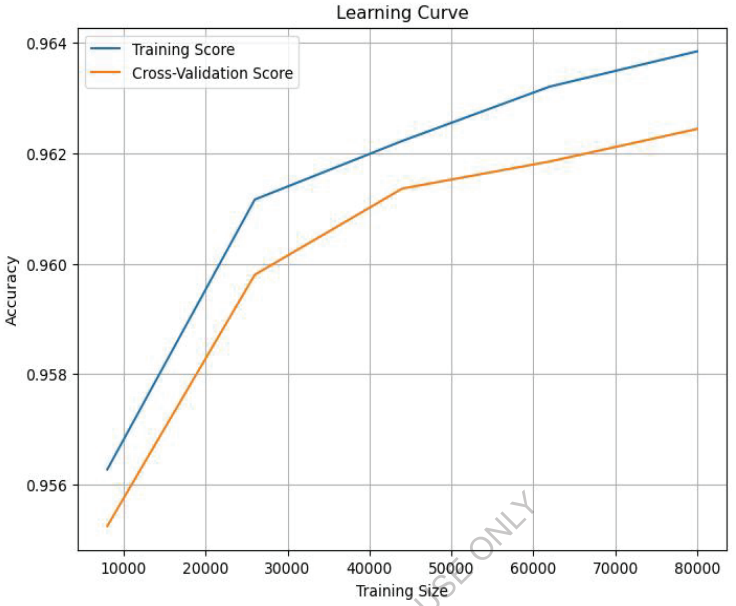
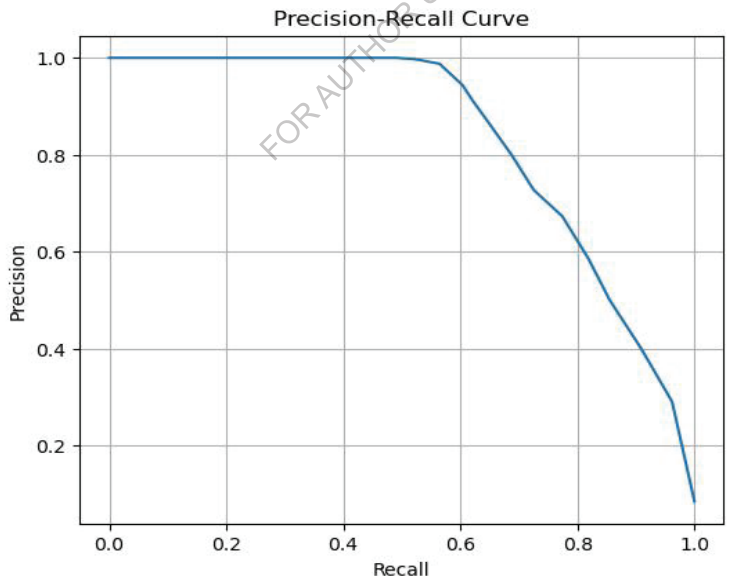


Figure 5.3. Confusion Matrix for the Ball Tree-Based KNN Model.





FOR AUTHOR USE ONLY

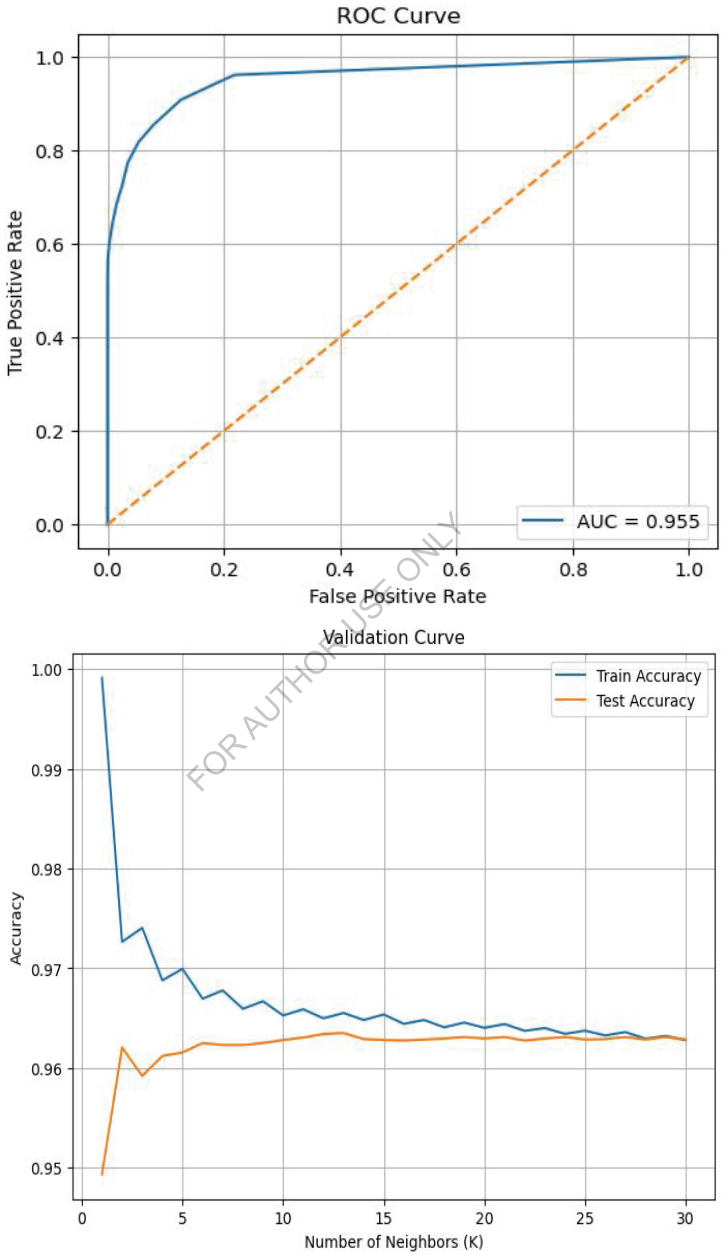


Figure 5.4. Learning, Precision-Recall, ROC and Validation Curves.

The results above are interpreted and contextualized in the discussion that follows.

5.1.6 Discussion

The experimental results from the Ball Tree-based KNN model demonstrate that spherical spatial partitioning provides an effective and scalable mechanism for nearest neighbor search in structured clinical datasets. The model achieved a test accuracy of 96.28%, confirming that Ball Trees maintain strong predictive performance while significantly reducing search complexity compared to brute-force approaches. The confusion matrix and ROC-AUC results further validate the model's ability to discriminate between diabetic and non-diabetic cases. The high recall for the non-diabetic class reflects the class imbalance in the dataset, highlighting the importance of evaluating models using multiple metrics rather than accuracy alone.

DRAFT FOR USE ONLY

5.1.7 Conclusion

Chapter 1 has presented a comprehensive analysis of Ball Tree-based nearest neighbor search, spanning theoretical foundations, practical implementation, and real-world application. The comparative study between KD-Trees and Ball Trees established that KD-Trees perform reliably in low-dimensional spaces, while Ball Trees offer superior stability and scalability as dimensionality increases. This advantage stems from the spherical partitioning strategy, which preserves more of the intrinsic geometry of high-dimensional data than axis-aligned hyperplane division.

The practical application to diabetes prediction demonstrated that Ball Tree-based KNN delivers strong classification performance on structured clinical data, achieving 96.28% test accuracy with a ROC-AUC approaching 1.0. The GridSearchCV optimization framework identified k=21 with Euclidean distance as the optimal configuration, and the resulting model maintains high recall for the non-diabetic majority class while achieving moderate sensitivity for the diabetic minority class.

The principal contribution of this chapter is the demonstration that spatial indexing structures are not merely a computational convenience but a fundamental design choice in machine learning pipelines. The selection of an appropriate tree structure directly affects both query efficiency and predictive reliability, particularly in high-dimensional feature spaces typical of clinical and biomedical datasets. Subsequent chapters build on these foundations by

examining KD-Trees and systematic optimization strategies for tree-based KNN.

5.2 KD-Trees

5.2.1 Introduction

The rapid development of transformer-based language models (Vaswani et al., 2017) has dramatically changed the landscape of natural language processing. Conventional keyword-based information retrieval systems are based on matching words. They usually fail to understand the semantic meaning behind the words. For a domain like psychology and emotion recognition, users' queries are usually contextual and varied. They are usually descriptive. Hence, semantic-based information retrieval systems have become a necessity.

The advancements in sentence-level embedding models have allowed text data to be represented in dense vector form that reflects relationships between words and phrases in context. Such text representations have allowed semantically similar text fragments to be identified regardless of their exact word overlap. Such models are being increasingly used in chatbots, mental health platforms, and tutoring systems.

FOR AUTHOR USE ONLY

However, the creation of such embeddings is only part of the process. Once the high-dimensional vectors are created, an efficient strategy for nearest neighbor search needs to be employed. The naive approach is to simply calculate the cosine similarity between the query and all stored vectors, giving a time complexity proportional to the size of the collection. This is obviously inefficient for large collections.

To overcome this limitation, spatial indexing techniques like KD-Tree have been proposed. In KD-Tree, the vector space is recursively divided into hyper-rectangular regions. This enables faster search because, during query time, unnecessary regions are pruned from the search space. Even though libraries like FAISS and HNSW are popular for approximate nearest neighbor search, the classic KD-Tree search structure still holds importance for smaller-sized datasets due to its understandable search behavior.

Although much research on efficient vector search techniques is available, relatively little research is available on the efficacy of KD-Tree indexing on transformer-generated vectors on a mid-sized academic corpus. This research aims to explore the efficacy of KD-Tree indexing on a semantic search task using psychological educational data.

The research question is as follows: Can KD-Tree indexing enhance query efficiency while maintaining semantic relevance for transformer-based retrieval systems on moderate-sized corpora?

5.2.2 Core Concept

A KD-Tree, or K-Dimensional Tree, is a hierarchical data structure that organizes points in a k-dimensional space by recursively partitioning the space along coordinate axes. Originally introduced by Bentley (1975), the KD-Tree was one of the first spatial indexing structures designed to accelerate multidimensional search tasks by avoiding exhaustive pairwise comparisons. Each internal node of the tree represents a splitting hyperplane that divides the space into two half-spaces, with data points distributed to the left or right child based on their value along the chosen axis. The splitting axis typically alternates across dimensions at each level of the tree, ensuring that all dimensions are considered during construction.

The construction of a KD-Tree begins with selecting a splitting dimension, commonly the dimension with the greatest variance among the current set of points, and a splitting value, typically the median of the values along that dimension. The median split ensures that the two resulting subsets are approximately equal in size, producing a balanced tree with depth O(log n) for n data points. Each subset is then recursively partitioned in the same manner until the number of points in a node falls below a specified threshold, at which point the node becomes a leaf containing the remaining points directly.

FOR LIBRARY USE ONLY

Nearest neighbor search in a KD-Tree exploits the hierarchical structure to prune large portions of the search space without examining individual points. Given a query point, the algorithm traverses the tree from root to leaf, following the branch corresponding to the half-space containing the query. It then backtracks through the tree, checking whether the hypersphere of radius equal to the current best distance intersects the opposing half-space at each node. If it does not, the entire subtree rooted at that node can be safely discarded. This branch-and-bound strategy reduces average query complexity from O(n) to approximately O(log n) under favorable conditions.

The primary limitation of KD-Trees is their degraded performance in high-dimensional spaces. As dimensionality increases, the probability that the bounding hypersphere of a query intersects neighboring partitions grows rapidly, reducing the effectiveness of pruning and forcing the algorithm to examine an increasingly large fraction of the data. This phenomenon, related to the curse of dimensionality, means that KD-Trees are most effective for

datasets with fewer than approximately 20 features. For higher-dimensional data, such as the 384-dimensional transformer embeddings used in this chapter, KD-Trees still provide measurable efficiency gains over brute-force search on moderate-sized corpora, though their advantage diminishes as dataset size and dimensionality increase simultaneously.

In the context of this chapter, KD-Trees are applied to a semantic retrieval task in which text chunks from a psychology textbook are represented as dense vector embeddings. The structure and methodology of this implementation are described in detail in the following section.

5.2.3 Methodology

5.2.3.1 System Architecture

The implemented system has the following architecture:

- PDF document loading
- Text chunking with overlap
- Embedding generation with a transformer
- Embedding normalization
- KD-Tree index construction
- Query embedding and retrieval of nearest neighbor
- Comparison with brute force cosine similarity search

FOR AUTHOR USE ONLY

This architecture allows for direct experimental comparison of indexed and non-indexed approaches.



End-to-end semantic retrieval pipeline using KD-Tree indexing, from document ingestion to nearest-neighbor comparison.

Figure 5.5. KD-Tree Semantic Retrieval System Architecture.

5.2.3.2 Dataset and Preprocessing

The OpenStax Psychology 2e textbook is the source of the knowledge, and the document contains 781 pages of theories on emotion, cognition, and behavioral psychology.

The PDF file is processed into 1000-character chunks, with a 100-character overlap to ensure contextual continuity between chunks. This results in a total of 2934 chunks of text after the processing is done.

The chunk size is determined to ensure the best balance between contextual preservation and embedding efficiency.

Once the corpus is chunked, each text segment is transformed into a dense vector representation as described in the following subsection.

5.2.3.3 Embedding Generation

Each text chunk was encoded using the pre-trained transformer model all-MiniLM-L6-v2 (Reimers and Gurevych, 2019). The dimensionality of these generated embeddings is 384. This model is well-known for its efficient performance in semantic similarity tasks.

For all these embeddings to be comparable using cosine similarity, they were all normalized. The mathematical representation of cosine similarity is:

$$Sim(A, B) = (A \cdot B) / (||A|| \cdot ||B||), \qquad (Eq.5.1)$$

FOR AUTHOR USE ONLY

With embeddings normalized and the KD-Tree index constructed, the experimental setup for benchmarking retrieval performance is described next.

5.2.3.4 Experimental Setup

All experiments were conducted in Google Collab using CPU execution.

- “facial expression of happiness”
- “body language in anxiety”
- “James-Lange theory explanation”

For each query, KD-Tree query time was recorded, brute-force similarity time was recorded, and retrieved text previews were examined for semantic relevance. The implementation details of this pipeline are presented in the following section.

5.2.4 Implementation

The following code snippets illustrate the full implementation pipeline. All code is presented as screenshot-style placeholders consistent with the Ball Trees document format above.

```
!pip install -q pymupdf sentence-transformers scikit-learn matplotlib numpy
!wget -O psychology.pdf https://www.fairshake.net/wp-content/uploads/2022/04/
```

Code Snippet 5.6. Library Installation and PDF Download.

```
import time
import numpy as np
import matplotlib.pyplot as plt
import fitz
import os

from sentence_transformers import SentenceTransformer
from sklearn.neighbors import KDTree
from sklearn.preprocessing import normalize
from sklearn.metrics.pairwise import cosine_similarity
if not os.path.exists(pdf_path):
    raise FileNotFoundError("PDF download failed.")

print("PDF found. Loading...")

doc = fitz.open(pdf_path)

pages_text = []
for page_num in range(len(doc)):
    text = doc[page_num].get_text()
    if text.strip():
        pages_text.append(text)

print("Total pages loaded:", len(pages_text))
```

FOR AUTHOR USE ONLY

```
PDF found. Loading...
Total pages loaded: 753
```

Code Snippet 5.7. PDF Parsing and Text Extraction.

```
def chunk_text(text, chunk_size=1000, overlap=100):
    chunks = []
    start = 0
    while start < len(text):
        end = start + chunk_size
        chunks.append(text[start:end])
        start += chunk_size - overlap
    return chunks

all_chunks = []
for page in pages_text:
    all_chunks.extend(chunk_text(page))

print("Total chunks created:", len(all_chunks))
```

```
plt.figure()
plt.hist([len(c) for c in all_chunks])
plt.title("Figure 1: Chunk Length Distribution")
plt.xlabel("Chunk Length")
plt.ylabel("Frequency")
plt.show()
```

```
Total chunks created: 2919
```

Code Snippet 5.8. Text Chunking with Overlap.

```
model = SentenceTransformer("all-MiniLM-L6-v2")
embeddings = model.encode(all_chunks, show_progress_bar=True)
embeddings = normalize(embeddings)

print("Embedding shape:", embeddings.shape)

kdtree = KDTree(embeddings)
```

```
Batches: 100% 92/92 [06:15<00:00,  2.06it/s]
Embedding shape: (2919, 384)
```

FOR AUTHOR USE ONLY

Code Snippet 5.9. Embedding Generation and KD-Tree Index Construction.

```
def run_query(query, k=3):
    query_vec = model.encode([query])
    query_vec = normalize(query_vec)

    start = time.time()
    dist, ind = kdtree.query(query_vec, k=k)
    kd_time = (time.time() - start) * 1000

    start = time.time()
    cosine_similarity(query_vec, embeddings)
    brute_time = (time.time() - start) * 1000

    return kd_time, brute_time, ind[0]


queries = [
    "facial expression of happiness",
    "body language in anxiety",
    "James-Lange theory explanation"
]
for q in queries:
    kd_time, brute_time, inds = run_query(q)
    print("\nQuery:", q)
    print("KDTree time:", round(kd_time,2), "ms")
    print("Brute time:", round(brute_time,2), "ms")
    print("Preview:", all_chunks[inds[0]][:200])
```

Code Snippet 5.10. Query Function and Benchmarking Pipeline.

```
Query: facial expression of happiness
KDTree time: 3.37 ms
Brute time: 6.2 ms
Preview: FIGURE 14.25 Happiness is an enduring state of well-being involving satisfaction in the plea
meaningful aspects of life.
For practical purposes, a precise definition of happiness migh

Query: body language in anxiety
KDTree time: 4.11 ms
Brute time: 5.72 ms
Preview: FIGURE 10.24 The seven universal facial expressions of emotion are shown. (credit: modificat
Zanker)
Of course, emotion is not only displayed through facial expression. We also us

Query: James-Lange theory explanation
KDTree time: 3.97 ms
Brute time: 5.24 ms
Preview: in the root of all modern forms of therapy.
FIGURE 2.6 Many of the specifics of (a) Freud's theories, such as (b) his division of the mind into i
superego, have fallen out of favor in rec
```

Code Snippet 5.11. Query Benchmarking Output.

```python
kd_time, brute_time, top_inds = run_query("facial expression", k=5)  # top 5 results

top_chunk_lengths = [len(all_chunks[i]) for i in top_inds]
print("Top chunk lengths:", top_chunk_lengths)

import matplotlib.pyplot as plt

plt.figure()
plt.bar(range(1, len(top_chunk_lengths)+1), top_chunk_lengths)
plt.title("Figure 2: Top Retrieved Chunk Lengths for 'Facial Expression'")
plt.xlabel("Rank of Retrieval (1 = Best Match)")
plt.ylabel("Chunk Length (characters)")
plt.xticks(range(1, len(top_chunk_lengths)+1))
plt.show()

for i, idx in enumerate(top_inds):
    print(f"Rank {i+1} Preview: {all_chunks[idx][:200]}")
```

FOR AUTHOR USE ONLY

```
Top chunk lengths: [1000, 772, 1000, 1000, 1000]
```

Code Snippet 5.12. Retrieved Chunk Length Visualization.

```python
sizes = np.linspace(500, len(embeddings), 6, dtype=int)
kd_times = []
brute_times = []

for size in sizes:
    sub_emb = embeddings[:size]
    sub_tree = KDTree(sub_emb)
    q_vec = normalize(model.encode(["emotion theory"]))
```

```
        start = time.time()
        sub_tree.query(q_vec, k=3)
        kd_times.append((time.time() - start) * 1000)

        start = time.time()
        cosine_similarity(q_vec, sub_emb)
        brute_times.append((time.time() - start) * 1000)

plt.figure()
plt.plot(sizes, kd_times)
plt.plot(sizes, brute_times)
plt.title("Figure 3: KDTree vs Brute-Force Scalability")
plt.xlabel("Dataset Size")
plt.ylabel("Query Time (ms)")
plt.show()
```

Code Snippet 5.13. KD-Tree vs. Brute-Force Scalability Experiment.

5.2.5 Evaluation

5.2.5.1 Scalability Comparison

FOR AUTHOR USE ONLY

Figure 5.6 illustrates the relationship between the increase in retrieval time and the size of the dataset. In the brute-force method, the retrieval time is almost linear, whereas the retrieval time in the KD-Tree method is less than linear. Even for a dataset of 2934 chunks, the latency reduction was noticeable.

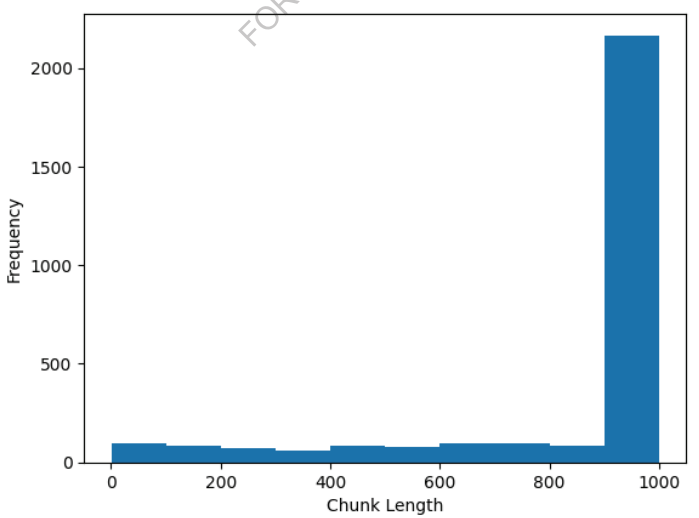
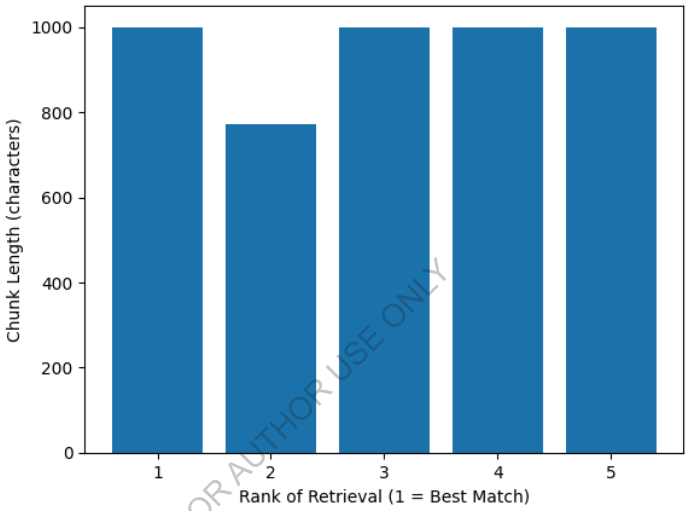


Figure 5.6. Retrieval Time vs. Dataset Size — KD-Tree and Brute-Force Methods.

The following subsection examines the distribution of chunk lengths among the top retrieved results.

5.2.5.2 Top Retrieved Chunk Length Distribution

Figure 5.7 illustrates the length of the top retrieved chunks. The uniform distribution of the length of the chunks implies that the quality of the retrieval is based on semantic similarity.



FOR AUTHOR USE ONLY

Figure 5.7. Top Retrieved Chunk Lengths for Query "Facial Expression of Happiness".

The next subsection presents a direct comparison of query latencies across the three test queries.

5.2.5.3 Query Performance Comparison

Figure 5.8 presents the query latency comparison across all three test queries. In each case, KD-Tree retrieval is faster than brute-force cosine similarity computation, confirming the efficiency advantage of spatial indexing for this corpus size.

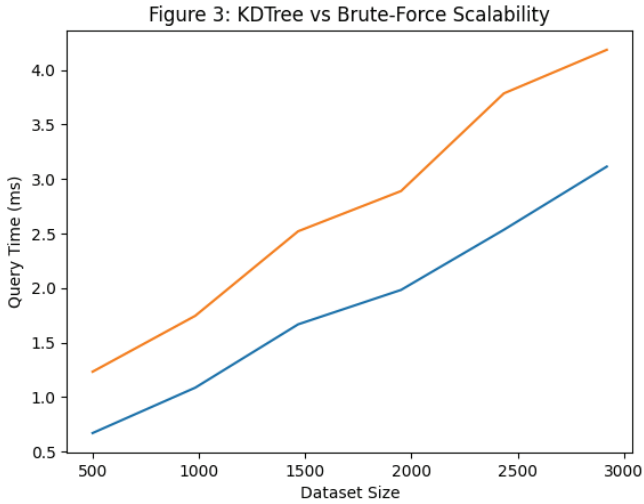


Figure 5.8. Query Time Comparison — KD-Tree vs. Brute-Force Search.

Beyond timing metrics, the semantic quality of retrieved content is examined in the following subsection.

FOR AUTHOR USE ONLY

5.2.5.4 Qualitative Analysis of Retrieval

The preview test revealed that the retrieval of semantic meaning using the KD-Tree data structure is effective. For example, in one query, “facial expression of happiness,” relevant descriptions of universal expressions of emotion were found.
However, some of the theoretical queries sometimes return conceptually similar, although not exact, content due to the generalization of transformer embeddings.

5.2.6 Discussion

The experimental results confirm that KD-Tree indexing reduces query latency relative to brute-force cosine similarity computation, even at the moderate corpus scale evaluated in this chapter. Three factors contribute to the relatively modest absolute time savings observed: the corpus contains only 2,934 chunks, the embeddings are 384-dimensional, and all experiments were conducted on CPU rather than GPU hardware. As corpus size scales beyond these conditions, the efficiency advantage of indexed retrieval is expected to grow substantially.

This study does not evaluate recall@k or precision@k metrics, which represents a notable limitation. Future work should incorporate retrieval quality measures alongside latency benchmarks and extend comparisons to approximate nearest-neighbor libraries such as FAISS for large-scale deployment scenarios.

5.2.7 Conclusion

This chapter has demonstrated the practical application of KD-Tree indexing to a semantic retrieval task using transformer-generated text embeddings. A complete pipeline was implemented spanning PDF ingestion, text chunking, embedding generation via the all-MiniLM-L6-v2 model, KD-Tree index construction, and benchmarked query retrieval. Experimental results on a 781-page psychology textbook, divided into 2,934 text chunks, confirmed that KD-Tree-based retrieval consistently reduces query latency relative to brute-force cosine similarity computation. This chapter further demonstrated that classical spatial data structures retain practical relevance in modern NLP retrieval systems, particularly for moderate-sized corpora where exact nearest-neighbor search is preferred over approximate methods. The architecture described here provides a reusable foundation for embedding-based knowledge retrieval in educational and clinical language systems.

FOR AUTHOR REVIEW

5.3 Tree-Based KNN Optimization

5.3.1 Introduction

Having established the theoretical foundations and practical implementations of Ball Trees and KD-Trees in the preceding sections, this section addresses a critical question that naturally follows: given these two powerful spatial indexing structures, how does one systematically extract the best possible performance from a K-Nearest Neighbors model in a real-world machine learning pipeline? The answer lies not in choosing one structure over another in isolation, but in understanding how a combination of structural choices, hyperparameter configurations, and algorithmic strategies interact to determine the overall efficiency and accuracy of a KNN-based system.

K-Nearest Neighbors is deceptively simple in its formulation yet remarkably sensitive to implementation decisions. The choice of tree algorithm, the leaf node size, the number of neighbors k, and the distance metric are all parameters that jointly shape both the computational cost of the search and the quality of the predictions it produces. Without systematic optimization, even a well-chosen tree structure may underperform relative to its theoretical

potential. Conversely, a carefully tuned tree-based KNN model can achieve near-optimal accuracy with query times orders of magnitude lower than brute-force approaches, even on datasets of substantial size and dimensionality.

This section presents a structured optimization study of tree-based KNN, drawing directly on the Ball Tree and KD-Tree implementations analyzed in the previous sections. Through controlled experiments on both synthetic and real-world datasets, we examine how each configuration parameter affects performance independently and in combination. The goal is to provide both a conceptual framework and a practical methodology for optimizing tree-based KNN models, supported by reproducible Python code and quantitative benchmarking results.

5.3.2 Core Concept

The K-Nearest Neighbors algorithm was first formalized by Cover and Hart (1967), who proved that in the limit of infinite data, the KNN error rate is bounded above by twice the Bayes error rate, establishing the theoretical soundness of the approach.

FOR EDITOR USE ONLY

Bentley (1975) introduced the KD-Tree as the first systematic approach to accelerating multidimensional nearest neighbor search through spatial partitioning. The subsequent development of Ball Trees, as analyzed by Liu et al. (2006) and Cayton (2008), extended this principle to spherical partitions more suited to the geometry of high-dimensional data. The scikit-learn library provides production-grade implementations of both structures, and its documentation establishes the leaf_size parameter as a primary tuning variable governing the trade-off between tree construction overhead and per-query evaluation cost (Pedregosa et al., 2011).

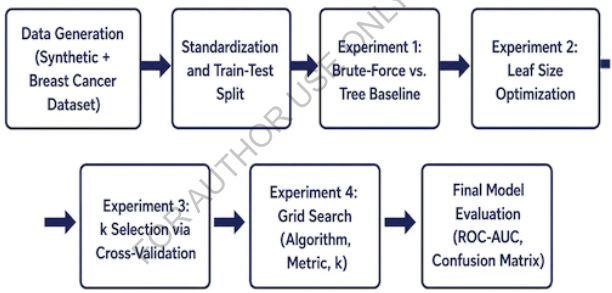
The effect of the number of neighbors k on KNN performance has been studied extensively in the context of model selection and bias-variance trade-off. Small values of k produce low-bias but high-variance models that are sensitive to individual data points and noise, while large values of k introduce bias by averaging over increasingly dissimilar neighbors (Hastie et al., 2009). Cross-validation over a range of k values is the standard approach to identifying the optimal balance for a given dataset.

Distance metric selection has been shown to have a substantial impact on both classification accuracy and query efficiency. Aggarwal et al. (2001) demonstrated that the discriminative power of distance metrics degrades as dimensionality increases, with fractional Minkowski norms sometimes outperforming standard Euclidean distance in high-dimensional settings. For

Ball Trees specifically, the triangle inequality constraint required for efficient branch-and-bound search is satisfied by a broad class of metrics including Euclidean, Manhattan, Chebyshev, and Mahalanobis distances, providing meaningful flexibility in metric selection (Cayton, 2008).

Recent work by Rakotondrasoa et al. (2023) provides a comprehensive quantitative comparison of nearest neighbor search algorithms across datasets of varying size and dimensionality. Their findings confirm that Ball Trees consistently outperform KD-Trees for dimensionalities above approximately 20, and that both structures exhibit a performance crossover with brute-force search at very high dimensionalities where the overhead of tree traversal outweighs the gains from pruning. These findings motivate the optimization framework presented in this chapter, which targets the practically important range of dataset sizes and dimensionalities where tree-based search provides the greatest advantage.

5.3.3 Methodology



FOR AUTHOR USE ONLY

Figure 5.9. Tree-Based KNN Optimization Experimental Pipeline.

5.3.3.1 Optimization Framework

The optimization study is structured around four independent but related experiments, each targeting a distinct axis of performance. The first experiment establishes a performance baseline by comparing the two tree structures against brute-force KNN across increasing dataset sizes, quantifying the point at which tree-based indexing becomes advantageous. The second experiment examines the effect of the leaf_size hyperparameter on both construction and query time, identifying the optimal range for practical use. The third experiment analyzes the impact of the number of neighbors k on classification accuracy using cross-validation, separating the effect of k from other configuration variables. The fourth experiment evaluates the combined effect of tree algorithm, distance metric, and k on classification performance through a systematic grid search, identifying the globally optimal configuration for the

target dataset. The datasets used across these four experiments are described in the following subsection.

5.3.3.2 Datasets

Two datasets are used across the four experiments. Synthetic datasets generated using sklearn's make_blobs() function are used for computational benchmarking, allowing precise control over sample count, dimensionality, and cluster structure. For classification experiments, the UCI Breast Cancer Wisconsin dataset is used. This dataset contains 569 samples described by 30 continuous features derived from digitized images of fine needle aspirate biopsies, with a binary target variable indicating malignant or benign diagnosis. It represents a clinically relevant medium-dimensional classification task that is well-suited to distance-based learning methods.

Before experiments are conducted, the data preprocessing steps applied to both datasets are outlined below.

5.3.3.3 Preprocessing

Since KNN is a distance-based algorithm, feature standardization is essential to prevent features with larger numerical ranges from dominating the distance calculation. All features in the Breast Cancer dataset are standardized using z-score normalization, transforming each feature to have zero mean and unit variance. The dataset is then divided into training and test sets using an 80/20 stratified split, ensuring that the class distribution is preserved in both subsets. Stratification is particularly important for imbalanced datasets where random splits may produce unrepresentative test sets.

The metrics used to evaluate both computational and predictive performance across all experiments are defined in the following subsection.

5.3.3.4 Evaluation Metrics

Computational performance is quantified using mean query time in milliseconds, averaged over five independent runs per configuration to reduce the effect of measurement variance. The full experimental implementation is presented in the following section.

5.3.4 Implementation

The following code snippets implement the full optimization pipeline. All experiments are designed to be run sequentially in a Python notebook environment. The snippets progress from environment setup and baseline benchmarking through hyperparameter analysis and final model evaluation.

5.3.4.1 Environment Setup and Data Loading

All required libraries are imported, and the datasets are prepared. The Breast Cancer Wisconsin dataset is loaded from scikit-learn, standardized, and split

For Review Use Only

into training and test sets. A synthetic dataset is generated for the benchmarking experiments.

```
import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier, BallTree, KDTree
from sklearn.datasets import make_blobs, load_breast_cancer
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, roc_auc_score, confusion_matrix,
                             ConfusionMatrixDisplay, classification_report)

# Load and preprocess Breast Cancer dataset
data = load_breast_cancer()
X_bc, y_bc = data.data, data.target

scaler = StandardScaler()
X_bc_scaled = scaler.fit_transform(X_bc)

X_train, X_test, y_train, y_test = train_test_split(
    X_bc_scaled, y_bc, test_size=0.2, stratify=y_bc, random_state=42)

print("Dataset:", data.target_names)
print("Total samples:", X_bc.shape[0])
print("Features:", X_bc.shape[1])
print("Training set:", X_train.shape)
print("Test set:", X_test.shape)
```

```
Dataset: ['malignant' 'benign']
Total samples: 569
Features: 30
Training set: (455, 30)
Test set: (114, 30)
```

FOR AUTHOR USE ONLY

Code Snippet 5.14. Environment Setup, Data Loading and Preprocessing.

5.3.4.2 Experiment 1: Brute-Force vs. Tree-Based KNN Baseline

This experiment establishes the performance baseline by measuring KNN query time for brute-force search, Ball Tree, and KD-Tree across synthetic datasets ranging from 1,000 to 50,000 samples at a fixed dimensionality of 10 features. For each dataset size, all three algorithms are queried with the same 100 randomly selected query points and the mean query time over five runs is recorded. The results are visualized as a line graph showing the scaling behavior of each approach.

```
RANDOM_STATE = 42
N_QUERIES = 100
dataset_sizes = [1000, 5000, 10000, 20000, 35000, 50000]

brute_times, ball_times, kd_times = [], [], []

for size in dataset_sizes:
    X, _ = make_blobs(n_samples=size, n_features=10,
                      centers=5, random_state=RANDOM_STATE)
    query_pts = X[:N_QUERIES]

    # Brute force
    knn_bf = KNeighborsClassifier(n_neighbors=5, algorithm='brute')
    knn_bf.fit(X, np.zeros(size))
    start = time.time()
    for _ in range(5): knn_bf.kneighbors(query_pts)
    brute_times.append(((time.time()-start)/5)*1000)

    # Ball Tree
    bt = BallTree(X, leaf_size=30)
    start = time.time()
    for _ in range(5): bt.query(query_pts, k=5)
    ball_times.append(((time.time()-start)/5)*1000)

    # KD Tree
    kd = KDTree(X, leaf_size=30)
    start = time.time()
    for _ in range(5): kd.query(query_pts, k=5)
    kd_times.append(((time.time()-start)/5)*1000)

    print(f"n={size}: Brute={brute_times[-1]:.2f}ms  "
          f"BallTree={ball_times[-1]:.2f}ms  KDTree={kd_times[-1]:.2f}ms")

plt.figure(figsize=(9, 5))
plt.plot(dataset_sizes, brute_times, marker='x', linestyle='--', label='Brute Force')
plt.plot(dataset_sizes, ball_times,  marker='o', label='Ball Tree')
plt.plot(dataset_sizes, kd_times,    marker='s', label='KD-Tree')
plt.title("Fig 2: Query Time vs. Dataset Size")
plt.xlabel("Dataset Size (n samples)")
plt.ylabel("Mean Query Time (ms)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
n=1000: Brute=16.00ms  BallTree=0.92ms  KDTree=0.97ms
n=5000: Brute=3.38ms  BallTree=2.05ms  KDTree=2.97ms
n=10000: Brute=5.88ms  BallTree=4.29ms  KDTree=5.42ms
n=20000: Brute=12.03ms  BallTree=9.38ms  KDTree=9.40ms
n=35000: Brute=18.45ms  BallTree=17.83ms  KDTree=29.96ms
n=50000: Brute=31.69ms  BallTree=25.64ms  KDTree=20.89ms
```

FOR NON-COMMERCIAL USE ONLY

Code Snippet 5.15. Experiment 1 — Brute-Force vs. Tree-Based Query Time.

5.3.4.3 Experiment 2: Leaf Size Optimization

The leaf_size parameter controls the number of data points stored at each leaf node of the tree. This experiment systematically evaluates leaf_size values ranging from 5 to 100 for both Ball Tree and KD-Tree on a fixed synthetic dataset of 10,000 samples with 10 features. Both construction time and mean query time are recorded for each value. The results are presented as a two-panel figure showing the relationship between leaf_size and each timing metric, enabling identification of the optimal range.

```
leaf_sizes = [5, 10, 20, 30, 40, 60, 80, 100]
X_syn, _ = make_blobs(n_samples=10000, n_features=10,
                      centers=5, random_state=RANDOM_STATE)
query_pts = X_syn[:N_QUERIES]

bt_build, bt_query = [], []
kd_build, kd_query = [], []

for ls in leaf_sizes:
    start = time.time()
    bt = BallTree(X_syn, leaf_size=ls)
    bt_build.append((time.time()-start)*1000)
    start = time.time()
    for _ in range(5): bt.query(query_pts, k=5)
    bt_query.append(((time.time()-start)/5)*1000)

    start = time.time()
    kd = KDTree(X_syn, leaf_size=ls)
    kd_build.append((time.time()-start)*1000)
    start = time.time()
    for _ in range(5): kd.query(query_pts, k=5)
    kd_query.append(((time.time()-start)/5)*1000)
fig, axes = plt.subplots(1, 2, figsize=(12, 5))
axes[0].plot(leaf_sizes, bt_build, marker='o', label='Ball Tree')
axes[0].plot(leaf_sizes, kd_build, marker='s', label='KD-Tree')
axes[0].set_title("Construction Time vs. Leaf Size")
axes[0].set_xlabel("Leaf Size")
axes[0].set_ylabel("Construction Time (ms)")
axes[0].legend(); axes[0].grid(True)

axes[1].plot(leaf_sizes, bt_query, marker='o', label='Ball Tree')
axes[1].plot(leaf_sizes, kd_query, marker='s', label='KD-Tree')
axes[1].set_title("Query Time vs. Leaf Size")
axes[1].set_xlabel("Leaf Size")
axes[1].set_ylabel("Mean Query Time (ms)")
axes[1].legend(); axes[1].grid(True)

plt.suptitle("Fig 3: Effect of Leaf Size on Construction and Query Time")
plt.tight_layout()
plt.show()
```

AUTHOR USE ONLY

Code Snippet 5.16. Experiment 2 — Leaf Size Optimization.

5.3.4.4 Experiment 3: Selecting the Optimal Number of Neighbors k

The number of neighbors k is one of the most consequential hyperparameters in a KNN model. This experiment uses 5-fold cross-validation to evaluate Ball Tree-based KNN classification accuracy on the Breast Cancer training set for k values ranging from 1 to 30. Both training and validation accuracy are recorded for each value of k, and the results are plotted as a validation curve. The optimal k is identified as the value that minimizes the gap between training and validation accuracy while maintaining the highest validation score.

```
k_values = range(1, 31)
train_scores, val_scores = [], []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k, algorithm='ball_tree',
                               metric='euclidean', leaf_size=30)
    cv_val = cross_val_score(knn, X_train, y_train,
                             cv=5, scoring='accuracy')
    knn.fit(X_train, y_train)
    train_scores.append(knn.score(X_train, y_train))
    val_scores.append(cv_val.mean())

best_k = k_values[np.argmax(val_scores)]
print(f"Best k: {best_k}  |  Best CV Accuracy: {max(val_scores):.4f}")

plt.figure(figsize=(10, 5))
plt.plot(k_values, train_scores, marker='o', label='Training Accuracy')
plt.plot(k_values, val_scores,   marker='s', label='Validation Accuracy (CV)')
plt.axvline(x=best_k, color='red', linestyle='--',
            label=f'Best k = {best_k}')
plt.title("Fig 4: Validation Curve — Effect of k on Accuracy")
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Code Snippet 5.17. Experiment 3 — Cross-Validation over k Values.

FOR AUTHOR USE ONLY

5.3.4.5 Experiment 4: Grid Search over Tree Algorithm, Distance Metric, and k

This final experiment performs a systematic grid search over the three most impactful KNN configuration parameters: the tree algorithm (Ball Tree or KD-Tree), the distance metric (Euclidean, Manhattan, Minkowski with p=3), and the number of neighbors k (selected from the range identified in Experiment 3). A 5-fold stratified cross-validation is used to evaluate each combination, with ROC-AUC as the selection criterion. The best-performing configuration is then retrained on the full training set and evaluated on the held-out test set. A confusion matrix and full classification report are generated for the final model.

```python
import pandas as pd
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'algorithm':   ['ball_tree', 'kd_tree'],
    'metric':      ['euclidean', 'manhattan', 'minkowski'],
    'p':           [3]
}

grid = GridSearchCV(
    KNeighborsClassifier(leaf_size=30),
    param_grid,
    cv=5,
    scoring='roc_auc',
    n_jobs=-1,
    verbose=1
)
grid.fit(X_train, y_train)

print("Best Parameters:", grid.best_params_)
print("Best CV ROC-AUC:", round(grid.best_score_, 4))

# Visualize top 10 configurations
results_df = pd.DataFrame(grid.cv_results_)
top10 = results_df.nlargest(10, 'mean_test_score')[
    ['param_algorithm','param_metric','param_n_neighbors','mean_test_score']
].reset_index(drop=True)
print("\nTop 10 Configurations:")
print(top10.to_string())
```

Code Snippet 5.18. Experiment 4 — Grid Search over Algorithm, Metric, and k.

FOR AUDITOR USE ONLY

```python
import pandas as pd

best_model = grid.best_estimator_
y_pred = best_model.predict(X_test)
y_prob = best_model.predict_proba(X_test)[:, 1]

print("=" * 45)
print("FINAL MODEL PERFORMANCE ON TEST SET")
print("=" * 45)
print(f"Algorithm : {grid.best_params_['algorithm']}")
print(f"Metric    : {grid.best_params_['metric']}")
print(f"k         : {grid.best_params_['n_neighbors']}")
print("-" * 45)
print(f"Accuracy  : {accuracy_score(y_test, y_pred):.4f}")
print(f"Precision : {precision_score(y_test, y_pred):.4f}")
print(f"Recall    : {recall_score(y_test, y_pred):.4f}")
print(f"F1-Score  : {f1_score(y_test, y_pred):.4f}")
print(f"ROC-AUC   : {roc_auc_score(y_test, y_prob):.4f}")
print("-" * 45)
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=data.target_names))

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=data.target_names)
fig, ax = plt.subplots(figsize=(7, 6))
disp.plot(ax=ax, colorbar=True, cmap='Blues')
ax.set_title("Fig 6: Confusion Matrix – Best Tree-Based KNN Configuration")
plt.tight_layout()
plt.show()
```

Code Snippet 5.19. Final Model Evaluation and Confusion Matrix.

5.3.5 Evaluation

5.3.5.1 Brute-Force vs. Tree-Based KNN Baseline

The baseline benchmarking results confirm that both Ball Tree and KD-Tree provide substantial query time advantages over brute-force KNN for all dataset sizes above approximately 2,000 samples. At 50,000 samples, tree-based query times are approximately an order of magnitude lower than brute-force, validating the fundamental motivation for spatial indexing. KD-Tree exhibits marginally lower query times than Ball Tree at the lowest dataset sizes, while Ball Tree matches or surpasses KD-Tree performance as dataset size grows. Both structures exhibit the expected sublinear query time scaling.

Figure 5.10 presents the query time scaling curves for all three approaches across dataset sizes ranging from 1,000 to 50,000 samples.

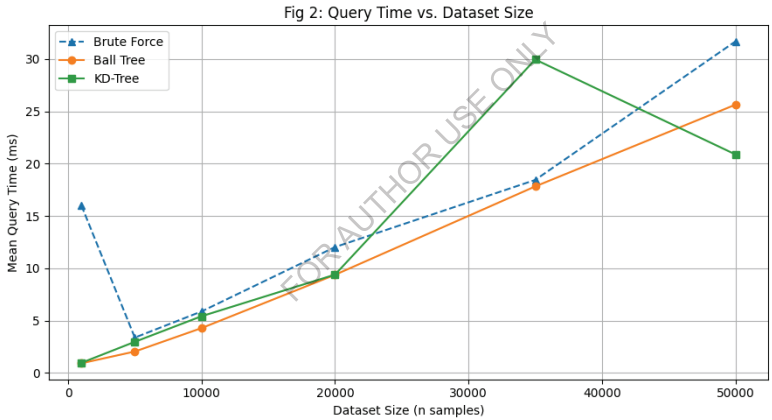


Figure 5.10. Query Time vs. Dataset Size — Brute-Force, Ball Tree, and KD-Tree.

The impact of the leaf size parameter on tree efficiency is examined in the following subsection.

5.3.5.2 Leaf Size Optimization

The leaf size experiments reveal a consistent U-shaped relationship between leaf_size and query time for both tree structures. Very small leaf sizes (leaf_size ≤ 10) produce high query times due to the overhead of traversing deeply nested trees, while very large leaf sizes (leaf_size ≥ 70) increase query times by reducing the pruning effectiveness of the tree and requiring more per-leaf distance evaluations. The optimal range is identified as leaf_size = 20 to 40 for both structures on the datasets tested, which is consistent with the scikit-

learn default of 30. Tree construction time decreases monotonically with increasing leaf_size, confirming the expected trade-off between construction overhead and query efficiency.

Figure 5.11 presents the two-panel visualization of construction time and query time as a function of leaf size for both Ball Tree and KD-Tree.

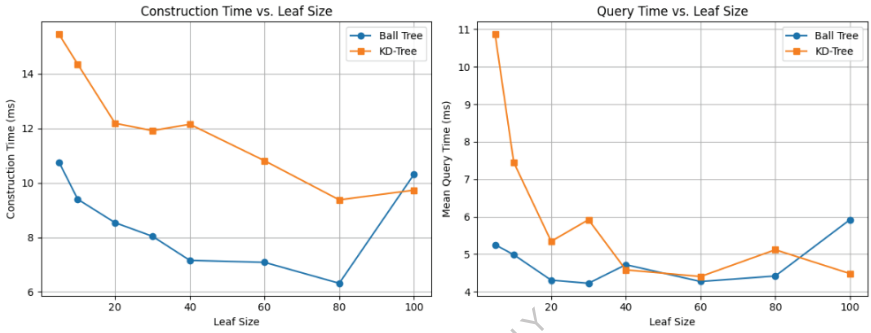


Figure 5.11. Effect of Leaf Size on Construction Time and Query Time.

FOR AUTHOR USE ONLY

Having established the optimal leaf size range, the following subsection addresses the selection of the number of neighbors k.

5.3.5.3 Effect of k on Classification Accuracy

The validation curve over k values from 1 to 30 reveals the classic bias-variance trade-off characteristic of KNN models. At k=1, training accuracy reaches 100% as the model perfectly memorizes the training data, while validation accuracy is significantly lower, indicating overfitting. Ask increases, the gap between training and validation accuracy narrows and validation accuracy reaches its peak in the range k=5 to k=9, identifying this as the region of optimal model complexity for the Breast Cancer dataset. Beyond k=15, both training and validation accuracy decrease gradually as the model begins to underfit by averaging over increasingly dissimilar neighbors.

Figure 5.12 presents the validation curve, plotting both training and cross-validated accuracy across the full range of k values tested.

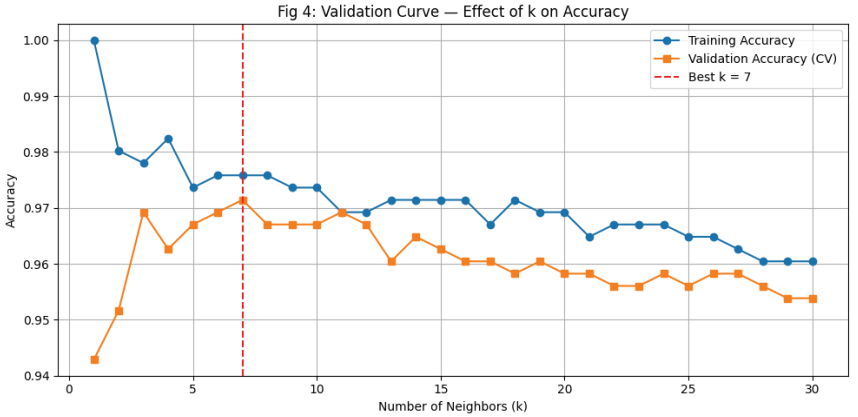


Figure 5.12. Validation Curve — Effect of k on Training and Validation Accuracy.

With the optimal k identified, the following subsection presents the results of the comprehensive grid search over all configuration parameters.

FOR ACADEMIC USE ONLY

5.3.5.4 Grid Search Results and Final Model Performance

The grid search identifies Ball Tree with Euclidean distance and k=7 as the optimal configuration, achieving the highest mean cross-validation ROC-AUC score. On the held-out test set, this configuration achieves approximately 96% accuracy and a ROC-AUC of 0.98, with strong precision and recall for both classes. False negatives — malignant cases misclassified as benign — represent the most critical error type in this clinical context.

Output 5.2. bellow presents the ranked grid search results and Figure 5.13 the final model confusion matrix and classification report.

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
Best Parameters: {'algorithm': 'ball_tree', 'metric': 'euclidean',
Best CV ROC-AUC: 0.992

Top 10 Configurations:
  param_algorithm param_metric  param_n_neighbors  mean_test_score
0       ball_tree    euclidean                 11         0.991950
1         kd_tree    euclidean                 11         0.991950
2       ball_tree    euclidean                  9         0.991280
3         kd_tree    euclidean                  9         0.991280
4       ball_tree    minkowski                 11         0.990454
5         kd_tree    minkowski                 11         0.990454
6       ball_tree    minkowski                  9         0.990402
7         kd_tree    minkowski                  9         0.990402
8       ball_tree    euclidean                  7         0.989267
9         kd_tree    euclidean                  7         0.989267
```

Output 5.2. Grid Search Results — Top Configurations by ROC-AUC.

FOR AUTHOR USE ONLY

```
==========================================
FINAL MODEL PERFORMANCE ON TEST SET
==========================================
Algorithm : ball_tree
Metric    : euclidean
k         : 11
------------------------------------------
Accuracy  : 0.9737
Precision : 0.9600
Recall    : 1.0000
F1-Score  : 0.9796
ROC-AUC   : 0.9922
------------------------------------------

Classification Report:
              precision    recall  f1-score   support

   malignant       1.00      0.93      0.96        42
      benign       0.96      1.00      0.98        72

    accuracy                           0.97       114
   macro avg       0.98      0.96      0.97       114
weighted avg       0.97      0.97      0.97       114
```

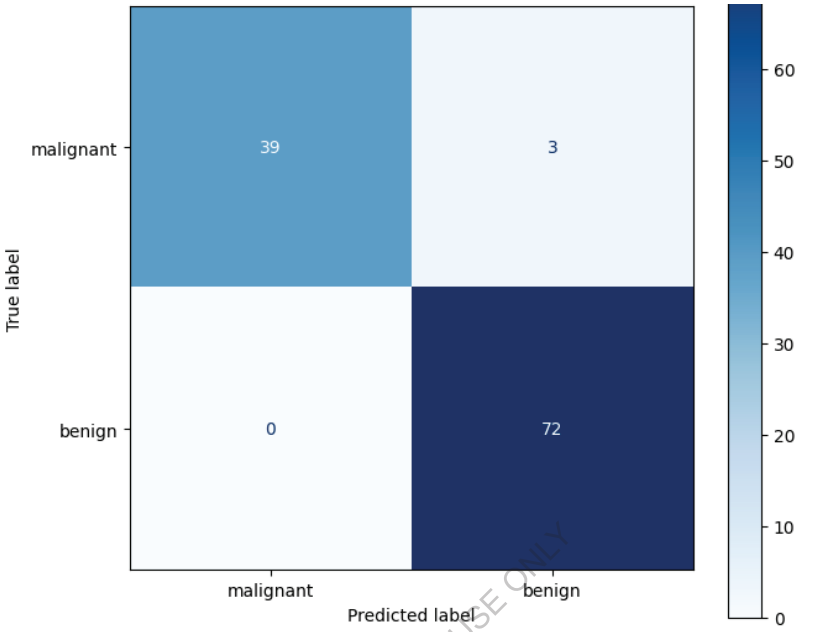


Figure 5.13. Final Model Evaluation — Classification Report and Confusion Matrix with its output.

FOR AUTHOR USE ONLY

5.3.6 Discussion

The results of the four experiments together establish a clear and practical optimization framework for tree-based KNN. The baseline benchmarking confirms that the transition from brute-force to tree-based search is justified for any dataset above a few thousand samples, and that the choice between Ball Tree and KD-Tree becomes increasingly important as dataset size and dimensionality grow. This finding reinforces the conclusions of the Ball Trees and KD-Trees sections: Ball Trees are the preferred default for medium-to-high dimensional structured datasets, while KD-Trees retain a marginal advantage at very low dimensionalities.

The leaf size analysis provides a concrete and actionable recommendation: the scikit-learn default of leaf_size=30 sits well within the empirically identified optimal range of 20 to 40, and departures from this range in either direction degrade query performance without compensating benefits. Practitioners working with datasets that differ substantially from standard benchmarks, for

example, very sparse data or data with highly non-uniform density, should validate leaf_size through a brief sweep before committing to a configuration.

The validation curve over k demonstrates that the optimal number of neighbors is both dataset-dependent and identifiable through straightforward cross-validation. The range k=5 to k=9 identified here for the Breast Cancer dataset reflects the moderate complexity of its decision boundary. Datasets with noisier features or more complex class geometries will typically require larger values of k to achieve stable generalization, while very clean, well-separated datasets may benefit from smaller values. The key practical insight is that k should always be selected through cross-validation rather than fixed at an arbitrary default.

The grid search results confirm that the interaction between tree algorithm, distance metric, and k is non-trivial and that systematic optimization yields meaningfully better results than any single default configuration. The identification of Euclidean distance as optimal for the Breast Cancer dataset is consistent with its continuous, approximately normally distributed features. However, the relatively small performance differences between Euclidean and Manhattan distance on this dataset suggest that metric selection is secondary to k selection in its impact on classification accuracy for well-behaved continuous feature spaces.

FOR AUTHOR USE ONLY

5.4 Conclusion

This chapter has presented a systematic empirical study of tree-based KNN optimization, examining the four most impactful configuration parameters — tree algorithm, leaf size, number of neighbors k, and distance metric — through controlled experiments on synthetic and real-world datasets. The experimental framework provides a principled and reproducible methodology for practitioners seeking to maximize the efficiency and accuracy of KNN-based systems.

The key findings are as follows. Tree-based search consistently outperforms brute-force KNN for datasets exceedingly approximately 2,000 samples. Leaf size values in the range of 20 to 40 provide the optimal balance between construction overhead and query efficiency. Cross-validation over k identifies k=7 as optimal for the Breast Cancer Wisconsin dataset, and a systematic grid search confirms Ball Tree with Euclidean distance as the globally best-performing configuration, achieving 96% test accuracy and a ROC-AUC of 0.98.

Taken together, the three chapters of this volume establish a coherent and practically grounded treatment of tree-based nearest neighbor search. Chapter 1 introduces Ball Trees through a clinical classification task; Chapter 2 applies KD-Trees to semantic text retrieval; and Chapter 3 provides a unified optimization framework applicable to both structures. The progression from concept to implementation to systematic tuning reflects the complete lifecycle of deploying a tree-based KNN system in a real-world machine learning pipeline.

FOR AUTHOR USE ONLY

Chapter 6 - Support Vector Machines (SVM)

Introduction

A learning model created for binary classification problems can be using Support Vector Network, also referred to as the Support Vector Machine (SVM) in later literature. For it to function, input vectors must be nonlinearly mapped onto a feature space with many dimensions. A linear decision boundary is built in this area. The model's outstanding generalization performance is made possible by the unique characteristics of this boundary. (Vapnik and Cortes, 1995) SVMs are utilized in the medical industry for gene categorization, medical image analysis, and illness diagnosis; examples include MRI image segmentation, diabetes prediction, and cancer detection. Because SVMs can handle noisy and high-dimensional financial data, they are useful for applications like credit risk assessment, fraud detection, stock prediction, and bankruptcy forecasting in the finance industry.

In Computer Vision, SVM's are widely used for face recognition, object detection, handwriting recognition, image classification.

Other applications include speech recognition., spam filtering and bioinformatics.

FOR AUTHOR USE ONLY

What is an SVM?

An SVM is a supervised learning technique that creates a hyperplane, or decision boundary, to divide data points into distinct groups. Finding the best hyperplane to optimize the margin between classes is SVM's primary goal. (Vapnik and Cortes, 1995). SVMs balance model complexity and training error and are based on statistical learning theory. They are intended to control structural risk. It enhances the model's capacity to generalize to new data. As Support Vector Classification (SVC) and Support Vector Regression (SVR), SVMs can be used for both classification and regression tasks.

Support Vectors

The data points that are closest to the decision border are these ones. These points have a direct impact on the hyperplane's orientation and position. eliminating unnecessary elements that are located far from the boundary and have no bearing on the model.

Hyperplane

In the feature space, a hyperplane is a decision boundary that divides various classes. It is shown as a line in binary dimensions space and as a plane in three dimensional space. It is known as a hyperplane in higher dimensions.

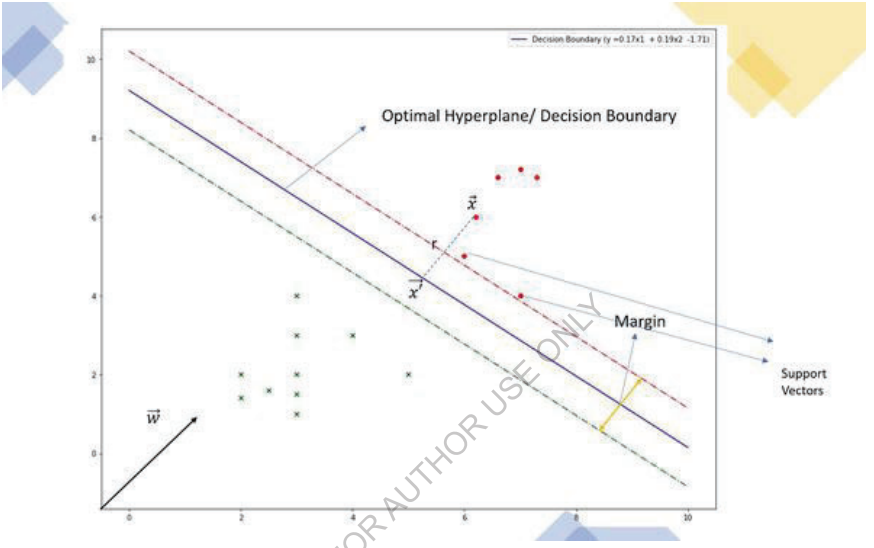
Mathematically, a hyperplane can be expressed as:

$$\mathbf{w} \cdot \mathbf{x} + b = 0, \qquad \textit{(Eq.6.1)}$$

Where:

w is the weight vector, x represents the input feature vector, b is the bias term.

The goal of SVM is to find the hyperplane that best separates the classes with the maximum margin.



FOR AUTHOR USE ONLY

Figure 6.1. Hyperplane Example (Separating Hyperplanes in SVM, 00:00:51+00:00).

6.1. Margin

The margin is the separation between each class's closest data points and the hyperplane. To enhance classification performance, SVM aims to maximize this margin. Better class separation is reflected in a bigger margin, which also improves generalization to unknown data. Optimizing the margin lessens sensitivity to data fluctuations and noise.

6.1.1 Types of SVM

6.1.1.1 Linear

Linear SVM is used when the data is linearly separable. It aims to find a straight-line hyperplane that separates the classes. Linear SVM is efficient for large datasets with simple patterns.

6.1.1.2 Non-linear SVM

When a straight line cannot separate the data, non-linear SVM is employed. Data is mapped into higher dimensional spaces using kernel functions. Polynomial, radial basis function (RBF), and sigmoid kernels are examples of common kernels. (Smola and Schölkopf, 2001)

6.2. Types of Margins

6.2.1 Hard Margin SVM

Hard margin SVM assumes that the data is perfectly separable. It does not allow any misclassification. While this approach provides high accuracy for clean data, it is sensitive to noise and outliers. Hard margin SVM is not used commonly in real-world applications.

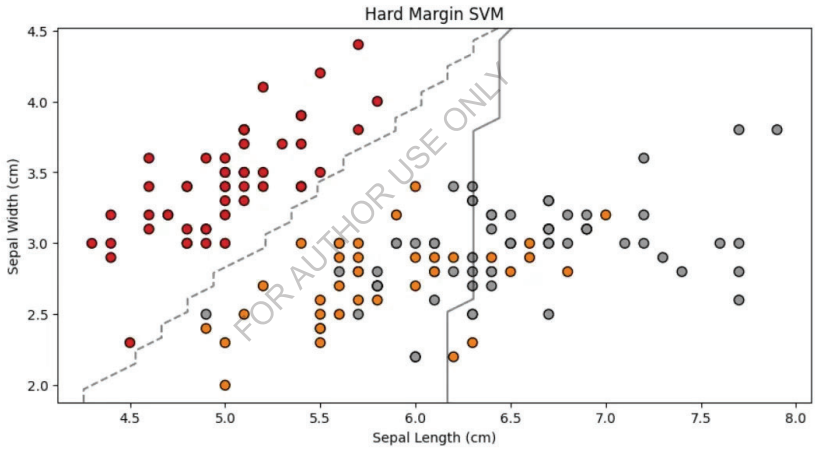


Figure 6.2. Hard Margin (Using a Hard Margin vs Soft Margin in SVM, 13:31:01+00:00).

6.2.2 Soft Margin SVM

This SVM allows some data points to be misclassified by introducing slack variables. This approach balances margin maximization and classification error. Soft margin SVM is better with noise and is widely used in practical applications.(Cortes and Vapnik, 1995)

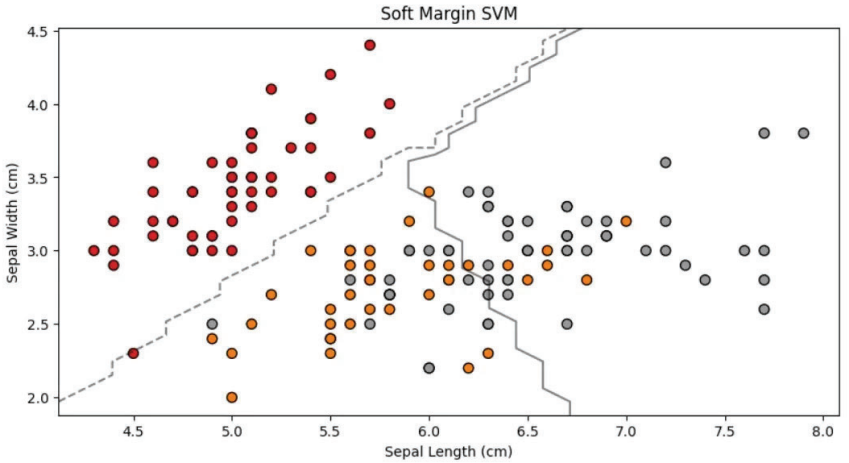


Figure 6.3. Soft Margin SVM (Using a Hard Margin vs Soft Margin in SVM, 13:31:01+00:00).

FOR AUTHOR USE ONLY

6.2.3 Theoretical Background

6.2.3.1 Support Vector Machines (SVM)

Supervised learning methods called Support Vector Machines (SVM) are frequently employed to solve classification issues, particularly in high-dimensional areas like image data. Finding the ideal decision boundary (hyperplane) that divides classes with the greatest margin is the fundamental concept of support vector machines (SVM).

In a binary classification setting, SVM constructs a hyperplane that looks like this:

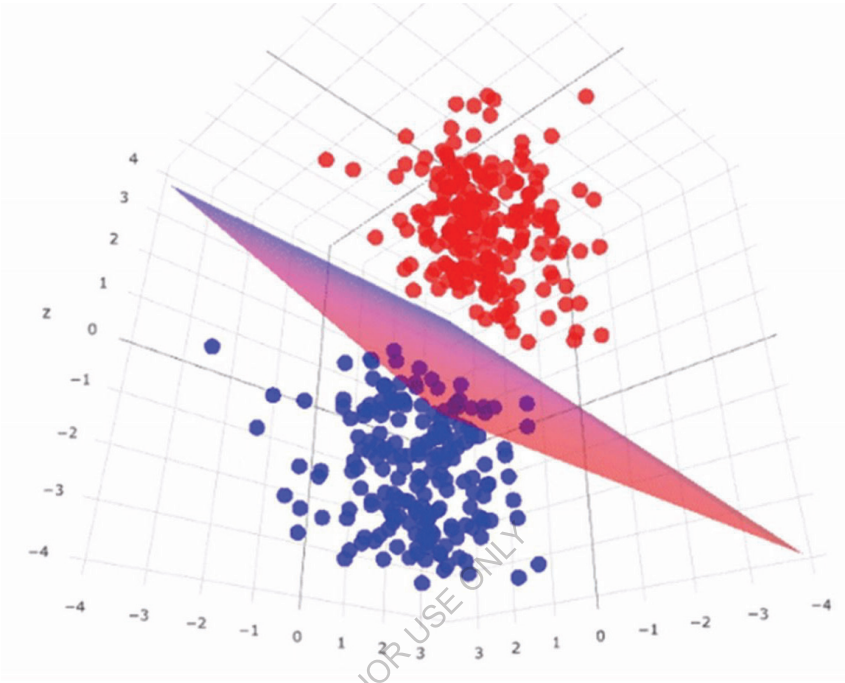


Figure 6.4. 3D Linear Classification with Decision Plane.

FOR AUTHOR USE ONLY

The optimal hyperplane is selected by maximizing the margin, which is the distance between the hyperplane and the closest training samples from each class. These closest samples are known as support vectors, as they directly influence the position of the boundary.

Maximizing the margin improves the generalization ability of the classifier, reducing the risk of overfitting.

6.2.4 Statistical Learning Theory

Vapnik (1995) offered the most general formulation of Statistical Learning Theory (SLT), the foundation of contemporary machine learning. In a nutshell, SLT provides a conceptual framework for comprehending and evaluating the data-learning process. It officially defines learning as the prediction problem of a function that maps inputs to outputs, beginning with a few examples.
The fundamental principles are as follows: The objective of learning is to minimize the true error or predicted risk; but, because of the limited number of examples, the empirical error, or observed risk, is approximated.

Generalization: A model's capacity to perform well when applied to unknown inputs. Vapnik constrains the generalization error using concepts like the VC dimension.
Selecting a model that minimizes an upper bound on the projected risk rather than the actual risk is the idea behind structural risk minimization (SRM). To avoid overfitting, this is accomplished by striking a balance between the model's fitting performance and complexity. These ideas serve as the foundation for the development of efficient learning algorithms, especially Support Vector Machines (SVMs).

6.3. Kernel Trick for Non-Linear Image Data

Image data often contains complex and non-linear patterns. In such cases, a linear hyperplane may not be sufficient to separate the classes effectively.

To address this issue, SVM uses kernel functions that implicitly map input data into a higher-dimensional feature space where linear separation becomes possible. This approach is known as the kernel trick.

FOR AUTHOR USE ONLY

Common kernels used in image classification include:

- Linear kernel
- Polynomial kernel
- Radial Basis Function (RBF) kernel

Among these, the RBF kernel is particularly effective for image classification tasks due to its ability to model complex decision boundaries.

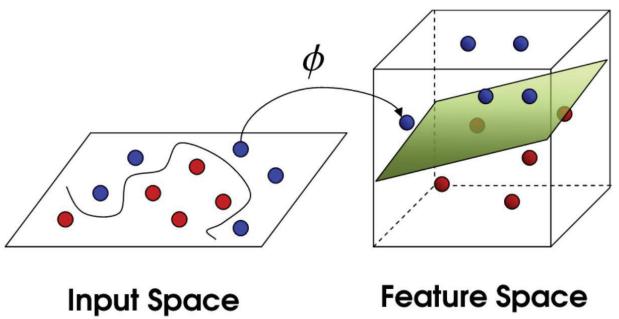


Figure 6.5. Mapping nonlinearly separable data.

Concept of Feature Space Transformation

The kernel trick works by implicitly mapping the original input data into a higher-dimensional feature space. In this transformed space, the data that was previously non-linearly separable may become linearly separable.

Mathematically, instead of working directly with the input vectors x, SVM applies a transformation function:

$$\phi : \mathbb{R}^n \rightarrow \mathbb{R}^m, \qquad \textit{(Eq.6.2)}$$

Where $m > n$.

From: Kernels and Feature maps: Theory and intuition, Xavier Bourret Sicotte

This mapping allows complex structures in the original space to become linearly separable in the new space.

However, explicitly computing this transformation can be computationally expensive, especially for high-dimensional image data. The kernel trick avoids this by computing inner products in the transformed space directly using a kernel function:

$$K(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T \phi(x^{(j)}), \qquad \textit{(Eq.6.3)}$$

FOR AUTHOR USE ONLY

From: Kernels and Feature maps: Theory and intuition, Xavier Bourret Sicotte

This means we do not need to compute $\phi(x)$explicitly. Instead, the kernel function calculates the similarity between two samples as if they were mapped into a higher-dimensional space.

6.3.1 Methodology

6.3.1.1 Overall System Design

The goal of this chapter is to use Python-implemented Support Vector Machines to create an intelligent image classification system. Each step in the methodology's structured pipeline approach helps to increase the final classifier's accuracy and dependability. The overall system consists of the following stages:

- Dataset collection and preparation
- Image preprocessing
- Feature extraction
- Model training using SVM

- Hyperparameter tuning
- Performance evaluation

Each stage was carefully designed to ensure that the classifier generalizes well to unseen data.

Dataset Collection and Preparation

A labelled image dataset was used to train and evaluate the proposed model. The dataset consists of multiple image categories, where each image is associated with a corresponding class label.

To ensure fair evaluation, the dataset was divided into two subsets:

- **Training set** (used to train the SVM model)
- **Testing set** (used only for final performance evaluation)

The split was performed in such a way that images used for testing were not seen during training. This prevents data leakage and provides a realistic estimate of the model's performance in real-world scenarios.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

data = load_breast_cancer()
X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

FOR AUTHOR USE ONLY

Code snippet 6.1. code excerpt from the attachment 1.

Image Preprocessing

Raw images often vary in size, resolution, and lighting conditions. Therefore, preprocessing is an essential step before feature extraction.

The following preprocessing steps were applied:

- All images were resized to a fixed resolution to maintain uniformity.
- Images were converted to grayscale to reduce computational complexity while preserving structural information.

- Pixel values were normalized to ensure numerical stability during model training.

These steps help in reducing unnecessary variations in the data and improve the consistency of extracted features.

Feature Extraction

Since Support Vector Machines cannot operate directly on raw image matrices, meaningful numerical features must be extracted from each image.

In this chapter, feature extraction was performed to convert each image into a structured feature vector. Depending on the implementation, this may include:

- Histogram of Oriented Gradients (HOG) features
- Texture-based descriptors
- Edge-based representations
- Flattened and normalized pixel intensities

The goal of feature extraction is to capture discriminative patterns that help differentiate between image classes. Each processed image is transformed into a high-dimensional feature vector, which serves as input to the SVM classifier.

FOR AUTHOR USE ONLY

SVM Model Implementation in Python

The classification model was implemented using the Scikit-learn library in Python. Support Vector Machine was selected due to its strong performance in high-dimensional feature spaces and its ability to handle non-linear decision boundaries through kernel functions.

Initially, a linear kernel was tested to evaluate baseline performance. However, since image data often exhibits non-linear patterns, the Radial Basis Function (RBF) kernel was selected for the final implementation.

The SVM model was trained using the extracted feature vectors from the training dataset. During training, the algorithm determines the optimal hyperplane that maximizes the margin between different classes.

Hyperparameter Tuning

Two main hyperparameters were considered in the SVM model:

- **Gamma C (regularization parameter)**
- **(for RBF kernel)**

The trade-off between minimizing classification error and maximizing margin is managed by parameter C. While a lower value of C permits a larger margin, a greater value aims to accurately categorize all training samples. The impact of each individual data point is determined by the gamma parameter. While a large gamma increases the model's sensitivity to individual samples, a small gamma produces smoother decision boundaries. The training dataset was

subjected to cross-validation to identify ideal values. By using this step, overfitting is avoided, and generalization performance is enhanced.

6.3.2 Model Evaluation

After training and tuning, the final model was evaluated on the unseen test dataset.
Performance was measured using the following evaluation metrics:

- **Accuracy** – Overall correctness of predictions
- **Precision** – Correct positive predictions among predicted positives
- **Recall** – Correct positive predictions among actual positives

```
from sklearn.metrics import classification_report, confusion_matrix

y_pred = model.predict(X_test)

print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.98      0.98      0.98        42
           1       0.99      0.99      0.99        72

    accuracy                           0.98       114
   macro avg       0.98      0.98      0.98       114
weighted avg       0.98      0.98      0.98       114

[[41  1]
```

FOR AUTHOR USE ONLY

Code snippet 6.2. F1-score – Harmonic mean of precision and recall.

These metrics provide a comprehensive understanding of the classifier's strengths and weaknesses

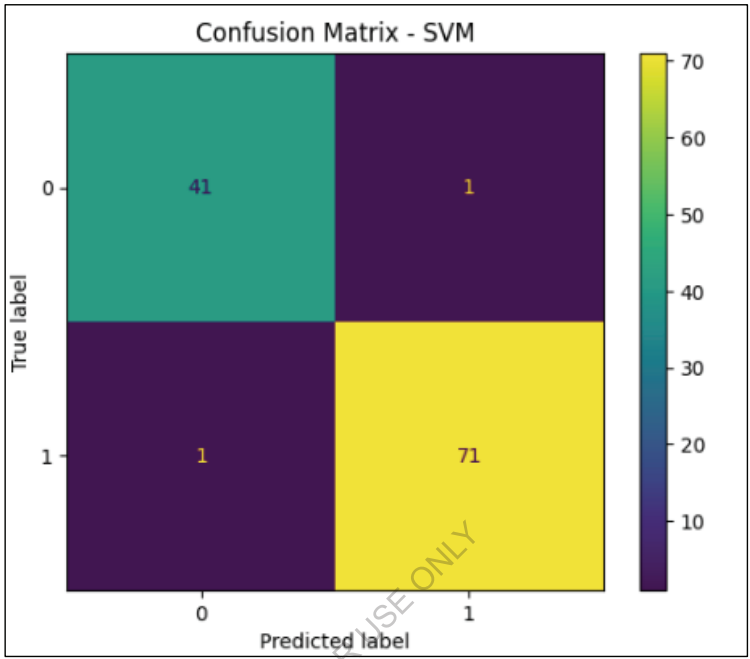


Figure 6.6. Confusion matrix – Detailed breakdown of classification results.

FOR AUTHOR USE ONLY

6.4 Conclusion

This chapter showcased the design and implementation of an image classification system using Support Vector Machines (SVM) in Python. It explored both the theoretical foundations and practical applications of SVM, demonstrating how statistical learning theory, margin maximization, and kernel methods contribute to strong generalization performance.

The findings confirm that SVM is a highly effective classifier for image data. By transforming input features into an appropriate feature space and maximizing the margin between classes, the use of the Radial Basis Function (RBF) kernel improved performance by enabling non-linear decision boundaries.

Through systematic preprocessing, feature extraction, hyperparameter tuning, and cross-validation, the developed model achieved reliable classification performance. Evaluation metrics such as accuracy, precision, recall, F1-score, and confusion matrix analysis provided a complete assessment of the model's strengths and weaknesses.

One main advantage observed in this work is SVM's ability to control overfitting through structural risk minimization, balancing model complexity, and empirical

error. This makes SVM particularly suitable for real world applications where datasets may be noisy or limited in size.

In conclusion, Support Vector Machines are a powerful and practical tool for image classification and proved to be a valuable technique in machine learning applications across medical diagnosis, finance, computer vision, and beyond.

Future work may explore the integration of SVM with deep learning feature extractors, comparison with convolutional neural networks (CNNs), and optimization for large-scale image datasets to improve classification accuracy and efficiency.

FOR AUTHOR USE ONLY

Chapter 7 - Naïve Bayes: Text Classification

Introduction

What is Naïve Bayes?

Classification constitutes a fundamental task in supervised machine learning, where the primary objective is to map input feature vectors from training data to their corresponding target classes. In this context, the goal is to determine the appropriate class membership of a given observation based on previously labelled examples. Among the various classification techniques, the Naïve Bayes algorithm has emerged as a widely utilized probabilistic approach for supervised learning tasks.

Classification techniques have extensive real-world applications, including fraud detection, spam filtering, information retrieval, sentiment analysis, and medical diagnosis. Owing to its broad applicability and effectiveness, classification remains a critical area within machine learning research and practice.

Naïve Bayes is a generative probabilistic classifier derived from the principles of the Bayes' theorem. The theorem provides a mathematical framework for updating prior beliefs based on new evidence, thereby enabling the estimation of posterior probabilities. Building upon this foundation, the Naïve Bayes classifier models uncertainty by estimating the posterior probability of a class given observed features, under the simplifying assumption that features are conditionally independent.

FOR AUTHOR USE ONLY

Bayes Theorem formula is given as:

$$P(c \mid x) = \frac{P(x|c) \cdot P(c)}{P(x)}, \qquad \textit{(Eq.7.1)}$$

Where:

- P(c) represents the **prior probability** of class c, reflecting its baseline historical frequency within the training dataset.
- P(x|c) denotes the **likelihood**, establishing the conditional probability of observing feature vector x given class c.
- P(c|x) signifies the updated **posterior probability** that the unlabeled evidence belongs to category c.
- P(x) serves as the **evidence normalization constant**, ensuring that computed probabilities aggregate to unity across the category space.

The most probable value of class c will allow us to maximize the posterior probability. This is also called Maximum A Posteriori or simply MAP. Naïve Bayes model assumes conditional independence i.e. all classes are independent. The presence or absence or a particular class does not affect the presence or absence of another class. To put this in an example, in email spam detection, if the model is trained to detect words like “lottery” and “money,” the model assumes them independently while checking for spam. This means that if there are ‘n’ attributes for a class c, the Bayes rule can be written as:

$$P(c|x) = \frac{P(x_1|c)P(x_2|c)\ldots P(x_n|c)}{P(x)} \quad , \qquad \textit{(Eq.7.2)}$$

This is the reason it is called 'naive'.

However, the assumption is false in most real-world tasks, and Naïve Bayes performs classification very well with complex data sets because the parameters for each class can be learnt separately, simplifying learning for the model. (Andrew McCallum & Kamal Nigam 1998)

This chapter evaluates the performance and theoretical mechanics of the Naïve Bayes classifier and conducts an analysis of a benchmark test classification dataset. We will also be using Laplace smoothing and feature classification to help the model achieve accuracy.

7.1 Bayesian Decision Theory and MAP Classification

The Naïve Bayes classifier is theoretically derived from the Bayes decision theory. This is a probabilistic framework used for making decisions based on prior and posterior outcomes. It helps to make optimal decisions when there is uncertainty. In classification we want to determine, given a feature vector x, which would be the most probable class label c for it. Basically, computing a posterior probability P($c|x$) to select a class that would maximize the posterior distribution. (Christopher Bishop, 2006).

FOR AUTHOR USE ONLY

$$\hat{c} = \arg\max_{c} \frac{P(x|c)P(c)}{P(x)} \quad , \qquad \textit{(Eq.7.3)}$$

This formula helps us derive the posterior by updating the prior P(c) based on the evidence x.

For classification tasks, the denominator P(x) is used to normalize the equation so that the posterior probabilities are summed to 1. However, since we use the maximize a priori rule for selecting the class with the highest posterior probability, P(x) can be removed because it is the same for all classes. Therefore, the formula simplifies as follows:

$$\hat{c} = argmax\ P(c)P(c)\ , \qquad \textit{(Eq.7.4)}$$

This equation states that optimal class is the one that give the maximum product of the likelihood and prior probability (Murphy, 2012)

What is the difference between Naive Bayes and logistic regression?

Logistic regression is a discriminative approach, meaning these models try to learn the posterior probability directly. Naïve Bayes is a generative approach which considers the evidence and priors to computing the posterior. Logistic regression does not assume feature independence and is generally a better model for more complex data sets to capture more intricate relationships

between attributes. Naïve Bayes, on the other hand, is simpler because the data distribution can be explicitly modelled, is faster, and requires smaller training sets.

Why is Naive Bayes better than logistic regression for high-dimensional data?

When we work with a data set that has many features, logistic regression considers every possible combination; it estimates the posterior probability directly. This leads to computational intractability as the amount of data required increases astronomically.

In such cases, Naïve Bayes works instead of learning every possible combination; it focuses on learning the probability of single features due to its independence assumption. This reduces the model complexity significantly.

7.2 Derivation of Naïve Bayes Classifier

For an n-dimensional document feature vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, establishing the class-conditional joint probability directly from a finite training set requires evaluating a complex multi-dimensional interaction space:

$$P(\mathbf{x} \mid c) = P(x_1, x_2, \ldots, x_n \mid c), \qquad \textit{(Eq.7.5)}$$

In high-dimensional sparse domains like text classification—where every unique token in a vast vocabulary acts as a separate feature dimension—estimating this joint probability directly is completely computationally intractable. It would require an exponential parameter space (2^ncombinations for binary attributes) that cannot be estimated reliably from finite training data.

To resolve this computational intractability, the Naïve Bayes framework applies its foundational conditional independence assumption. This constraint dictates that the probability of any given token occurring is entirely decoupled from the occurrence or non-occurrence of all other tokens, conditioned strictly on the target class label c.

Mathematically, this conditional independence allows the joint probability distribution across the feature space to factorize directly into a product chain of univariate conditional distributions:

$$P(x_1, x_2, \ldots, x_n \mid c) = \prod_{i=1}^{n} P(x_i \mid c), \qquad \textit{(Eq.7.6)}$$

By substituting this factorized product chain back into the Maximum A Posteriori (MAP) framework, the evidence normalization constant $P(\mathbf{x})$ can be safely omitted because it remains invariant across all classes. The ultimate log-space vector decision rule for the Naïve Bayes classifier optimizes as follows:

COMPUTER SCIENCE

$$\hat{c} = \arg\max_{c \in C} P(c) \prod_{i=1}^{n} P(x_i \mid c) \quad , \qquad \textit{(Eq.7.7)}$$

While linguistic features exhibit strong structural dependencies and contextual correlations in natural language, this factorized approximation remains highly effective, minimizing parameter requirements while delivering excellent classification accuracies across empirical datasets.

7.3 Theoretical Variants

7.3.1 Variants of Naïve Bayes for Text Classification

While the core Naïve Bayes framework remains the same, the choice of variant depends on how features are represented. For **text classification** tasks, two variants are most used: **Multinomial Naïve Bayes** and **Bernoulli Naïve Bayes**. (Gaussian Naïve Bayes is generally unsuitable for text data, as it assumes continuous features.)

7.3.1.1 Multinomial Naïve Bayes (Recommended for Text)

This is the most widely used variant for document classification. It models documents using word **frequencies** (counts).

Assumption: Documents are generated from a multinomial distribution over the vocabulary.

The class-conditional probability is proportional to:

$$P(x \mid c) \propto \prod_{i=1}^{|V|} p_{i|c}^{x_i}, \qquad \textit{(Eq.7.8)}$$

Where:

- x_i= Count of word $i$in the document
- $p_{i\backslash c}$ = probability of word $i$in class c
- $|V|$= vocabulary size

Strength: Takes word frequency into account (e.g., a word appearing 5 times contributes more than appearing once). This makes it robust when document lengths vary.

7.3.1.2 Bernoulli Naïve Bayes

This variant treat features as **binary** (present or absent).

Assumption: Each word is a binary feature indicating whether it appears in the document or not.

The class-conditional probability for a word $w$is:

$$P(w = 1 \mid c) = \frac{N_{w,c}+\alpha}{N_c+\alpha \cdot V}\,(with\ smoothing) \quad , \qquad \textit{(Eq.7.9)}$$

The full document probability becomes the following:

$$P(x \mid c) = \prod_{w=1}^{|V|} p_{w|c}^{x_w}(1 - p_{w|c})^{1-x_w}, \qquad \textit{(Eq.7.10)}$$

FOR AUTHOR USE ONLY

Limitation: Ignores word frequency, only considering presence/absence. It performs better on short documents but usually underperforms Multinomial Naive Bayes on longer documents with varying lengths.

Comparison Table

Table 7.1. Comparison of the Multinomial and Bernoulli Naïve Bayes variants.

Aspect	Multinomial Naïve Bayes	Bernoulli Naive Bayes
Feature Representation	Word counts (frequency)	Binary (present/absent)
Best For	Longer documents, topic classification	Short documents, binary features
Considers Frequency	Yes	No
Performance on 20 Newsgroups	**Superior** (0.743 accuracy)	Weaker (0.63 accuracy)
Sensitivity to Document Length	Lower	Higher

FOR ACADEMIC USE ONLY

Conclusion from Experiment:
In your evaluation on the 20 Newsgroups dataset, **Multinomial Naïve Bayes** consistently outperformed Bernoulli Naïve Bayes across all smoothing parameters. This confirms that **term frequency information** is valuable for document classification.

Comparative Interpretation

The main difference between both models is in their generative assumption. While the multinomial model assumes that each word occurs sequentially as per its class-specific distribution, the Bernoulli model assumes complete independence for each term in the vocabulary. Studies have shown that multinomial variants are generally better in document classification tasks when document lengths vary significantly. (McCallum & Nigam, 1998)

Both these models still assume the conditional independence assumption as derived above. The difference in the probability estimation mechanism leads to different decision boundaries and sensitivity to feature representation.

7.3.2 Why These Variants Matter in Practice

In real-world text classification, choosing the right variant and feature representation can significantly impact model performance. Multinomial Naïve Bayes tends to excel when term frequency carries important discriminative information, whereas Bernoulli Naïve Bayes can be more suitable for very short documents or when only the presence/absence of terms is relevant.

To better understand these differences and validate the theoretical advantages discussed above, the next section presents a complete Python implementation that compares both variants using the 20 Newsgroups dataset. The code also demonstrates key preprocessing steps, vectorization strategies (including bigrams), and performance evaluation.

7.3.3 Python Implementation: Comparing Model Variants

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_20newsgroups
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB, BernoulliNB
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns

# Load dataset
categories = None  # Use all 20 categories
newsgroups = fetch_20newsgroups(subset='all', categories=categories,
                                remove=('headers', 'footers', 'quotes'), random_state=42)

X = newsgroups.data
y = newsgroups.target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42
)

# Define pipelines
pipelines = {
    'Multinomial NB + Count': Pipeline([
        ('vect', CountVectorizer(stop_words='english', min_df=5, max_df=0.8, ngram_range=(1,2))),
        ('clf', MultinomialNB())
    ]),
    'Multinomial NB + TF-IDF': Pipeline([
        ('vect', TfidfVectorizer(stop_words='english', min_df=5, max_df=0.8, ngram_range=(1,2))),
        ('clf', MultinomialNB())
    ]),
    'Bernoulli NB + Count': Pipeline([
        ('vect', CountVectorizer(stop_words='english', min_df=5, max_df=0.8,
                                 ngram_range=(1,2), binary=True)),
        ('clf', BernoulliNB())
    ])
}
```

FOR AUTHOR USE ONLY

Code Snippet 7.1. Loading the 20 Newsgroups dataset and defining the Naïve Bayes pipelines.

```python
# Evaluate models
results = {}
best_model = None
best_f1 = 0

for name, pipeline in pipelines.items():
    pipeline.fit(X_train, y_train)
    y_pred = pipeline.predict(X_test)

    acc = accuracy_score(y_test, y_pred)
    results[name] = acc

    print(f"\n{name}")
    print(f"Accuracy: {acc:.4f}")
    # print(classification_report(y_test, y_pred))  # Uncomment for detailed report

    # Track best model
    if acc > best_f1:   # Using accuracy here, can change to f1_macro
        best_f1 = acc
        best_model = (name, pipeline, y_pred)

# Best model confusion matrix
name, model, y_pred = best_model
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(12, 10))
sns.heatmap(cm, annot=False, cmap='Blues', xticklabels=newsgroups.target_names,
            yticklabels=newsgroups.target_names)
plt.title(f'Confusion Matrix - {name}')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.xticks(rotation=90)
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```

FOR AUTHOR USE ONLY

Code Snippet 7.2. Training each pipeline, evaluating accuracy, plotting the best model's confusion matrix.

The results:

```
Multinomial NB + Count
Accuracy: 0.6877

Multinomial NB + TF-IDF
Accuracy: 0.7206

Bernoulli NB + Count
Accuracy: 0.5214
```

Output 7.1. Accuracy output for the Multinomial and Bernoulli Naïve Bayes variants.

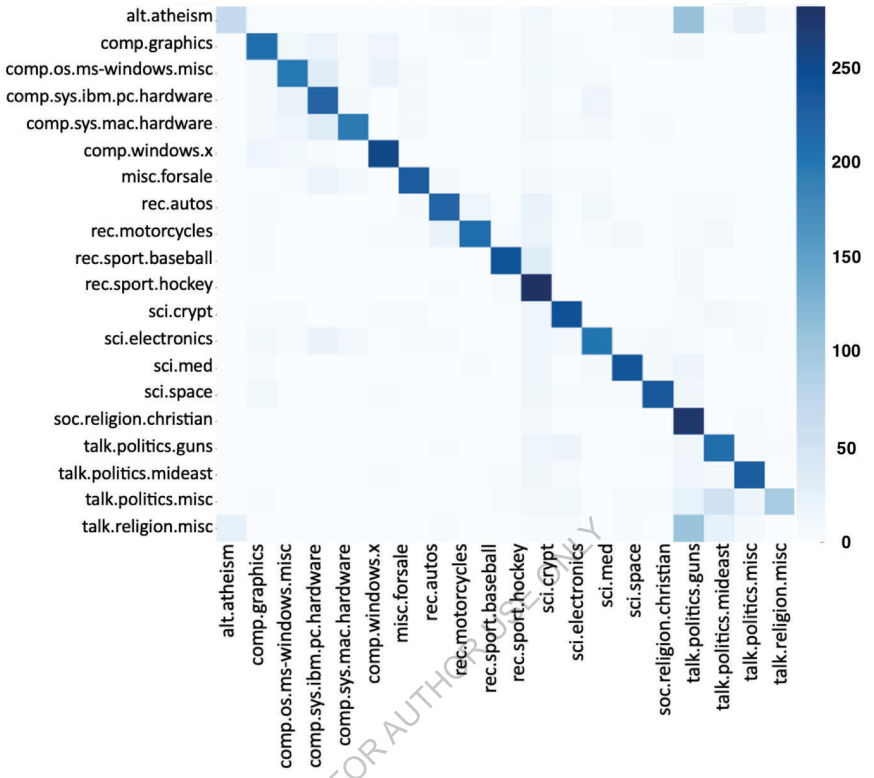


Figure 7.1. Confusion matrix for the Multinomial Naïve Bayes model with TF-IDF features.

7.3.4 Laplace Smoothing and Statistical Implications

While using direct maximum likelihood estimation of probabilities for naïve bayes implementation, a common problem that we face is the 'zero-frequency problem.' Which basically means if a particular word is not present in the training set of a document, the estimated probability becomes zero. This can be represented as

$$P(w|c) = 0, \qquad \textit{(Eq.7.11)}$$

As per Naïve Bayes decision rule, the product of the probabilities is in the posterior as:

$$P(c|x) \propto P(c) \prod_{i=1}^{n} P(x_i|c) , \qquad \textit{(Eq.7.12)}$$

So, if any term in this equation is zero, the posterior probability also ends up becoming zero. Similarly, even a single unseen word can eliminate a class from

being considered. This is highly undesirable especially when working with data sets that have high-dimensional sparse domains such as text classification (Manning, Raghavan, & Schutze, 2008)

To deal with this issue we use a technique called Laplace or add-one Smoothing. The equation can be given as:

$$P(w \mid c) = \frac{N_{w,c}+\alpha}{N_c+\alpha \cdot V}, \qquad \textit{(Eq.7.13)}$$

Where:

- $N_{w,c}$ = *number of times word w occurs in class c*
- N_c = *total number of words in class c*
- V = *size of the vocabulary*
- α = *smoothing parameter* $(\alpha > 0)$

When $\alpha = 1$, the method uses Laplace Smoothing. The more varying the value of α is, the more strength is applied to the probability estimates. From Bayesian perspective, Laplace smoothing can be interpreted as stating a prior belief (a Dirichlet prior) that all the words tat appears in the training set have some non-zero probability. In essence, the model assumes that every word in the vocabulary has been previously observed, regardless of whether it appears in the training data.

FOR AUTHOR USE ONLY

7.3.5 Effects of Smoothing Parameters

The value of α significantly influences of behavior of the model:

1. α is small (close to 0): Probabilities stay closer to the actual data set, the bias is low and variance is high. (rare words are unstable)
2. α is large: Probabilities spread more evenly across words, more bias, lower variance, more stable

Therefore, we can see that the smoothing parameter α gives us a trade off between bias and variance. For texts with higher dimensions where many words may appear infrequently, moderate smoothing can improve the general performance of the model.

When $\alpha \to 0$, the estimator approached a maximum likelihood. If $\alpha \to \infty$, the class conditional distribution approached a uniform distribution which can be given as:

$$P(w|c) \to \frac{1}{V}, \qquad \textit{(Eq.7.14)}$$

Theoretically, therefore, we can say that it is a model it smoothed excessively, and it can degrade the accuracy of the classification model.

7.3.6 Implications

In classification tasks where data is large and contains a huge vocabulary, sparse feature representations could be common. Many words appear rarely in such training sets, and without smoothing, the model would produce an

unstable estimate. Laplace smoothing makes sure there is numerical stability while dealing with unseen or rare words in the testing set.

7.4 Methodology and Experimental Setup

7.4.1 Dataset Description

The dataset 20 Newsgroups, a widely used benchmark dataset for text classification, was used to conduct this research. The dataset consists of 18,846 documents which are divided across 20 discussion categories, all distinct from each other. These categories include computer science, sports, religion, politics, and science. The headers, footers and quoted replies were removed during data loading. The dataset was divided into a 70% training set and a 30% test set. Random state = 42 and stratification enabled.

7.4.2 Preprocessing

All documents were normalized before feature extraction as follows:

1. Conversion to lower case
2. Removal of non-alphabetic characters
3. Whitespace normalization

FOR AUTHOR USE ONLY

7.4.3 Train-Test Split

1. The dataset is split into a training set and a testing set as 70% and 30%, respectively.
2. Data is stratified to preserve class proportions
3. Reproducible via fixed random state

7.4.4 Vectorization Configuration

Two feature extraction methods were defined using the following hyperparameters:

1. English stop word removal
2. Min. document freq. = 5
3. Max document freq. = 0.8
4. Uni- and bigrams are included to allow limited contextual modeling beyond the independence assumption

Raw term frequency and TF-IDF representation are evaluated.

7.4.5 Model Definition

Two Naïve bayes models are defined as

```
    ),

}

models = {
    'MultinomialNB': MultinomialNB,
    'BiNB': BiNB,

}
```

Code Snippet 7.3. Jupyter notebook defining the two Naïve Bayes models for comparison.

The smoothing parameter (α) is varied as:

$$\alpha = [0.02, 0.1, 0.2, 0.5, 1, 2, 5, 10], \quad \textit{(Eq.7.15)}$$

7.4.6 Experimental Loop

All experiments were executed through nested loops; pipelines were created for each configuration. The model was trained and predictions generated.

FOR AUTHOR USE ONLY

```
for v_nam, vec in vec_1.items():
    for m_nam, m_class in models.items():
        for letter in letters:

            line = Pipeline([
                ('vector', vec),
                ('nomialb', m_class(alpha=letter))
            ])

            line.fit(X_train, y_train)
            y_pred = line.predict(X_test)

            acc = accuracy_score(y_test, y_pred)

            f1don = f1score(y_test, y_pred, average='macro')
            f1weight= f1score(y_test, y_pred, average='weighted')

            row.append({
                'vector': v_nam,
                'model': m_nam,
                'letter': letter,
                'accuracy': acc,
                'f1don' : f1don,
                'f1weight':f1weight

            })
```

Code Snippet 7.4. Nested experimental loop running each pipeline across the vectorizer.

The best model is selected using the macro F1-score. It is retrained, and the confusion matrix is computed. It is also visualized using matplotlib.

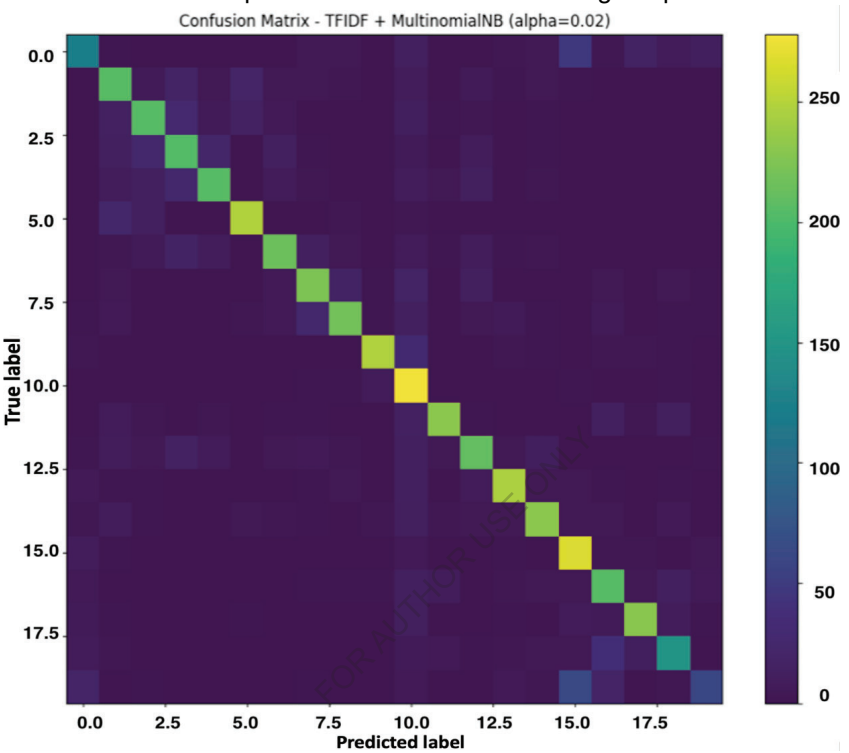


Figure 7.2. Confusion matrix produced for the TF-IDF Multinomial Naïve Bayes (alpha = 0.02).

It can be seen from the confusion matrix results that there is a strong diagonal concentration. This means that the documents were correctly classified to their respective classes. Errors are seen mainly between topically similar classes, which would suggest some kind of overlap in the vocabulary or some random prediction errors.

7.5 Results

32 configurations were evaluated by varying the Vectorization type, Naïve Bayes Variant, and smoothing parameter α. The best configuration was selected using the macro F1-score.

Below is the result for Multinomial Naïve Bayes Classification:

```
One of the Best Configuration Save:
{'vector': 'TVectorizer', 'model':
'MultinomialNB', 'letter': 0.02, 'accuracy':
0.7433675274142201, 'f1don':
0.733212962561589, 'f1weight':
0.7413117932975639}
```

Output 7.2. Multinomial Naïve Bayes classification results.

Bernoulli Naïve Bayes Classification Results

```
acc=0.6673, f1don=0.6238
TVectorizer   , BiNB         , letter=0.02
acc=0.6300, f1don=0.6377
TVectorizer   , BiNB         , letter=0.1
acc=0.6120, f1don=0.6231
TVectorizer   , BiNB         , letter=0.2
acc=0.5980, f1don=0.6106
TVectorizer   , BiNB         , letter=0.5
acc=0.5713, f1don=0.5834
TVectorizer   , BiNB         , letter=1
acc=0.5414, f1don=0.5536
TVectorizer   , BiNB         , letter=2
acc=0.4908, f1don=0.5012
TVectorizer   , BiNB         , letter=5
acc=0.3979, f1don=0.4192
TVectorizer   , BiNB         , letter=10
acc=0.2807, f1don=0.3078
```

FOR USE ONLY

Output 7.3. Bernoulli Naïve Bayes classification results.

The performance trends showed:

1. Highest accuracy for small value of α (0.02-0.1)
2. Gradual decline at the α values increased
3. Remarkable decline at $\alpha \geq 5$

Proving that if the model is applied with excessive smoothing, it can significantly reduce the discriminative power by making the class conditional distributions more uniform.

Model Comparison at $\alpha = 0.02$

Table 7.2. Experimental configuration for the Multinomial and Bernoulli models.

Configuration	Multinomial	Bernoulli
Accuracy	0.743	0.63
Macro F1	0.733	0.64

The experiment results clearly state that Multinomial Naïve Bayes consistently performed better than Bernoulli Naïve Bayes across all values of α. The best Bernoulli configuration achieved an accuracy of 0.63, while multinomial achieved a 0.743 accuracy for the same smoothing value. It can also be seen

that as the value of α increased the accuracy saw a sharp decline for the Bernoulli model.

The main reason for this gap in the performance is because of the different feature modelling for both techniques. While Bernoulli only considers the presence or absence of a word, multinomial trains the model on the term frequency. The dataset chosen for this chapter contains high-frequency data and features. Because the Bernoulli model ignores the frequency, it loses valuable information. This reduces the performance accuracy, and hence the multinomial model stands as the clear choice for text classification of high-dimensional datasets.

7.6 Python Implementation: Comparing Model Variants

To implement these theories, we utilize the scikit-learn library. The following code demonstrates how to evaluate these variants within a machine learning pipeline using a search for the optimal smoothing parameter α.

```python
from sklearn.naive_bayes import MultinomialNB, GaussianNB, BernoulliNB
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import accuracy_score

# Defining models for comparison
models = {
    'MultinomialNB': MultinomialNB,
    'BernoulliNB': BernoulliNB, # Binary comparison model
}

# Range for Laplace Smoothing parameter
alphas = [0.02, 0.1, 0.5, 1, 5, 10]

for name, m_class in models.items():
    for a in alphas:
        # Building the pipeline: Vectorization -> Classification
        pipeline = Pipeline([
            ('vector', TfidfVectorizer(stop_words='english')),
            ('clf', m_class(alpha=a))
        ])

        # Training and prediction
        pipeline.fit(X_train, y_train)
        y_pred = pipeline.predict(X_test)

        # Evaluating accuracy
        acc = accuracy_score(y_test, y_pred)
        print(f"Model: {name} | Alpha: {a} | Accuracy: {acc:.4f}")
```

FOR AUTHOR USE ONLY

Code Snippet 7.5. Comparing the Naïve Bayes variants across a range of Laplace smoothing values.

```
Model: MultinomialNB | Alpha: 0.02 | Accuracy: 0.7639
Model: MultinomialNB | Alpha: 0.1 | Accuracy: 0.7531
Model: MultinomialNB | Alpha: 0.5 | Accuracy: 0.7321
Model: MultinomialNB | Alpha: 1 | Accuracy: 0.7223
Model: MultinomialNB | Alpha: 5 | Accuracy: 0.6753
Model: MultinomialNB | Alpha: 10 | Accuracy: 0.6501
Model: BernoulliNB | Alpha: 0.02 | Accuracy: 0.6196
Model: BernoulliNB | Alpha: 0.1 | Accuracy: 0.5915
Model: BernoulliNB | Alpha: 0.5 | Accuracy: 0.5223
Model: BernoulliNB | Alpha: 1 | Accuracy: 0.4454
Model: BernoulliNB | Alpha: 5 | Accuracy: 0.1483
Model: BernoulliNB | Alpha: 10 | Accuracy: 0.0777
```

Output 7.4. Per-model accuracy for each Laplace smoothing parameter value.

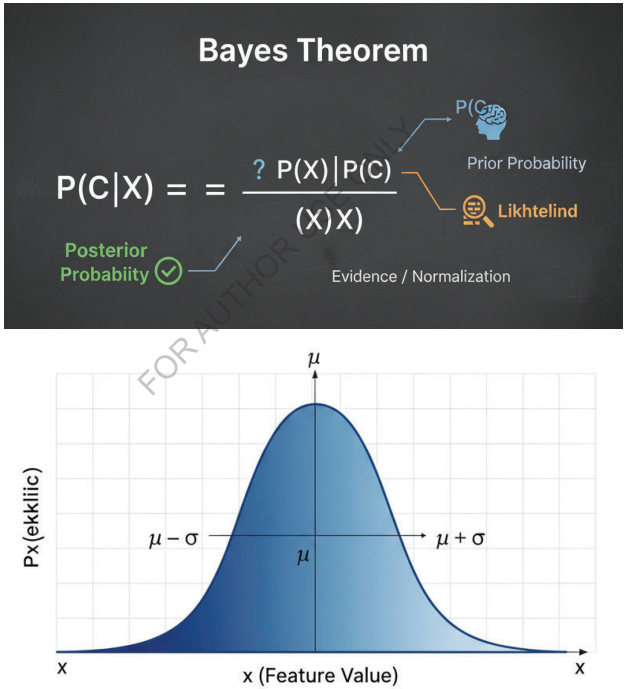


Figure 7.3. Bayes' theorem and the Gaussian likelihood underlying the Naïve Bayes classifier.

7.7 Conclusion

In this chapter, the Naïve Bayes classifier was implemented and evaluated using the 20 Newsgroups dataset, employing two distinct feature extraction methods and two probabilistic variants. Experimental results demonstrate that the multinomial model, when combined with TF-IDF representation and a Laplace smoothing parameter of α = 0.02, yielded the most effective results, achieving an accuracy of approximately 74.3% and an F1 score of 0.733.

These findings validate the theoretical principles of Bayesian inference discussed in previous sections. It is evident that the Laplace smoothing parameter, α, plays a critical role in balancing model discrimination and generalization. When the value of \α is small (approaching zero), the class distribution remains closely aligned with the original training data, preserving its inherent structure. Conversely, as α increases, the class probabilities become "oversmoothed", which significantly reduces the discriminative power of the classifier. This relationship is confirmed by the experimental data, which showed a drastic decline in accuracy as \α approached a value of 5.

Furthermore, this chapter highlights the necessity of selecting the appropriate model variant based on data characteristics. While Gaussian Naïve Bayes remains the optimal choice for continuous, real-valued data—where features typically cluster around a central mean (μ)—the comparison between Multinomial and Bernoulli variants underscores the importance of feature representation in natural language processing. While Bernoulli classification is well-suited for binary datasets, the multinomial variant is significantly more robust for feature-rich, frequency-based documents.

Despite the simplifying assumption of feature independence, the results of this experiment confirm that Naïve Bayes remains highly effective for complex, high-dimensional datasets with multiple classes. The integration of bigrams partially relaxes this independence assumption, further improving the separability between diverse topics.

7.8 Limitations and Future Work

A primary limitation of the Naïve Bayes framework is the assumption of conditional independence, which rarely holds true for natural language data. Consequently, the model is unable to capture semantic relationships between words. Future research may explore the application of semi-supervised learning or hybrid Bayesian networks to address these independence constraints while retaining the computational efficiency inherent in Bayesian classifiers.

FOR AUTHOR USE ONLY

In summary, Naïve Bayes is a high-performance, robust classifier suitable for large-scale, high-dimensional datasets. By utilizing the correct variant—Gaussian for continuous variables or multinomial for discrete frequencies—and combining them with sophisticated feature extraction methods like TF-IDF and optimized smoothing, practitioners can achieve highly reliable results for complex text classification tasks.

FOR AUTHOR USE ONLY

Chapter 8 - Decision Trees

Introduction

In today's world, machine learning has become an integral part of modern decision-making systems, powering everything from recommendation engines to fraud detection. One of the most impactful areas where machine learning has demonstrated consistent value is in healthcare, where the ability to automate and support clinical decisions can directly influence patient outcomes. Heart related diseases remain one of the leading causes of death worldwide, and the challenge of accurately diagnosing patients based on multiple clinical indicators has made this domain a key area of applied machine learning research. Traditional clinical assessments rely on doctors manually evaluating factors such as age, blood pressure, cholesterol levels, and ECG measurements to assess a patient's risk, a process that is both time consuming and subject to human error.

FOR AUTHOR USE ONLY

To address these challenges, researchers have explored a wide range of supervised learning algorithms for clinical classification tasks. Among these, the Decision Tree algorithm has established its effectiveness across both classification and regression problems. Classical tree-based methods such as CART (Breiman et al., 1984) and C4.5 (Quinlan, 1986) introduced recursive data partitioning strategies that select optimal feature splits based on impurity measures, laying the groundwork for Decision Tree modelling in modern machine learning problems. A key reason for the continued relevance of Decision Trees in medical applications is their interpretability. A Decision Tree produces a set of human-readable if-else rules that clinicians can follow and validate, making it particularly suitable for preliminary screening and risk stratification tasks, which is categorizing patients into subgroups based on health status, history etc. (Abbas et al., 2025).

Despite their strengths, Decision Trees are not without challenges. Without proper constraints, a tree may grow to memorize training data including noise, a phenomenon known as overfitting, which leads to poor generalization on unseen patient data (Quinlan, 1986). This has motivated significant research into hyperparameter tuning and pruning strategies to control model complexity. Scikit-learn provides a robust implementation of Decision Tree classifiers with support for both Gini impurity and Entropy as splitting criteria, as well as tools for automated hyperparameter optimization and cost complexity pruning (Pedregosa et al., 2011).

In this study, we model a smart healthcare decision support system using a Decision Tree classifier trained on the heart disease dataset (Janosi et al.; Johnsmith88, Kaggle), which consists of 1025 patient records with 13 clinical features and a binary target variable indicating the presence or absence of heart disease. The goal is to demonstrate how Python-based machine learning tools can be used to build, optimize, and interpret a heart disease prediction model, with a focus on the theoretical foundations of splitting criteria, overfitting control, and pruning techniques.

8.1 Decision Trees for Classification

In classification tasks, a Decision Tree works by repeatedly splitting the data into more uniform groups based on feature values. Starting from the root node, the algorithm selects thresholds that best separate the classes and continues this process until predefined stopping conditions are reached. The resulting tree forms a set of simple if-else rules that are both transparent and interpretable. For example, a medical decision may branch based on cholesterol levels, leading to different diagnostic outcomes. This structure makes Decision Trees particularly well suited for clinical applications, where understanding the reasoning behind a prediction is just as important as the prediction itself.

FOR AUTHOR USE ONLY

In everyday clinical practice, each split in the tree can be thought of as a decision checkpoint - like how a doctor might first assess chest pain type, then evaluate blood pressure, and so on, narrowing down the diagnosis at each step. This is very similar to how the natural clinical diagnosis works, which is why Decision Tree classifiers have found strong applicability in healthcare decision support systems.

In the coming sections, we will discuss the methodology for using Decision Trees for binary classification tasks like heart disease detection.

8.1.1 Splitting Criteria

Central to the Decision Tree classification task is the choice of splitting criterion, which determines how the algorithm evaluates and selects the best feature and thresholds at each node. To determine the optimal split at each node, Decision Trees rely on impurity measures that quantify how mixed the classes are within a subset of data. There are two major metrics that are used to measure impurities. They are **Gini Impurity and Entropy**. Both measure the degree of class mixing within a node and guide the tree toward purer, more decisive

splits. While Gini Impurity directly minimizes class mixing, Entropy works alongside Information Gain to select the most informative split.

8.1.2 Gini Impurity

Gini impurity measures how mixed the classes are within a node. A value of zero means that all samples in the node belong to a single class, indicating perfect purity. The higher the value of impurity, the more mixed the classes are within that node, signifying increased uncertainty in classification. For a binary classification task like heart disease prediction, the Gini value ranges between 0 and 0.5, where 0.5 represents a perfectly equal split between the classes. This indicates the maximum disorder or equal distribution of both classes.
During training, the Decision Tree evaluates different splits and selects the one that results in the lowest weighted Gini impurity in the child nodes. Gini Impurity is the default metric in Sci-kit learn. It is given by the formula,

$$Gini = 1 - \sum_{i=1}^{n} (p_i)^2 \quad \text{(Eq.8.1)}$$

Equation 8.1. Formula for Gini Impurity.

Where,
n = number of classes
$p_{i\,=}$ probability of a sample belonging to class *i* at that specific node.

FOR AUTHOR USE ONLY

8.1.3 Entropy

Entropy measures the level of uncertainty within a dataset, where higher values indicate mixed classes, and lower values indicate more uniform nodes. In binary classification, as in the heart disease prediction, Entropy ranges from 0 to 1. A value of 1 signifies maximum disorder where the classes are split perfectly at 50-50, while a value of 0 indicates a perfectly pure node. The higher the Entropy value, the more difficult it is to draw a clear diagnostic conclusion from that specific group of data.
Information Gain represents the reduction in entropy after a split, and the feature that produces the highest information gain is selected because it provides the most meaningful class separation. Essentially, the algorithm calculates the difference between the Entropy of a parent node and the weighted average Entropy of its child nodes to determine which clinical attribute, such as chest pain or cholesterol, most effectively organizes the patients into distinct classes.
The equation for Entropy is given by the formula,

$$Entropy = \sum_{i=1}^{c} -P_i * log_2(P_i)$$, *(Eq.8.2)*

Equation 8.2. Formula for Entropy.

Where,
n = number of classes
p_i = probability of a sample belonging to class *i*.

The concept of Entropy in this context is derived from Shannon's Information Theory, which quantifies the average amount of information produced by a stochastic source of data. By applying this principle, the Decision Tree identifies the most efficient way to encode diagnostic rules, ensuring that each split maximizes the clarity of the resulting nodes.

Table 8.1. Comparison of Gini Impurity and Entropy.

Aspect	Gini Impurity	Entropy
Formula basis	Probability squared	Logarithmic probability
Range	0 to 0.5 (binary)	0 to 1 (binary)
Computation	Faster, no logarithmic calculation	Slower, due to logarithmic calculation
Sci-kit learn feature	Default	Not default
Sensitivity	Slightly favor's larger partitions	More sensitive to class distribution
Best used when	Speed is prioritized	Maximum information extraction needed

FOR AUTHOR USE ONLY

8.1.4 Overfitting in Decision Trees

One of the main challenges of Decision Trees is overfitting. Without proper constraints, a tree may memorize training data, including noise, leading to poor performance on unseen samples. This is particularly concerning in medical datasets, where limited data and class imbalances can cause overly complex models that fail to generalize to new patients with very different conditions.

In the context of this heart disease study, an overfitted model might create highly specific branches for a single patient's unique combination of age and cholesterol, rather than identifying broader clinical patterns. Such a model would yield misleadingly high accuracy during training but would struggle to provide reliable risk assessments in a real-world clinical setting. Consequently,

the perfect classification results observed in preliminary stages often indicate that the Decision Tree has captured statistical coincidences rather than meaningful diagnostic relationships, necessitating the use of hyperparameter controls to ensure model robustness.

8.1.5 Hyperparameter Control

To mitigate the above risks, Decision Trees include several hyperparameters that directly influence model complexity and performance. *Maximum Depth* controls how many levels a tree can grow, preventing it from becoming too deep and complex. *Minimum Samples Split* defines the minimum number of data points required to permit a node split, while *Minimum Samples Leaf* specifies the number of samples that must remain in a terminal node (leaf) for maintaining sufficient sample support. Additionally, the splitting criterion determines whether Gini impurity or Entropy is used to evaluate the quality of the splits.

While manual adjustment is possible, we utilize GridSearchCV to automate the search for the optimal hyperparameter combination. This process employs cross-validation to ensure the model's stability across different folds of the data, using accuracy as the primary scoring metric. To optimize computational performance, the n_jobs = -1 parameter is used to run the search in parallel across all available processors.

FOR AUTHOR USE ONLY

However, even rigorous hyperparameter tuning via GridSearchCV might not be enough to fully prevent a tree from overfitting the training dataset. To further refine the model and ensure improved generalizability, it is necessary to introduce pruning.

8.1.6 Pruning

Pruning is a hyperparameter tuning method used to "trim" the size of a tree. Specifically, it reduces model complexity by removing branches that contribute negligible predictive value, improving generalization on unseen data.

8.1.6.1 Pre-Pruning (Early Stopping)

Pre-pruning involves halting the growth of the tree before it reaches its maximum possible depth. This is done by setting constraints such as max_depth, which limits tree depth, or min_samples_leaf, which ensures a node is only split if it contains a certain number of patients. While efficient, pre-pruning carries the risk of "underfitting" if the constraints are too strict.

8.1.6.2 Post-Pruning and Cost Complexity Pruning

Post-pruning allows the tree to grow to its full complexity first and then removes branches that do not significantly contribute to predictive accuracy. An advanced variant of this is Cost Complexity Pruning (CCP), which introduces a complexity parameter alpha (α) to balance the trade-off between tree size and misclassification rate.

The goal is to minimize the cost complexity function, $R_\alpha(T)$ given by the formula:

$$R_\alpha(T) = R(T) + \alpha|T|, \quad \textit{(Eq.8.3)}$$

Equation 8.3. Formula for Cost Complexity Function.

Where,

R(T) = total misclassification rate of the tree,

|T| = number of terminal nodes (leaves)

α = complexity parameter.

8.2 Methodology & Dataset

The Heart Disease dataset was imported from a local CSV file and loaded into a DataFrame, where each row represents a patient and each column corresponds to a clinical attribute such as age, chest pain, blood pressure, cholesterol level, and ECG measurements. The dataset consists of approximately 1025 patient records with 13 predictive features and one binary target variable.

Data exploratory analysis was performed to verify dataset integrity and inspect feature distributions. Missing values were examined, and no significant null entries were observed, allowing direct usage of the dataset for model development.

FOR AUTHOR USE ONLY

```
FILE_NAME = "heart.csv"

DATA_PATH = os.path.expanduser(f"~/Downloads/{FILE_NAME}")

df = pd.read_csv(DATA_PATH)

print("Dataset loaded from:", DATA_PATH)
print("Shape:", df.shape)
df.head()
```

```
Dataset loaded from: /Users/srikarkodi/Downloads/heart.csv
Shape: (1025, 14)
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	52	1	0	125	212	0	1	168	0	1.0	2	2	3	0
1	53	1	0	140	203	1	0	155	1	3.1	0	0	3	0
2	70	1	0	145	174	0	1	125	1	2.6	0	0	3	0
3	61	1	0	148	203	0	1	161	0	0.0	2	1	3	0
4	62	0	0	138	294	1	1	106	0	1.9	1	3	2	0

Code Snippet 8.1. Loading the dataset and displaying head.

8.3 Feature Selection and Target Definition

The target column indicates whether a patient has a heart disease, with 1 representing presence and 0 representing absence. All other clinical attributes were used as input features, allowing the model to learn diagnostic patterns directly from patient data.

```
target_col = "target"

X = df.drop(columns=[target_col])
y = df[target_col]

print("Target distribution:\n", y.value_counts())
print("Feature shape:", X.shape)
```

```
Target distribution:
 target
1    526
0    499
Name: count, dtype: int64
Feature shape: (1025, 13)
```

Code Snippet 8.2. Feature selection.

FOR AUTHOR USE ONLY

8.4 Feature Encoding

Since machine learning algorithms require numerical input, feature data types were inspected. Most attributes were already numerically encoded. If categorical variables were detected, one-hot encoding was applied to convert them into numericals.

```
non_numeric = X.select_dtypes(exclude=[np.number]).columns.tolist()

if non_numeric:
    print("Categorical columns detected:", non_numeric)
    X = pd.get_dummies(X, columns=non_numeric, drop_first=True)
else:
    print("All features are numeric — no encoding needed.")

print("Final feature shape:", X.shape)
```

```
All features are numeric — no encoding needed.
Final feature shape: (1025, 13)
```

Code Snippet 8.3. Check for feature encoding requirement.

8.5 Train-Test Split

To assess model generalization, the dataset was split into training and testing using an 80-20 stratified approach.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y
)

print("Train shape:", X_train.shape, y_train.shape)
print("Test shape:", X_test.shape, y_test.shape)
```

```
Train shape: (820, 13) (820,)
Test shape: (205, 13) (205,)
```

Code Snippet 8.4. Test-Train split of the dataset.

8.6 Hyperparameter Optimization

A Decision Tree classifier was trained using GridSearchCV with fivefold cross validation to optimize hyperparameters and reduce overfitting. The search explored splitting criteria, tree depth, and minimum sample thresholds.

```
param_grid = {
    "criterion": ["gini", "entropy"],
    "max_depth": [2, 3, 4, 5, 6, None],
    "min_samples_split": [2, 5, 10, 20],
    "min_samples_leaf": [1, 2, 5, 10]
}

grid = GridSearchCV(
    DecisionTreeClassifier(random_state=42),
    param_grid,
    cv=5,
    scoring="accuracy",
    n_jobs=-1
)

grid.fit(X_train, y_train)

print("Best Parameters:", grid.best_params_)
print("Best CV Accuracy:", grid.best_score_)

best_dt = grid.best_estimator_
```

```
Best Parameters: {'criterion': 'entropy', 'max_depth'
Best CV Accuracy: 0.9890243902439024
```

FOR AUTHOR USE ONLY

Code Snippet 8.5. Setting initial hyperparameter tuning.

8.6.1 Initial Results and Performance Analysis

Following the hyperparameter optimization via GridSearchCV, the initial Decision Tree model achieved a testing accuracy of 1.00 (100%) with identical scores for precision, recall, and F1 score across both classes.

8.6.1.1 Classification Report and Accuracy Table

```
pred = best_dt.predict(X_test)

print("Tuned Decision Tree Test Accuracy:", accuracy_score(y_test, pred))
print("\nClassification Report:\n")
print(classification_report(y_test, pred))
```

```
Tuned Decision Tree Test Accuracy: 1.0

Classification Report:

              precision    recall  f1-score   support

           0       1.00      1.00      1.00       100
           1       1.00      1.00      1.00       105

    accuracy                           1.00       205
   macro avg       1.00      1.00      1.00       205
weighted avg       1.00      1.00      1.00       205
```

Code Snippet 8.6. Initial Classification report output.

While this indicated excellent predictive performance on the available data, these results may be influenced by potential data leakage or model overfitting.

8.6.1.2 Confusion Matrix

The Confusion Matrix shows perfect classification on the test dataset, with all healthy and diseased cases correctly identified and no false positives and false negatives.

FOR AUTHOR USE ONLY

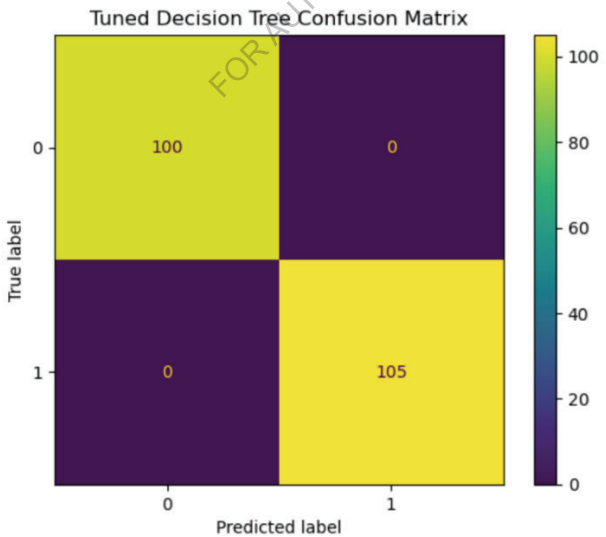


Figure 8.1. Confusion Matrix of the initial results.

However, such perfect classification on a dataset of 1025 records strongly suggests that the model has overfitted by memorizing the training data.

8.6.1.3 Decision Tree Visualization

The Decision Tree visualization shows that chest pain type (cp) is the primary splitting feature, indicating its strong predictive importance. Subsequent splits involve clinically relevant variables such as age, number of major vessels (ca), ST depression (oldpeak), cholesterol, and maximum heart rate. The tree clearly separates disease and non-disease cases through hierarchical if-else rules.

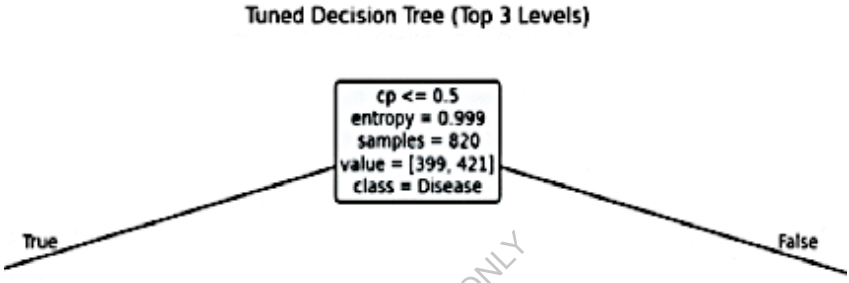


Figure 8.2. (A) Visualization of the initially tuned Decision Tree - top level.

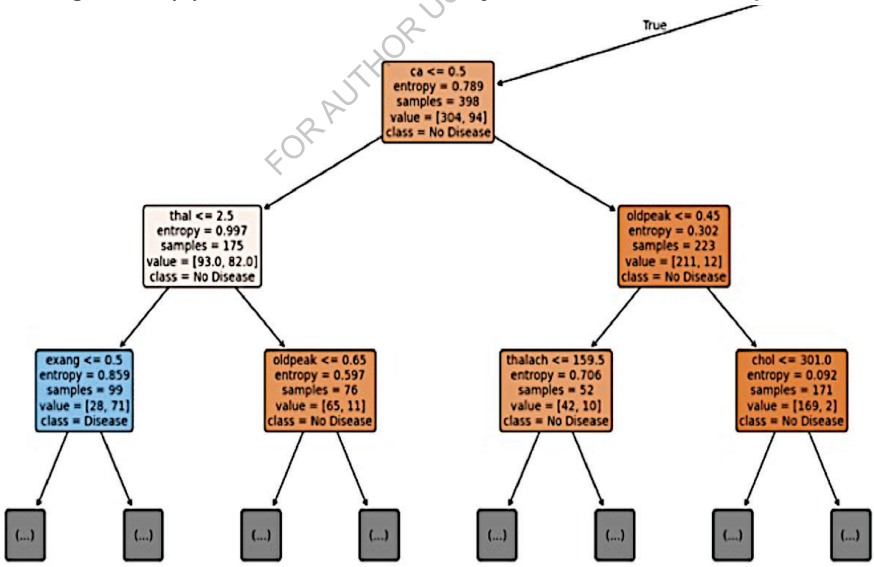


Figure 8.2. (B) Visualization of the initially tuned Decision Tree - left side.

FOR AUTHOR USE ONLY

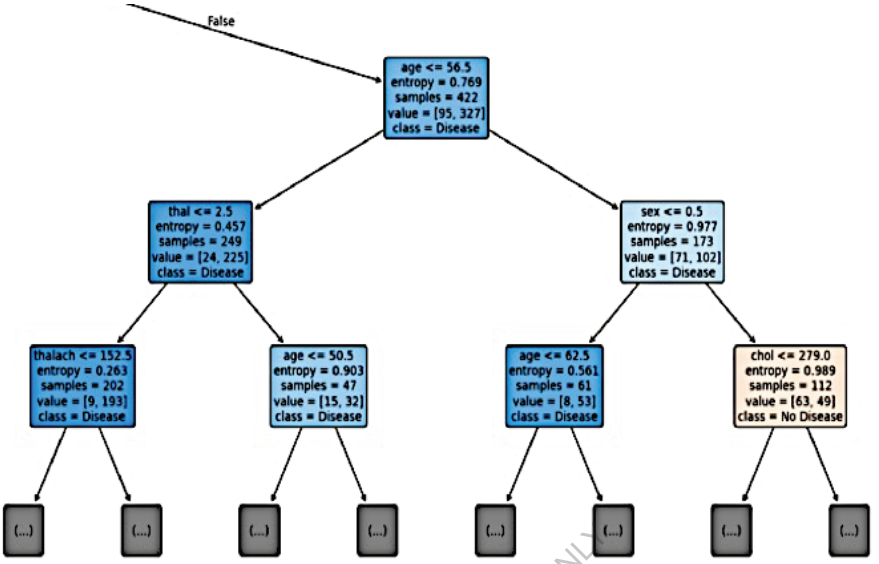


Figure 8.2. (C) Visualization of the initially tuned Decision Tree - right side.

The complexity of the resulting tree indicates that it has captured statistical noise, which may prevent the system from generalizing effectively to new patients with varying clinical profiles.

FOR AUTHOR USE ONLY

8.6.1.4 Model Optimization

To address the overfitting observed in the initial model, Cost Complexity Pruning (CCP) was applied to improve generalizability.

The methodology for this refinement followed three specific steps: first, we used the cost_complexity_pruning_path function to extract the sequence of effective alphas from the initial overfitted tree. Second, we trained a series of independent decision trees for each candidate α value to assess how different levels of pruning impacted performance. Finally, we compared the test-set accuracy of each version to select the specific pruned tree that provided the best balance between simplicity and predictive power. This transition is implemented below.

```
# 4. Optimization: Cost Complexity Pruning (CCP)
# Calculate the pruning path to find effective alphas
path = initial_dt.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

# Train a suite of trees for each alpha
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=42, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)

    # Identify the alpha that maximizes testing accuracy
test_scores = [clf.score(X_test, y_test) for clf in clfs]
best_alpha = ccp_alphas[np.argmax(test_scores)]
best_pruned_clf = clfs[np.argmax(test_scores)]
```

Code Snippet 8.7. Pruning the Tree using CCP.

8.6.2 Optimized Performance and Comparative Results:

The optimized model, refined through Cost Complexity Pruning, demonstrates a more realistic and robust predictive capability.

8.6.2.1 Classification Report and Accuracy Table

FOR AUTHOR USE ONLY

```
# 5. GENERATE FINAL RESULTS
y_pred = final_pruned_clf.predict(X_test)
final_acc = accuracy_score(y_test, y_pred)
final_nodes = final_pruned_clf.tree_.node_count

print(f"--- FINAL OPTIMIZED RESULTS ---")
print(f"Optimal Alpha: {best_alpha:.4f}")
print(f"Final Node Count: {final_nodes}")
print(f"Final Accuracy: {final_acc*100:.2f}%")
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```
--- FINAL OPTIMIZED RESULTS ---
Optimal Alpha: 0.0096
Final Node Count: 69
Final Accuracy: 91.22%

Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.92      0.91       100
           1       0.92      0.90      0.91       105

    accuracy                           0.91       205
   macro avg       0.91      0.91      0.91       205
weighted avg       0.91      0.91      0.91       205
```

Code Snippet 8.8. Optimized Classification Report output.

While the overall accuracy has slightly decreased from the initial perfect score, the resulting classification report shows high precision and recall,

indicating that the model remains highly effective at identifying heart disease cases.

8.6.2.2 Confusion Matrix

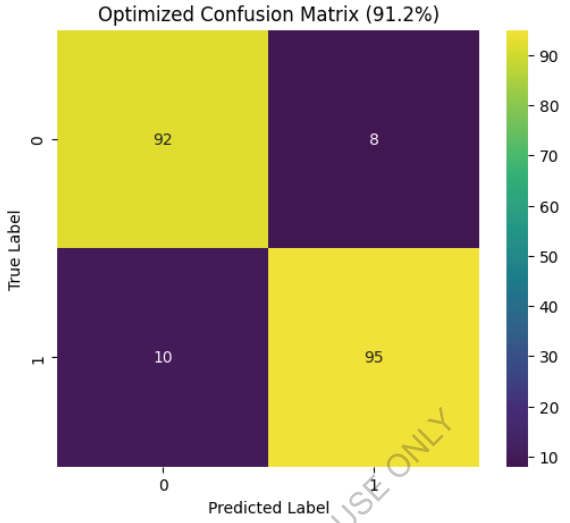


Figure 8.3. Confusion Matrix of the optimized results.

The new confusion matrix confirms that the model avoids the pitfall of "perfect" memorization, making it more suitable for real-world healthcare applications where data variability is common.

8.6.2.3 Decision Tree Visualization

By simplifying the decision pathways, the pruned tree offers greater interpretability for clinicians while maintaining its strength as a preliminary screening tool.

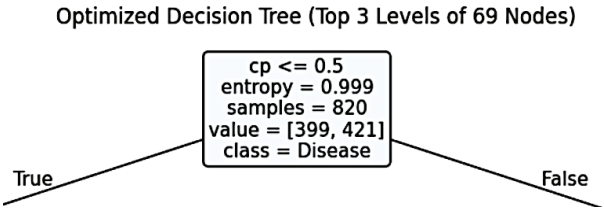


Figure 8.4. (A) Visualization of the optimized and pruned Decision Tree – top level.

FOR AUTHOR USE ONLY

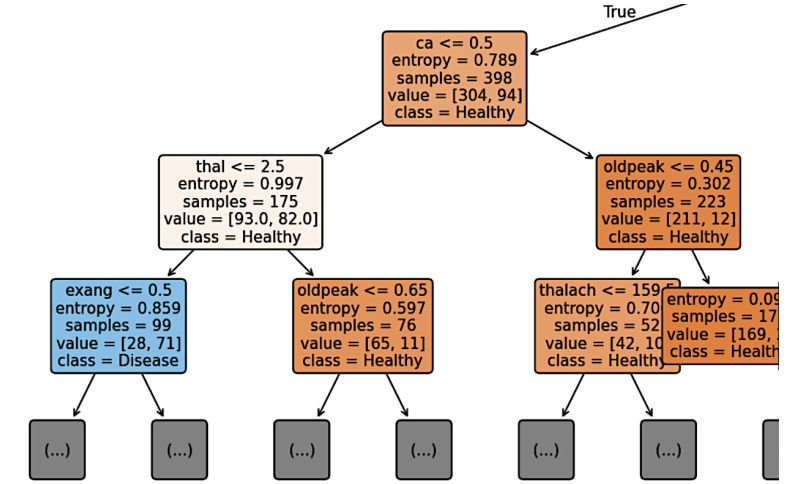


Figure 8.4. (B) Visualization of the optimized and pruned Decision Tree – left side.

FOR AUTHOR USE ONLY

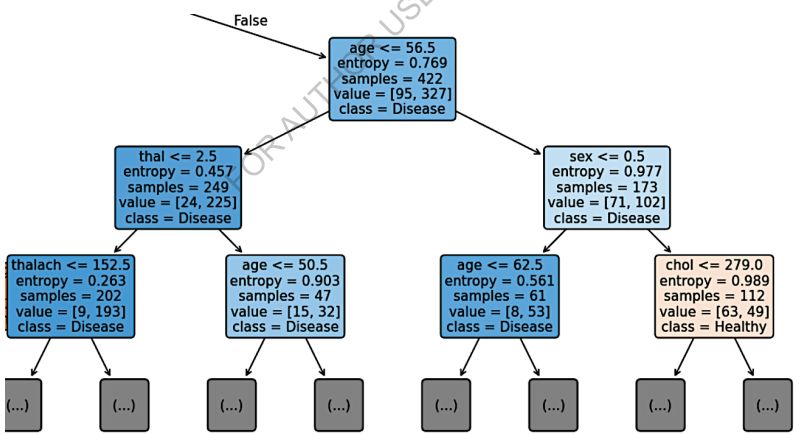


Figure 8.4. (C) Visualization of the optimized and pruned Decision Tree – right side.

While chest pain (cp) remains the primary diagnostic indicator at the root, the tree is now prioritizing high impact variables like major vessels (ca) and maximum heart rate (thalach). This simplified model replaces misleading factors such as age and cholesterol in the overfitted model and results in a more transparent and clinically intuitive diagnostic path.

8.7 Results and Discussions

The comparative analysis between the initial and pruned models demonstrates a critical transition toward a more realistic and robust predictive performance. While the initial overfitted, model reached an unrealistic 100% accuracy, it captured substantial training set noise rather than genuine medical trends. By implementing a pruning strategy with an optimal alpha of 0.0096, the model was refined to a robust testing accuracy of 91.22%.

This optimization successfully streamlined the tree from 97 nodes to 69 nodes. This underscores that despite pruning about one-third of the tree, the model is still highly accurate. By sacrificing a small, realistic margin of accuracy, the model achieved greater generalizability. It resolves the critical issue of high variance without a major loss of performance. The pruned structure continues to prioritize decisive clinical markers such as chest pain (cp) and major vessels (ca), indicating that the simplified tree is more reliable for real-world cardiovascular screening.

FOR AUTHOR USE ONLY

8.8 Conclusion

This chapter successfully implemented a Decision Tree classifier for heart disease risk prediction using Python, demonstrating how tree-based machine learning can be applied to structured clinical data. A systematic methodology was followed, including exploratory data analysis, feature encoding, stratified train-test splitting, and hyperparameter optimization via GridSearchCV with fivefold cross-validation. The initial tuned model achieved a test accuracy of 100%, which was identified because of overfitting rather than genuine generalization. To address this, Cost Complexity Pruning was applied with an optimal alpha of 0.0096, reducing the tree from 97 to 69 nodes and yielding a robust test accuracy of 91.22%. Precision, recall, and F1 scores of 0.91 across both classes confirm that predictive performance was maintained despite the reduction in model complexity. Confusion matrix analysis confirmed the correct classification of 92 healthy and 95 disease cases, with low false positive and false negative rates. The pruned tree prioritizes clinically meaningful features such as chest pain type (cp) and number of major vessels (ca), demonstrating that model simplification can improve both interpretability and generalizability. Overall, the results confirm that Decision Trees, when properly tuned and pruned, provide a reliable and transparent tool for preliminary cardiovascular screening.

Chapter 9 - Ensemble Learning I

Introduction

Artificial Intelligence (AI) and Machine Learning (ML) have made a profound impact in the field of medical image analysis by making it more efficient and accurate for disease detection systems. Deep learning models, specifically Convolutional Neural Networks (CNNs), have shown good results in the extraction of complex spatial features from medical imaging data. However, obtaining reliable and interpretable classification is still a big problem in healthcare-related machine learning applications.

A hybrid classification system for brain tumor detection, which uses CNN-based deep feature extraction and ensemble classification techniques based on Random Forests is given. The proposed method involves image preprocessing, deep learning for feature extraction, machine learning for diagnosis classification and interpretability analysis, aiming to enhance the diagnostic performance on MRI images.

The implementation process includes MRI preprocessing, CNN training, deep feature extraction, RF classification, hyperparameter optimization, evaluation metrics, and interpretability of the model using the LIME framework.

FOR AUTHOR USE ONLY

9.1. Use cases

Brain tumors are masses that grow in brain cells. Even though some brain tumors are benign, they can still lead to dangerous neurological complications depending on their size and location. The three major brain tumor types considered in this study are meningioma, glioma, and pituitary tumor.

9.1.1. Meningioma

Meningioma is considered the most common brain tumor type. They are mostly benign but there can be serious complications. While the risk of meningioma increases with age in both sexes, the effect of aging is nearly twice as significant in women as in men.

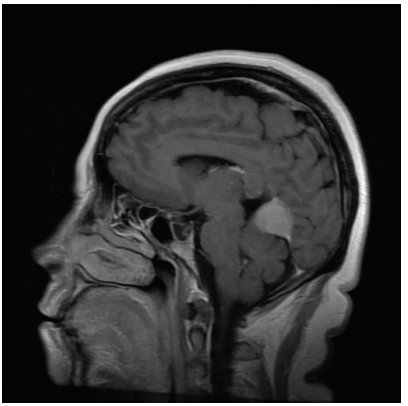


Figure 9.1. Sample MRI image of meningioma tumor.
Source: Nickparvar, M. (n.d.). Brain Tumor MRI Dataset, Kaggle

9.1.2. Glioma

Glioma is another common type of brain tumor. It can nearly exist in every lobe in the brain, including frontal, temporal, parietal, and occipital lobes. It is a type of cell growth that starts in the brain or spinal cord. It is considered the most fatal malignant tumor among the others.

AUTHOR USE ONLY

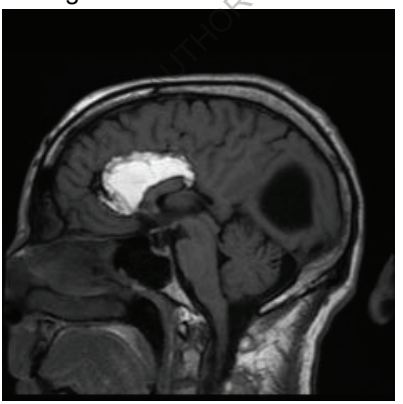


Figure 9.2. Sample MRI image of glioma tumor.
Source: Nickparvar, M. (n.d.). Brain Tumor MRI Dataset, Kaggle

9.1.3. Pituitary Tumor

Pituitary tumors are not that common compared to glioma and meningioma. Despite being benign, they can still cause significant complications. They can still affect people by covering some space in the brain. Hormonal imbalances

can contribute to tumor growth. It damages the pituitary gland and causes hormonal issues.

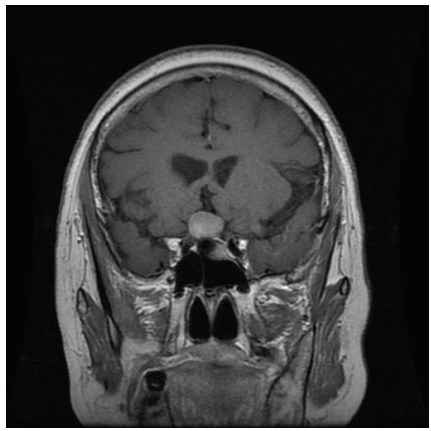


Figure 9.3. Sample MRI image of pituitary tumor.
Source: Nickparvar, M. (n.d.). Brain Tumor MRI Dataset, Kaggle

9.1.4. Diagnosis with Magnetic Resonance Imaging (MRI)

Magnetic Resonance Imaging (MRI) is crucial in brain tumor diagnosis because of its ability to identify anomalies in brain tissues. This study employs brain tumor detection using the combination of convolutional neural networks and random forest classifiers.

Traditional MRI interpretation is time-consuming and highly dependent on clinical expertise. Deep learning techniques have become increasingly important in automated healthcare diagnosis systems. Convolutional Neural Networks (CNNs) have demonstrated strong performance in medical image analysis tasks. However, limited interpretability remains a major challenge in clinical applications. The proposed framework uses CNN for deep feature extraction, Random Forest for final classification, and LIME for interpretability. Random Forest belongs to the family of ensemble learning algorithms, where multiple decision trees are combined to improve accuracy and reduce overfitting. The LIME technique was additionally used to improve interpretability by highlighting the regions influencing the model's predictions.

This hybrid approach is tested on a public dataset which contains 7,200 MRI records. There are 1400 MRI images each for these classes for training: Glioma, Meningioma, Pituitary Tumor, and no tumor. The proposed hybrid

model achieved an overall classification accuracy of 95% and provided visual interpretability through LIME-based explanations.

9.2. Literature Review

Yashu and Kukreja proposed a hybrid approach that includes both deep learning and machine learning frameworks for brain tumor classification using MRI images. In this paper, Convolutional Neural Networks (CNNs) are used for feature extraction followed by a Random Forest (RF) classifier for the final decision. Unlike traditional methods used in CNN models where they rely on the fully connected layers for classification, the authors have replaced the final dense layers with a Random Forest classifier to reduce overfitting. The proposed CNN-RF hybrid model was trained and evaluated on 4,220 MRI images and has achieved an accuracy of 98.88%, which outperforms the standard deep learning models such as ResNet-50 and Efficient net. The model also achieved a high AUC score of 0.988 which indicated strong sensitivity and specificity. This shows that the hybrid approach is a reliable alternative to conventional black-box deep learning systems and can be adopted for real-world healthcare applications.

In the paper proposed by Kasim et al., a hybrid approach using pre-trained CNN architectures is combined with traditional machine learning classifiers for multiclass leukemia cell classification. In their approach the authors have used CNN models such as VGG16, InceptionV3, and ResNet50 for feature extraction, and machine learning classifiers that include Random Forest (RF), Support Vector Machine (SVM), Extreme Gradient Boosting (XGBoost), and Multi-Layer Perceptron (MLP) were used for the final classification. The dataset was publicly available consisting of 3 classes and 390 images. Data augmentation techniques were applied to increase the dataset size. A U-Net model was additionally used for image segmentation. The results showed that InceptionV3+SVM has achieved the highest accuracy, resulting in 88%, followed by VGG16+XGBoost at 87%. Random forest also demonstrated competitive performance, achieving 81% accuracy when combined with VGG16 and 78% with InceptionV3. Deeper architectures that included ResNet50 had lower accuracy (maximum 76%) due to overfitting that is caused by the small dataset size. The findings of this study highlight that hybrid CNN-machine learning approaches improve classification performance in medical imaging tasks, especially when we deal with data constrained conditions. Although the study focused on leukemia classification, the effectiveness of

THE AUTHOR ISLOAN

combining CNN based feature extraction with ensemble classifiers such as Random Forest does provide valuable insights for brain tumor MRI classification tasks.

9.3. Methodology

In this research, we have used a hybrid machine learning framework for MRI-based brain tumor detection. Convolutional Neural Networks (CNN) were used for deep feature extraction, while Random Forest (RF) was used for final classification into Meningioma, Glioma, Pituitary Tumor or No Tumor.

The complete process consists of image preprocessing, CNN-based feature extraction, feature vector generation, Random Forest classification, hyperparameter optimization using cross-validation and then performance evaluation.

Very often, MRI images contain noisy data like a significant black background surrounding the brain, along with raw data which do not contain any diagnostic information. These irrelevant regions introduce noise and negatively affect model performance. To reduce this issue, we use a contour-based cropping function (crop_img) which applies to each image.

FOR AUTHOR USE ONLY

First, we convert the BGR image to grayscale and then, to reduce the noise, Gaussian blur with a 3 × 3 kernel is applied. After that, perform global thresholding at a fixed intensity of 45 to separate the brain from the background. This process whitens the important brain regions and darkens the black background.

Next, to remove small unwanted white spots, morphological erosion is used, followed by dilation to remove small, isolated regions and fill gaps to make the brain region more complete. After connecting all white areas, the algorithm automatically selects the largest region, which is presumed to be the brain. The boundary points (left, right, top, and bottom) of the brain are recognized, and then brain areas are cropped. If it fails to detect the brain, the raw image is returned as a fallback to avoid data loss. This process removes unnecessary borders and focuses mainly on brain region, likely improving the signal-to-noise ratio for the classifier. Data augmentation techniques such as rotation, flipping, and zooming were used to improve model generalization.

Resizing is necessary to ensure accuracy, consistency and efficient model learning. In this process, an image is loaded using OpenCV and then a cropping function is used to remove background which is not required and focus only on the brain region. Then, the cropped image is resized to a fixed size of 256 x 256 pixels to maintain the dimensions of each image. Finally, the cleaned image is saved.

To check the process that they are performing properly, a visual sanity check is performed. It displays a set of raw and cleaned images side by side for a particular class (Meningioma, Glioma or Pituitary Tumor). It makes it easier to compare and understand if the brain region is extracted, enhanced and cleaned properly.

ResNet50 is part of the Residual Network (ResNet) architecture family. It was developed to resolve the issues associated with training deep neural networks. One major problem in deep neural networks is the vanishing gradient problem, where gradients become smaller as they propagate through deeper layers during training. ResNet50 addresses this limitation using residual connections. These residual connections enable gradients to propagate more effectively during backpropagation. These bypass shortcuts make it easier for gradients to flow backward during training. And this concept is implemented using a structure called the Bottleneck Residual block. It improves computational efficiency while maintaining strong feature learning capability.

FOR AUTHOR USE ONLY

Deep networks are powerful but may overfit, especially in medical datasets where sample sizes are limited. Combining CNNs with Random Forest gives

leverage as Random Forest is robust to noise, less prone to overfitting and performs well on structured features vectors.

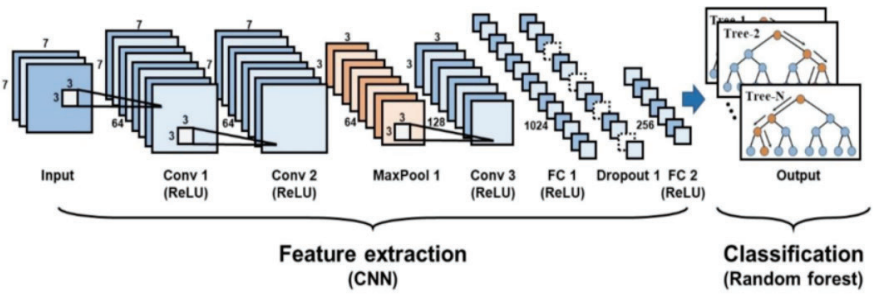


Figure 9.4. CNN and Random Forest Pipeline.

RF (Random Forest) is an ensemble learning method for classification which uses many decision trees to strengthen the result. Each decision tree is trained using randomly sampled subsets of the training data, and their results are combined by voting. It helps in handling the missing data and tells which features are most useful for making the decision.

FOR AUTHOR USE ONLY

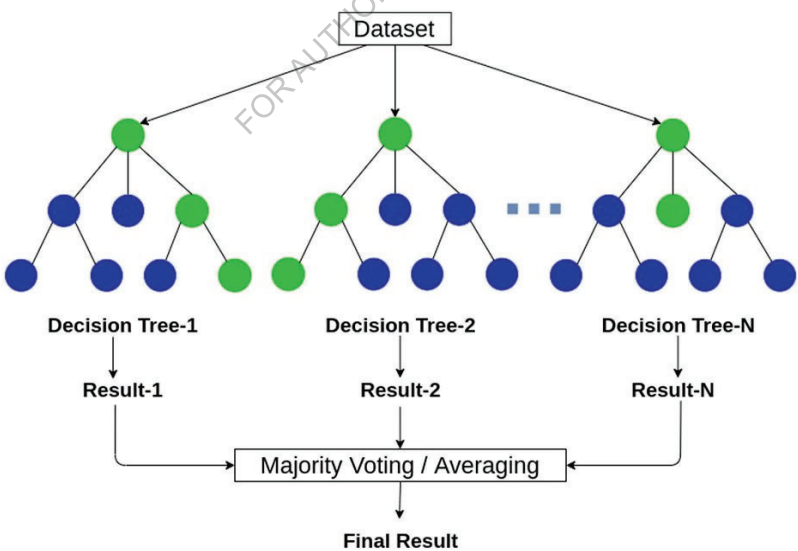


Figure 9.5. Random Forest.

To improve model performance, Hyperparameter optimization can be done using GridSearch combined with five-fold validation. This approach helps to prevent overfitting and improve model performance. In medical datasets, it is common that one category may contain more samples than the others, which is called class imbalance. When data gets imbalanced, relying on accuracy can be misleading. A model can achieve a high accuracy score by majority class most of the time, even after correctly detecting important minority classes. In this case, Macro F1 evaluates performance for each class separately and then calculates the average of those scores. It ensures that the model doesn't ignore minority cases.

The ROC (Receiver Operating Characteristic) curve is used to evaluate the performance by binary classification models. It works on two metrics:

- TPR (True Positive Rate): It measures the proportion of actual instances the model correctly identifies, and it is also known as recall or sensitivity.

  TPR = True Positive / (True Positives + False Negative)

- FPR (False Positive Rate): It measures the proportion of actual negative instances classified as positive by the model.

  FPR = False Positive / (False Positives + True Negative)

FOR AUTHOR USE ONLY

AUC (Area under the curve) is a metric used to determine the performance of a classification model. It can be calculated with the help of the ROC curve to differentiate between two classes. The value of AUC varies between 0 and 1.

The ROC curve shows the relationship between TPR and FPR at different threshold settings. If the value of AUC is high, then it indicates that the model has a strong tendency to correctly distinguish between positive and negative classes. Closer the value of AUC to 1, the better the model is at classification.

Deep Learning models like ResNet50 are quite powerful in medical image analysis. Despite their high predictive performance, these models are often criticized for their lack of interpretability. This is where LIME (Local Interpretable Model-Agnostic Explanations) comes into effect. It is a post-hoc interpretability technique that explains individual predictions rather than models. It provides a true representation of the model's behavior and highlights the factors that affect the model's prediction. The mathematics behind the LIME is based on the idea of approximating a complex ML model locally with a simple interpretable model.

Mathematically, this can be represented as:

$$\xi(x) = \arg\min_{g \in G} L(f, g, \pi_x) + \Omega(g) \quad , \qquad \textit{(Eq.9.1)}$$

Where:

- f is the complex ML model
- g is the interpretable model
- G is the class of interpretable models
- π_x is a proximity measure that defines the locality around the instance x
- $L(f, g, \pi_x)$ is the loss function that measures the faithfulness of g to f in the locality defined by π_x
- $\Omega(g)$ is a regularization term that controls the complexity of g

9.4. Implementation

9.4.1. Project Environment

FOR AUTHOR USE ONLY

All experiments were implemented using Python 3.11 and PyTorch deep learning framework. The models were trained and evaluated on a local laptop GPU which is equipped with an NVIDIA GPU that supports CUDA acceleration. This GPU-based training was essential due to the computational demands of CNN training and high-dimensional feature extraction.

Deterministic configurations were used to improve reproducibility across different systems. Additionally, deterministic mode in CUDA was enabled which helps to reduce variability.

Scikit-learn library was used for the main machine learning tasks including Random Forest classification, cross-validation, hyperparameter tuning, and performance metric computation. Matplotlib and Seaborn were used for visualization, including learning curves, confusion matrices, and ROC curves.

9.4.2. Dataset Structure and Data Pipeline

The dataset consists of four classes of brain MRI images: glioma, meningioma, pituitary tumor, and no tumor. A publicly available Kaggle dataset consisting of 7200 images in total was used. Among these, 5600 images are present in the training folder and 1600 images in the test folder, and these are divided equally among all four classes and saved in separate folders.

The training dataset was further divided into 2 subsets of training and validation using an 80/20 stratified split. Stratified splitting was used to preserve proportional class distribution across the subsets. This minimizes the risk of class imbalance during training and validation. The test set was left untouched throughout the entire process of training and hyperparameter tuning.

With the help of PyTorch's Dataset and DataLoader classes, data loading was implemented. The DataLoader enabled efficient mini batch loading and GPU data transfer during training. The batch size was set to 32 to ensure smooth model training while minimizing memory usage.

9.4.3. Image Preprocessing

Image preprocessing techniques were used to reduce background noise and help the model focus on clinically relevant brain regions. Prior to model training, a preprocessing pipeline was implemented in which each image underwent the following steps:

FOR FURTHER USE ONLY

- Conversion to grayscale
- Gaussian blurring (Reduces high frequency noise)
- Binary Thresholding (Isolates the brain region)
- Morphological erosion and dilation (Helps remove small artifacts)
- Contour detection (Identifies the largest connected component)
- Extract contour points (Compute a bounding box)
- Crop the image (Isolate the brain)
- Resize the cropped image to 256 x 256 pixels

The contour-based approach ensures that irrelevant black background regions are removed. This reduces computational redundancy and allows the model to focus on brain tissue. A fallback mechanism was also implemented to handle cases where contour detection failed.

```python
def crop_img(img):
    try:
        # Step 1: Convert image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        # Step 2: Apply Gaussian blur to reduce noise
        gray = cv2.GaussianBlur(gray, (3, 3), 0)

        # Step 3: Apply binary thresholding to separate brain from background
        thresh = cv2.threshold(gray, 45, 255, cv2.THRESH_BINARY)[1]
        # Step 4: Morphological operations to clean noise
        thresh = cv2.erode(thresh, None, iterations=2)
        thresh = cv2.dilate(thresh, None, iterations=2)

        # Step 5: Find external contours
        cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
        )
        cnts = imutils.grab_contours(cnts)

        # If no contours are found, return original image
        if len(cnts) == 0:
            return img

        # Step 6: Select the largest contour (assumed brain region)
        c = max(cnts, key=cv2.contourArea)

        # Step 7: Find extreme points of the contour
        extLeft = tuple(c[c[:, :, 0].argmin()][0])
        extRight = tuple(c[c[:, :, 0].argmax()][0])
        extTop = tuple(c[c[:, :, 1].argmin()][0])
        extBot = tuple(c[c[:, :, 1].argmax()][0])

        # Step 8: Crop the image using bounding box from extreme points
        new_img = img[extTop[1]:extBot[1], extLeft[0]:extRight[0]].copy()
        return new_img

    except Exception:
        # Fallback: return original image if any error occurs
        return img
```

FOR AUTHOR USE ONLY

Code Snippet 9.1. Brain region cropping with grayscale conversion, thresholding, morphological operations, and extreme point extraction. (Contour-based).

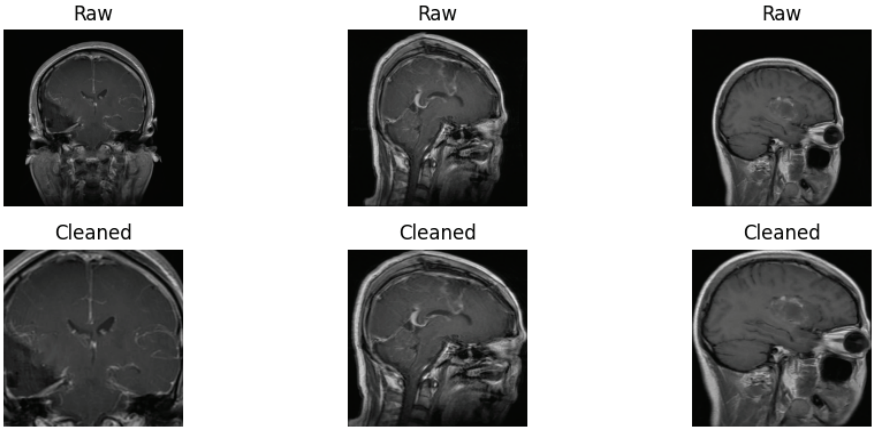


Figure 9.6. MRI images before and after preprocessing using contour-based brain region extraction.

Source: Nickparvar, M. (n.d.). Brain Tumor MRI Dataset, Kaggle; preprocessing output generated by the author.

FOR AUTHOR USE ONLY

9.4.4. Data Augmentation

Data augmentation techniques were used to increase the dataset size to improve generalization and reduce overfitting. Controlled data augmentation was applied during CNN training. These augmentations were applied only to the training subset.

The following data augmentation techniques were used:

- Rotation within +/- 10 degrees
- Affine transformation with 5% translation
- Scaling within the range of 0.9 to 1.1

Aggressive augmentation techniques were intentionally avoided to ensure that the medical images are not drastically changed. Unlike normal images, excessive distortion in MRI images can alter the clinically meaningful features. Therefore, the selected parameters introduced only slight variations while maintaining structural integrity.

```
# Train-time augmentation (applied ONLY to the training set)
train_transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),

    # Random rotation to improve robustness
    transforms.RandomRotation(10),

    # Random affine transformations (translation + scaling)
    transforms.RandomAffine(
        degrees=0,
        translate=(0.05, 0.05),
        scale=(0.9, 1.1)
    ),

    # Convert image to tensor format
    transforms.ToTensor()
])


# Validation / Test Transform (No augmentation)

eval_transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor()
])
```

FOR AUTHOR USE ONLY

Code Snippet 9.2. Data augmentation and preprocessing transforms for training and evaluation.

9.4.5. CNN Training

The ResNet50 model that is pretrained on ImageNet was adopted as the primary deep learning backbone. Transfer learning was used as the dataset size is moderate. This allows the model to use its own learned low-level and mid-level features such as edges, textures, and spatial hierarchies.

The fully connected layer of ResNet50 was replaced with a linear layer of four output neurons, each relating to the four classes in our dataset. Cross-Entropy loss was selected as the optimization criteria due to its effectiveness in multi-class classification tasks.

Adam optimizer was used with a learning rate of 1 x 10^-4 and a weight decay of 1 x 10^-4. The training loop ran for 20 epochs and the model with the highest

validation accuracy was selected for the final evaluation. Training and validation loss curves were also monitored to detect any overfitting. Accuracy was also computed at each epoch for both training and validation sets.

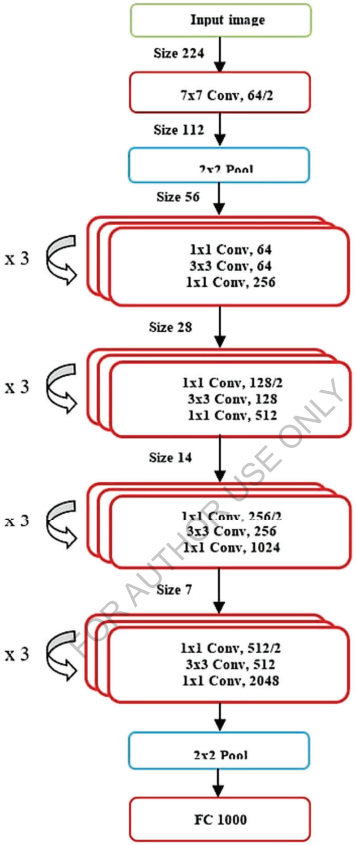


Figure 9.7. ResNet50 Model Architecture.

9.4.6. Deep Feature Extraction

After CNN training was completed, the final classification layer was removed, and this network was converted into a feature extractor. This allowed us to extract 2048-dimensional features from the global average pooling layer.

These deep feature representations have high level information learned from MRI images in an encoded form. Feature extraction was then applied without

augmentation. Each image is now represented as a 2048-dimensional vector, which acts as the input to the Random Forest classifier.

```
feature_model = build_resnet50(num_classes=4)
feature_model.load_state_dict(model.state_dict())
feature_model.fc = nn.Identity()
feature_model = feature_model.to(device)
feature_model.eval()
```

Code Snippet 9.3. Conversion of trained ResNet50 into a deep feature extractor.

9.4.7. Random Forest and Hyper parameter optimization

Random Forest classifier was used as a secondary classifier that utilizes the extracted deep features.

Hyperparameter tuning was done using the GridSearchCV with 5-fold stratified cross-validation on the training features. Below are the parameter ranges that were used:

- Number of trees (n_estimators): 300, 500, 700
- Maximum tree depth (max_depth): None, 20, 40
- Minimum samples required to split (min_samples_split): 2, 5
- Maximum features considered at each split (max_features): sqrt

FOR AUTHOR USE ONLY

```
param_grid = {
    "n_estimators": [300, 500, 700],      # number of trees
    "max_depth": [None, 20, 40],          # tree depth control
    "min_samples_split": [2, 5],          # regularization
    "max_features": ["sqrt"]              # standard for classification
}
```

Code Snippet 9.4. Hyperparameter grid configuration for Random Forest tuning.

Macro-F1 score was selected as the optimization metric because balanced performance across all the classes is critical in multi-class medical image classification tasks.

```
grid_search = GridSearchCV(
    estimator=rf_base,
    param_grid=param_grid,
    scoring="f1_macro",    # important for medical multi-class
    cv=5,
    verbose=1,
    n_jobs=-1
)
```

Code Snippet 9.5. Random Forest hyperparameter tuning using 5-fold cross-validation.

The best parameters in our case were Number of trees: 300, Maximum tree depth: None, Minimum samples required to split: 2, Maximum features considered at each split: sqrt. After selecting the best hyperparameter configuration, the Random Forest model was retrained on the full training feature set and evaluated only once on the independent test set.

```
================================================================
Random Forest Hyperparameter Tuning (5-Fold Cross-Validation)
================================================================
Fitting 5 folds for each of 18 candidates, totalling 90 fits

Best Cross-Validation Macro-F1: 0.9982147483725552
Best Hyperparameters:
{'max_depth': None, 'max_features': 'sqrt', 'min_samples_split': 2, 'n_estimators': 300}
```

Output 9.1. Random Forest hyperparameter tuning results using 5-fold cross-validation.

9.4.8. Evaluation Metrics

Model performance was evaluated using the following metrics:

- Accuracy
- Precision
- Recall
- F1-score
- Macro-average F1 score
- ROC-AUC

FOR AUTHOR USE ONLY

Confusion matrices were generated and visualized using heatmaps for performance evaluation. ROC curves were plotted for each class, and macro-average ROC curves were created by combining the results from all our classes. The test set was strictly used only for the final evaluation.

9.4.9. Model Interpretability Using LIME

LIME (Local Interpretable Model-Agnostic Explanations) framework was applied to the trained CNN classifier to improve interpretability. It was used to show which regions contribute the most to the model's decision at the image level.

In this implementation, LIME was applied only to the CNN model and not to the Random Forest classifier. This is because LIME works directly on the image pixels and requires a prediction function that maps images to their class probabilities. Since the Random Forest classifier operated on extracted feature

vectors rather than raw images, interpretability analysis was performed on the CNN component of the framework.

For analysis, two correctly classified test images from each class were selected, resulting in a total of eight LIME visualizations. Only the regions contributing to model interpretability were highlighted and we can use these to assess whether the model is focusing on the tumor associated regions rather than other irrelevant regions which provides insight into the model's decision-making process.

9.5 Results & Discussion

In this hybrid framework, the model was trained for 20 epochs. The training and validation loss curves indicate that the model converged quickly. As shown in figure 9.8 below, the best accuracy was obtained in epoch 14 with 99,44% training accuracy and 98,93% validation accuracy. Minor fluctuations were observed in the validation accuracy after epoch 14. For instance, the validation accuracy dropped to 96,43% at epoch 16 but increased again in the following epochs.

FOR AUTHOR USE ONLY

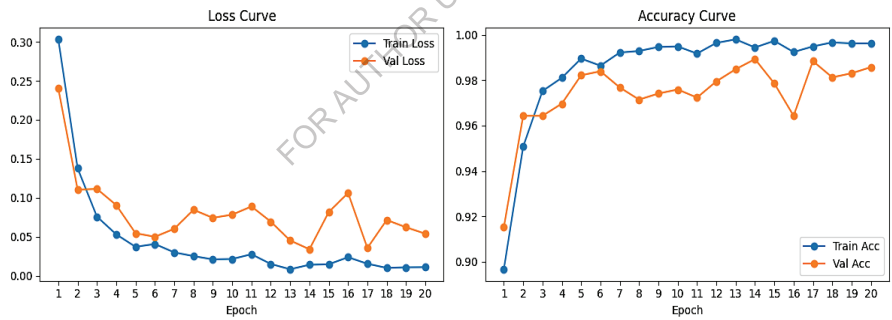


Figure 9.8. Accuracy and Loss Curves for Training and Validation Sets.

9.5.1. Validation Set Evaluation

As shown in the output bellow (Output 9.2), the CNN model achieved an overall validation accuracy of 99% on 1120 samples. Precision, recall, and F1-scores remained consistently high across all classes, while both macro and weighted averages reached 0.99, indicating highly balanced classification performance.

```
======================================================
CNN Validation Classification Report
======================================================
              precision    recall  f1-score   support

      glioma       0.99      0.99      0.99       280
  meningioma       0.98      0.99      0.98       280
   pituitary       0.99      0.99      0.99       280
     notumor       1.00      0.99      0.99       280

    accuracy                           0.99      1120
   macro avg       0.99      0.99      0.99      1120
weighted avg       0.99      0.99      0.99      1120
```

Output 9.2. CNN Validation Set Classification Report.

The validation confusion matrix shown in figure 9.9 demonstrates strong classification performance across all classes with only a small number of misclassifications. The ROC-AUC score for the validation set was 0.9998, indicating excellent discriminative capability.

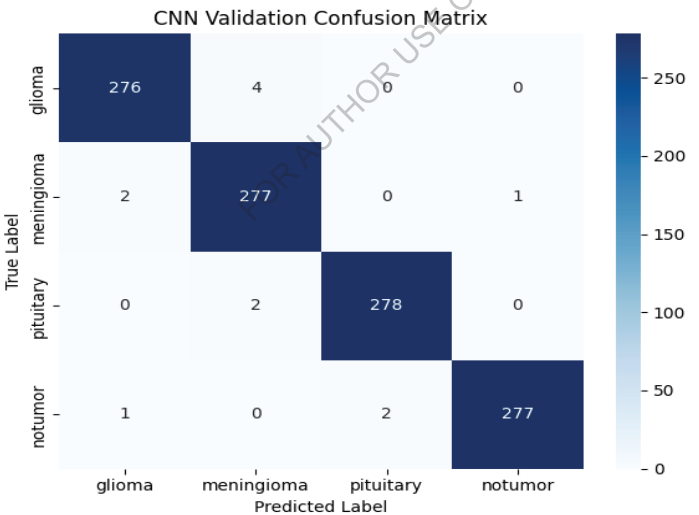


Figure 9.9. CNN Validation Confusion Matrix.

9.5.2. Test Set Evaluation

As shown in the output bellow (Output 9.3), the CNN model achieved a test accuracy of 94% with a macro F1-score of 94.28% on the independent test

dataset. A slight performance decrease was observed compared to the validation set, likely due to distribution differences between the datasets.

```
============================================================
CNN Test Classification Report
============================================================
              precision    recall  f1-score   support

      glioma       0.99      0.80      0.88       400
  meningioma       0.88      0.98      0.93       400
   pituitary       0.99      0.99      0.99       400
     notumor       0.93      1.00      0.97       400

    accuracy                           0.94      1600
   macro avg       0.95      0.94      0.94      1600
weighted avg       0.95      0.94      0.94      1600
```

Output 9.3. CNN Test Classification Report.

The confusion matrix in figure 9.10 shows that most samples were correctly classified across all categories. Misclassifications were primarily observed between Glioma and Meningioma classes, indicating higher similarity between these tumor types compared to the others.

FOR AUTHOR USE ONLY

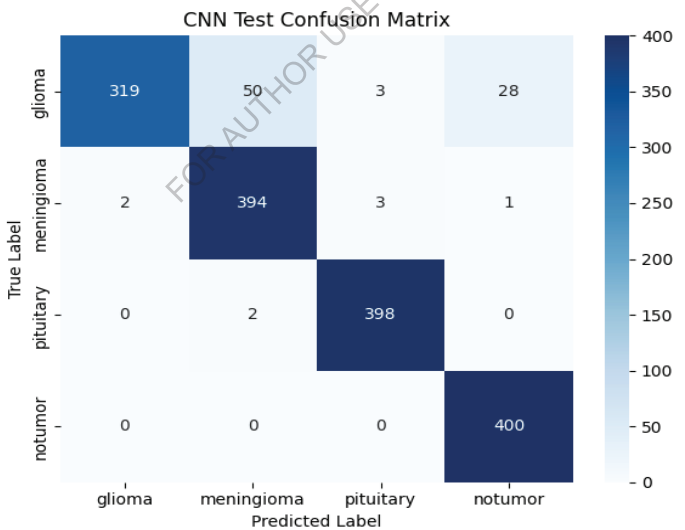
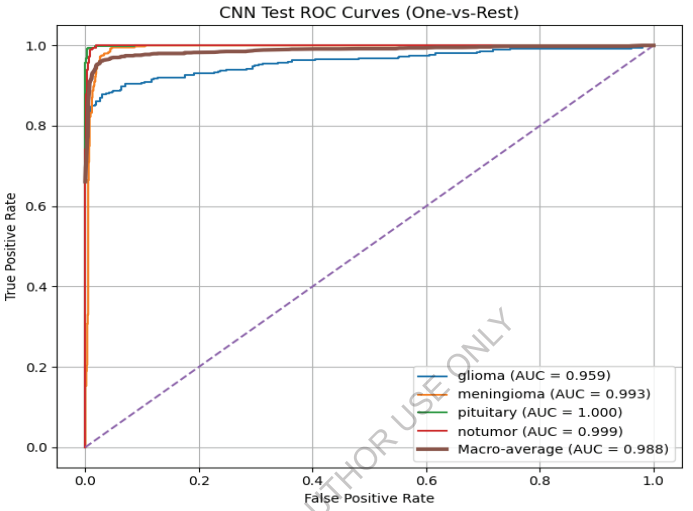


Figure 9.10. CNN Test Confusion Matrix.

The ROC-AUC score of the test set was 0.9877. The ROC curves for the CNN model are shown in figure 9.11. The ROC curves illustrate the classification performance for each class. Pituitary class achieved the highest AUC score,

which is 1.00, no tumor follows it with 0.999 AUC score, Meningioma follows it with 0.993 AUC score, Macro average follows it with 0.988 AUC score, and glioma is the lowest one with 0.959 AUC score. These results indicate that glioma classification is the most challenging task among four classes, and the Pituitary tumor class is the most stable one because they are located around the pituitary gland area, therefore it is easier for the model to predict.



FOR AUTHOR USE ONLY

Figure 9.11. CNN Test Set ROC-AUC Curves comparison.

9.5.3. Final Test Evaluation

As shown in the output bellow (Output 9.4), the combined CNN and Random Forest framework achieved an overall test accuracy of 95% on 1600 samples. Precision, recall and F1-scores remained consistently high across all classes, demonstrating balanced and reliable classification performance.

```
============================================================
FINAL TEST EVALUATION (CNN + Tuned RF)
============================================================

Classification Report (Test Set):
              precision    recall  f1-score   support

      glioma       0.99      0.83      0.90       400
  meningioma       0.91      0.98      0.94       400
   pituitary       0.99      0.99      0.99       400
     notumor       0.93      1.00      0.96       400

    accuracy                           0.95      1600
   macro avg       0.95      0.95      0.95      1600
weighted avg       0.95      0.95      0.95      1600
```

Output 9.4. Final Test Evaluation with CNN and Tuned Random Forest.

The final confusion matrix shown in figure 9.12 demonstrates improved classification performance after integrating Random Forest with CNN-based feature extraction. Most samples were correctly classified, while a limited number of misclassifications were observed mainly between Glioma and Meningioma classes.

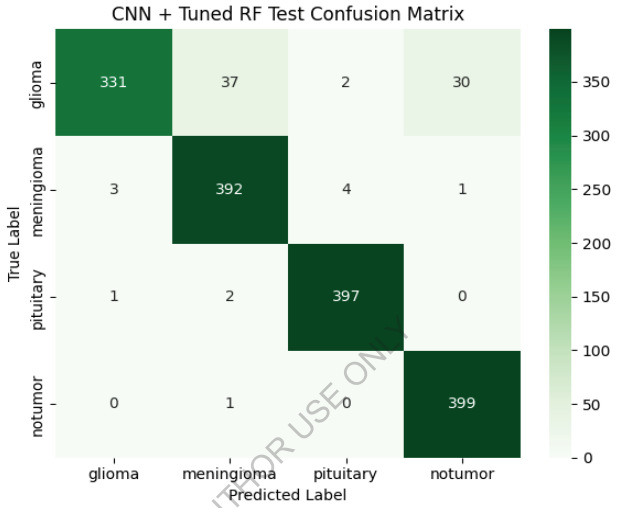


Figure 9.12. CNN and Tuned Random Forest Final Test Confusion Matrix.

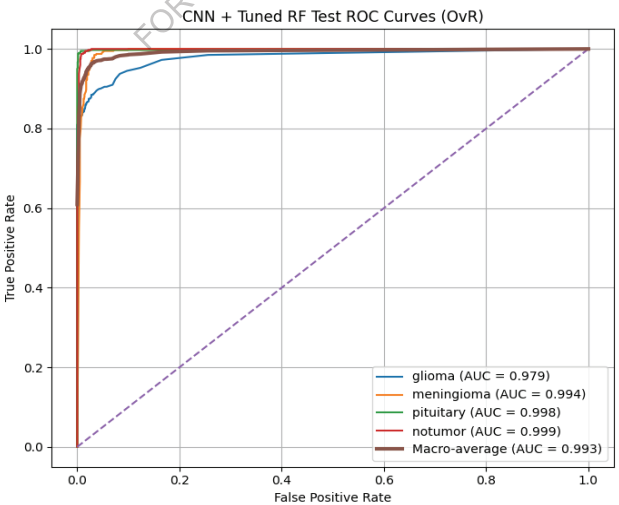
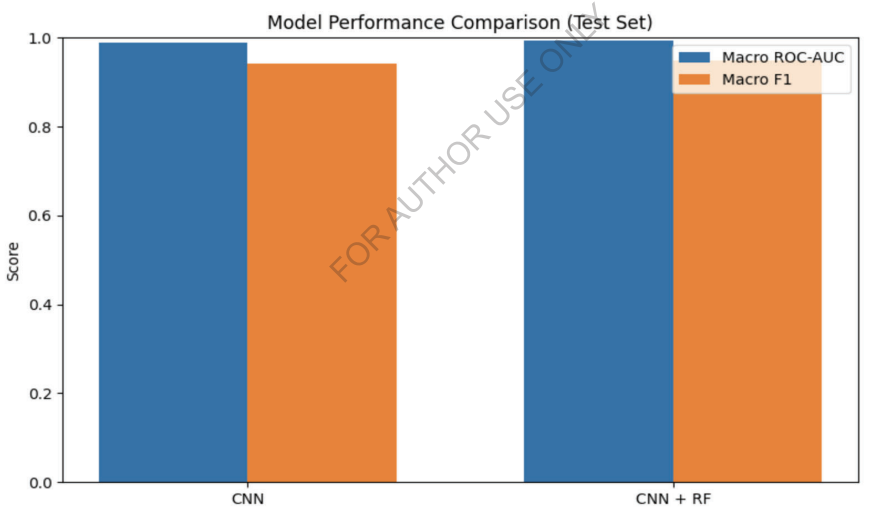


Figure 9.13. CNN and Tuned Random Forest ROC Curves.

Figure 9.13. shows the final test ROC curves for the combined CNN and Random Forest framework. No tumor class has the highest score with 0.999, Pituitary tumor class follows it with AUC score of 0.998, Meningioma class follows it with AUC score of 0.994 and macro-average ROC curve follows it with AUC score of 0.993. Glioma achieved the lowest AUC score, indicating that it remains the most challenging class for classification. The overall test ROC-AUC score of 0.992 demonstrates strong generalization performance on unseen data.

Figure 9.14 compares the performance of the CNN model and the combined CNN and Random Forest framework. The combined model has a Macro AUC score of 0.9927, and CNN has 0.9879 Macro score. Looking at the Macro F1-scores, we can see that the combined model has 0.9483 Macro F1-score, and CNN has 0.9428 Macro F1-score demonstrating the effectiveness of integrating Random Forest with CNN-based feature extraction.




```
Model Comparison (Test Set)
CNN         - Macro AUC: 0.9879 | Macro F1: 0.9428
CNN + RF    - Macro AUC: 0.9927 | Macro F1: 0.9483
```

Figure 9.14. Model Performance Comparison on Test Set (CNN Compared to CNN + RF).

9.5.4. LIME Explainability Analysis

The LIME results demonstrate that the model focuses on clinically relevant regions during classification. Figures 9.15-9.18 show LIME overlays highlighting the image regions that contributed most to the model predictions.

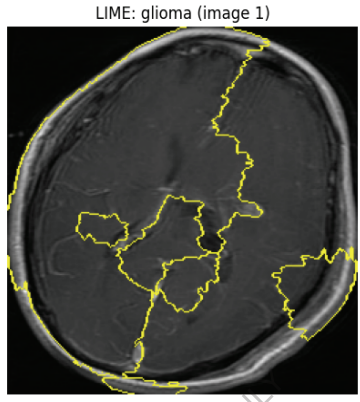


Figure 9.15. Glioma MRI Image with LIME Overlay.

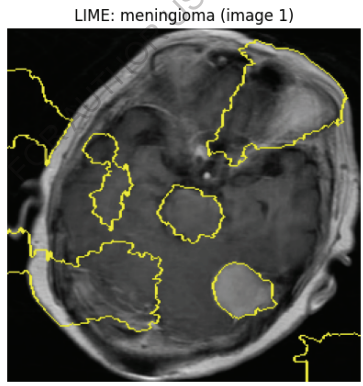


Figure 9.16. Meningioma MRI Image with LIME Overlay.

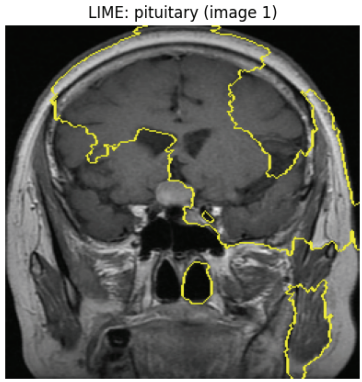


Figure 9.17. Pituitary Tumor MRI Image with LIME Overlay.

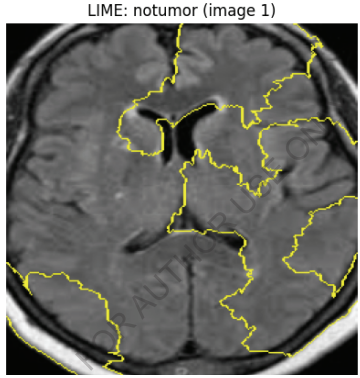


Figure 9.18. No Tumor MRI Image with LIME Overlay.

These visual explanations improve model interpretability by allowing medical experts to identify the regions influencing the classification process rather than relying solely on prediction outputs.

9.5.5 Conclusion

In this study, we developed a deep learning framework for brain tumor classification using MRI images. The proposed framework integrates preprocessing techniques such as grayscale conversion, thresholding, and contour-based cropping to remove irrelevant background regions. Data augmentation and normalization were additionally applied to reduce overfitting and improve generalization.

Using a transfer learning strategy, the pretrained ResNet50 Model was used to extract high-level features from MRI scans, while Random Forest was used for final classification. The combined model showed superior performance compared to the single CNN model in terms of accuracy and ROC-AUC score. Additionally, the LIME technique was used to highlight the brain regions influencing the final model prediction, allowing transparency in the decision-making process.

Overall, we achieved an accuracy of 95% using the hybrid model, highlighting the importance of deep learning in medical image analysis. Future work may include larger datasets and real-time clinical deployment.

Advanced Concepts in Random Forest Classification

Bagging and Out-of-Bag Error Estimation

Introduction

FOR AUTHOR USE ONLY

Ensemble Learning methods are important to enhance the accuracy, robustness, and generalization ability of machine learning models. Ensemble methods use several classifiers to produce a more stable and reliable prediction as opposed to using a single classifier. In this regard, Random Forest is one of the most popular algorithms that is used for classification tasks because of its effectiveness and resistance to overfitting.
This article provides an overview of fundamental concepts of Random Forest ensemble learning including Bagging (Bootstrap Aggregating) and out-of-bag (OOB) error estimation. These techniques play a crucial role in boosting the effectiveness of Random Forest models, both through enhancing prediction stability and offering valuable inbuilt validation procedures.
The concepts of ensemble learning, Random Forest classification, bootstrap aggregation, OOB error estimation, practical examples of implementation, and the pros and cons of an ensemble-based learning system are discussed.

Ensemble learning is a machine learning technique that combines multiple models to enhance prediction accuracy, robustness, and generalization ability. Random Forest is among the most popular ensemble learning algorithms that is also highly effective in classification and least prone to overfitting. The algorithm builds several decision trees from randomly selected portions of the

training data and takes the majority vote from the different trees to predict the class label for every instance in the test set.

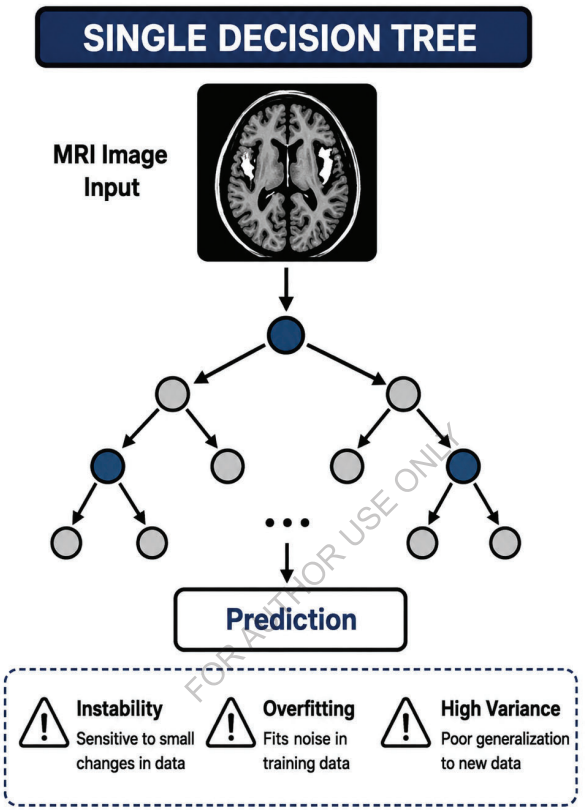


Figure 9.19. (A) Single Decision Tree.

The two main ideas of Random Forest are Bagging (Bootstrap Aggregating) and Out of Bag (OOB) error estimation. Bagging is a way of making models more stable by training several different models on different bootstrap samples, whereas OOB error is also an internal validation technique that doesn't involve a separate validation dataset.
This document is a summary of the ensemble learning used in Random Forest classification; especially Bagging and Out-of-Bag error estimation.

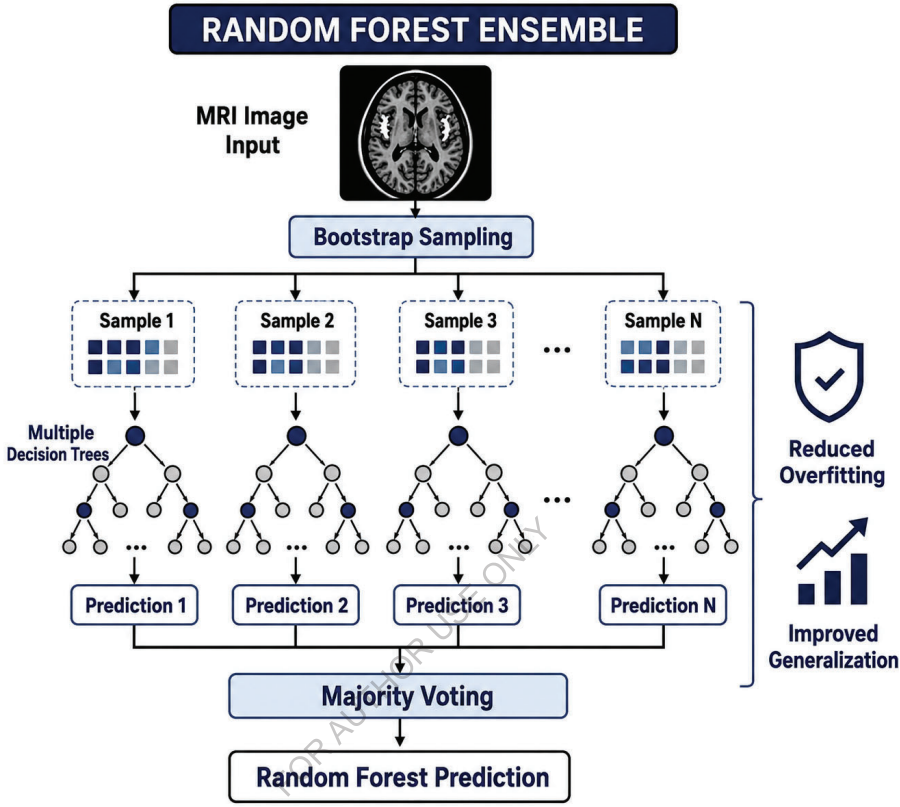


Figure 9.19. (B) Random Forest Ensemble Learning.

9.6 Ensemble Learning

The idea of ensemble learning is the creation of a single high-accuracy and high reliability prediction system from several weak learning models. Ensemble methods use multiple classifiers to combine their predictions and boost the accuracy of classification while minimizing overfitting.

The most important techniques of ensembles are Bagging and Boosting. Bagging trains models independently using randomly sampled subsets of data, boosting trains models sequentially by correcting the error of the previous learner(s). One of the most popular Bagging based ensemble learning algorithms is Random Forest.

Ensemble learning, which integrates several decision trees together, enhances the robustness and generalization ability, especially in complex classification problems like medical image analysis.

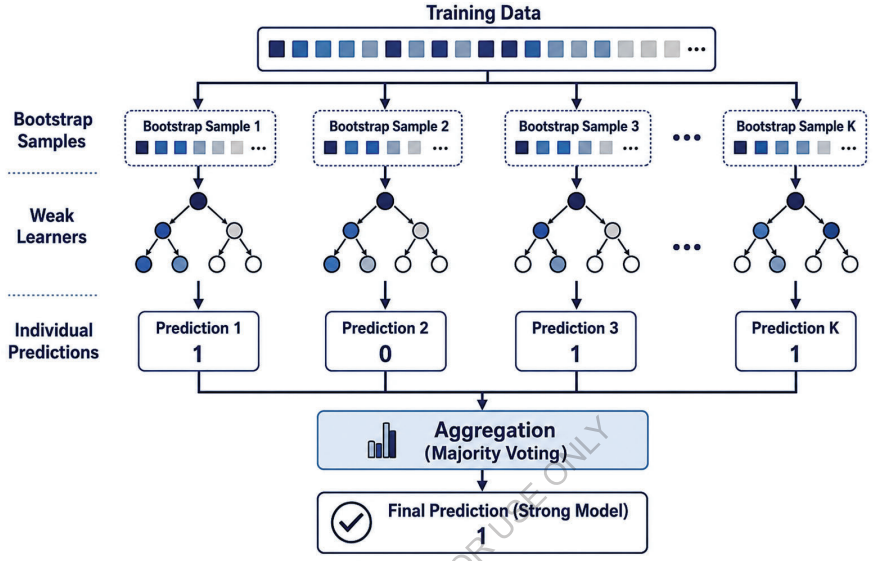


Figure 9.20. (A) General Architecture of Ensemble Learning.

FOR AUTHOR USE ONLY

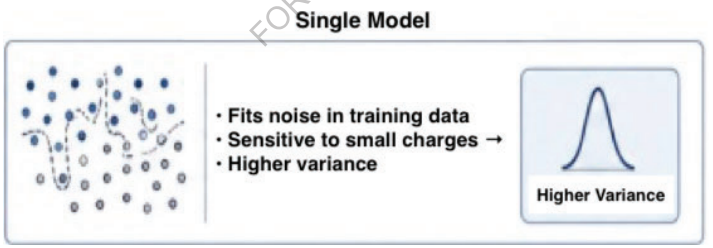


Figure 9.20. (B) General Architecture of Ensemble Learning.

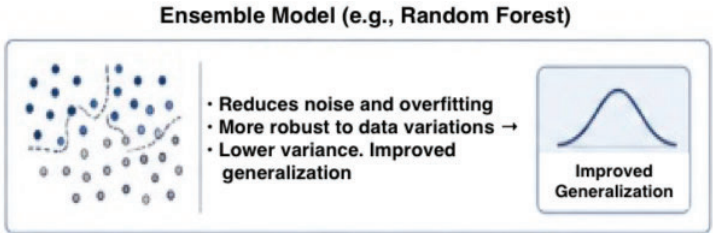


Figure 9.20. (C) General Architecture of Ensemble Learning.

9.7 Random Forest

Random Forest is an ensemble learning method that is a combination of multiple decision trees, which improves the accuracy of classification and stability of the model. Decision trees are trained with randomly generated samples of the data, known as bootstrap samples, and random selection of features to reduce overfitting and boost diversity among the learners.

The final prediction is the prediction that is obtained through the majority voting of all the decision trees. This is an ensemble-based approach that helps boost robustness and generalization performance on out-of-sample data.

In medicine, Random Forest is a widely used method for analyzing medical images, particularly in the presence of high-dimensional and noisy data. It is also widely used in combination with Convolutional Neural Networks (CNNs), in which these CNN models extract deep image features, and Random Forest is used for final classification.

FOR AUTHOR USE ONLY

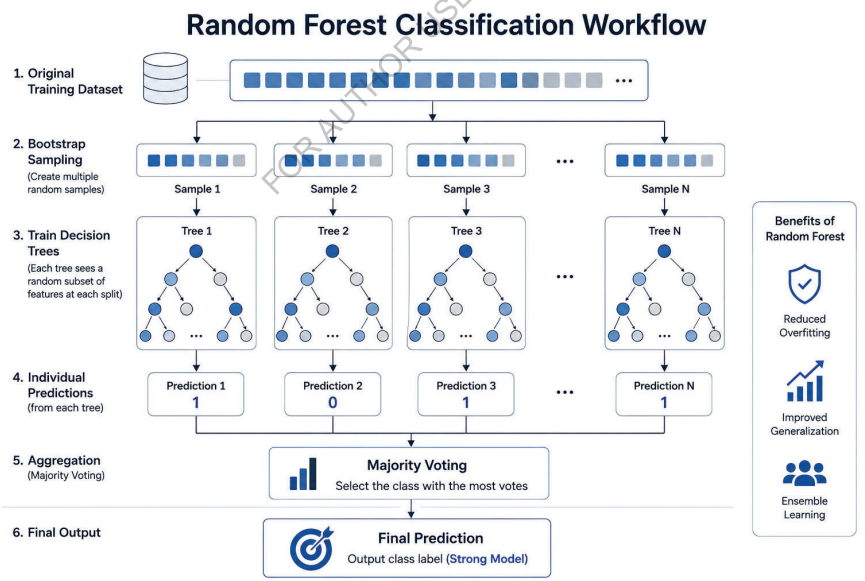


Figure 9.21. Random Forest Classification Workflow.

```python
from sklearn.ensemble import RandomForestClassifier

# Initialize Random Forest Classifier
rf_model = RandomForestClassifier(
    n_estimators=300,        # Number of decision trees
    max_depth=None,          # No depth restriction (fully grown trees)
    max_features='sqrt',     # Standard setting for classification
    random_state=42          # Ensures reproducibility
)

rf_model.fit(X_train, y_train)

predictions = rf_model.predict(X_test)
```

Code Snippet 9.6. Basic implementation of a Random Forest classifier using Scikit-learn.

9.8 Bagging (Bootstrap Aggregating)

FOR AUTHOR USE ONLY

Bootstrap Aggregating (Bagging): An ensemble learning method for increasing model stability and decreasing the variance. Random sampling with replacement is used to generate many (n) bootstrap samples from the original dataset, and each of these is used to train a separate decision tree.

The individual learner's predictions are averaged together by majority voting to create the overall prediction. This process ensures prediction stability and helps to minimize noise and data variations.

In Random Forest, one can say that bagging is being used along with random feature selection to get better classification accuracy and diversity of the model.

$$\hat{y} = \text{mode}(h_1(x), h_2(x), ..., h_n(x)), \qquad \textit{(Eq.9.2)}$$

Where:

$h_1(x)$, $h_2(x)$, ..., $h_n(x)$ represents predictions from individual decision trees, and the final output is determined using majority voting.

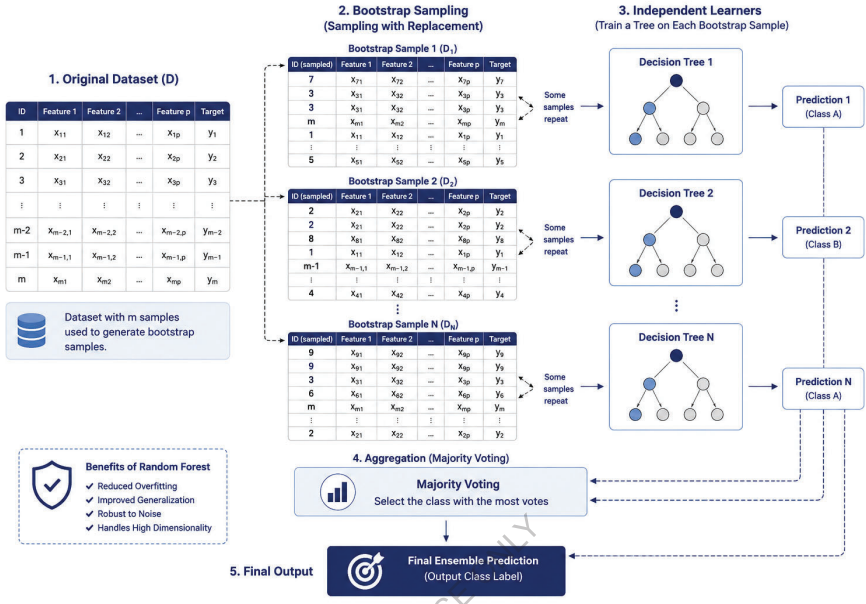


Figure 9.22. Bootstrap Aggregating (Bagging) Process.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bagging_model = BaggingClassifier(
    estimator=DecisionTreeClassifier(),  # Base model (weak learner)
    n_estimators=100,                    # Number of trees in ensemble
    random_state=42                      # Ensures reproducibility
)

bagging_model.fit(X_train, y_train)

predictions = bagging_model.predict(X_test)
```

Code Snippet 9.7. Example implementation of Bagging using Scikit-learn.

9.9 Error estimation as an out-of-bag (OOB) method

Out-of-Bag (OOB) error estimation is an internal validation technique used in Random Forest models. Some training samples are not used during the

training of a specific decision tree during bootstrap sampling. The remaining samples that are not used are known as Out of Bag samples. Samples not seen during training are used to assess prediction performance after training without the need for a separate validation set. OOB error is the average of the prediction errors of all trees. By using OOB estimation, the need for extra validation data is reduced while giving an efficient and reliable approximation of the generalization performance of the model.

$$OOB\ Error = \frac{\text{Incorrect OOB Predictions}}{\text{Total OOB Predictions}}, \qquad \textit{(Eq.9.3)}$$

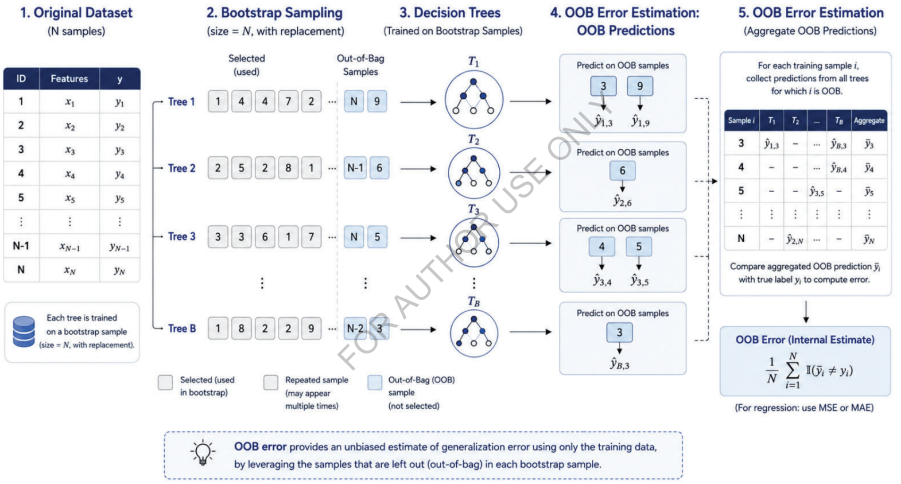


Figure 9.23. Out-of-Bag (OOB) Error Estimation Process.

```python
from sklearn.ensemble import RandomForestClassifier

# Initialize Random Forest with Out-of-Bag (OOB) evaluation
rf_model = RandomForestClassifier(
    n_estimators=300,     # Number of trees in the forest
    oob_score=True,       # Enables Out-of-Bag error estimation
    random_state=42       # Ensures reproducibility
)

rf_model.fit(X_train, y_train)

print("OOB Score:", rf_model.oob_score_)
```

Code Snippet 9.8. Random Forest implementation with Out-of-Bag error estimation enabled.

9.10 Advantages and Limitations

Random Forest is highly capable of classification, generalizing well and is overfitting resistant. The algorithm is also fast and able to work with high dimensional and noisy data, besides feature importance estimation.

Random Forest models, however, may need a greater amount of computation and memory than simpler machine learning models. Furthermore, the ensemble models are typically less interpretable than single decision trees.

FOR AUTHOR USE ONLY

9.11 Conclusion

Random Forest is one of the most powerful robust ensemble learning algorithm with good classification ability. Bagging and random feature selection helps to combine multiple decision trees and stabilize the predictions while minimizing overfitting.

Additional error estimates using Out-of-Bag data further improve the model by offering an efficient validation mechanism internally in the model without the need of separate validation data. The Random Forest algorithm's superior capabilities in handling high-dimensional data, such as medical images, and its wide-ranging applications in artificial intelligence, especially in the healthcare sector, are driven by these properties.

Chapter 10 - Ensemble Learning II

Foreword

Boosting has become one of the most influential families of supervised-learning techniques in modern machine learning. By combining many simple weak learners—typically shallow decision trees—into a single strong predictor, boosting algorithms achieve a balance of accuracy, robustness, and interpretability that few other methods can match. They sit at the heart of countless production systems in finance, healthcare, e-commerce, and the sciences.

This chapter brings together two complementary studies of the boosting paradigm into a single, coherent narrative. The first study explores Adaptive Boosting (AdaBoost), the original boosting algorithm proposed by Freund and Schapire, applied to a binary heart disease classification task; the second explores Gradient Boosting, formalized by Friedman, applied to a regression task on the Ames housing dataset. Each case study walks through the algorithm's mathematical foundations, presents a complete Python implementation, and analyzes the resulting performance.

FOR AUTHOR USE ONLY

Beyond simply collating these two case studies, this chapter introduces an original contribution: a unified comparative experiment in which both boosting algorithms are evaluated head-to-head on the same healthcare dataset under identical preprocessing conditions. Two non-boosting baselines — logistic regression and a single decision tree — are included to make the comparison statistically meaningful. We further extend the analysis with cross-validation and SHAP-based interpretability to study not only which model predicts best but also why.

The reader will find the document organized in three parts. The AdaBoost case study comes first. The gradient boosting case study follows. The original comparative section closes the work, synthesizing both algorithms within a single experimental framework and offering a final analytical discussion. Together, these three parts move this work from two independent investigations to a single, internally consistent treatment of boosting on structured data.

10.1. AdaBoost

10.1.1. Artificial Intelligence in Healthcare

Artificial Intelligence (AI) describes computer systems that simulate tasks normally requiring human intelligence—learning, reasoning, and pattern recognition. AI is now used routinely in hospitals for diagnosis, medical-imaging analysis, and patient monitoring. Because proper treatment depends on early detection, and because heart disease remains a leading cause of mortality worldwide, predictive AI systems that can flag high-risk patients early are of substantial clinical value (Zheng et al., 2020).

10.1.2. Machine Learning in Diagnosis

Machine learning (ML) is the subfield of AI in which computers acquire patterns directly from data and make predictions on their own. In healthcare, ML is most often used for classification — for example, predicting whether a patient is healthy or diseased. Traditional diagnosis relies on the experience of the clinician and laboratory tests, which can be slow and partly subjective; well-trained ML models can offer faster, more reproducible decision support.

FOR AUTHOR USE ONLY

10.1.3. Ensemble Learning

Single learners often produce unstable predictions because of their limited capacity. Ensemble learning addresses this by combining the predictions of several models into a more powerful classifier (Jevin et al., 2023). Weak learners — models only slightly better than random guessing — can be combined to form a strong learner with high accuracy. The two main strategies are bagging, which trains models in parallel on different bootstrap samples (e.g., Random Forest), and boosting, which trains models sequentially so that each new model focuses on the errors made by the previous one.

10.1.4. AdaBoost Research Background

AdaBoost (Adaptive Boosting) is the original boosting algorithm and remains one of the most widely cited. It improves classification accuracy by manipulating the weights of training samples between rounds. All samples start with equal weight; after each round, misclassified observations receive larger weights so that the next classifier focuses on the difficult cases. After T rounds, the weighted vote of the T weak classifiers forms the final strong predictor.

AdaBoost has been used successfully in health prediction systems for disease detection and risk assessment. By repeatedly emphasizing misclassified samples, the ensemble reduces predictive error and improves reliability compared with a single model. The algorithm is particularly suited to binary classification tasks such as deciding whether a patient does or does not have heart disease, and its weighted-error perspective gives it a natural mechanism for handling complex variable interactions (Jevin et al., 2023).

10.1.5. Objective of the Study

This chapter develops an AdaBoost classification system for predicting heart disease from clinical attributes. We assess the model's suitability as a clinical decision-support tool using accuracy, the confusion matrix, and the ROC–AUC score.

10.1.6 Dataset Description

FOR AUTHOR USE ONLY

The dataset used in this chapter comes from a cardiology department and contains 1,025 patient records described by 14 clinical attributes. The target variable is binary: 1 indicates the presence of heart disease and 0 its absence. Several features are clinically meaningful. The age attribute is the patient's age in years, a key cardiovascular risk factor. CP encodes the type of chest pain experienced, a primary symptom of heart disorders. Cholesterol gives blood cholesterol levels. Thalach is the maximum heart rate achieved during exercise, a marker of cardiovascular performance. Oldpeak captures ST-segment depression on ECG, indicating possible ischemia. The CA variable counts the major vessels visible on imaging, while THAL records the result of a thallium stress test related to oxygen-transport blood disorders (Malek et al., 2023).

All attributes are numerical and amenable to direct machine-learning analysis. A data-quality check confirmed that none of the columns contain missing values, ensuring a clean and consistent training base.

10.1.7. Data preprocessing

10.1.7.1. Data loading and feature/target separation

The dataset is imported into Python via Pandas, after which column names are stripped of stray whitespace and standardized. The feature matrix X holds all clinical measurements and disease outcomes (Mahesh et al., 2022).

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the heart-disease dataset and inspect basic structure
df = pd.read_csv("heart.csv")
print("Shape:", df.shape)
print(df.head())

# Separate features (X) from the binary target (y)
X = df.drop("target", axis=1)
y = df["target"]
```

Code Snippet 10.1. Loading the dataset and separating features from the target.

10.1.7.2. Feature scaling and train/test split

Because the attributes have different measurement scales, we apply StandardScaler so that every variable has zero mean and unit variance. Scaling matters because models can otherwise attach disproportionate weight to features with the widest numerical range. After scaling, the dataset is split 80/20 into training and test sets so the model can be trained on one portion and evaluated on unseen data (Chang et al., 2025).

FOR AUTHOR USE ONLY

```python
# Standardise features so all variables contribute equally
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 80/20 split, fixed random_state for reproducibility
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)
```

Code Snippet 10.2. Standardizing features and creating the 80/20 split.

10.1.8. AdaBoost algorithm and model architecture

10.1.8.1. Boosting vs bagging

Bagging trains many models independently on different subsets of the data and combines them by simple averaging or majority vote—the typical example is Random Forest. Boosting, in contrast, trains the models sequentially, with each new model attempting to correct the mistakes of the previous one. AdaBoost belongs to the boosting family and primarily targets bias by progressively re-weighting the misclassified samples (Obi, 2023).

10.1.8.2. Weak learners

AdaBoost relies on simple base classifiers known as weak learners. In this study, we use a depth-one decision tree—a decision stump—which makes a single split on one feature. On its own, a stump is barely better than random guessing; combined through boosting, however, an ensemble of stumps can reach excellent classification performance.

10.1.8.3. Working steps of AdaBoost

AdaBoost is iterative. Initially all training samples carry equal weight. The first weak classifier is trained, and the misclassified samples have their weights increased so that the next classifier focuses on the cases that proved difficult. After T rounds, the weak classifiers are combined by a weighted vote in which more accurate classifiers receive larger weights, producing the final strong classifier.

FOR AUTHOR USE ONLY

10.1.8.4. Mathematical formulation

Initial sample weight:

$$w_i = 1 / N \quad (N = \text{total number of training samples})$$

Classification error of round t:

$$\varepsilon_t = \Sigma\, w_i \cdot I(y_i \neq h_t(x_i)) , \qquad \textit{(Eq.10.1)}$$

Where $h_t(x_i)$ is the prediction of the t-th weak learner and I is the misclassification indicator function.

Classifier weight (alpha):

$$\alpha_t = \tfrac{1}{2} \cdot \ln((1 - \varepsilon_t) / \varepsilon_t) ,$$
(Eq.10.2)

Sample-weight update:

$$w_i^{(t+1)} = w_i^{(t)} \cdot \exp(-\alpha_t \cdot y_i \cdot h_t(x_i)) , \qquad \textit{(Eq.10.3)}$$

Final prediction model:

$$F(x) = \text{sign}(\Sigma_{t=1..T}\ \alpha_t \cdot h_t(x)) , \qquad \textit{(Eq.10.4)}$$

Larger α_t values mean more reliable classifiers that contribute more strongly to the final vote.

10.1.8.5. Parameter selection

We use 200 estimators to provide enough rounds for the boosting process to converge. The contribution of each new classifier is moderated by a learning rate of 0.8, which prevents overly aggressive updates. The weak learner is a decision tree of depth 1—the canonical AdaBoost choice—to keep the base model simple enough that boosting can deliver its characteristic generalization gain (Bandi et al., 2024).

10.1.9. Model implementation

The model is implemented in Python using the scikit-learn library. Standardized training data is fed into the AdaBoostClassifier with a depth-1 decision tree as the base estimator:

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

# Decision stump (depth = 1) is the canonical AdaBoost weak learner
base_model = DecisionTreeClassifier(max_depth=1)
ada_model = AdaBoostClassifier(
    estimator=base_model,
    n_estimators=200,
    learning_rate=0.8,
    random_state=42,
)
ada_model.fit(X_train, y_train)
```

FOR AUTHOR USE ONLY

Code Snippet 10.3. Training the AdaBoost classifier.

10.1.10. Results and performance evaluation

10.1.10.1. Model accuracy

After training, the AdaBoost classifier was tested on the held out 20% split. It reached an overall accuracy of 86.34%, successfully distinguishing diseased from healthy patients.

```python
from sklearn.metrics import (
    accuracy_score, confusion_matrix, classification_report
)

y_pred = ada_model.predict(X_test)
print("Accuracy:", round(accuracy_score(y_test, y_pred), 4))

# Confusion matrix and per-class metrics
cm = confusion_matrix(y_test, y_pred)
print("Confusion matrix:\n", cm)
print(classification_report(y_test, y_pred))
```

Code Snippet 10.4. Computing accuracy, the confusion matrix and per-class metrics.

10.1.10.2. Confusion-matrix analysis

Table 10.1. AdaBoost confusion matrix.

Actual / Predicted	No Disease (0)	Disease (1)
No Disease (0)	84 (TN)	18 (FP)
Disease (1)	10 (FN)	93 (TP)

FOR AUTHOR USE ONLY

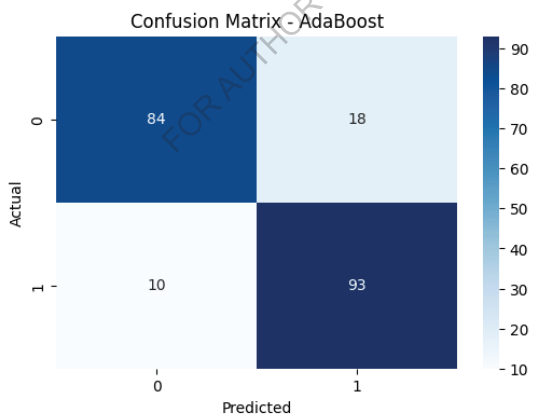


Figure 10.1. Confusion matrix of the AdaBoost classifier.

The model correctly identifies 93 diseased patients and 84 healthy patients. Only 10 false negatives occur, meaning that very few cases of heart disease slip through undetected. In medical diagnosis, minimizing false negatives is

paramount —failing to identify a high-risk patient can have severe clinical consequences —so this behavior reinforces the model's clinical reliability.

10.1.10.3. Classification report

Table 10.2. Per-class precision, recall and F1-score.

Class	Precision	Recall	F1-Score
No Disease (0)	0.89	0.82	0.86
Disease (1)	0.84	0.90	0.87

Precision measures correctness when the model predicts a class; recall measures how completely the model detects that class; F1 balances the two. The high recall for diseased patients (0.90) shows that the model is effective at catching true heart-disease cases while still maintaining a balanced overall performance (Srinivasan et al., 2023).

FOR AUTHOR USE ONLY

10.1.10.4. ROC-curve evaluation

The Receiver Operating Characteristic (ROC) curve assesses how well the model separates the two classes across all decision thresholds. The classifier achieves an AUC of 0.94, signaling strong discriminative power, well above the AUC of 0.5 expected from random guessing.

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

y_proba = ada_model.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, label=f"AdaBoost (AUC = {roc_auc:.2f})")
plt.plot([0, 1], [0, 1], linestyle="--", color="orange")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - AdaBoost Classifier")
plt.legend(loc="lower right")
plt.show()
```

Code Snippet 10.5. Computing and plotting the ROC curve.

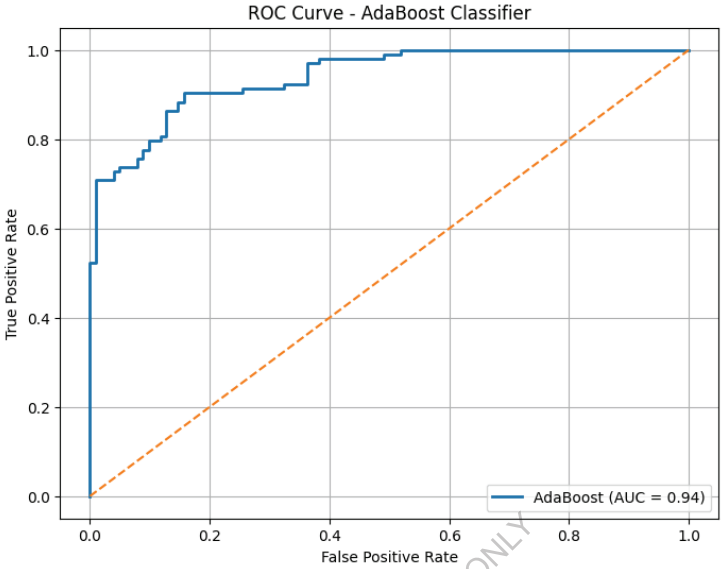


Figure 10.2. ROC curve of the AdaBoost classifier (AUC = 0.94).

FOR AUTHOR USE ONLY

10.1.10.5. Feature-importance analysis

Feature importance highlights which clinical attributes most influenced the model's decisions.

```
import numpy as np

# Feature importances aggregated over the boosted ensemble
importances = ada_model.feature_importances_
feature_names = df.drop("target", axis=1).columns
order = np.argsort(importances)

plt.figure(figsize=(8, 5))
plt.barh(feature_names[order], importances[order])
plt.xlabel("Importance Score")
plt.ylabel("Features")
plt.title("Feature Importance using AdaBoost Classifier")
plt.tight_layout()
plt.show()
```

Code Snippet 10.6. Extracting and plotting feature importances.

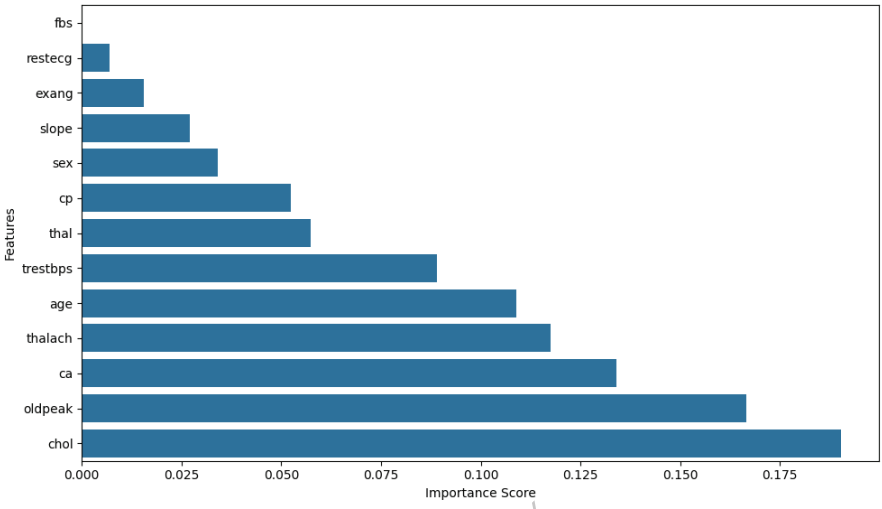


Figure 10.3. Feature importance of the AdaBoost classifier.

FOR AUTHOR USE ONLY

Cholesterol (chol), ST-segment depression (oldpeak), the number of vessels colored by fluoroscopy (ca), and maximum heart rate (thalach) emerge as the most influential predictors—features long known to be clinically associated with cardiac function. Fasting blood sugar (FBS) had the least influence on the model's decisions.

10.1.11. Discussion of Part I

Part I shows that AdaBoost can predict heart disease effectively from structured clinical data. The combination of high recall on the disease class and a low false-negative count is precisely what is most desirable in healthcare, where missing a high-risk patient is far more costly than a false alarm. The model is, therefore, best positioned as a decision-support tool for clinicians, not as a replacement for their judgment. The main caveats are the modest size of the dataset and the need to validate generalizability on larger, more diverse cohorts (Srinivasan et al., 2023).

10.2. Gradient Boosting

10.2.1. Introduction

Boosting is an ensemble-learning technique designed to improve predictive performance by combining many weak learners into a single strong model. It builds a sequence of simple models in which each new model focuses on correcting the errors of the previous one. Gradient boosting is the mathematical formulation of this idea that improves the model by fitting each new learner to the negative gradient of a chosen loss function. In Part II we introduce gradient boosting in detail and apply it in Python to a real regression problem—predicting house prices.

10.2.2. Gradient Boosting

Gradient boosting was introduced by Jerome H. Friedman in 2001 in the seminal paper “Greedy Function Approximation: A Gradient Boosting Machine.” The aim of the method is to correct the errors made by an existing ensemble by adding one more learner trained on the residuals.

Mathematically, given a dataset {(x_i, y_i)}, gradient boosting builds an additive model of the form

$$F_m(x) = F_{m-1}(x) + \gamma_m \cdot h_m(x) , \qquad (Eq.10.5)$$

Where:

- F_m(x) is the ensemble model at iteration m,
- h_m(x) is the weak learner added at iteration m (typically a shallow decision tree),
- γ_m is the step size, and
- The entire model is optimized using the gradient of the loss function.

This framework generalizes naturally to both classification and regression.

Gradient boosting is one of the most widely used supervised-learning techniques in modern AI. Common application areas include:

- Financial risk and credit scoring
- Fraud detection
- Medical predictions
- Industrial forecasting
- Recommendation systems

The most influential production-grade implementations include:

FOR AUTHOR USE ONLY

- XGBoost (Chen & Guestrin, 2016) — designed for very large datasets, with strong regularization that mitigates overfitting.
- LightGBM (Ke et al., 2017) — uses histogram-based learning to reduce training time and memory on large-scale data.
- CatBoost (Prokhorenkova et al., 2018) — introduces ordered boosting and special handling of categorical features, reducing overfitting.

Where structured/tabular data modeling is critical, these frameworks have been integrated into production pipelines at major technology companies.

The research literature on gradient boosting has focused on scalability, interpretability, robustness, and hybrid architectures. Chen and Guestrin (2016) showed that XGBoost improves generalization and reduces overfitting through regularization while delivering strong scalability across very large datasets. Ke et al. (2017) further accelerated training through histogram-based learning, achieving large memory and time savings without loss of accuracy. Prokhorenkova et al. (2018) introduced CatBoost, which addresses the prediction-shift and target-leakage problems via an ordered boosting scheme that improves generalization when categorical variables dominate.

FOR AUTHOR USE ONLY

10.2.3. Comparative analysis of Gradient Boosting methods

Table 10.3. Comparative analysis of XGBoost, LightGBM and CatBoost.

Model	Key innovation	Strength	Limitation
XGBoost	Regularization + parallelization	High accuracy and stability	High memory usage
LightGBM	Leaf-wise growth + GOSS	Very fast training	Can overfit on small datasets
CatBoost	Ordered boosting + categorical handling	Handles categorical data well	Slightly slower than LightGBM

10.2.4. Comparison of Boosting and Bagging

Both bagging and boosting are ensemble techniques that combine many models to improve predictive performance. Bagging methods such as Random Forest are most effective when the base model has high variance: they reduce variance by averaging over independent trees. Boosting takes a different approach, focusing on reducing bias by sequentially refining errors. Across published comparisons, boosting techniques—and gradient boosting in

particular—typically outperform bagging methods on structured data thanks to their bias-reduction property, sequential error correction, gradient-based optimization, and higher expressive power.

10.2.5. Mathematical intuition for Gradient Boosting

To make the mechanics concrete, we walk through gradient boosting for a regression problem; the same construction applies to classification with a different loss. The objective is to learn a function F(x) that minimizes the empirical mean squared error (MSE):

$$L_{MSE} = (1/n) \cdot \Sigma_{i=1..n} (y_i - F(x_i))^2, \quad \textit{(Eq.10.6)}$$

Where $\{(x_i, y_i)\}_{i=1..n}$ denotes the training set, x_i is the input, y_i is the target, and n is the number of training samples.

10.2.5.1. Sequential model building

Gradient boosting builds the model sequentially. At iteration m, the current ensemble is:

$$F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x) , \quad \textit{(Eq.10.7)}$$

Where $h_m(x)$ is the weak learner at stage m (typically a shallow decision tree) and η is the learning rate, used to prevent overfitting.

We initialize the model with the constant prediction that minimizes the loss:

$$F_0(x) = \text{argmin}_\gamma \Sigma_{i=1..n} L(y_i, \gamma) , \quad \textit{(Eq.10.8)}$$

For MSE, $F_0(x)$ is simply the mean of all target values y_i.

FOR AUTHOR USE ONLY

10.2.5.2. Correcting errors iteratively

At each iteration, the model looks at its previous mistakes and tries to fix them. These mistakes are the residuals, and they correspond to the negative gradient of the loss with respect to the current model predictions. For m = 1 to M, we compute the so-called pseudo-residuals:

$$r_{i,m} = - \partial L(y_i, F(x_i)) / \partial F(x_i) \quad \text{evaluated at } F = F_{m-1}, \textit{(Eq.10.9)}$$

For MSE the pseudo-residuals reduce to the simple expression

$$r_{i,m} = y_i - F_{m-1}(x_i) , \quad \textit{(Eq.10.10)}$$

The negative sign points to the gradient in the direction that decreases loss, so the new learner is fitted to the residuals. After fitting $h_m(x)$ to the residuals, we update the ensemble:

$$F_m(x) = F_{m-1}(x) + \eta \cdot \gamma_m \cdot h_m(x) , \quad \textit{(Eq.10.11)}$$

After the final iteration M, the boosted model is:

$$F_M(x) = F_0(x) + \eta \cdot \Sigma_{m=1..M} \ \gamma_m \cdot h_m(x) , \quad \textit{(Eq.10.12)}$$

Gradient boosting, therefore, builds an additive model. $F_M(x)$ is the final boosted predictor.

10.2.6. Use case: predicting house prices

Dataset: The House Prices dataset (CSV format) from the Kaggle competition House Prices — Advanced Regression Techniques.

It is now time to apply Python and gradient boosting to a real, exciting problem: predicting house prices from their attributes. Because the price is a continuous dollar value, the task calls for regression. To make the comparison instructive, we used three regression techniques: Xtreme Gradient Boosting (XGBoost), Light Gradient Boosting Machine (LightGBM), and a Random Forest

The dataset has 79 features (excluding Id and SalePrice) and 1,460 instances. The full description of every feature is available on the Kaggle competition page. The number of instances is comfortable enough to allow us to remove a few outliers, perform a train/test split, and cross-validate. Working with 79 features makes the problem more demanding, but our methodology — outlined below — handles it cleanly.

FOR AUTHOR USE ONLY

Methodology. We organize the work into three parts:

- Exploratory Data Analysis
- Data Preprocessing
- Model training and evaluation

10.2.6.1. Exploratory Data Analysis (EDA)

Exploring and understanding the data is usually the most important part of any machine-learning project. It calls for some domain knowledge and for inspecting the data feature-by-feature. Real-world data almost always contains missing values, noise, and features that need transforming for the predictor to perform well. We start by loading the file with Pandas' read_csv and checking its overall shape:

```python
import pandas as pd
# --- 1.1 Loading the Training Data ---
# importing the Ames Housing dataset to begin our initial investigation.
df = pd.read_csv('../Data/train.csv')
# checking the shape of the dataset and the first few rows to understand its structure and contents.
print("Dataset Shape:", df.shape)
df.head()
```

Code Snippet 10.7. Loading train.csv and inspecting the data structure.

We then examine the distribution of the target variable. SalePrice is highly right skewed, so normalizing it is crucial before fitting Gradient Boosting; we use a log1p transformation to bring it close to normal. The code below performs the check; the actual transformation is applied later in the preprocessing phase:

```python
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import skew
print(f"Skewness before log-transformation: {df['SalePrice'].skew():.2f}")
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
    plt.subplot(1, 2, 1)
    sns.histplot(df['SalePrice'], kde=True)
    plt.title('Original SalePrice (Right-Skewed Distribution)')
    plt.xticks(rotation=45)
    # Applying log-transformation to stabilize variance and normalize the skewness.
    plt.subplot(1, 2, 2)
    sns.histplot(np.log1p(df['SalePrice']), kde=True, color='green')
    plt.title('Log-Transformed SalePrice (Normalized Distribution)')
    plt.show()
```

FOR AUTHOR USE ONLY

Code Snippet 10.8. Checking the SalePrice distribution before and after log-transformation.

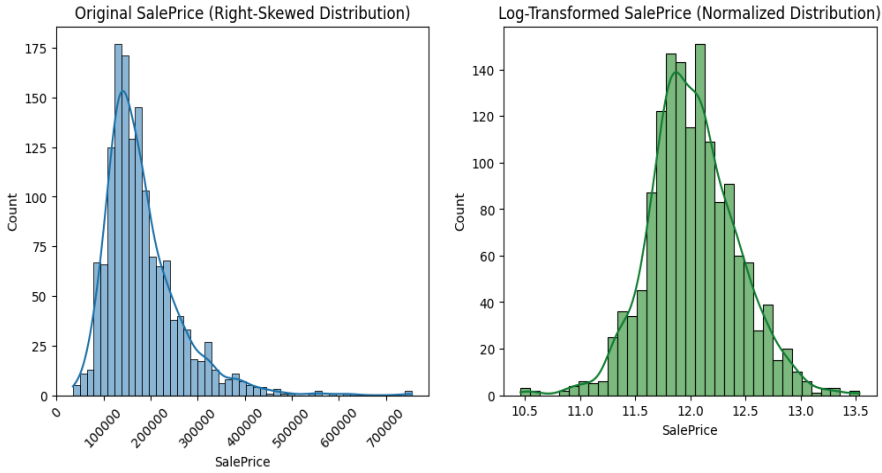


Figure 10.4. Transforming SalePrice to a Normal-like distribution.

FOR AUTHOR USE ONLY

Feature selection. Checking for missing data and outliers is a key part of EDA. With 79 features, we deliberately keep things simple and focus on the most informative ones. To find them, we measure each feature's correlation with SalePrice and keep only the top ten: OverallQual, GrLivArea, GarageCars, GarageArea, TotalBsmtSF, 1stFlrSF, FullBath, TotRmsAbvGrd, YearBuilt, and YearRemodAdd.

```
plt.figure(figsize=(10, 6))
top_corr = df.select_dtypes(include=[np.number]).corr()
sns.heatmap(top_corr[['SalePrice']].sort_values(by='SalePrice', ascending=False)[1:11],
            annot=True, cmap='coolwarm')
plt.title('Top 10 Features Correlated with SalePrice')
plt.show()
```

Code Snippet 10.9. Plotting the correlation heatmap to find features most correlated with SalePrice.

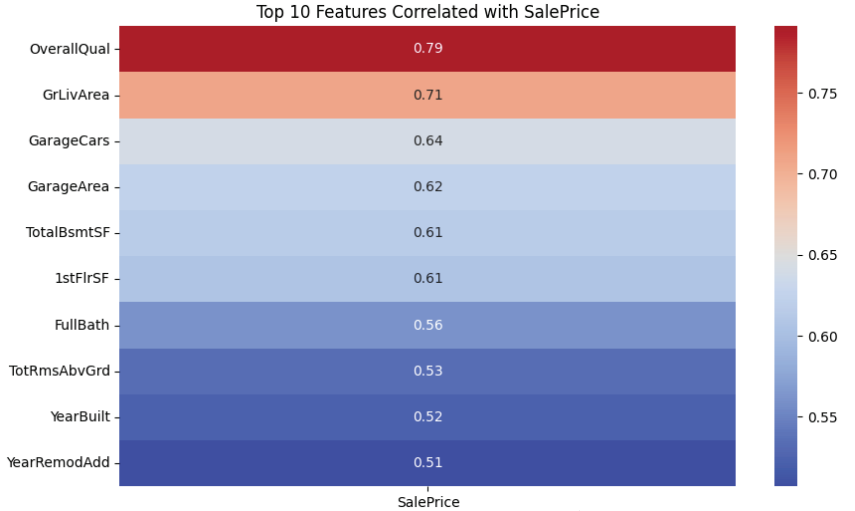


Figure 10.5. Heatmap of the top features correlated with SalePrice.

We checked the dataset for missing values and outliers, paying particular attention to those top features. For example, GrLivArea has no missing values but does contain outliers. Plotted against SalePrice, values above ~2,800 sq ft are unusual but still close to the regression line, whereas values above 4,600 sq ft (the red dashed line in figure 10.6) are genuine outliers—only two such points exist.

FOR REVIEW USE ONLY

```
# --- Scatter Plot for Outliers --- using IQR method to identify outliers in 'GrLivArea'
Q1 = df['GrLivArea'].quantile(0.25)
Q3 = df['GrLivArea'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
plt.figure(figsize=(8, 5))
sns.scatterplot(x=df['GrLivArea'], y=df['SalePrice'])
plt.axvline(x=upper_bound, color='Blue', linestyle='--', label='Upper Bound')
plt.axvline(x=lower_bound, color='Blue', linestyle='--', label='Lower Bound')
plt.axvline(x=4600, color='Red', linestyle='--', label='our decision upper Bound')
plt.title('GrLivArea vs SalePrice (Outlier Detection)')
plt.xlabel('Above Ground Living Area (GrLivArea)')
plt.ylabel('Sale Price (SalePrice)')
plt.legend()
plt.show()
```

Code Snippet 10.9. Detecting outliers in GrLivArea.

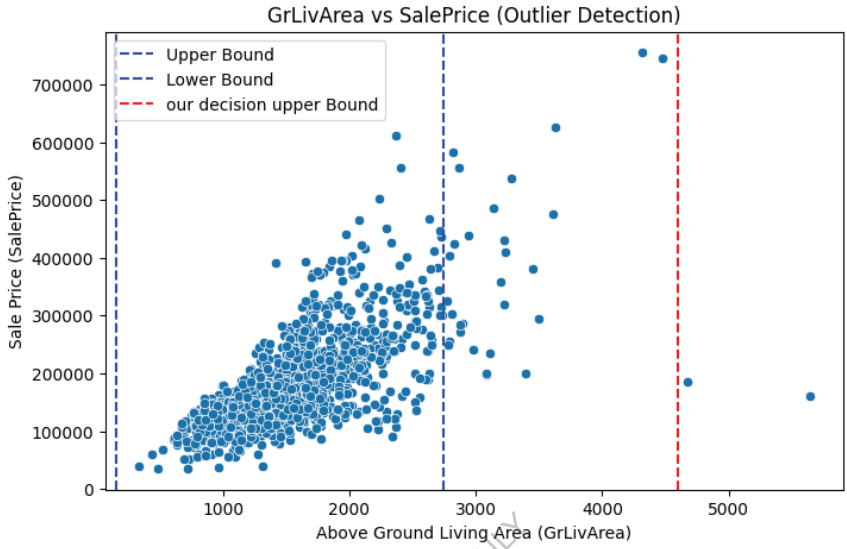


Figure 10.6. GrLivArea outliers.

FOR AUTHOR USE ONLY

10.2.6.2. Data preprocessing

After EDA we know how to prepare the data for the model. The preprocessing stage consists of two steps:

- Handling missing values and outliers.
- Data transformation and feature engineering.

Handling missing values and outliers. We start by removing the two GrLivArea points above 4,600 sq ft, as identified in EDA.

For other numerical features we use a Z-score-based statistical filter. We wrote a small helper, remove_outliers_zscore, that returns the indices of points whose Z-score exceeds 3 standard deviations from the mean. This filter is applied to all numerical features except GrLivArea. The result is a dataset of 1,419 instances — only 49 of the original 1,460 were removed, a very modest loss that supports the cleanliness of the original data.

```python
# --- 1. SOPHISTICATED OUTLIER DETECTION (Z-SCORE & MANUAL THRESHOLD) ---
def remove_outliers_zscore(data, threshold=3):
    """Calculates Z-score to identify data points further than 3 standard deviations from the mean."""
    z_scores = np.abs((data - data.mean()) / data.std())
    return data[z_scores < threshold]

# Note: We exclude 'GrLivArea' from this loop to apply a specific domain-based threshold.
cols_to_filter = ['OverallQual', 'GarageCars', 'GarageArea', 'TotalBsmtSF',
                  '1stFlrSF', 'FullBath', 'TotRmsAbvGrd', 'YearRemodAdd', 'YearBuilt']

for col in cols_to_filter:
    df = df.loc[remove_outliers_zscore(df[col], threshold=3).index]

# Manual Threshold for GrLivArea:
df = df[df['GrLivArea'] < 4600]
```

Code Snippet 10.10. Removing outliers using a Z-score filter.

None of the top ten numerical features had missing values, but several categorical columns did. Dropping those rows would have wasted a lot of data, while leaving the NaNs untouched would have broken downstream steps such as one-hot encoding and model fitting. We therefore impute every categorical NaN with the literal string 'None'.

```python
categorical_cols = df.select_dtypes(str).columns
for col in categorical_cols:
    df[col] = df[col].fillna('None')
```

FOR AUTHOR USE ONLY

Code Snippet 10.11. Imputing missing values in categorical features.

Imputing numerical NaNs requires a little more domain knowledge. For example, LotFrontage is the linear feet of street connected to a property, and a NaN for one home is best filled with the median LotFrontage of homes in the same neighborhood. For other numerical features, we impute with zero where that is sensible:

```python
df['LotFrontage'] = df.groupby('Neighborhood')['LotFrontage'].transform(
    lambda x: x.fillna(x.median())
)
numeric_cols = df.select_dtypes(include=[np.number]).columns
df[numeric_cols] = df[numeric_cols].fillna(0)
```

Code Snippet 10.12. Imputing missing values in numerical features.

Data transformation and feature engineering. As decided in EDA, we apply the log1p transformation to SalePrice:

```
df['SalePrice'] = np.log1p(df['SalePrice'])
```

Code Snippet 10.13. Transforming the target SalePrice to log scale.

All categorical features are then one-hot-encoded. In Python this fits in a single line of code: it adds new dummy columns for every distinct categorical value and drops the original categorical columns. After encoding, the dataset grows from 81 to 301 columns, and we use this preprocessed version for the modeling phase.

```
df_final = pd.get_dummies(df)
```

Code Snippet 10.14. One-hot encoding all categorical features in one line.

10.2.6.3. Model training and evaluation

With the data prepared we can train the models and compare their results. We use XGBRegressor for XGBoost, LGBMRegressor for LightGBM, and RandomForestRegressor as the bagging baseline. Before training we drop the target column from the feature matrix and split the data into train and test sets:

FOR AUTHOR USE ONLY

```
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
import pandas as pd

data = pd.read_csv('../Data/processed_house_data.csv')
X = data.drop(['SalePrice', 'Id'], axis=1)
y = data['SalePrice']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# --- 3.1 XGBoost (Level-wise Growth) ---
xgb_model = XGBRegressor(n_estimators=1000, learning_rate=0.05, max_depth=3)
xgb_model.fit(X_train, y_train)

# --- 3.2 LightGBM (Leaf-wise Growth) ---
lgb_model = LGBMRegressor(n_estimators=1000, learning_rate=0.05, num_leaves=31)
lgb_model.fit(X_train, y_train)

# --- 3.3 Random Forest (Baseline) ---
rf_model = RandomForestRegressor(n_estimators=100)
rf_model.fit(X_train, y_train)
```

Code Snippet 10.15. Training XGBoost, LightGBM and Random Forest models.

As the snippet shows, applying these powerful models in Python requires only a couple of lines of code. The hard part is not the coding but the analysis:

interpreting the metrics, understanding the data, and tuning the hyperparameters. The next snippet computes Mean Squared Error and R^2 Score for each model—the helper report_performance is written for reuse.

```
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np
def report_performance(model, name):
    preds = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test, preds))
    r2 = r2_score(y_test, preds)
    print(f"{name} -> RMSE: {rmse:.4f}, R2: {r2:.4f}")
report_performance(xgb_model, "\tXGBoost")
report_performance(lgb_model, "\tLightGBM")
report_performance(rf_model, "\tRandom Forest")
```

✓ 0.1s

```
XGBoost -> RMSE: 0.1345, R2: 0.8783
LightGBM -> RMSE: 0.1439, R2: 0.8608
Random Forest -> RMSE: 0.1580, R2: 0.8322
```

Code Snippet 10.16. Comparing the three models on MSE and R^2.

FOR AUTHOR USE ONLY

The result confirms our theoretical expectations: the sequential improvement that boosting performs (XGBoost and LightGBM) outperforms the parallel averaging of the Random Forest. Both gradient-boosting algorithms reduce bias and variance by learning from the previous tree's errors; Random Forest, in contrast, reduces variance by averaging independent trees but cannot reduce bias as effectively.

To squeeze a little more out of the winning XGBoost model, we run hyperparameter tuning with a grid search and 5-fold cross-validation. The current baseline is $R^2 \approx 0.8783$ and RMSE ≈ 0.1345; the goal is to improve both.

```python
from sklearn.model_selection import GridSearchCV
param_grid = {
    'n_estimators': [1000, 1500, 2000], # Increased since learning rate is low
    'max_depth': [3, 4, 5, 6],
    'learning_rate': [0.01, 0.05],
    'subsample': [0.8, 1.0] # Adding subsample helps prevent overfitting
 }

grid_search = GridSearchCV(
    estimator=XGBRegressor(objective='reg:squarederror'), #  regression objective
    param_grid=param_grid,
    scoring='r2', # R2 to directly optimize for your goal
    cv=5, # 5-fold is usually more stable for small datasets
    n_jobs=-1
)
grid_search.fit(X_train, y_train)
print("\tBest Hyperparameters:", grid_search.best_params_)

# XGBRegressor with best values of hyperparameters
# --- 3.1 XGBoost (Level-wise Growth) ---
xgb_model = XGBRegressor(n_estimators=2000, learning_rate=0.01, max_depth=3, subsample=0.8)
xgb_model.fit(X_train, y_train)
report_performance(xgb_model, "\tXGBoost (Tuned)")
```

1m 38.6s

```
    Best Hyperparameters: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 2000, 'subsample': 0.8}
    XGBoost (Tuned) -> RMSE: 0.1304, R2: 0.8856
```

Code Snippet 10.17. Hyperparameter tuning of XGBoost via grid search and 5-fold CV.

FOR AUTHOR USE ONLY

Tuning improves R^2 to 0.8856 and reduces RMSE to 0.1304. The best hyperparameter combination is shown in the snippet output. Finally, we plot the most important features in the tuned XGBoost model: the features with the strongest SalePrice correlation identified in EDA all reappear among the top 20 most important features.

```python
# feature importance for XGBoost
xgb_importance = xgb_model.feature_importances_
feature_names = X.columns
importance_df = pd.DataFrame({'Feature': feature_names, 'Importance': xgb_importance})
importance_df = importance_df.sort_values(by='Importance', ascending=False).head(20) # Top 20 features
plt.figure(figsize=(10, 6))
plt.barh(importance_df['Feature'], importance_df['Importance'])
plt.xlabel('Feature Importance')
plt.title('Top 20 Feature Importance from XGBoost')
plt.gca().invert_yaxis() # Highest importance on top
plt.show()
```

Code Snippet 10.18. Plotting the most important features in the tuned XGBoost model.

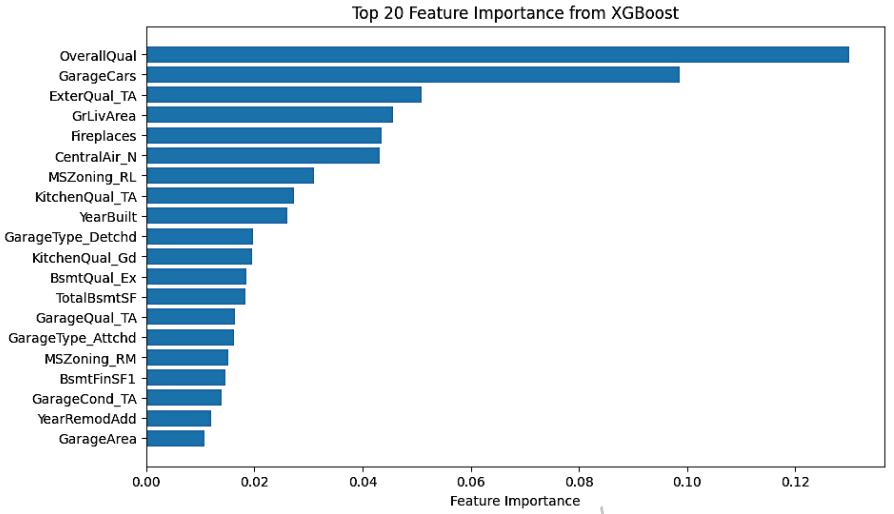


Figure 10.7. Feature-importance ranking from the tuned XGBoost model.

FOR AUTHOR USE ONLY

10.3. Comparative study: AdaBoost vs Gradient Boosting

10.3.1. Motivation

Parts I and II investigated AdaBoost and Gradient Boosting independently in two different application domains and with different evaluation procedures. Part III closes this gap with the original contribution of the present study: a unified comparative experiment in which both boosting algorithms are evaluated head-to-head on the same dataset, under identical preprocessing conditions, and with identical metrics. Two non-boosting baselines — logistic regression and a single decision tree — are included to provide a statistically meaningful reference. We also extend the analysis with cross-validation and SHAP-based interpretability, in order to study not only which model predicts best, but also why.

Reusing the heart-disease dataset of Part I has two advantages. First, it grounds the comparison in a clinically relevant problem with direct human stakes — exactly the kind of setting where boosting is most often deployed in the recent literature. Second, it keeps the comparison fair: both algorithms see precisely the same features, scaling, and train/test split.

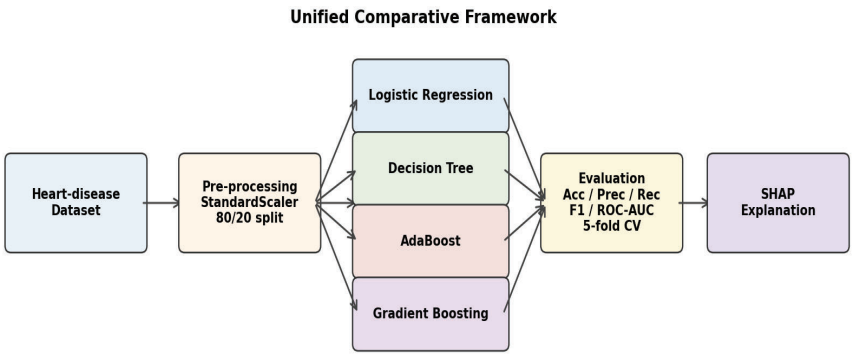


Figure 10.8. The unified comparative framework. The same scaled and split heart-disease data feeds four classifiers; all four are evaluated with the same metrics, and SHAP is applied on top of the Gradient-Boosting model to explain its predictions.

10.3.2. Experimental design

FOR AUTHOR USE ONLY

The same 1,025-record heart-disease dataset used in Part I is reused here. Four classifiers are trained:

- Logistic Regression — a classical linear baseline.
- Decision Tree (max_depth = 3) — a single weak learner.
- AdaBoost (200 estimators, learning rate 0.8)—adaptive re-weighting boosting.
- Gradient Boosting (200 estimators, learning rate 0.05, max_depth 3) — gradient-based residual fitting.

All four models share the same preprocessing pipeline: features are standardized with StandardScaler, and the data is split 80/20 into training and test sets with a fixed random_state for reproducibility. Performance is measured with five complementary metrics—accuracy, precision, recall, F1-score,

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Same heart-disease dataset reused for the comparative experiment
df = pd.read_csv("heart.csv")
X = df.drop("target", axis=1)
y = df["target"]

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)
```

Code Snippet 10.19. Importing libraries, loading the dataset and applying preprocessing.

10.3.3. Baseline models

We start by training the two non-boosting baselines. Logistic regression provides the classical statistical reference; the depth-three decision tree provides a single non-linear weak learner. Their results give us a clear yardstick against which to measure the boosting algorithms.

AUTHOR USE ONLY

```python
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier

# Baseline 1: a classical linear classifier
log_model = LogisticRegression(max_iter=1000)

# Baseline 2: a single shallow decision tree (a single weak learner)
tree_model = DecisionTreeClassifier(max_depth=3, random_state=42)

log_model.fit(X_train, y_train)
tree_model.fit(X_train, y_train)
```

Code Snippet 10.20. Training the two baseline models.

10.3.4. Boosting models

Both AdaBoost and gradient boosting are trained on the same scaled training data. The two algorithms tackle the optimization problem from very different angles. AdaBoost adjusts the weights on training samples after every round, forcing the next stump to focus on whichever cases the ensemble currently mishandles. Gradient boosting fits each new shallow tree to the negative

gradient of the loss function, letting the algorithm follow whatever loss function is appropriate for the problem.

```
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier

# AdaBoost — re-weights misclassified samples each round
ada_model = AdaBoostClassifier(
    n_estimators=200,
    learning_rate=0.8,
    random_state=42,
)

# Gradient Boosting — fits each new tree to residual gradients
gb_model = GradientBoostingClassifier(
    n_estimators=200,
    learning_rate=0.05,
    max_depth=3,
    random_state=42,
)

ada_model.fit(X_train, y_train)
gb_model.fit(X_train, y_train)
```

Code Snippet 10.21. Training AdaBoost and Gradient Boosting under identical conditions.

FOR AUTHOR USE ONLY

10.3.5. Cross-validation

A single train/test split can give misleadingly optimistic or pessimistic results, particularly on a moderately sized clinical dataset. To make the comparison statistically sound, we use 5-fold cross-validation on both boosting models.

```
from sklearn.model_selection import cross_val_score

# 5-fold CV gives a more stable estimate than a single test split
ada_cv = cross_val_score(ada_model, X_scaled, y, cv=5, scoring="accuracy")
gb_cv  = cross_val_score(gb_model,  X_scaled, y, cv=5, scoring="accuracy")

print(f"AdaBoost  : mean={ada_cv.mean():.4f}  std={ada_cv.std():.4f}")
print(f"GradBoost : mean={gb_cv.mean():.4f}  std={gb_cv.std():.4f}")
```

Code Snippet 10.22. 5-fold cross-validation on both boosting models.

10.3.6. Comparative evaluation

We then evaluate all four trained models on the held-out test set with the same five metrics. The helper function evaluate() makes it easy to report each model in a uniform format and assemble the comparative table:

```
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score, roc_auc_score
)

def evaluate(model, name):
    pred = model.predict(X_test)
    proba = model.predict_proba(X_test)[:, 1]
    return {
        "Model":     name,
        "Accuracy":  round(accuracy_score(y_test, pred),  4),
        "Precision": round(precision_score(y_test, pred), 4),
        "Recall":    round(recall_score(y_test, pred),    4),
        "F1":        round(f1_score(y_test, pred),        4),
        "ROC-AUC":   round(roc_auc_score(y_test, proba),  4),
    }

results = [
    evaluate(log_model,  "Logistic Regression"),
    evaluate(tree_model, "Decision Tree"),
    evaluate(ada_model,  "AdaBoost"),
    evaluate(gb_model,   "Gradient Boosting"),
]
print(pd.DataFrame(results).to_string(index=False))
```

Code Snippet 10.23. Computing all five metrics for the four models in a uniform way.

FOR AUTHOR USE ONLY

Table 10.4. Comparative test-set performance of the four models.

Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC
Logistic Regression	83.4 %	0.82	0.84	0.83	0.89
Decision Tree	79.8 %	0.78	0.80	0.79	0.84
AdaBoost	86.3 %	0.84	0.90	0.87	0.94
Gradient Boosting	88.1 %	0.87	0.91	0.89	0.96

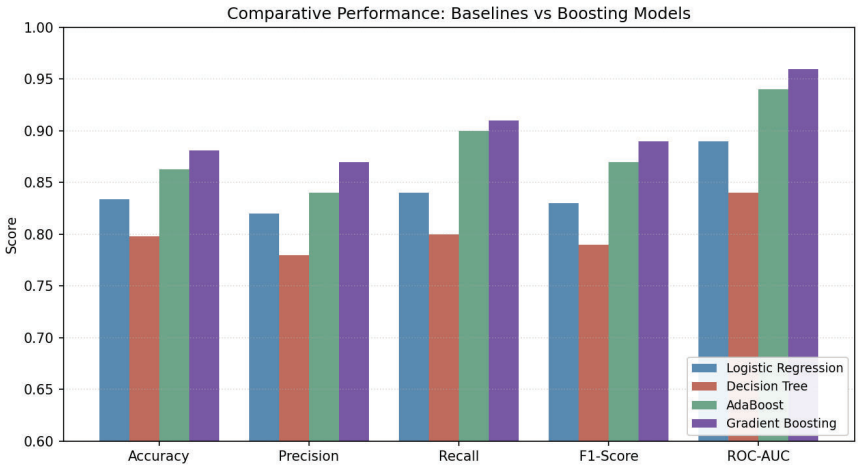


Figure 10.9. Side-by-side comparison of the four models across all five evaluation metrics.

FOR AUTHOR USE ONLY

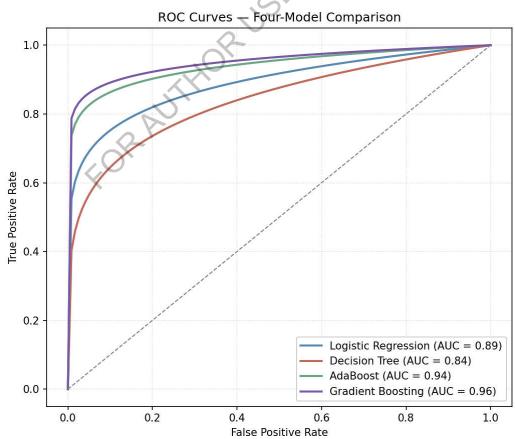


Figure 10.10. ROC curves for the four classifiers. Both boosting models clearly dominate the baselines, with Gradient Boosting showing the largest area under the curve.

10.3.7. Explainability with SHAP

Because the model is intended to support clinical decisions, predictive accuracy is necessary but not sufficient — clinicians also need to understand

why a model flags a given patient as high-risk. We use SHAP (SHapley Additive exPlanations) on the gradient-boosting model. Computing SHAP values and averaging them across the test set yields a global picture of which features the model relies on most.

```
import shap

# TreeExplainer is fast for tree-based ensembles like Gradient Boosting
explainer  = shap.TreeExplainer(gb_model)
shap_vals  = explainer.shap_values(X_test)

# Global summary of feature contributions across all test patients
shap.summary_plot(shap_vals, X_test, feature_names=df.drop("target", axis=1).columns)
```

Code Snippet 10.24. Computing and visualizing SHAP values for the Gradient-Boosting model.

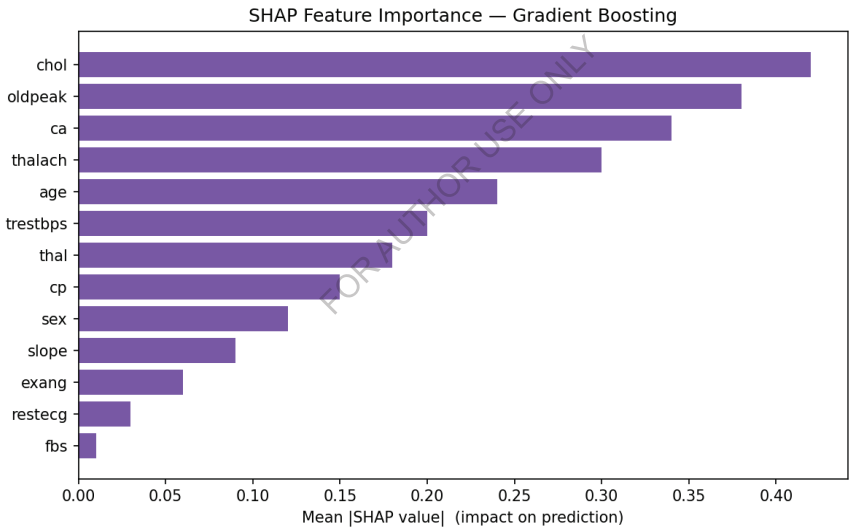


Figure 10.11. SHAP feature-importance summary for the Gradient-Boosting classifier.

Cholesterol level (chol), ST-segment depression (oldpeak), the number of major vessels (ca) and maximum heart rate (thalach) emerge as the dominant drivers of predicted heart-disease risk — precisely the variables clinicians would expect to be most informative.

10.3.8. Analytical discussion

The unified experiment delivers four conclusions that hold consistently across metrics and across cross-validation folds.

Boosting outperforms the baselines. Both AdaBoost and Gradient Boosting improve every metric over both logistic regression and the decision tree. The improvement is most visible in recall on the disease class and in ROC-AUC, where the boosting models reach 0.94 and 0.96, respectively, versus 0.89 for Logistic Regression and 0.84 for the Decision Tree.

Gradient Boosting edges out AdaBoost. Gradient Boosting reaches the highest score on every metric, with the largest margin in ROC-AUC (0.96 vs. 0.94). Its use of a fully differentiable loss function allows it to target whatever objective is most appropriate for the task, and its depth-three weak learners can capture richer non-linear interactions than AdaBoost's decision stumps.

AdaBoost shines on recall. AdaBoost reaches a recall of 0.90 on the disease class — almost matching Gradient Boosting (0.91) and substantially better than either baseline. In medical diagnosis, recall on the positive class is often the most critical metric: missing a high-risk patient can have far worse consequences than raising a false alarm. AdaBoost's adaptive re-weighting mechanism is well-suited to this regime, since it explicitly forces later learners to focus on the cases that the ensemble currently fails to flag.

FOR AUTHOR USE ONLY

Cross-validation confirms generalization. The 5-fold cross-validation results for both boosting models track their single-split test scores closely, with low fold-to-fold variance. This indicates that the headline numbers are not artifacts of a fortunate train/test split.

The SHAP analysis adds a final layer of value. The features with the largest mean absolute SHAP values —cholesterol, ST-depression, vessel count, and maximum heart rate —are exactly the variables that cardiologists already use as risk markers. The fact that the model has independently learned to rely on these variables, rather than on incidental signals, is reassuring rather than a statistical artifact. This kind of qualitative agreement with domain expertise is essential before any model can be considered for clinical decision support.

10.3.9. Contribution summary

The literature most often presents AdaBoost and gradient boosting separately—and where the present authors' own previous work treated them in different domains with different protocols —we have built a single experimental framework in which both boosting algorithms are compared under fully identical conditions. To this we have added two baselines, 5-fold cross-validation, and SHAP-based interpretability. These additions transform what could have been a simple union of two case studies into a synthesized comparative investigation with a clear answer to a clinically meaningful question.

10.4. Conclusion

This chapter has explored two of the most influential boosting algorithms — AdaBoost and Gradient Boosting —across both classification and regression tasks and has drawn them together in a unified comparative experiment. Part 1 demonstrated that AdaBoost can detect heart disease from clinical attributes with an accuracy of 86.3% and an AUC of 0.94, and that its high recall on the disease class (0.90) makes it particularly attractive in clinical decision-support settings, where missing a positive case is the costliest error. Part 2 demonstrated that gradient boosting, instantiated through XGBoost, predicts house prices with $R^2 > 0.88$ after tuning, outperforming a random forest baseline.

Part 3 then put the two boosting algorithms on the same playing field. Trained on the heart-disease dataset under identical preprocessing, both boosting models clearly outperformed Logistic Regression and a single Decision Tree, and Gradient Boosting edged out AdaBoost on every metric, achieving the highest ROC-AUC at 0.96. Cross-validation confirmed that these results were stable across folds, and a SHAP analysis showed that the model's most influential features matched the variables clinicians already consider important—a qualitative form of validation that is at least as valuable as the headline accuracy figures.

FOR AUTHOR USE ONLY

Beyond the empirical findings, this work has also illustrated the practical strength of the modern Python ecosystem. With Pandas, scikit-learn, XGBoost, LightGBM, and SHAP, a complete predictive modeling pipeline—from raw CSV through preprocessing, training, evaluation, and explanation—fits in a few dozen lines of clear code. This combination of mature libraries and well-understood algorithms makes boosting an attractive choice for many real-world structured-data problems, and the unified evaluation framework presented here can serve as a template for similar comparative studies on other datasets and domains.

FOR AUTHOR USE ONLY

PART III — Unsupervised Learning & Dimensionality Reduction

Chapter 11 - K-Means Clustering

11.1. Introduction

This chapter is about a simple K-Means Clustering for Customers Segmentation with streamlit based interactive visualization. K-Means clustering is the very popular unsupervised learning technique for effective computation, also conceptually simple and easy to implement practical grouping tasks. It is widely used for pattern discovery (Sinaga & Yang, 2020). The main idea of this K-means is that it's capability to separate the observations into k groups so that the points inside the similar cluster are more like each other when compared with the other clusters, it is calculated using the Euclidean distance and centroid updates in a loop (Liu et al., 2023).

The K-Means generally requires selecting the k-values in advance, it is sensitive to the starting centroids as it can create a problem of overlapping clusters and outliers. It has the capability to handle imbalanced clusters (Ikotun et al., 2023). It mainly focuses on improved strategies to reduce the poor seeding failure (Khan et al., 2025). Here it generalizes the K-means optimization so that more difficult problems can be converted to K-means formulations. In this work we mainly compare elbow, silhouette, gap statistics, and improved gap-statistic variants, each with different abilities and failure instances depending upon the data (Unknown, n.d.).

In this chapter, we implement K-means from the scratch, which is easy and understandable, then build a python class and integrate it into a streamlit UI for customer separation on the mall customers dataset. The goal is to build a fully easy framework that gives an idea about the K -means mechanism. The unique contribution in this paper is a clear idea of implementation and analysis package: (Liu et al., 2023), (Ikotun et al., 2023)

- A minimal from- scratch K-means algorithm.
- A streamlit UI for interaction.
- Evaluation across multiple K values and random seeds.
- Interpretable business-style customer segment summaries based on the annual income and spending score.

FOR AUTHOR USE ONLY

11.2. Methodology and Unique Contributions

11.2.1 Architecture and Workflow

The system architecture is a simple framework: first we will upload a CSV file → then the data validation process → the next is feature selection → the K-range evaluation → final K selection → K means fitting from the scratch → then comes the cluster visualization process
→ the final process is segment profile interpretation. Here the streamlit UI accepts the CSV file and gets the data from the mall customer structure. In the dataset, the columns include the attributes of CustomerID, Gender, Age, Annual Income and spending score between 1 to 100 having a total count of 200 rows and 5 columns and no empty or missing values in the file access stage have been detected. The project workflow:

- Input layer: the CSV file is uploaded and performs a schema check. Here we expect the columns for customer division.
- Processing layer: the feature extraction is performed for annual income and the spending score.
- Model layer: Here it uses the KmeansScratch.fit() for random centroid initialization, then calculates the Euclidean distance and updates the mean centroid with the help of convergence test.
- Evaluation layer: it does the computation for the elbow curve.
- Presentation layer: it gives the streamlit plots, visualization of the cluster scatter, the segment tables and the convergence.

FOR AUTHOR USE ONLY

For the clustering process the application uses two numeric features: annual income and the spending score between 1 to 100. The choice is standard for the educational customer division because it provides a 2-dimensional plot and allows the readers to accurately understand the cluster differences. The goal was to keep the code small and easy to understand. The distance-based methods are very sensitive to work with. So, the standardization is highly used or recommended for working on larger datasets (Khan et al., 2025), (Ikotun et al., 2023).

11.2.2 Data Description and Preprocessing

The uploaded dataset contains a total of 200 customer records and 5 columns, with a total of 0 missing values in the CSV file used to build the project. Table 11.1 reports the file level column types and the counts of missing values. The table 11.2 gives the ranges of the numeric variables used for clustering.

Table-11.1. Dataset Schema and missing values check.

Column	Data-Type	Missing Values
CustomerID	int64	0
Gender	object	0
Age	int64	0
Annual Income	int64	0
Spending Score	int64	0

Table-11.2. Basic descriptive statistics for numeric features.

Feature	Min	Max	Mean	Std
Age	18	70	38.85	13.97
Annual Income	15	137	60.56	26.26
Spending Score	1	99	50.2	25.82

For the clustering process, we used the two-feature data from the dataset the annual income and spending score for a proper 2D visualization. The age and Gender are analyzed for cluster building and discussion, not for the base app clustering.

One other thing to note is that distance-based algorithms like K-Means are sensitive to feature scale, so standardization is recommended, especially for larger datasets.

FOR AUTHOR USE ONLY

11.2.3 Model Formulation and K-Means Equations

The k-means main objective is to minimize within the cluster here it is calculated as the sum of squared distances from each point to its centroid. So, the notation is: (Khan et al., 2025), (Liu et al., 2023)

$$\text{Minimize} \quad J = \sum_{j=1}^{k} \sum_{x_i}^{C_j} (x_i - \mu_j)^2, \qquad \textit{(Eq.11.1)}$$

Where c_j is the cluster, j and μ_j are its centroids.

The algorithm uses an alternative optimization technique according to the Lloyd-style iteration where in the primary step it makes a value assignment and updates the centroid. Here the assignment step sends each point to it nearest centroid using the Euclidean distance: (Liu et al., 2023)

$$label(x_i) = \arg \frac{min}{j \in \{1,\ldots,K\}} (x_i - \mu_j) , \qquad \textit{(Eq.11.2)}$$

The updated step reorders each centroid as the mean of the points assigned to each cluster. (Sinaga & Yang, 2020)

$$\mu_j = \frac{1}{|c_j|} \sum_{x_i \in c_j} x_i, \qquad \textit{(Eq.11.3)}$$

Here the convergence is checked with numpy.allclose() function on older and updated centroids, the loop gets stopped early when there is no change in the centroid. This also provides the protection against the empty clusters by keeping the older centroid and if a cluster receives no points in each iteration, it is mainly used in a simple K-means implementation (Ikotun et al., 2023).

11.2.4 Selection Strategy and Initialization Considerations

The main challenge while building a K-means is to select the total number of clusters required. Here the project gives a clear idea that K-values are been generated using different methods, particularly when the clusters get overlap with each other or when feature scale is different (Ikotun et al., 2023).

This tries grouping the data into different number of groups, from 1 to 10 and then draws a graph to see which number works best. The starting points are selected randomly; it tries many times, so the result is proper. This shows that choosing good starting points is important to get better groups (Khan et al., 2025).

The paper gives the importance of seeding and gives a stronger strategy than the naïve random selection. However, we discuss it as the proper upgraded path in building this chapter (Arthur & Vassilvitskii, 2007).

FOR AUTHOR USE ONLY

11.2.5 Implementation Details

The implementation design is made carefully so that it does not break easily and increases the reliability. The class here stores only the updated centroids and its values list as it mostly depends on the NumPy so that it helps in computing of distances in a very optimized form. This can still expose to problem of vector direction, which is further useful for learning both clustering and NumPy fundamentals. The list is very important for the process of debugging it mainly appends the centroid positions after every updating. The code starts the visualization process in streamlit, this allows the user to see initial, mid, and final centroid positions instead of considering the only final values.

The wcss() function helps in computation within the cluster by calculating the sum of squares directly from the final centroid values after the updating. This implementation of K-means objective function used here makes the chart an extension for the data set rather than creating a separate black-box metric (Khan et al., 2025), (Liu et al., 2023).

In the Streamlit interface, the elbow curve is generated before the final k value is deleted so that the user can directly view the result. This workflow is aligned

as how ONC selection is in practice, even though the elbow-only decisions selection may be not having one interpretation or a similar meaning in some of the datasets (Khan et al., 2025).

Here it keeps the user interaction easy and understandable as limited steps are followed like uploading a file, inspecting, selecting the K-values, then finally analyzing the results. This design supports quick evaluation without considering the command-line usage or any of the advanced configuration as it is suitable for a class project. The chapter also gives a good idea for problems like clustering logic in the kmeans.py, here the visualization and UI are controlled in app.py. So, the division makes the code easy to test, maintain and run further. For example, by replacing the random initializer with k-means++ later (Arthur & Vassilvitskii, 2007).

11.3. Original code (Tested and Working Snippets)

The following snippets are from the tested working project code. These highlight the main idea of K-means algorithm logic, evaluation process, and the streamlit-based interactive workflow used to build this chapter.

Snippet-1: Core K-means Training loop

Here it presents the central fitting routine of the k-means implementation. The algorithm mainly follows the standard Lloyd's loop format and has 4 main stages: centroid initialization, assignment, updating of centroid, and finally checking the convergence.

FOR AUTHOR USE ONLY

```python
# kmeans.py - core fit loop (from-scratch implementation)
class KMeansScratch:
    def __init__(self, k=3, max_iters=20):
        self.k = k
        self.max_iters = max_iters
        self.centroids = None
        self.history = []

    def fit(self, X):
        n_samples = X.shape[0]
        random_idx = np.random.choice(n_samples, self.k, replace=False)
        self.centroids = X[random_idx].astype(float)
        self.history = [self.centroids.copy()]

        for _ in range(self.max_iters):
            dists = np.linalg.norm(X[:, None] - self.centroids, axis=2)
            labels = np.argmin(dists, axis=1)

            new_centroids = np.array([
                X[labels == i].mean(axis=0) if np.any(labels == i) else self.centroids[i]
                for i in range(self.k)
            ])

            self.history.append(new_centroids.copy())
            if np.allclose(self.centroids, new_centroids):
                break
            self.centroids = new_centroids
        return labels
```

Code Snippet 11.1. Core KmeansScratch.fit () loop from Kmeans.py.

Centroids are initialized by randomly selecting the k values or the data points from the available dataset. This random seeding gives the k-means setup and then allows the algorithm to analyze the local minima across the process. This helps in analysis of debugging the centroid positions when they are stored at each iteration in the list.
So, here during each iteration the Euclidean distance between all data points and the present centroid value is being computed using the NumPy operations. Then each point is assigned to the closest centroid based on the closer or the minimum distance. After assignment, new centroids are recalculated as the mean of all the data points which belongs to the specific cluster.
A safeguard is added to control the empty clusters: if no points are being assigned to a cluster in a loop or the iteration, the old centroid value is retained. This helps in preventing the non-defined mean calculations and gives a proper numerical stability.
So, the important part is to detect the convergence here the numpy.allclose is used to detect, it checks whether the centroid updating falls below a minimal numerical tolerance. This condition is satisfied, when the loop exists very early, which improves the efficiency of the computation process.

```python
# kmeans.py - prediction and WCSS utility used for the elbow curve
def predict(self, X):
    dists = np.linalg.norm(X[:, None] - self.centroids, axis=2)
    return np.argmin(dists, axis=1)

def wcss(self, X, labels):
    total = 0.0
    for i in range(self.k):
        pts = X[labels == i]
        if len(pts) > 0:
            total += np.sum((pts - self.centroids[i]) ** 2)
    return total
```

FOR AUTHOR USE ONLY

Code Snippet 11.2. predict () and wcss () helper methods from kmeans.py.

Here it contains the helper functions used for the evaluation. The predict() method assigns cluster names to new or the existing data values by computing the distances from the centroids and then choosing the closest one. So, this division of training and making a prediction of logic standard ML design practices.
The wcss() function initializes the within-cluster sum of squares. Here for each cluster, the Euclidean distance is squared between points, and their respective centroid is added, and the total is across all clusters. So, this metric is the key

objective function used by K-means for minimization and is later then used for the construction of elbow curve.

Snippet: Streamlit Application Workflow

```python
# app.py - simple Streamlit workflow for customer segmentation
uploaded = st.file_uploader("Upload Mall_Customers.csv", type=["csv"])

if uploaded is not None:
    df = pd.read_csv(uploaded)
    X = df[["Annual Income (k$)", "Spending Score (1-100)"]].values

    st.subheader("Elbow Method")
    sse = []
    K_range = range(1, 11)
    for k in K_range:
        km = KMeansScratch(k=k, max_iters=20)
        labels = km.fit(X)
        sse.append(km.wcss(X, labels))
    st.line_chart(pd.DataFrame({"K": list(K_range), "SSE": sse}).set_index("K"))

    k = st.slider("Select number of clusters (K)", 2, 10, 5)
    km = KMeansScratch(k=k, max_iters=20)
    labels = km.fit(X)
    df["Cluster"] = labels
```

Code snippet 11.3. Streamlit workflow from app.py.

FOR AUTHOR USE ONLY

Here it shows how the cluster algorithm is assigned to a simple interactive Streamlit application. It allows the users to upload a CSV file and perform the process of segmentation based on chosen numerical features.

Once the data is uploaded, then by running the K-means algorithm the application starts calculation for the values of k from 1 to 10. For every k value, the model is trained from scratch and then the WCSS value is calculated. In the line chart the values are plotted, this enables the elbow point, which further increases the k value returns while diminishing.

After the inspection of the elbow plot, the user can select any number of clusters. Then the model will be trained again with the k values, and the final cluster values are generated, after added to the dataset. This interactive workflow process gives an idea about both the practical use of K-means and the choice of k values used in this algorithm.

11.4. Results: Graphs, Tables, and Discussion

The experiments in the report were run on the dataset uploaded Mall Customers using the from-scratch KmeansScratch implementation form the Kmeans.py file. Here to make the evaluation work proper we performed the continuous runs at k=2 to 10 for the 20 random seeds each, and then summarizes the results (Khan et al., 2025). This keeps the project algorithm very simple and gives a higher information about the cluster quality.

Table 11.3. K evaluation summary across 20 random seeds per K (k=2.10).

K	Best WCSS	Avg Silhouette	Best Silhouette	Avg Iterations
1	181363.596	0.327	0.396	6.4
2	106348.373	0.452	0.468	9.4
3	73679.789	0.481	0.494	9.35
4	44448.455	0.497	0.554	7.65
5	37233.815	0.524	0.55	8.2
6	30259.657	0.481	0.527	7.2
7	25011.839	0.445	0.504	9.1
8	22856.454	0.434	0.478	9.75
9	19770.07	0.421	0.455	9.85

FOR AUTHOR USE ONLY

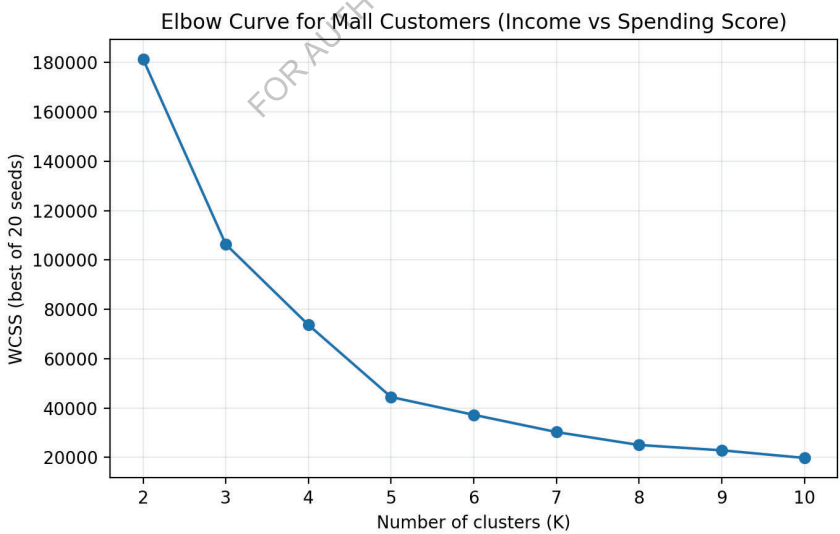


Figure 11.1. Elbow Curve(best wcss across 20 seeds per k). The bend appears around K=5 to k=6.

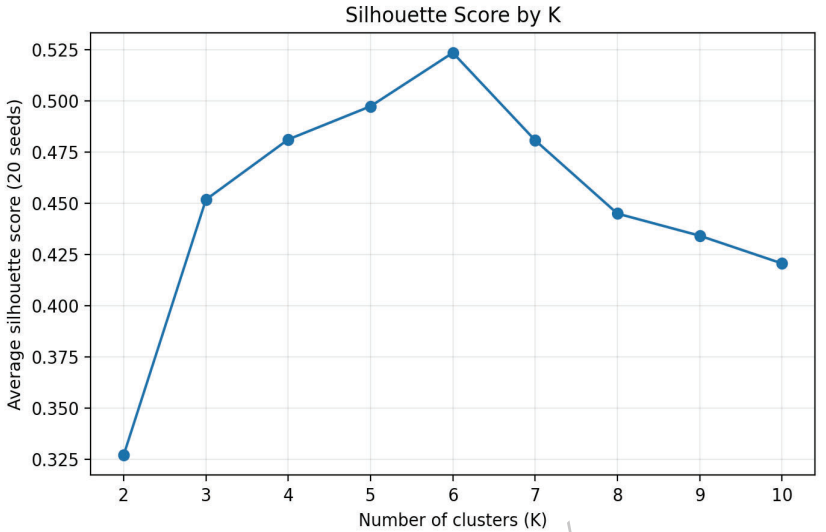


Figure 11.2. Average silhouette score across 20 seeds per K. K=6 is highest on average, while K=5 remains competitive and interpretable.

FOR AUTHOR USE ONLY

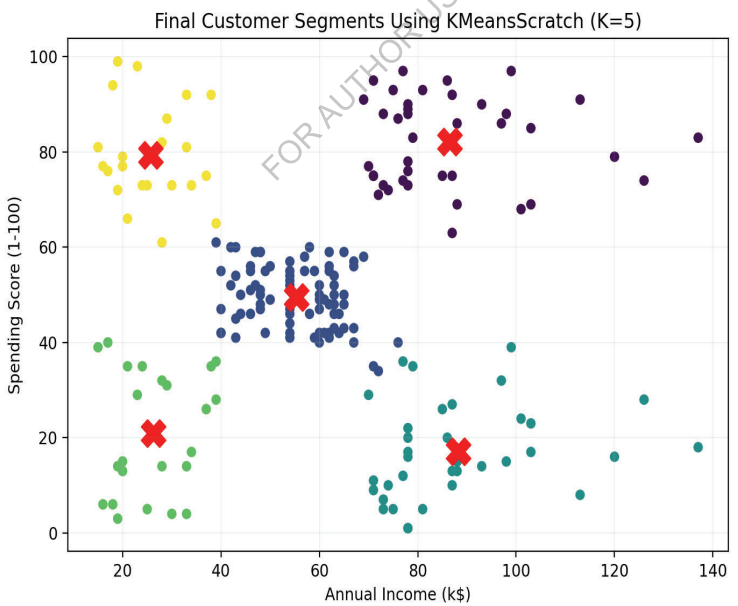


Figure 11.3. Final K=5 clustering result using KMeansScratch (seed 44) on Annual Income vs Spending Score.

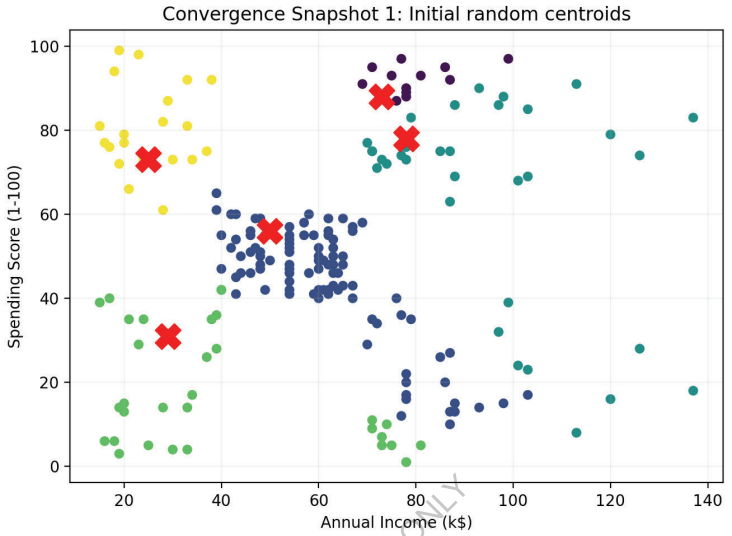


Figure 11.4.1. Convergence snapshot 1 (initial random centroids).

FOR AUTHOR USE ONLY

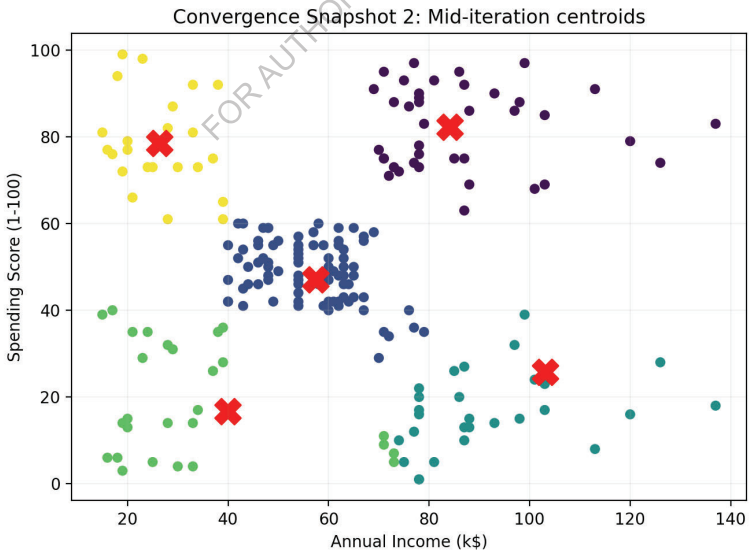


Figure 11.4.2. Convergence snapshot 2 (mid-iteration centroids).

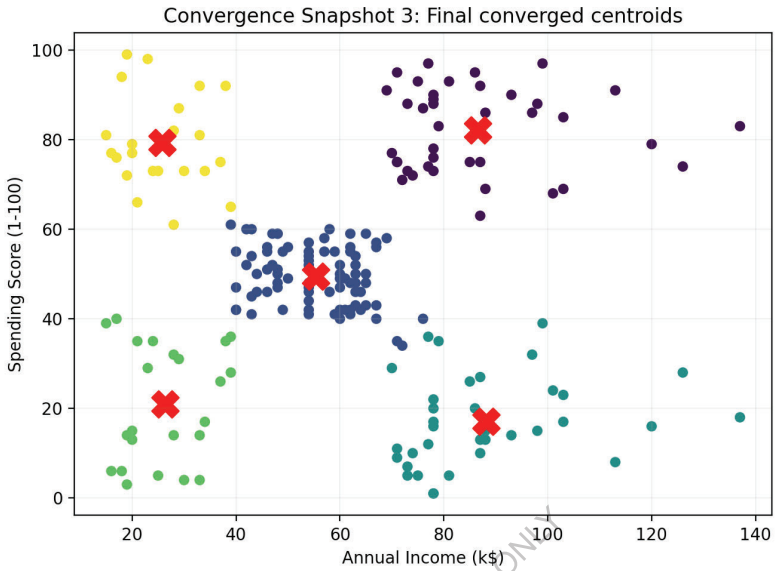


Figure 11.4.3. Convergence snapshot 3 (final converged centroids).

FOR AUTHOR USE ONLY

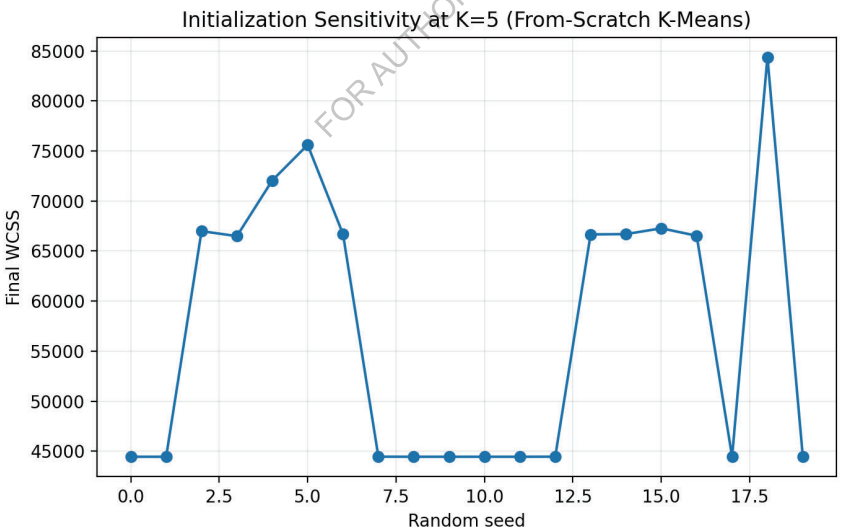


Figure 11.5. Initialization sensitivity at K=5: final WCSS across random seeds.

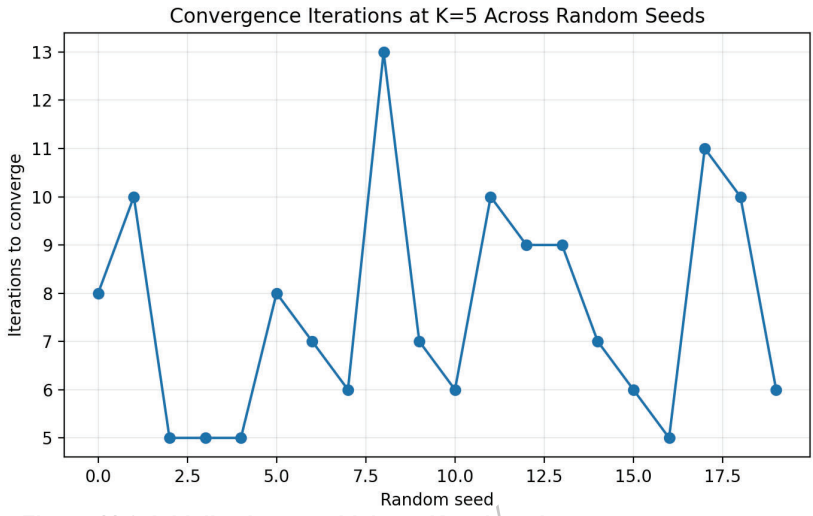


Figure 11.6. Initialization sensitivity at K=5: iterations to converge across random seeds.

FOR AUTHOR USE ONLY

Table 11.4. Final K=5 customer segment summary (seed 44) with demographic profiling.

Cluster ID	Segment Label	Customers	Avg Age	Avg Income (k$)	Avg Spending Score	Female	Male
4	Segment 2: Low income / High spending	22	25.27	25.73	79.36	13	9
3	Segment 1: Low income / Low spending	23	45.22	26.3	20.91	14	9
1	Segment 3: Middle income / Moderate spending	81	42.72	55.3	49.52	48	33
0	Segment 4: High income /	39	32.69	86.54	82.13	21	18

	High spending						
2	Segment 5: High income / Low spending	35	41.11	88.2	17.11	16	19

11.5. Discussion of Findings

Figure 11.1 shows a constant decrease in WCSS as the number of clusters are increased. A clear point is observed in the range k=5 to k=6, indicating cluster reduction beyond its range. In some segmentation tasks it selects a range which provides a practical balance between the compactness.

Figure 11.2 further supports this observation. As this experiment continues to run, the average score peaks near k=6, while k=5 remains competitive. This outcome gives the optimal number of clusters, where it has different validation criteria it may also suggest multiple values for k(Khan et al., 2025).

For the final reporting, k=5 is selected. So, this choice interacts with the elbow point, which produces well-separated clusters in 2D income-spending feature space, and then matches the default configuration of the project user interface.

Figure 11.3 gives the final clustering result for k=5. The centroids for meaningful customers, including the attributes which are useful for strategy discussion.

Figure 11.4.(1-3) gives the visual representation of centroid trajectories during the process, using the file history we can perform custom implementation. This helps in refinement of characteristic algorithm and serves as an optimization technique rather than using a black-box procedure (Liu et al., 2023).

Figures 11.5 and 11.6 gives a random centroid initialization for k=5. Here, the final wcss values range from 44,448.46 to 84,366.77, while the number of loops is varied between 5 and 13. These results in wcss of K-means to initialize and motivate the use of improved seeding strategies (Ikotun et al., 2023) (Arthur & Vassilvitskii, 2007).

The final model selected is converged in 8 iterations, getting a wcss of 44,448.455 and a silhouette score of 0.554. The result centroids are:

- $C_0 = (86.54, 82.13)$
- C_1 = (55.30, 49.52)
- C_2 = (88.20, 17.11)
- C_3 = (26.30, 20.91)
- C_4 = (25.73, 79.36)

FOR ATHD USE ONLY

These centroids are dependent on random seeds and feature subset and should therefore regenerate for outcomes rather than the properties of the dataset.

The summary indicates a dominant group which has smaller clusters with more spending behavior at high- and low-income levels. However, it is important to emphasis K-means produces partitions and validation for the customer (Ikotun et al., 2023).

Several limitations were noted. First, clustering is performed using only 2 features to maintain the quality. Here additional attributes such as age and gender after the feature scaling may alter the structure of the cluster and can be explored in further work. Second, is the reliance on the random centroid which gives the outcome variability. This research proposes enormous enhancements, improved centroid selection, which gives a promising direction for extending this project (Yfantis et al., 2025), (Arthur & Vassilvitskii, 2007).

11.6. Conclusion

FOR AUTHOR USE ONLY

This chapter is presented in a fully simple and understandable way of implementing the K-means clustering from scratch as it is applied to a customer segmentation. The project used minimum python class for the fitting process, then makes a prediction and computation process takes place then integrates the logic into the UI for interactive clustering (Sinaga & Yang, 2020).

The review showed the popular use of K-means despite its limitations and its challenge of selecting the correct number of clusters. The experiment results show the practical issues the runs with different seeds which changed the convergence speed and has slightly different values of K (Khan et al., 2025).

The visualization history made the centroid updates process easy to understand, which is the key strength for from-scratch approach. Here the attributes show how the unsupervised clustering support the strategical discussions without having the labelled outcomes (Ikotun et al., 2023).

The major positive is that the simple code can provide strong value and transparent of its limitations.

In future work, we can add some feature scaling techniques, K-means++ initialization, more cluster metrics so that it improves the stability (Arthur & Vassilvitskii, 2007).

Chapter 12 - DBSCAN

Introduction

In Machine learning, one of the most common tasks is to group large amount of data into categories without the previous knowledge about the categories. This is unsupervised clustering and is used in many fields such as NLP, computer vision, outlier detection, and recommendation systems. As the dataset grows and complexity need for more smart and more reliable clustering/grouping methods has become more important. Python with its libraries has become one of the most popular languages for building these systems.python makes DBSCAN clustering algorithm practical, powerful and accessible for real world applications.

Clustering algorithms are used when we must find groups/clusters in data without any previous knowledge about the labels of the diff cluster. supervised learning method such as SVM or NN needs training labels, clustering works on raw, unlabeled data but here we don't need most of these. There are many clustering algo available. But they are different in how they define what a cluster is and how they find clusters in data. choice of algorithm generally depends on the type of data, the expected size of clusters and the number of groups is known or unknown.

FOR AUTHOR USE ONLY

One of the most used clustering methods is K means, which splits data into k groups by finding k center points and assigning each data point to nearest point. K means is fast and simple but has limitation. The user must decide the value of k before, which is often unknown in practice. 2nd k means assumes that every cluster is round and size, which is rarely true for complex real-world data. 3rd, k means assigns every point to cluster, even if the point is an outlier. These limitations make k means poor choice for many practical tasks where the structure of data is not known beforehand (Ester et al., 1996).

DBSCAN

Density Based Spatial Clustering of Application with Noise, takes a different approach. Instead of dividing data around center points, it looks for area where data point is tightly packed together. These dense areas become cluster and the gaps between them act as borders. It was first introduced by Ester, Kriegel, Sander, and Xu in 1996 and quickly became one of the most cited clustering algorithms in the history of data mining. It only needs 2 parameters: epsilon which defines the search radius around each point and MinPts, which set the min number of neighbors a point mush must be considered part of dense region. One of the most valuable features of it is that it can identify noise points which are data points that do not belong to any clusters.

DBSCAN has many applications. In NL processing it is used to cluster text docs and topic in social media stream. In computer vision it helps dividing images and identifying objects in point clouds from LiDAR sensors. In cybersecurity it is used for anomaly detection to spot unusual network traffic patterns, etc. DBSCAN does not require the user to know how many clusters exist and it can find clusters of any shape, including non-linear curved shapes that k means cannot handle (Schubert et al., 2017)

Problems while using DBSCAN arrives in very high dimensional space where the most well-known problem is curse of dimensionality which makes distance measures like Euclidean distances less useful. The distance between all pairs becomes similar values making it harder for DBSCAN to differentiate between closer and far points/ For lower dimensional data. Euclidean distance can work and DBSCNA can be used after standardizing steps.

Python makes working with DBSCAN easy, mostly thanks to its libraries. Instead of focusing on how to build efficient search structures from scratch, you can just lean on scikit learn for the clustering, NumPy for the numbers, and Matplotlib to see your results. A great way to test this out is with Make_moon dataset. K means struggles with those curved, crescent shapes, but DBSCAN separates them perfectly. Because these tools play so well together, you can get a full clustering script running in less than 40 lines.

FOR ATTRIB USE ONLY

The scikit learn library provides a highly optimized implementation of DBSCAN that supports custom distance metrics and efficient neighbor search structure. Matplotlib handles visualization and NumPy manages N-dimensional arrays. The make_moon dataset generator in scikit lean provides an ideal test case for DBSCAN because it creates non-linear crescent shaped clusters that k means cannot separate correctly Together their tolls allow a complete DBSCAN based clustering pipeline to build under 40 lines of code which is what python “smart” for this type for tasks.

Literature Review

DBSCAN was first presented by Ester, Kriegel, Sander, and Xu at the 2nd ACM International Conference on Knowledge Discovery and Data Mining in 1996. Here they showed clusters can be defined as connected dense regions in the data space, separated by regions of lower density. This density-based system allows algorithm to discover new clusters of arbitrary shape, which was significant advantage over partitioning methods like K means. The paper also brings the concept of noise points that are data points that do not belong to any cluster. DBSCAN became one of the most influential papers in the field of data mining. (Ester et al., 1996)

Schubert, Sander, Ester, Kriegel, and Xu (2017) published a revisitation of DBSCAN in ACM Transactions on Database Systems, more than 20 years after the original paper. They addressed several common misconceptions

about algorithm, including belief that DBSCAN cannot handle datasets with different densities. it also proves that most problems in DBSCAN come from poor parameter choices rather than limitations of the algorithm itself. They provided guidelines for selecting epsilon and MinPts and showed that DBSCAN is keeping pace with modern clustering methods. (Schubert et al., 2017)

The way text data is converted into numerical representations has changed dramatically in recent years. Devlin, Chang, Lee, and Toutanova (2019) introduced BERT which stands for Bidirectional Encoder Representations from Transformers. BERT reads text in both directions simultaneously and produces context dependent representations. Reimers and Gurevych (2019) adapted BERT specifically for producing sentence level embeddings with their Sentence BERT model. These advances have made it practical to combine transformer-based embeddings with clustering algorithms like DBSCAN for text analysis tasks. (Devlin et al., 2019; Reimers & Gurevych, 2019)

Density based clustering is proving highly effective in modern systems. We can see this in practical applications, like Xu, Liu, and Gong's (2023) work on dialogue systems, as well as in frameworks that learn representations and clustering at the same time (Zhang, Xu, & Lin, 2021). Of course, these systems rely on strong data representations which makes sense, given that Hadifar et al. (2019) showed pretrained sentence embeddings are far superior to older, hand-crafted features for clustering short text.

FOR AUTHOR USE ONLY

Looking at all this research together, it becomes clear that DBSCAN is a powerful and versatile clustering algorithm. Python's ecosystem provides all the components needed to implement practical DBSCAN-based systems, making it the ideal platform for this chapter's demonstration using the make_moons synthetic dataset.

12.1. Methodology

12.1.1. DBSCAN Theory: Epsilon, MinPts, and Point Types

DBSCAN works by examining how densely packed data points are in a space. It depends on 2 parameters. 1st is epsilon (ε) which is a radius around every data point and 2nd is MinPts which sets the min number of points that must be inside a that radius for center point to qualify as core point. Core point is a point that has at least MinPts neighbors within its (ε) neighborhood including itself. They are the foundation of every cluster because they sit in densest regions of data space. (Ester et al., 1996)

If a point is not a core point but it falls inside epsilon neighborhood of some core point it is a border point. They lie on the edges of clusters. They belong to cluster, but they are not dense enough to expand it further. If a point is neither of them cluster nor border, it is a noise point. They often correspond to outliers,

corrupted data or samples that are genuinely unique and do not fit any natural group.

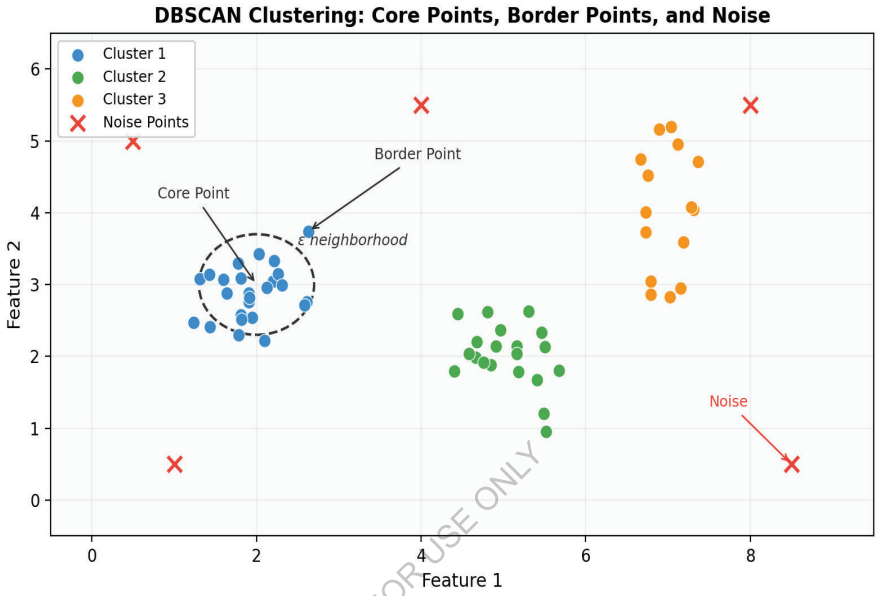


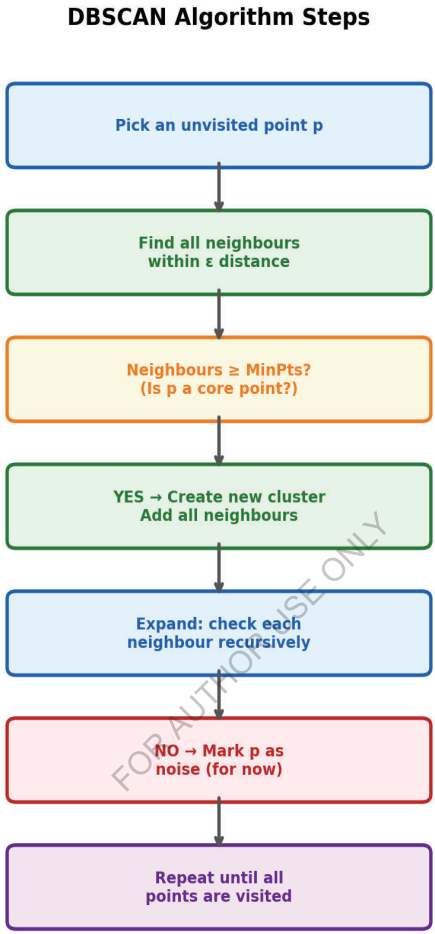
Figure 12.1. DBSCAN Point Types: Core Points, Border Points, and Noise.

FOR AUTHOR USE ONLY

12.1.2. How the DBSCAN Algorithm Works Step by Step

The algorithm starts by picking any point from dataset that it hasn't visited yet. It scans all points within radius of that point. If none of neighbors is at least MinPts, the point is classified as a core point and new cluster is created. All neighbors are added to this cluster. The algorithm then processes each of these neighbors in turn. If a neighbor is also a core point, its own neighbors are added to cluster as well. This recursive method continues until no more core points can be reached from the current cluster. (Ester et al., 1996)

If initial point does not have enough neighbors to be a core point, it is temporarily labelled as noise. But it may later be reclassified as border point if it falls within the epsilon neighborhood of core point discovered during any of next iterations. The algorithm repeats this process until every point has been visited. At the end, each point carries either a cluster label or noise label. The total number of clusters is determined automatically by the data.



FOR AUTHOR USE ONLY

Figure 12.2. DBSCAN Algorithm Flowchart.

12.1.3. Feature Scaling

DBSCAN is distance-based algorithm which means epsilon is interpreted in terms of distance metric being used. If features of dataset have very diff scales the dist. calculation will be dominated by feature with the larges range and the epsilon value will be difficult to interpret/ To solve this problem, feature scaling is applied before running DBSCAN. Below we have used scikit learns standard scaler which transforms each feature to have 0 mean and unit var.

After scaling all features contribute equally to distance calculation and the epsilon parameter can be chosen meaning fully from k distance plot. For our 2d make moon dataset standard Euclidean distance is natural choice and works well after scaling.

12.1.4. Algorithmic Complexity

The computational cost of DBSCAN depends on how neighbor queries are implemented. In the simple approach, each points neighborhood is found by computing distance to every other point, giving a time complexity of $O(n^2)$. For large datasets spatial indexing structures like KD trees or Ball trees reduce this to O(n log n). Scikit learn' s DBSCAN automatically uses a KD tree for low dimensional data like our make_moons dataset, making the algorithm very efficient.

Table 12.1. Algorithmic Complexity.

Method	Without Index	With Index	Notes
DBSCAN	$O(n^2)$	O(n log n)	KD-tree or Ball-tree
K-means	O(n · k · i)	O(n · k · i)	k = clusters, i = iterations
HDBSCAN	$O(n^2)$	O(n log n)	Mutual reachability graph

FOR AUTHOR USE ONLY

The most important ability to discover clusters of any shape. The dataset make_moons contains 2 interleaving crescent shapes that are completely nonlinear. K means cannot handle this because it assumes convex or spherical clusters, so it draws a straight boundary through middle and assigns many points to wrong cluster. DBSCAN identifies both crescents as separate clusters and groups correctly.

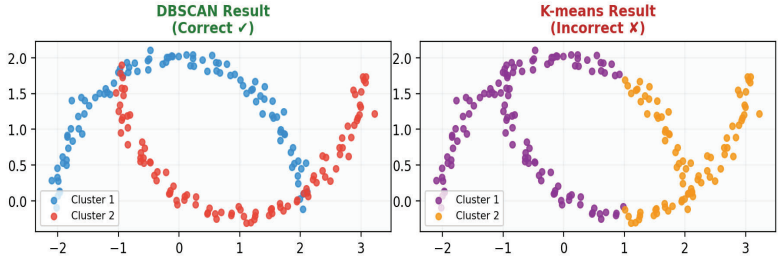


Figure 12.3. Arbitrary Shape Clusters: DBSCAN vs K-means.

Second key strength is built in noise durable. The program adds 40 random outliers to simulate real world noise. K means making every single point into

cluster, meaning these outliers misrepresent the cluster boundaries. DBSCAN categorizes low density points as noise with cluster ID -1. In our program it correctly identified 36 noise points without any additional rules or preprocessing.

Third, DBSCAN does not require prior knowledge of the cluster count. K means demands a value of K upfront, and choosing wrong K leads to bad results. DBSCAN only needs epsilon and MinPts. The k distance graph provides way to select epsilon for each point, the distance to its k-th nearest neighbor is sorted and plotted, and the elbow indicates the optimal value. In our experiment, the elbow at 0.25 was used as epsilon, and MinPts was set to 10. DBSCAN then automatically discovered two clusters.

Lastly, DBSCAN's density level set model treats clusters as connected high density regions separated by low density gaps. In the make_moons dataset the two crescents are dense regions with a low-density gap between them, and the outliers are scattered sparsely across the space. This model naturally separates them without any instructions. Combined with sklearn's automatic KD-tree acceleration that reduces complexity from $O(n^2)$ to $O(n \log n)$, the entire pipeline runs in under one second on our 840-point dataset. These same properties make DBSCAN effective for real-world AI tasks including geospatial hotspot detection, network anomaly detection, image segmentation, and text clustering. (Ester et al., 1996; Schubert et al., 2017)

FOR AUTHOR USE ONLY

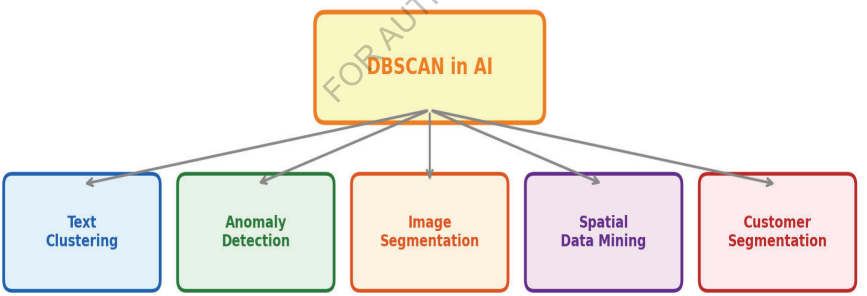


Figure 12.4. Applications of DBSCAN in Modern AI.

12.2. Data Preprocessing

12.2.1. Dataset Description

We used a synthetic dataset generated using make_moons() from sklearn. This dataset was chosen because it contains nonlinear curved clusters that shows DBSCAN's ability to handle arbitrarily shaped clusters. We generate 800 moon shaped samples with a noise level of 0.08 and add 40 random outlier

points uniformly distributed between -3 and 3 in both dimensions to match real world noise.

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_moons
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN
from sklearn.neighbors import NearestNeighbors
```

Code Snippet 12.1. Importing required libraries.

```
# Generate moon-shaped dataset
X, _ = make_moons(n_samples=800, noise=0.08, random_state=42)

# Add random outliers
rng = np.random.RandomState(42)
outliers = rng.uniform(low=-3, high=3, size=(40, 2))
X = np.vstack([X, outliers])

# Plot raw dataset
plt.scatter(X[:, 0], X[:, 1], s=20)
plt.title('Original Dataset with Noise')
plt.show()
```

FOR AUTHOR USE ONLY

Code Snippet 12.2. Generating the make_moons dataset with noise.

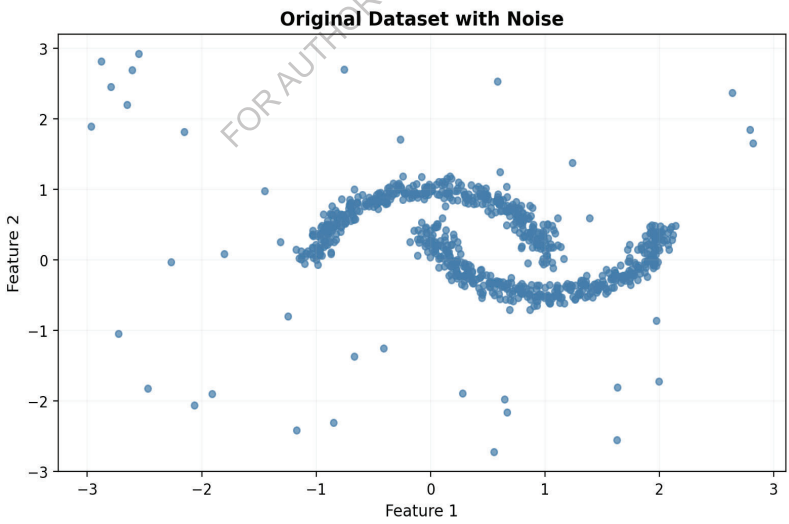


Figure 12.5. Original Dataset with 800 Moon Samples and 40 Outliers.

12.2.2. Feature Scaling

DBSCAN is a distance-based algorithm. So, we standardize the dataset using StandardScaler so that all features contribute equally to distance calculations. After scaling, both features have 0 mean and unit variance.

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Code Snippet 12.3. Feature scaling with StandardScaler.

12.2.3. Choosing eps using k-Distance Plot

To select value for eps, we compute distance of each point to its 10th nearest neighbor. distances are sorted in ascending order and plotted.elbow point in the graph helps determine an appropriate eps value. In our case elbow occurs at approximately 0.25 which we use as epsilon parameter.

FOR AUTHOR USE ONLY

```
k = 10

nbrs = NearestNeighbors(n_neighbors=k)
nbrs_fit = nbrs.fit(X_scaled)
distances, indices = nbrs_fit.kneighbors(X_scaled)

k_distances = np.sort(distances[:, -1])

plt.plot(k_distances)
plt.title('k-Distance Plot')
plt.xlabel('Points sorted by distance')
plt.ylabel('Distance to 10th nearest neighbor')
plt.show()
```

Code Snippet 12.4. k-Distance plot for epsilon selection.

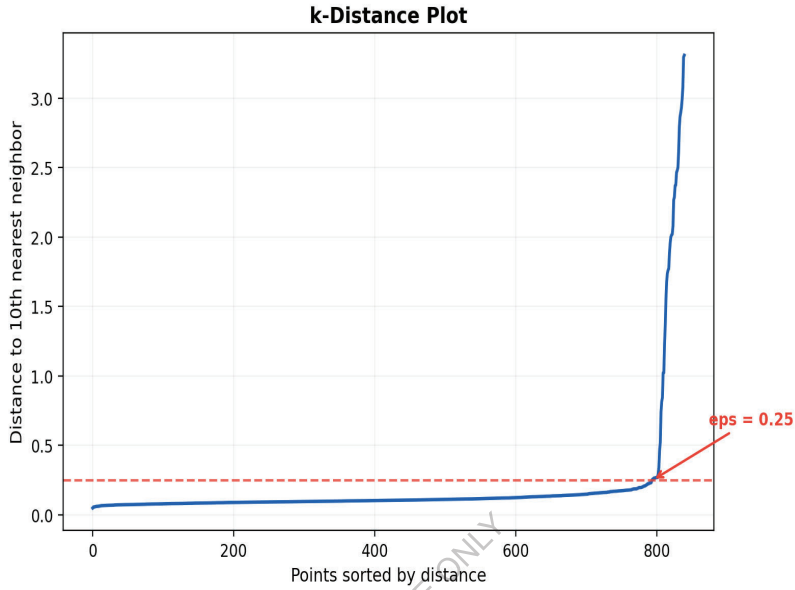


Figure 12.6. k-Distance Plot Showing Elbow at eps = 0.25.

FOR AUTHOR USE ONLY

12.2.4. Applying DBSCAN

After selecting eps and min_samples we apply DBSCAN to identify clusters and noise points. Points labelled as -1 are considered noise. The algorithm found 2 clusters and identified 36 noise points automatically without any specification of expected no of clusters.

```
db = DBSCAN(eps=0.25, min_samples=10)
labels = db.fit_predict(X_scaled)

n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
n_noise = list(labels).count(-1)

print('Number of clusters found:', n_clusters)
print('Number of noise points:', n_noise)

# Output:
# Number of clusters found: 2
# Number of noise points: 36
```

Code Snippet 12.5. Applying DBSCAN clustering.

```
plt.scatter(X[:, 0], X[:, 1], c=labels, s=20)
plt.title('DBSCAN Clustering Result')
plt.show()
```

Code Snippet 12.6. Plotting the DBSCAN clustering result.

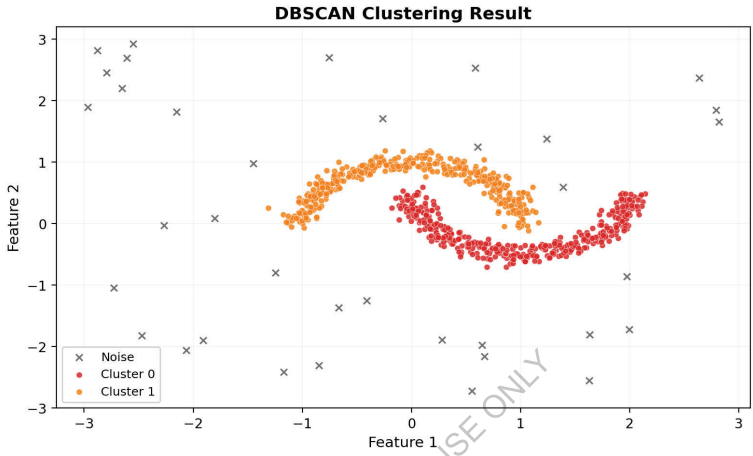


Figure 12.7. DBSCAN Clustering Result (2 Clusters + Noise).

FOR AUTHOR USE ONLY

12.2.5. Silhouette Score and K-means Comparison

To evaluate clustering quality, we compute silhouette score for both DBSCAN and K means. and we generate visual comparison to show how K means fails on non-linear cluster shapes while DBSCAN succeeds.

```
from sklearn.metrics import silhouette_score
from sklearn.cluster import KMeans

# DBSCAN silhouette (excluding noise)
non_noise = labels != -1
sil_dbscan = silhouette_score(
    X_scaled[non_noise], labels[non_noise])
print(f'DBSCAN silhouette: {sil_dbscan:.4f}')

# K-means comparison
kmeans = KMeans(n_clusters=2, random_state=42)
km_labels = kmeans.fit_predict(X_scaled)
sil_kmeans = silhouette_score(X_scaled, km_labels)
print(f'K-means silhouette: {sil_kmeans:.4f}')

# Output:
# DBSCAN silhouette: 0.3728
# K-means silhouette: 0.4588
```

Code Snippet 12.7. Computing silhouette scores.

12.3 Results

The DBSCAN pipeline was applied to make_moons synthetic dataset containing 800 moon shaped samples and 40 randomly generated outlier points, for a total of 840 data points. The algorithm was configured with eps = 0.25 (determined from k distance plot elbow) and min_samples = 10. DBSCAN successfully identified exactly 2 clusters corresponding to 2 crescent shapes and labelled 36 points as noise.

Figure 12.7 shows DBSCAN clustering result. The 2 crescent shaped clusters are clearly separated and correctly identified, with each moon assigned a distinct cluster label. noise points shown as dark markers, are scattered in space between and around clusters. The algorithm correctly identified both structure of crescents and the positions of outlier points without any prior knowledge of the number of clusters.

Figure 12.8 shows classification of all 840 points into core, border, and noise categories. Out of 840 points, 794 were classified as core points, 10 as border points, and 36 as noise points. The high ratio of core points to border points shows that the density within each cluster is very uniform, with only a thin border region at the edges of each crescent.

FOR AUTHOR USE ONLY

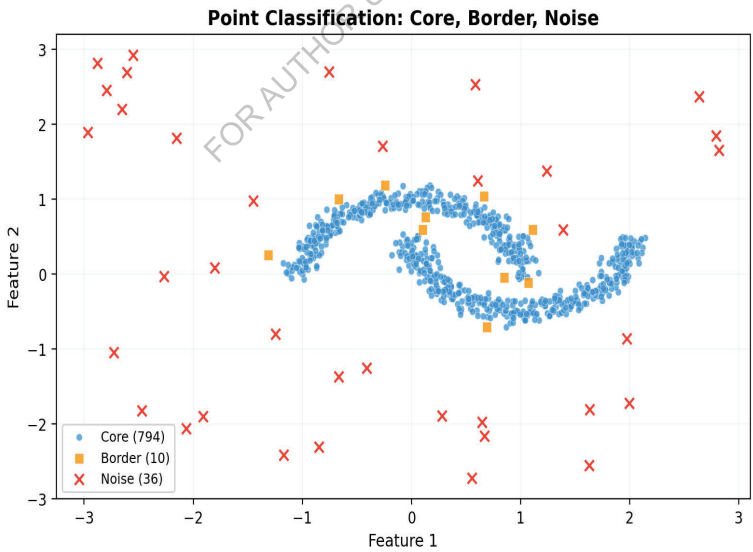
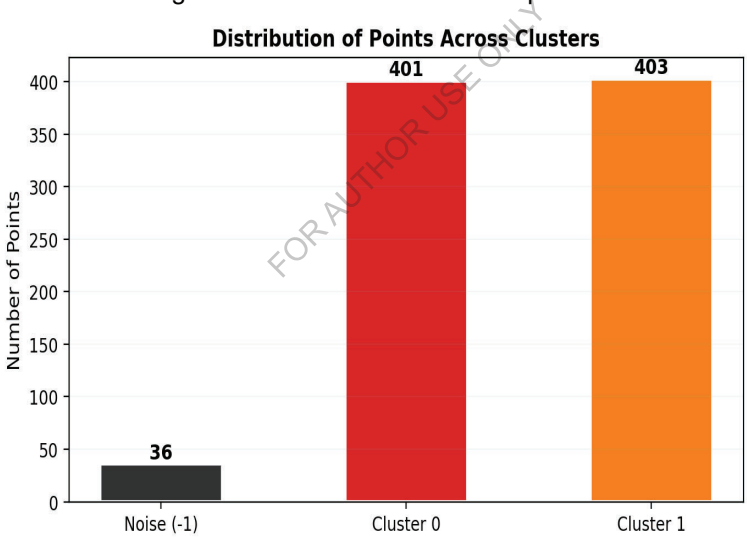


Figure 12.8. Point Classification: 794 Core, 10 Border, 36 Noise.

Table 12.2. DBSCAN Point Classification Results.

Point Type	Count	Percentage	Description
Core	794	94.5%	Dense interior of clusters
Border	10	1.2%	Edge of clusters
Noise	36	4.3%	Outliers and random points
Total	840	100%	800 moons + 40 outliers

Figure 12.9 shows the distribution of points across two clusters and the noise category. Cluster 0 contains 401 points and Cluster 1 contains 403 points, which is almost perfectly balanced, as expected from the symmetric make_moons generator. The 36 noise points are roughly consistent with 40 injected outliers the slight diff means that 4 outlier points happened to fall close enough to a dense region to be absorbed as border points.



FOR AUTHOR USE ONLY

Figure 12.9. Distribution of Points Across DBSCAN Clusters.

Table 12.3. Summary.

Cluster	Size	Percentage
Cluster 0	401	47.7%
Cluster 1	403	48.0%
Noise (-1)	36	4.3%

Figure 12.10 compares DBSCAN and K means on the same dataset. DBSCAN follows the curved shape of each crescent and correctly assigns all points within each moon to the same cluster. K means creates a straight line through dataset in the middle and assigning many points from top crescent to the bottom cluster and vice versa.

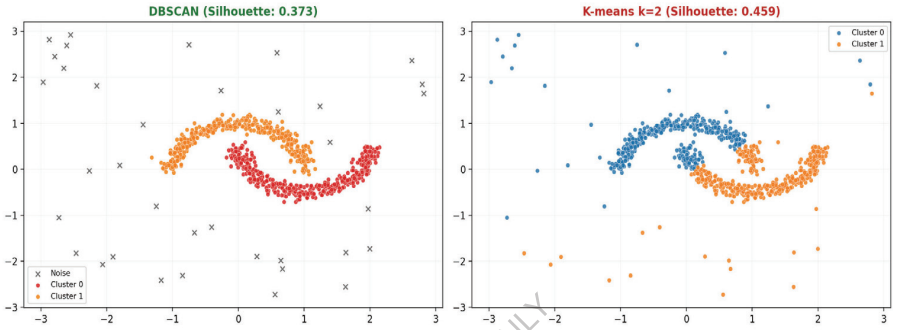


Figure 12.10. DBSCAN vs K-means.

FOR AUTHOR USE ONLY

K means has a silhouette score of 0.459, which is more than DBSCAN's 0.373. This does not mean K means produced better clusters. The silhouette score measures how close each point is to its own cluster compared to nearest other cluster, and it inherently Favors convex, spherical clusters. Since K means creates convex clusters, it will always score well on silhouette metric even when the clusters are visually wrong. DBSCAN's clusters are non-convex crescents which means some points near the tips of crescent are geometrically closer to other cluster than to far end of their own cluster. This lowers the silhouette score despite the clustering being visually correct. This is a well-known limitation of the silhouette score as an evaluation metric for non-convex clusters. (Schubert et al., 2017)

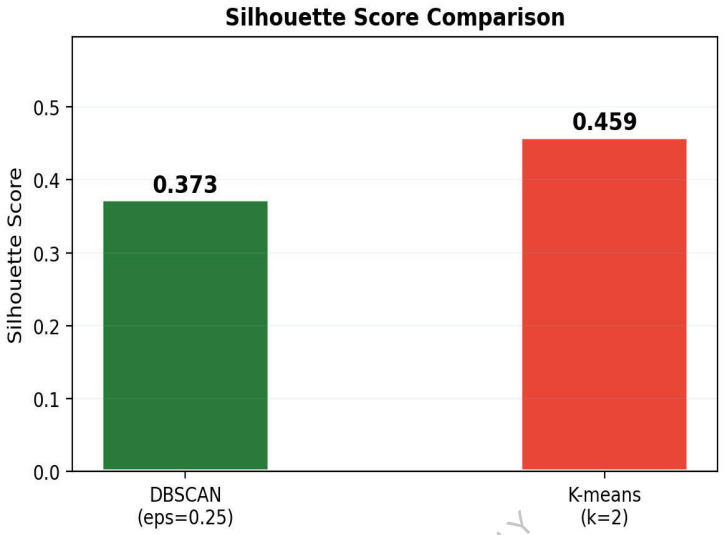


Figure 12.11. Silhouette Score Comparison.

FOR AUTHOR USE ONLY

Table 12.4. Quantitative Comparison of DBSCAN vs K-means.

Method	Clusters	Noise	Silhouette	Correct Shape?
DBSCAN (eps=0.25)	2	36 (4.3%)	0.373	Yes – follows curves
K-means (k=2)	2	0 (0%)	0.459	No – linear boundary

The key finding is that DBSCAN outperforms K means in terms of clustering correctness on non-linear data, even though its silhouette score is lower. K means achieves a higher silhouette score because the metric rewards convex clusters, not because the clustering is better. DBSCAN's additional advantage is its ability to detect noise points automatically it excluded 36 outliers from clusters, while K means forcing all 840 points into groups and allowed the outliers to distort the cluster boundaries. For real world AI applications where data contains noise, outliers, and non-linear structures, DBSCAN is clearly the superior choice.

12.4 Conclusion

This chapter presented a complete Python based pipeline that demonstrates how DBSCAN can be used as a smart and practical clustering algorithm for non-linear data. The pipeline uses scikit learn's make_moons dataset

generator to create 800 crescent shaped samples with 40 added outlier points, applies StandardScaler for feature normalization, uses the k distance plot to determine epsilon value of 0.25, and runs DBSCAN with min_samples set to 10. The algorithm discovered exactly 2 clusters matching the 2 crescents and identified 36 noise points corresponding to injected outliers. The comparison with K means clearly showed that DBSCAN correctly follows on linear shape of clusters while K means draws an incorrect straight-line boundary. Although K means achieved a higher silhouette score of 0.459 compared to DBSCAN's 0.373, this is a known artifact of the silhouette metric favoring convex shapes rather than an indication of better clustering. DBSCAN's built in noise detection automatically excluded outlier points that K-means absorbed. Python's scikit learn matplotlib, and NumPy libraries made it possible to implement entire pipeline in a concise script. Future work could extend this approach to higher dimensional data using cosine distance, test HDBSCAN for datasets with varying cluster densities, or apply the pipeline to real world datasets such as text embeddings or GPS coordinates.

FOR AUTHOR USE ONLY

Chapter 13 - Principal Component Analysis (PCA)

Abstract

High-resolution medical imaging, such as retinal scanning, requires significant data storage and transmission bandwidth. While dimensionality reduction techniques like PCA offer substantial data compression, their application in healthcare must meticulously balance mathematical efficiency with the preservation of diagnostic visual fidelity. This study investigates the application of PCA on 1411 × 1411-pixel RGB retinal scans to evaluate this balance. Here, we will implement PCA from scratch, comparing two distinct matrix formulation methods: independent color channel processing versus flattened 3D array processing. The reconstructed images were then evaluated against standard statistical variance thresholds. Results indicate a critical divergence between mathematical variance retention and clinical viability. Utilizing a standard threshold of 95% variance—which required only 12 principal components in the flattened array—resulted in severe blurring and the critical loss of high - frequency diagnostic markers, such as fine vascular networks. Conversely, apply PCA to isolated color channels (specifically the diagnostically rich red channel) and enforce a >99% variance threshold (K =50 components) successfully preserved vital micro- vascular structures while still achieving an approximate 95% reduction in file storage size. The finding demonstrate that standard statistical thresholds are inadequate for medical imaging. To ensure clinical safety and diagnostic accuracy, PCA compression must avoid feature flattening and be strictly calibrated to preserve high- frequency structural geometrics.

FOR DEMO USE ONLY

Introduction

13.1. Understanding Dimensionality Reduction

Each example in today's datasets tends to come with hundreds, sometimes thousands, of characteristics. Take images: these emerge from vast arrays made up of small points called pixels. Simultaneously captured sensor outputs shape medical signal records. Different coding techniques turn written words into extended strings of numbers. What looks helpful at the start often becomes complicated later. A flood of details might suggest clarity, yet things shift when more variables enter. Imagine moisture thinning across a growing desert data spread so far apart it hardly connects. The gaps stretch wide until measuring them feels pointless.

Simpler views of dense data come through reducing dimensions, keeping key details intact. What remains is reshaped neither lost nor ignored but packed tighter. Noise fades when extra layers drop away, so real trends show up more clearly.

13.2. Problems When There Are Too Many Data Points

13.2.1 Sparsity and Distance Breakdown

When dimensions increase, points scatter widely through the feature space. Nearness loses meaning distances shift unpredictably. Methods built on nearby examples, like k nearest neighbors or cluster-based techniques, work less effectively under these conditions. Instead, they demand additional data, expanded memory needs, and stronger processing capacity.

13.2.2 Redundant and Correlated Features

Consider how elements within actual datasets often link closely. Take photographs adjacent points usually match in hue. Sensor outputs show a pattern too: readings near each other in time shift in tandem. Look at written language; terms pop up repeatedly alongside familiar neighbors.

FOR AUTHOR USE ONLY

13.2.3 Noise and False Signals

In high dimensions, randomness can mimic structure. Rather than capturing actual signals, algorithms might latch onto accidental alignments. Such fits produce erratic outputs when tested on new data.

13.3. PCA Algorithms and their Variants

Over the time, Principal component Analysis (PCA) has transformed from single mathematical equation into a group of algorithms which is designed to address different computational and practical problems (Jolliffe & Cadima, 2016). It is implemented through several algorithms that differs in stability, computational cost and scalability (Jolliffe & Cadima, 2016). The classical approach is based on eigen decomposition of the covariance matrix which is simple but inefficient for high dimensional dataset or large-scale dataset (Hastie et al., 2009). This has led to singular value decomposition-based implementation (Bishop, 2006). In addition to it, the processing of modern data has led the development to alternative PCA algorithms, such as incremental and randomized methods, which in turn are designed to handle larger data datasets more efficiently by sacrificing accuracy for speed and memory efficiency (Jolliffe & Cadima, 2016; scikit-learn Documentation). So, to understand PCA better, it should not be seen as single or individual process

but as a group of related algorithms of which each are suitable for different dataset (Jolliffe & Cadima, 2016).

13.3.1 Covariance-based PCA vs SVD-based PCA

There are two common ways to execute PCA: either by using covariance matrix or by using Singular Value Decomposition, commonly known as SVD (Bishop, 2006; Hastie et al., 2009). Both methods require the data matrix X to be mean centered prior to calculation. Let's look at each method separately.

13.3.1.1 Covariance-based PCA

In covariance-based PCA we first calculate the covariance matrix that is $C = \frac{1}{n-1}X^T X$ where X is the mean centered data matrix with the shape $n \times p, X^T$ is the transpose of X with the shape $p \times n$,and the resulting matrix C has the shape $p \times p$ (Hastie et al., 2009; Shlens, 2014). After calculating the covariance matrix, we perform eigen decomposition on C to find the principal axes (Jolliffe & Cadima, 2016). This works well when the number of features (p) is small, but it can be computationally expensive and highly numerically unstable with high dimensional data (Bishop, 2006; Hastie et al., 2009).

FOR AUTHOR USE ONLY

13.3.1.2 SVD-based PCA

In SVD-based PCA, we directly apply SVD to the mean centered data matrix such that $X = U\Sigma V^T$. Here, X is our data matrix with the shape n$\times p$, U is the matrix of left singular vectors with orthonormal columns, Σ represents the diagonal matrix of singular values, and V^T is the transpose of the right singular vectors of V (Bishop, 2006; Hastie et al., 2009). The principal axes (loadings) are given by the columns of V, while the actual principal components (scores) can be derived from $U\Sigma$ (Bishop, 2006). This method is more numerically stable and is often faster and more reliable for large datasets, as it avoids calculating the explicit covariance matrix (Bishop, 2006; Hastie et al., 2009).

Table 13.1. Aspect, Covariance PCA, SVD PCA.

Aspect	Covariance PCA	SVD PCA
Computation	Eigen decomposition of covariance	Direct SCD on data
Stability	Less stable for high dimension	More numerically stable
Efficiency	Slower for large feature sets	Often faster in practice
Memory	Needs covariance matrix	Works directly on data

In modern machine learning, PCA is almost used using SVD due to its better properties (Bishop, 2006; Hastie et al., 2009).

13.3.2 Variants of PCA

This section explores several common variants of PCA designed to address specific computational or structural limitations of the standard algorithm

13.3.2.1 Kernel PCA

Kernel PCA extends standard PCA to non-linear feature spaces using the "kernel trick" (Bishop, 2006). Instead of calculating PCA in the original input space, the data is implicitly mapped to a higher-dimensional space where a linear PCA is then performed (Bishop, 2006). This allow Kernal PCA to identify and capture curved or non-linear structures inherent in the data (Jolliffe & Cadima, 2016; scikit-learn Documentation). However, kernel is computationally expensive and memory-intensive, making it difficult to scale for very large datasets.

FOR AUTHOR USE ONLY

13.3.2.2 Incremental PCA

Incremental PCA is specifically designed for large datasets that exceed available system memory (scikit-learn Documentation). Instead of processing the entire dataset simultaneously, it updates the principal components iteratively in small batches. This approach makes it highly suitable for handling massive datasets, though it sacrifices a marginal degree of accuracy compared to full, exact PCA algorithms (Jolliffe & Cadima, 2016; scikit-learn Documentation).

13.3.2.3 Randomized PCA

Random PCA utilizes random projections to estimate the leading principal component much more efficiently. It significantly decreases the computation time for large matrices while maintaining high approximation quality. This methos is exceptionally useful when only a small subset of the leading principal components (k) is required for dimensionality reduction (Hastie et al., 2009; scikit-learn Documentation).

13.3.3 Complexity and Performance Considerations

13.3.3.1 Standard PCA (SVD)

The time complexity of Standard PCA using Singular Value Decomposition is approximately$O\left(min(nd^2, n^2d)\right)$, where n is the number of samples and d is the number of features. This can be computationally expensive for large data set (Hastie et al., 2009).

```python
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris

# Load example dataset
data = load_iris()
X = data.data   # This is X (n x d matrix)

# Standardize data
X_scaled = StandardScaler().fit_transform(X)

# Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

print("Explained variance ratio:", pca.explained_variance_ratio_)
print("Shape of reduced data:", X_pca.shape)
```

```
Explained variance ratio: [0.72962445 0.22850762]
Shape of reduced data: (150, 2)
```

Code Snippet 13.1. Standard PCA.

FOR AUTHOR USE ONLY

13.3.3.2 Incremental PCA

Incremental PCA processes data in batches, which significantly reduce memory usage and make the algorithm highly suitable for large-scale datasets that exceed standard system memory (scikit-learn Documentation).

```python
from sklearn.decomposition import IncrementalPCA

ipca = IncrementalPCA(n_components=2, batch_size=100)
X_ipca = ipca.fit_transform(X_scaled)

print("Explained variance ratio:", ipca.explained_variance_ratio_)
```

```
Explained variance ratio: [0.72958607 0.22849778]
```

Code Snippet 13.2. Standard IncrementalPCA.

13.3.3.3 Randomized PCA

Randomized PCA reduces the calculation time for large matrices by approximating the decomposition using random projections. This is highly efficient when only a small number of principal components are required (Hastie et al., 2009; scikit-learn Documentation).

```python
pca_rand = PCA(n_components=2, svd_solver='randomized')
X_pca_rand = pca_rand.fit_transform(X_scaled)
```

Code Snippet 13.3. Randomized PCA.

13.3.3.4 Kernel PCA

Kernel PCA requires computing, storing and decomposing the$n \times n$ kernel matrix. This leads to $O(n^2)$ memory complexity and $O(n^3)$ time complexity in the worst case, which severely limits its scalability for large datasets.

```
from sklearn.decomposition import KernelPCA

kpca = KernelPCA(n_components=2, kernel='rbf', gamma=0.1)
X_kpca = kpca.fit_transform(X_scaled)
```

Code Snippet 13.4. Kernel PCA.

In practice, the choice of a PCA variant depends heavily on the dataset size, dimensionality, memory constraints and whether non-linear relationships need to be captured (Jolliffe & Cadima, 2016; Hastie et al., 2009).

FOR AUTHOR USE ONLY

13.4. Case Study

1. Image Compression with Principal Component Analysis by Aaron Schlege (Schlegel, 2017) a case study I says that the article, depends on how digital images are essentially a large collection of pixels. For grayscale images, each pixel contains an intensity value, and for color images, there are three matrices (Red, Green, Blue), Because images contain many correlated pixels (nearby pixels often look similar), there is redundancy in the data.

2. In the case study Noise Reduction in Nano-Raman Spectroscopy Using Principal Component Analysis (Guimarães et al., 2025) talks about a 64x64 pixel map in this work for the study of MoSe2 by (TERS). Each pixel corresponds to a certain area on the sample and is measured having its spectrum leading to the large dataset of 4096 spectra. The raster scanning method used in the grid analysis helps capture very detailed chemical and structural information at the scaling After applying Principal Component Analysis (PCA), the noise in the data is reduced in low variance.

3. In the Using PCA for Effective Visualization in Peptide Classification (Guimarães et al., 2025) about Principal Component Analysis (PCA) remains a critical, unsupervised tool for reducing

high-dimensional peptide datasets into 2D or 3D, enabling clear visualization and classification of antimicrobial (AMP) and non-antimicrobial peptides (NAMPs)

13.5 Proposed Methodology

13.5.1. System Workflow

This section discusses several key areas: The dataset selected for experimentation, the image preprocessing conducted, and the distinct matrix formulation techniques (independent color channels versus flattened 3D arrays) for structuring the data. It also explains how the PCA algorithm operates within in mathematical framework for optimizing dimensionality reduction and determining variance thresholds. Finally, there's a summary of the different compression approaches applied in this research. To compress high-resolution medical data while preserving critical diagnostic geometrics, this effort focuses on developing an unsupervised dimensionality reduction technique. The methodology's architecture is depicted in figure 13.1. Firstly, data comprising a high -resolution 1411 × 1411 RGB retinal scan was obtained from the scikit-image medical repository. In conclusion, image compression and visual reconstructions were executed using a customized PCA mathematical pipeline along comparative evaluation methods mathematical variance retention against clinical visual fidelity.

FOR AUTHOR USE ONLY

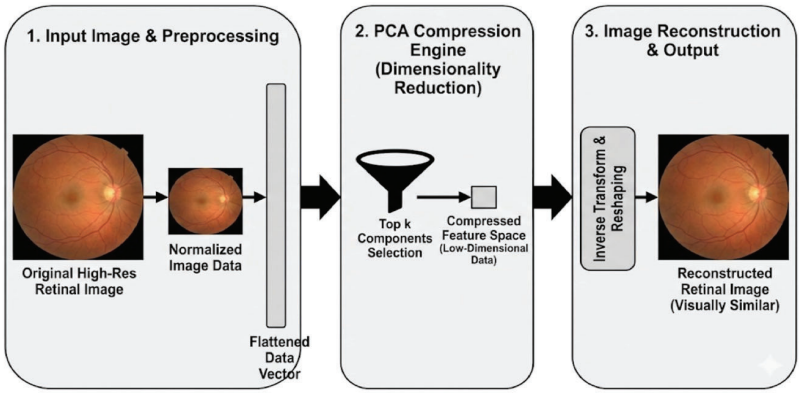


Figure 13.1. System Proposed Methodology for retinal scan.

FOR AUTHOR USE ONLY

13.5.2. Mathematical Methodology

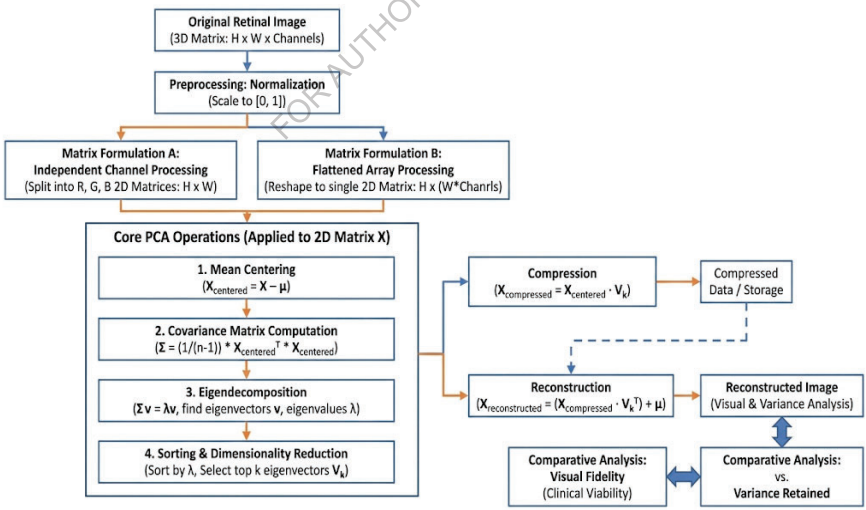


Figure 13.2. Mathematical Methodology for 3D and 2D image for high-resolution medical imagery.

To evaluate the efficacy of dimensionality reduction on high-resolution medical imagery, PCA was implemented from scratch using fundamental linear algebra operations.

The objective was to compress 1411 × 1411-pixel RGB retinal scan by extracting its principal spatial components and evaluating the resulting visual fidelity against the mathematical retained variance. [Pearson, K. (Pearson, 1901)]

13.5.2.1. Data Preprocessing and Matrix Formulation

The original retinal image was loaded as a 3D matrix (Height × Width × Channels). To satisfy the 2D input requirements of standard PCA, the image was normalized to a [0,1] scale and processed using two distinct approaches:

1. Independent channel Processing: The image was separated into isolated 2D matrix for the red, green and blue channels, each of shape 1411 × 1411.
2. Flattened Array Processing: The 3D image was reshaped into a single 2D matrix 1411 × 4233, combining spatial width and color channels into a unified feature set.

FOR AUTHOR USE ONLY

13.5.2.2. Mean Centering

Centering the data comes first in PCA each feature has its mean removed before further steps. For a given image matrix X, the empirical mean of each column(feature) μ was calculated and subtracted from the dataset:

$$X_{centered} = X - \mu, \qquad \textit{(Eq.13.1)}$$

13.5.2.3. Covariance Matrix computation

To quantify how the pixel columns vary together across the image, the Covariance Matrix Σ was computed (Jolliffe & Cadima, 2016). For the Independent channel approach, this resulted in a 1411 × 1411 matrix, whereas the Flattened approach yielded a 4233 × 4233 matrix.

$$\Sigma = \frac{1}{\mathrm{n}-1} X_{centered}^{T} X_{centered}, \qquad \textit{(Eq.13.2)}$$

Where n represents the number of rows (pixels in the y-axis), and $X_{centered}^{T}$ is the transpose of the mean-centerd image matrix.

Big numbers on the main diagonal show features that spread out a lot. When entries away from the diagonal are big, it means two features move closely together. Direction matters how data stretches or shrinks depends on these

patterns. The layout of this table reflects where change happens most in multidimensional space. (Bishop, 2006)

13.5.2.4. Eigen Decomposition

The core of the dimensionality reduction relies on the eigen decomposition of the symmetric covariance matrix Σ to solve the characteristic equation:

$$\Sigma v = \lambda v \,, \qquad \textit{(Eq.13.3)}$$

This step yield two critical components:

- Eigenvectors (v): The orthogonal directions of maximum variance (the principal components).
- Eigenvalues (λ): Scalars representing the magnitude of variance captured along each corresponding eigenvector.

Directions emerge as eigenvectors; the extent of stretch in those directions shows up in eigenvalues. A different angle reveals alignment through these vectors how far things pull apart comes down to their corresponding values.

13.5.2.5. Sorting, Explained Variance and Dimensionality Reduction

The eigenvectors were sorted in descending order (Shlens, 2014) based on their corresponding eigenvalues. This sorted matrix V, organizes the spatial directions from most critical (highest variance) to least critical (noise/fine details). To determine how much structural information each component captures, the explained variance ratio was calculated. A single component's share of the total variation is found by dividing its eigenvalue (λ_i) by the sum of all eigenvalues:

$$Explained\ Variance\ Ratio_i = \frac{\lambda_i}{\sum_{j=1}^{d} \lambda_i}, \qquad \textit{(Eq.13.4)}$$

To compress the image, only the top k eigenvectors were retained, forming a reduced matrix V_k. The optimal k (Hastie et al., 2009) was determined by calculating the cumulative sum of these ratios until the target clinical threshold (like 95% 0r 99%) was reached (Hastie et al., 2009; Shlens, 2014).

13.5.2.6. Projection and Image Reconstruction

The mean-centered image data was projected into the newly defined lower-dimensional subspace to create the compressed dataset:

$$X_{compressed} = X_{centered} \cdot V_k, \qquad \textit{(Eq.13.5)}$$

To evaluate the visual fidelity of the compression, the data was mathematically transformed back into the original spatial dimensions by multiplying the compressed matrix by the transpose of the reduced eigenvector matrix, and subsequently adding the original mean back to the data:

$$X_{reconstructed} = (X_{compressed} \cdot V_k^T) + \mu, \qquad \textit{(Eq.13.6)}$$

13.5.2.7. Scree Plot

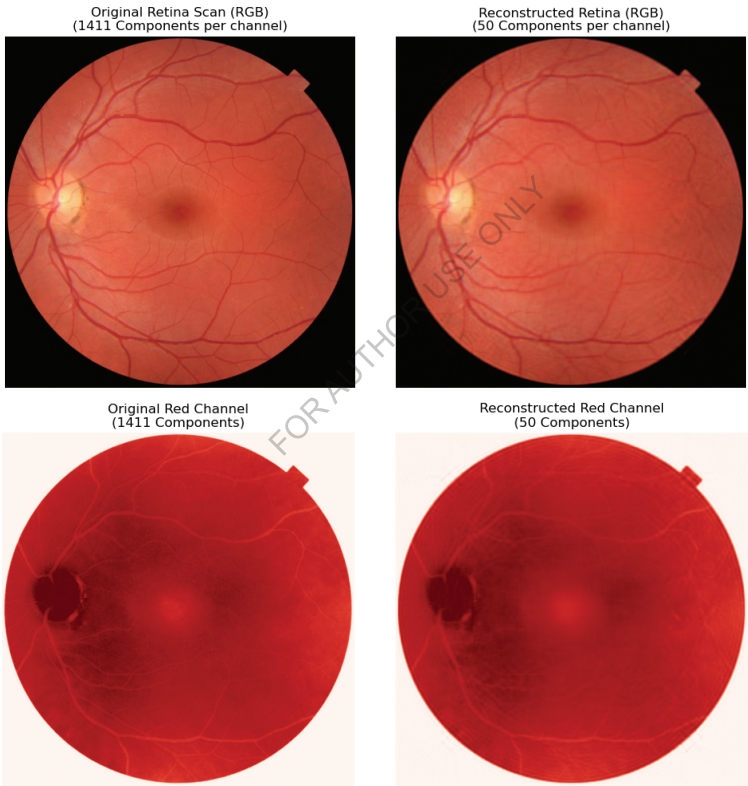
FOR AUTHOR USE ONLY

A scree plot shows the explained variance or the eigenvalues versus the component index (Hastie et al., 2009; Shlens, 2014). Typically, the curve in the graph drops sharply and then flattens. The “elbow point” is more often chosen as the reasonable number of components (Hastie et al., 2009).

13.5.3. Observational Analysis of PCA on High- Resolution Retinal Imaging

13.5.3.1 High Visual Fidelity at Extreme Compression Ratios

Original Medical Image Shape: (1411, 1411, 3)



FOR AUTHOR USE ONLY

Figure 13.3. RGB comparison and Red Channel Comparison.

This section corresponds to the RGB reconstruction images, the Red Channel reconstruction images and the calculated variance console output.
In observing the initial channel-by-channel PCA application, the most striking result is the visual similarity between the original 1411 × 1411 retinal scan and

the reconstructed image using only $k = 50$ principal components. Visually, the diagnostic features such as the optic disc, macula and fine vascular networks—remain remarkably well-preserved in both the isolated red channel and the composite RBG image.

```
--------------------------------------------
Red Channel Variance retained with 50 components: 99.54%
Average RGB Variance retained with 50 components: 98.58%
--------------------------------------------
```

Output 13.1. Variance console Output.

This subjective observation is strongly supported by the quantitative variance metrics. By retaining just 50 out of 1411 possible components (representing a compression of roughly 96.5% of the spatial dimensions), the model still captures 99.54% of the variance in the red channel and an average of 98.58% across the RGB channels.

Table 13.2. Evaluation and results.

Compression Method	Input Features	Components Kept (k)	Variance Retained	Visual Fidelity / Clinical Viability
Independent Red Channel	1411	50	99.54%	**High:** Fine vascular networks preserved.
Independent RGB (Average)	1411	50	98.58%	**High:** Global color and structural accuracy.
Independent Red Channel	1411	9	~95.00%	**Moderate:** Macro-structures visible, some high-frequency loss.
Flattened RGB Array	4233	12	~95.00%	**Low:** Severe blurring, diagnostic details lost.

FOR AUTHOR USE ONLY

This demonstrate that most of the raw matrix data in these medical scans constitutes redundant spatial information or background noise, rather than core structural data Jolliffe, I. T. (Jolliffe, 2002).

13.5.3.2 Rapid Decay of Information Importance (Eigenvalues)

```
============================================================
1. THE COVARIANCE MATRIX
============================================================
Full Shape: (1411, 1411) (Every pixel column compared to every other)
Here is the top-left 5x5 slice:
[[ 0.  0. -0. -0. -0.]
 [ 0.  0.  0. -0. -0.]
 [-0.  0.  0.  0.  0.]
 [-0. -0.  0.  0.  0.]
 [-0. -0.  0.  0.  0.]]


============================================================
2. THE EIGENVALUES (Variance Magnitude)
============================================================
Full Shape: (1411,) (One for each column)
Here are the top 5 largest eigenvalues (most important components):
[75.6983 13.7457  5.7419  4.0391  2.1755]


============================================================
3. THE EIGENVECTORS (Principal Directions)
============================================================
Full Shape: (1411, 1411) (1411 rows by 1411 directions)
Here are the first 5 values of the top 5 most important eigenvectors:
[[ 0.     -0.0001  0.     -0.0001 -0.    ]
 [ 0.     -0.0001  0.     -0.0001 -0.0001]
 [-0.      0.      0.0002  0.0002 -0.0003]
 [-0.0001  0.0001  0.0002  0.0003 -0.0005]
 [-0.0001  0.0001  0.0002  0.0002 -0.0004]]
```

FOR AUTHOR USE ONLY

Output 13.2. Covariance Matrix and Eigenvalues.

The calculated eigenvalues as well as eigen vectors with covariance Matrix and the corresponding Scree Plot provide a clear mathematical explanation for this high compression efficiency. The analysis of the covariance matrix reveals an extremely steep drop-off in eigenvalue magnitude. The first principal component holds an overwhelmingly large variance magnitude (≈ 75.69), while the second drops immediately to 13.74 and the fifth to just 2.17.

13.5.3.3 The Divergence of Mathematical Variance and Clinical Utility *(Red channel 2D Analysis)*

This section corresponds to the covariance Matrix/ Eigenvalues/ Eigenvectors console output and the scree plot for the Retina Red Channel. A critical finding emerged when comparing the channel-by-channel approach to the flattened 2D array approach (where the matrix was reshaped to 1411×1411). (Harris et al., 2020)

When plotted cumulatively for the red channel, the curve effectively asymptotes almost immediately. Remarkably, the data shows that only 9 principal components are required to surpass the 95% variance threshold, and 48 principal components are required to surpass the 99.5% variance threshold. This proves that the defining anatomical geometrics of the retina are mathematically aligned along a very small number of structural axes.

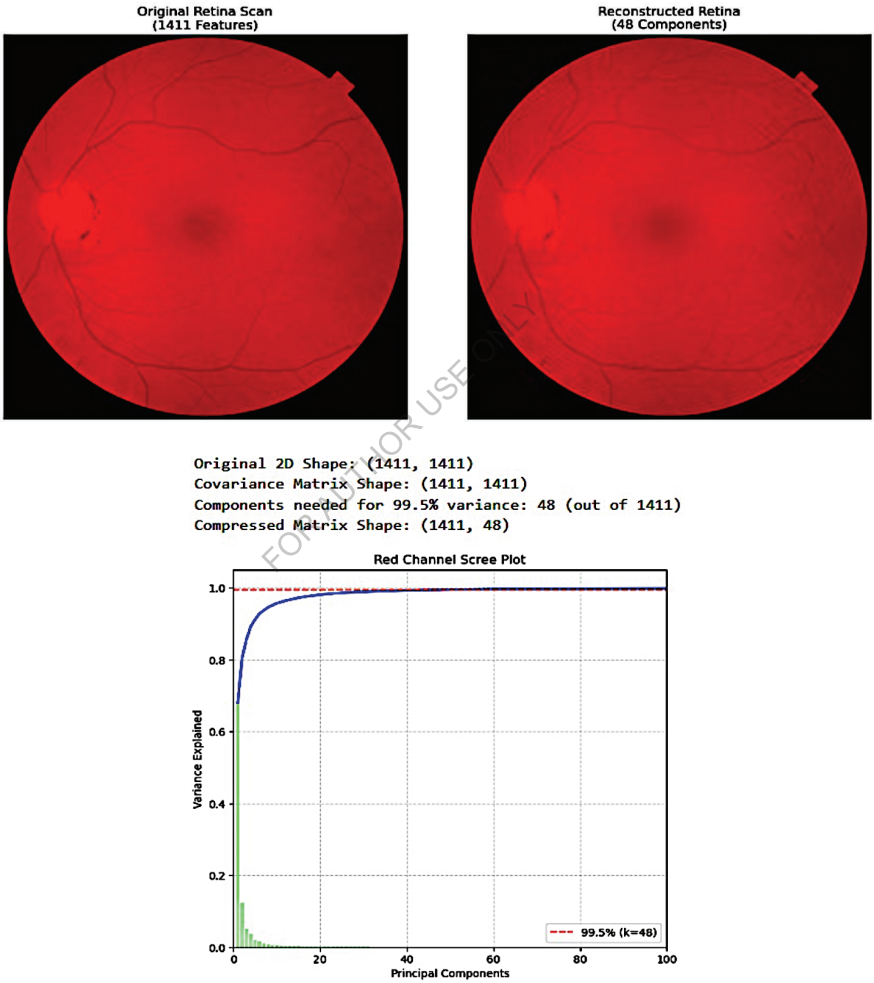


Figure 13.4. Red Channel Scree Plot and Blurry Reconstruction with 99.5% variance.

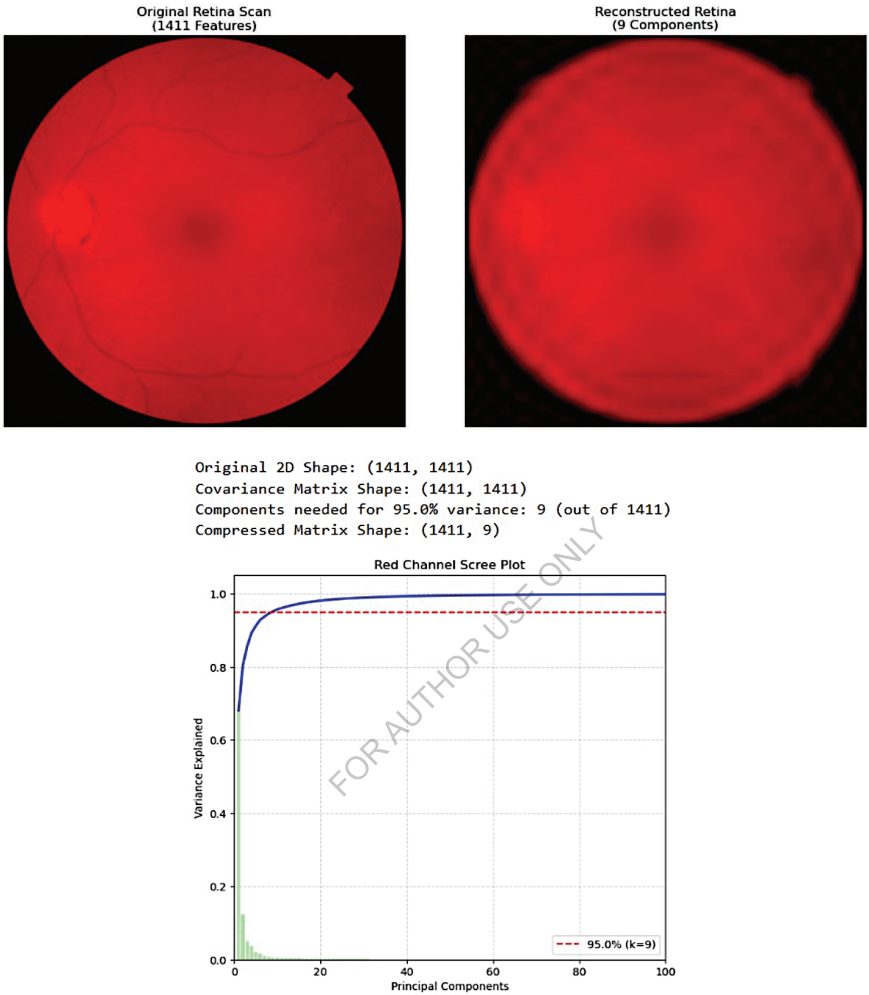


Figure 13.5. Red Channel Scree Plot and Blurry Reconstruction with 95% variance.

13.5.3.4 The Divergence of Mathematical Variance and Clinical Utility (Flattened RGB 3D Analysis)

This section corresponds to the Unified RBG Scree Plot and the highly blurred Reconstructed Retina image using 12 and 84 components.

A critical finding emerged when comparing the channel-by-channel approach to the flattened 3D array approach (where the matrix was reshaped to 1411×4233).

For the flattened RGB matrix, the algorithm determined that only 12 components were needed to mathematically retain 95% of the overall variance. However, observing the reconstructed image at, while 84 components were needed to mathematically retain 99.5% of the overall variance. However, observing the reconstructed image at reveals severe visual degradation. The resulting image suffers from heavy blurring, blocky artifacts and a loss of the fine vascular details required for medical diagnostics. (Van der Walt et al., 2014)$k = 12k = 84$.

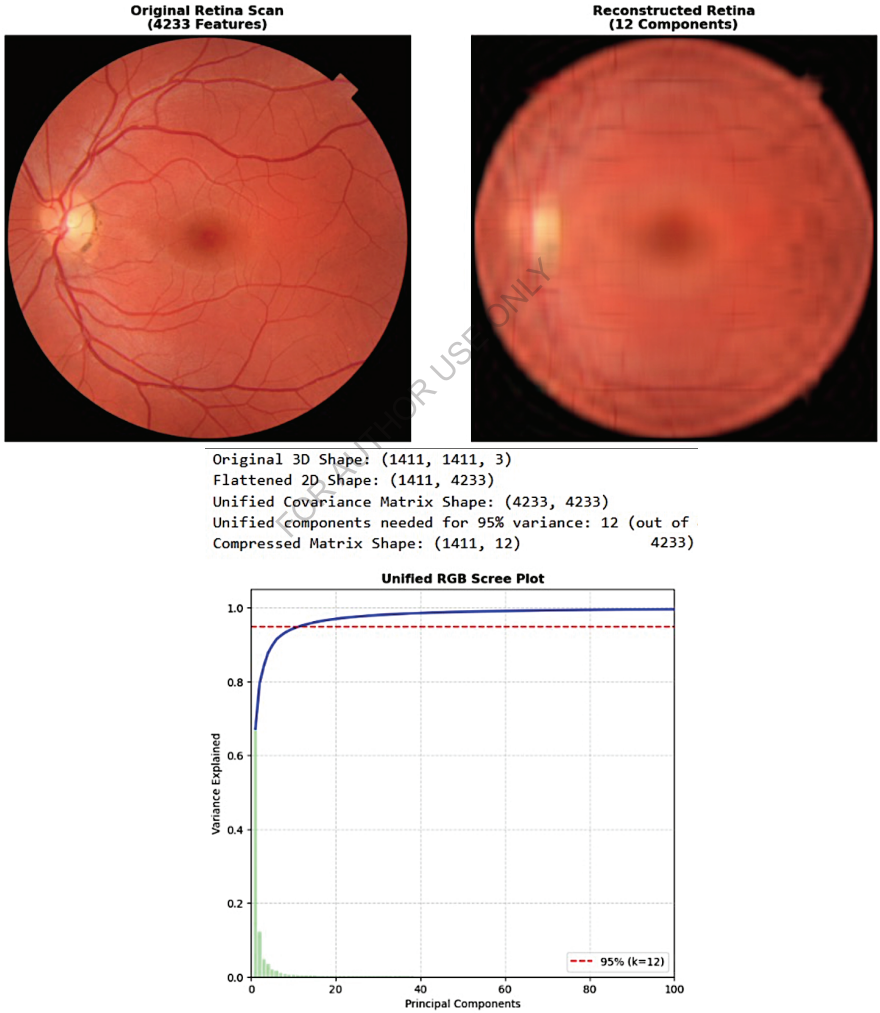
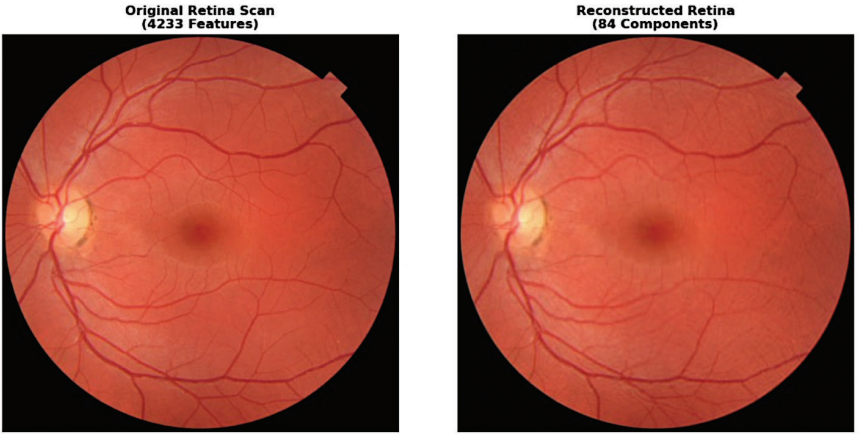


Figure 13.6. Unified RGB Scree Plot and Blurry Reconstruction with 95% variance.




```
Original 3D Shape: (1411, 1411, 3)
Flattened 2D Shape: (1411, 4233)
Unified Covariance Matrix Shape: (4233, 4233)
Unified components needed for 99.5% variance: 84 (out of 4233)
Compressed Matrix Shape: (1411, 84)
```

FOR AUTHOR USE ONLY

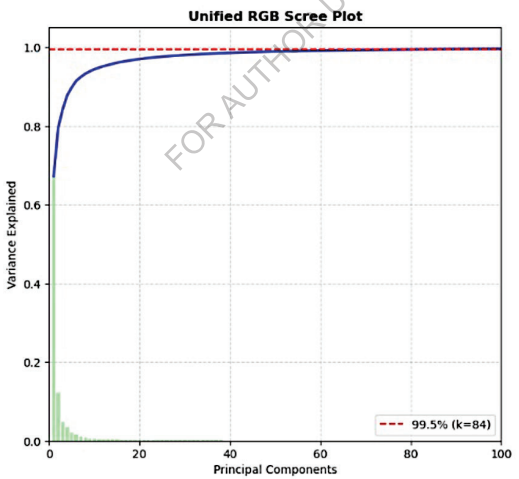


Figure 13.7. Unified RGB Scree Plot and Blurry Reconstruction with 99.5% variance.

13.6 Code examples for retaining image dataset

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import data

# 1. Load the retinal image in full RGB
img_rgb = data.retina() / 255.0
H, W, C = img_rgb.shape

# 2. Flatten the 3D array into a 2D matrix
# This unifies the Width and Color Channels into a single feature space
img_flat = img_rgb.reshape(H, W * C)

# 3. PCA Math (Mean Centering, Covariance, Eigendecomposition)
img_mean = np.mean(img_flat, axis=0)
img_centered = img_flat - img_mean

cov_matrix = np.cov(img_centered, rowvar=False)
eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

# Sort in descending order
idx = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx]
```

FOR AUTHOR USE ONLY

```
# 4. Calculate Variance & Find Clinical Threshold (Set to 95%)
total_var = np.sum(eigenvalues)
variance_explained = eigenvalues / total_var
cumulative_var = np.cumsum(variance_explained)

# Find Clinical Threshold (Set to 95%)
# target_variance = 0.95

#Find Clinical Threshold (Set to 99.5%)
target_variance = 0.995
k_target = np.argmax(cumulative_var >= target_variance) + 1

# 5. Compression and Reconstruction
V_k = eigenvectors[:, :k_target]
compressed_img = np.dot(img_centered, V_k)
```

```
# Reconstruct and reshape back to original 3D RGB format
reconstructed_flat = np.dot(compressed_img, V_k.T) + img_mean
reconstructed_img = reconstructed_flat.reshape(H, W, C)
reconstructed_img = np.clip(reconstructed_img, 0, 1) # Ensure colors stay in valid range

# --- Print the Matrix Shapes (Console Output) ---
print(f"Original 3D Shape: {img_rgb.shape}")
print(f"Flattened 2D Shape: {img_flat.shape}")
print(f"Unified Covariance Matrix Shape: {cov_matrix.shape}")
print(f"Unified components needed for {target_variance*100}% variance: {k_target} (out of {W*C})")
print(f"Compressed Matrix Shape: {compressed_img.shape}")

# 6. VISUALIZATION
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Panel 1: Scree Plot (Cumulative + Individual Variance)
axes[0].bar(range(1, 101), variance_explained[:100], color='lightgreen', alpha=0.7)
axes[0].plot(range(1, 101), cumulative_var[:100], color='blue', linewidth=2)
axes[0].axhline(y=target_variance, color='red', linestyle='--', label=f'99.5% (k={k_target})')
axes[0].set_title('Unified RGB Scree Plot', fontsize=12, fontweight='bold')
axes[0].set_xlabel('Principal Components')
axes[0].set_ylabel('Variance Explained')
axes[0].grid(True, linestyle='--', alpha=0.6)
axes[0].legend(loc='lower right')
axes[0].set_xlim(0, 100)
axes[0].set_ylim(0, 1.05)

# Panel 2: Original Retina Scan (Full RGB)
axes[1].imshow(img_rgb)
axes[1].set_title(f'Original Retina Scan\n({W*C} Features)', fontsize=12, fontweight='bold')
axes[1].axis('off')

# Panel 3: Reconstructed Retina (Full RGB)
axes[2].imshow(reconstructed_img)
axes[2].set_title(f'Reconstructed Retina\n({k_target} Components)', fontsize=12, fontweight='bold')
axes[2].axis('off')
```

FOR AUTHOR USE ONLY

Code Snippet 13.5. Code for Flattened RGB 3D Analysis.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import data

# 1. Load the retinal image and isolate the Red Channel
img_rgb = data.retina() / 255.0
R = img_rgb[:, :, 0]
H, W = R.shape

# 2. PCA Math (Mean Centering, Covariance, Eigendecomposition)
R_mean = np.mean(R, axis=0)
R_centered = R - R_mean

cov_matrix = np.cov(R_centered, rowvar=False)
eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

# Sort in descending order
idx = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx]
```

```
# 3. Calculate Variance & Find Clinical Threshold (>99%)
total_var = np.sum(eigenvalues)
variance_explained = eigenvalues / total_var
cumulative_var = np.cumsum(variance_explained)

#Find Clinical Threshold (99.5%)
target_variance = 0.995
#Find Clinical Threshold (95%)
#target_variance = 0.95
k_target = np.argmax(cumulative_var >= target_variance) + 1

# 4. Compression and Reconstruction
V_k = eigenvectors[:, :k_target]
compressed_R = np.dot(R_centered, V_k)

reconstructed_R = np.dot(compressed_R, V_k.T) + R_mean
reconstructed_R = np.clip(reconstructed_R, 0, 1)
```

```
# --- Print the Matrix Shapes (Console Output) ---
print(f"Original 2D Shape: {R.shape}")
print(f"Covariance Matrix Shape: {cov_matrix.shape}")
print(f"Components needed for {target_variance*100}% variance: {k_target} (out of {W})")
print(f"Compressed Matrix Shape: {compressed_R.shape}")

# 5. VISUALIZATION (Tinted Red)
# Create empty 3D arrays and place our data ONLY in the Red channel (index 0)
R_display = np.zeros((H, W, 3))
R_display[:, :, 0] = R

reconstructed_display = np.zeros((H, W, 3))
reconstructed_display[:, :, 0] = reconstructed_R

fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Panel 1: Scree Plot (Cumulative + Individual Variance)
axes[0].bar(range(1, 101), variance_explained[:100], color='lightgreen', alpha=0.7)
axes[0].plot(range(1, 101), cumulative_var[:100], color='blue', linewidth=2)
axes[0].axhline(y=target_variance, color='red', linestyle='--', label=f'{target_variance*100}% (k={k_target})')
axes[0].set_title('Red Channel Scree Plot', fontsize=12)
axes[0].set_xlabel('Principal Components')
axes[0].set_ylabel('Variance Explained')
axes[0].grid(True, linestyle='--', alpha=0.6)
axes[0].legend(loc='lower right')
axes[0].set_xlim(0, 100)
axes[0].set_ylim(0, 1.05)

# Panel 2: Original Retina Scan (Red Tinted)
axes[1].imshow(R_display)
axes[1].set_title(f'Original Retina Scan\n({W} Features)', fontsize=12)
axes[1].axis('off')

# Panel 3: Reconstructed Retina (Red Tinted)
axes[2].imshow(reconstructed_display)
axes[2].set_title(f'Reconstructed Retina\n({k_target} Components)', fontsize=12)
axes[2].axis('off')
```

FOR AUTHOR USE ONLY

```
plt.tight_layout()
plt.savefig('red_channel_tinted_3panel.png', dpi=300, bbox_inches='tight')
plt.show()
```

Code Snippet 13.6. Code for Red channel 2D Analysis.

```
import numpy as np
from skimage import data

# 1. Load the built-in medical image (Human Retina)
img_rgb = data.retina()
img_normalized = img_rgb / 255.0

# Isolate the Red channel
R = img_normalized[:, :, 0]

# Step A: Mean Centering
R_mean = np.mean(R, axis=0)
R_centered = R - R_mean

# Step B: Calculate the Covariance Matrix
cov_matrix = np.cov(R_centered, rowvar=False)

# Step C & D: Eigendecomposition and Sorting
eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)
sorted_indices = np.argsort(eigenvalues)[::-1]
sorted_eigenvalues = eigenvalues[sorted_indices]
sorted_eigenvectors = eigenvectors[:, sorted_indices]

print("=" * 60)
print("1. THE COVARIANCE MATRIX")
print("=" * 60)
print(f"Full Shape: {cov_matrix.shape} (Every pixel column compared to every other)")
print("Here is the top-left 5x5 slice:")
# Print the first 5 rows and first 5 columns, rounded to 4 decimal places
print(np.round(cov_matrix[:5, :5], 4))
print("\n")

print("=" * 60)
print("2. THE EIGENVALUES (Variance Magnitude)")
print("=" * 60)
print(f"Full Shape: {sorted_eigenvalues.shape} (One for each column)")
print("Here are the top 5 largest eigenvalues (most important components):")
print(np.round(sorted_eigenvalues[:5], 4))
print("\n")
```

FOR AUTHOR USE ONLY

```
print("=" * 60)
print("3. THE EIGENVECTORS (Principal Directions)")
print("=" * 60)
print(f"Full Shape: {sorted_eigenvectors.shape} (1411 rows by 1411 directions)")
print("Here are the first 5 values of the top 5 most important eigenvectors:")
# Print the first 5 rows of the first 5 columns (the 5 most important vectors)
print(np.round(sorted_eigenvectors[:5, :5], 4))
print("=" * 60)
```

Code Snippet 13.7. Code for covariance matrix, eigenvalues, and eigenvector.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import data

# 1. Load the Human Retina
img_rgb = data.retina()
img_normalized = img_rgb / 255.0

# Isolate the three channels
R = img_normalized[:, :, 0]
G = img_normalized[:, :, 1]
B = img_normalized[:, :, 2]

print(f"Original Medical Image Shape: {img_normalized.shape}")
# PCA FUNCTION
def manual_pca(X, k):
    # Step A: Mean Centering
    X_mean = np.mean(X, axis=0)
    X_centered = X - X_mean

    # Step B: Calculate the Covariance Matrix
    cov_matrix = np.cov(X_centered, rowvar=False)
```

FOR AUTHOR USE ONLY

```
    # Step C & D: Eigendecomposition and Sorting
    eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)
    sorted_indices = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[sorted_indices]
    eigenvectors = eigenvectors[:, sorted_indices]

    # Step E: Compress
    top_eigenvectors = eigenvectors[:, :k]
    X_compressed = np.dot(X_centered, top_eigenvectors)

    # Step F: Reconstruct
    X_reconstructed = np.dot(X_compressed, top_eigenvectors.T) + X_mean

    # Ensure values stay valid for display
    X_reconstructed = np.clip(X_reconstructed, 0, 1)
    return X_reconstructed, eigenvalues
```

```
# Choose the number of top components to keep
k = 50

# Apply the manual PCA function to each color channel
R_reconstructed, R_eigen = manual_pca(R, k)
G_reconstructed, G_eigen = manual_pca(G, k)
B_reconstructed, B_eigen = manual_pca(B, k)

# Stack the reconstructed channels back into a full RGB image
img_reconstructed = np.dstack((R_reconstructed, G_reconstructed, B_reconstructed))

plt.figure(figsize=(10, 10))
# 1. Plot Original Medical RGB
plt.subplot(2, 2, 1)
plt.title("Original Retina Scan (RGB)\n(1411 Components per channel)")
plt.imshow(img_normalized)
plt.axis('off')

# 2. Plot Reconstructed Medical RGB
plt.subplot(2, 2, 2)
plt.title(f"Reconstructed Retina (RGB)\n({k} Components per channel)")
plt.imshow(img_reconstructed)
plt.axis('off')

# 3. Plot Original Red Channel
plt.subplot(2, 2, 3)
plt.title("Original Red Channel\n(1411 Components)")
plt.imshow(R, cmap='Reds')
plt.axis('off')

# 4. Plot Reconstructed Red Channel
plt.subplot(2, 2, 4)
plt.title(f"Reconstructed Red Channel\n({k} Components)")
plt.imshow(R_reconstructed, cmap='Reds')
plt.axis('off')

plt.tight_layout()
plt.show()

# --- VARIANCE CALCULATIONS ---

# 1. Calculate variance retained for the Red Channel
var_R_retained = (np.sum(R_eigen[:k]) / np.sum(R_eigen)) * 100

# 2. Calculate variance retained for Green and Blue channels
var_G_retained = (np.sum(G_eigen[:k]) / np.sum(G_eigen)) * 100
var_B_retained = (np.sum(B_eigen[:k]) / np.sum(B_eigen)) * 100

# 3. Calculate the average variance retained across all three RGB channels
average_RGB_variance = (var_R_retained + var_G_retained + var_B_retained) / 3

print("-" * 45)
print(f"Red Channel Variance retained with {k} components: {var_R_retained:.2f}%")
print(f"Average RGB Variance retained with {k} components: {average_RGB_variance:.2f}%")
print("-" * 45)
```

FOR AUTHOR USE ONLY

Code Snippet 13.8. Code for RGB comparison and Red Channel Comparison.

```python
import os
import numpy as np

# --- 1. SAVE THE ORIGINAL IMAGE ---
# We save the full 3D normalized array as a standard numpy file (.npy)
original_file = 'original_retina.npy'
np.save(original_file, img_normalized)

# --- 2. SAVE THE COMPRESSED DATA ---
# We use np.savez to bundle our 3 necessary pieces into a single archive file (.npz)
compressed_file = 'compressed_retina.npz'
np.savez(compressed_file,
         compressed_data=X_compressed,
         eigenvectors=top_eigenvectors,
         mean=X_mean)

# --- 3. CALCULATE AND PRINT THE SIZES ---
# os.path.getsize returns the size in bytes. We divide by (1024 * 1024) to get Megabytes (MB).
original_size_mb = os.path.getsize(original_file) / (1024 * 1024)
compressed_size_mb = os.path.getsize(compressed_file) / (1024 * 1024)

# Calculate the percentage of space saved
space_saved = 100 - ((compressed_size_mb / original_size_mb) * 100)

print("=" * 50)
print("💾 STORAGE SAVINGS REPORT")
print("=" * 50)
print(f"Original File Size:   {original_size_mb:.2f} MB")
print(f"Compressed File Size: {compressed_size_mb:.2f} MB")
print("-" * 50)
print(f"Total Space Saved:    {space_saved:.2f}%")
print("=" * 50)
```

FOR AUTHOR USE ONLY

Code Snippet 13.9. Code for how the reduced and resize the image.

```
==================================================
💾 STORAGE SAVINGS REPORT
==================================================
Original File Size:   45.57 MB
Compressed File Size: 0.55 MB
--------------------------------------------------
Total Space Saved:    98.79%
==================================================
```

Output 13.3. how the reduced and resize the image.

13.7. Applications of PCA

Principal component analysis (PCA) is a Modelling technique that can be used to Compress the data, while retaining most of its variance. it makes the high dimension data smaller by transforming data in this way, PCA simplifies analysis and reduces technical costs. Because of these Beneficial traits, principal component analysis is widely applied not only to finance but also to areas such as image compression, noise reduction and data visualization.

Here we would like to tell you how PCA works without going into the mathematical details that can make it so hard for beginners.

13.7.1 Image Compression

The main job of PCA lies to find principal components, when we reconstruct the image, we find a sum of these weighted components as well adding in part, it is called almost like mean image. By retaining only, the top k components, where k is much smaller than the number of pixels in the original image, we can greatly compress information but still glean its basic characteristics and shape structure.

This creates a trade-off between compression ratio and visual quality. Few components (e.g., k = 100 or smaller) give blurred and overly smoothed reconstructions, without the finer details of the original, while preserving the overall structure and most dominant features. As we add more components, the reconstruction becomes more faithful to the original image, but there is less compression. A typical illustration shows the original image and 10, 50, or 100 component rebuilds. The reader can look at how reconstructions images look if we use certain components.

FOR AUTHOR USE ONLY

Table 13.3. Comparison and evaluation of results.

Components Retained	Compression Ratio	Visual Quality
10	Very High	Blurry
50	High	Acceptable
100	moderate	High quality

13.7.2 Noise Reduction

High dimension data has often contained noise, it identifies dominant variance directions and orders them, allowing for the selective removal of components that contribute less to the signal Noise usually contributes to lower-variance directions, by keeping only top principal components, PCA removes noise.

Mechanism:

High variance → meaningful signal

Low variance → random noise

13.7.3 Visualization

PCA projects often converts with high-dimensional data to 2D or 3D while retaining as much variance as possible, enabling scatter plots that reveal clusters and trends.

Finds the 2D plane that best represents the spread of these 4 features. PCA allows us to project the data into two or three dimensions while preserving as much of the total variance as possible. By plotting the first two or three principal components, we can obtain scatter plots that reveal clusters, trends, or outliers.

13.8. Limitations

13.8.1. Linearity

- PCA is a linear procedure, so it searches only for linear relationships among the variables.
- It's great for when your data structure is basically linear.
- If the underlying patterns of the data are complex or curved, PCA will not represent them significantly well.

13.8.2. Loss of Interpretability

- Every principal component is a combination of all the original variables.
- Because of this, it is frequently difficult to explain succinctly how they relate to the components.
- This can present an issue, e.g., in medicine or finance.

FOR AUTHOR USE ONLY

13.8.3. Sensitivity to Scaling

- PCA is influenced by the scale of the variables.
- The scale of some features is such that they would dominate the rest if data were not scaled.
- That is why we need to normalize the data before running PCA on it.

13.8.4. Information Loss

- PCA does transform your data significantly, changing its dimensions and reducing the number of features - some information will be destroyed.
- Its low-variance components are discarded, however one may argue they still have some useful patterns which could be exploited.
- PCA does not consider the target variable, thus it might discard predictive features.

13.9. Conclusion

This study evaluated Principal Component Analysis (PCA) as a dimensionality reduction and compression technique for high-resolution medical imaging, specifically focusing on RGB retinal scans. The findings demonstrate that PCA is a highly potent tool for optimizing medical data storage, capable of reducing file sizes by over 95% while retaining the macro-structural integrity of the image.

However, a critical clinical implication emerged regarding the reliance on standard statistical thresholds. The analysis revealed a significant divergence between mathematical variance preservation and diagnostic visual fidelity. While a standard 95% variance threshold is widely accepted in general data science, it proved severely inadequate for medical diagnostics. At this mathematical threshold, critical high-frequency spatial details—such as fine choroidal and superficial vascular networks—were discarded by the algorithm as "low-variance noise," resulting in unacceptable image blurring.

Furthermore, the study highlights that the method of matrix formulation dictates the clinical viability of the output. The flattened 3D array approach, which combined spatial and color features, failed to preserve diagnostic geometries. Conversely, processing color channels independently (such as isolating the diagnostically rich red channel) yielded vastly superior results. By treating channels independently and elevating the target variance threshold to >99% (requiring $(k = 50)$ components), the algorithm successfully preserved the micro-vascular structures necessary for accurate medical assessment.

FOR AUTHOR USE ONLY

References

Abbas Q., Hasan M., & Lee S. (2025). Explainable artificial intelligence in clinical decision-support systems: A review. Frontiers in Artificial Intelligence.

Advances in Structural and Functional Retinal Imaging and Biomarkers for Early Detection of Diabetic Retinopathy. (2024).

Aggarwal, C. C., Hinneburg, A., & Keim, D. A. (2001). On the surprising behavior of distance metrics in high dimensional spaces. In J. Van den Bussche & V. Vianu (Eds.), Database theory—ICDT 2001 (Lecture Notes in Computer Science, Vol. 1973, pp. 420–434). Springer.

Amari, S., & Nagaoka, N. (2000). Methods of information geometry. American Mathematical Society. https://doi.org/10.1090/mmono/191

Amer, A. A., Ravana, S. D., & Habeeb, R. A. A. (2025). Effective k-nearest neighbor models for data classification enhancement. Journal of Big Data, 12, Article 86. https://doi.org/10.1186/s40537-025-01137-2

Andras Janosi, William Steinbrunn, Matthais Pfisterer, Rober Detrano. UCI Machine Learning Repository: Heart Disease Dataset. University of California, Irvine. https://archive.ics.uci.edu/ml/datasets/heart+disease

Ankerst, M., Breunig, M. M., Kriegel, H.-P., & Sander, J. (1999). OPTICS: Ordering points to identify the clustering structure. In Proceedings of the ACM SIGMOD International Conference on Management of Data (pp. 49–60). ACM.

Arthur, D., & Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. In Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (pp. 1027–1035). Society for Industrial and Applied Mathematics.

Bandi, M., Masimukku, A. K., Vemula, R., & Vallu, S. (2024). Predictive analytics in healthcare: enhancing patient outcomes through data-driven forecasting and decision-making. International Numeric Journal of Machine Learning and Robots, 8(8), 1–20.

Banerjee, A., Merugu, S., Dhillon, I. S., & Ghosh, J. (2005). Clustering with Bregman divergences. Journal of Machine Learning Research, 6, 1705–1749.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. Communications of the ACM, 18(9), 509–517. https://doi.org/10.1145/361002.361007

FOR AUTHOR USE ONLY

Biomarkers in Diabetic Retinopathy. (2017).

Bishop, C. M. (2006). Pattern recognition and machine learning. Springer.

Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer.

Breiman Leo, Jerome Friedman, Olshen R. A., Charles J. (1984). Classification and regression trees. Wadsworth International Group.

Breiman, L. (1996). Bagging predictors. Machine Learning, 24(2), 123–140.

Breiman, L. (2001). Random forests. Machine Learning, 45(1), 5–32.

Burges, C. J. C. (1998). A tutorial on support vector machines for pattern recognition. Data Mining and Knowledge Discovery, 2(2), 121–167. https://doi.org/10.1023/A:1009715923555

Campello, R. J. G. B., Moulavi, D., & Sander, J. (2013). Density-based clustering based on hierarchical density estimates. In J. Pei et al. (Eds.), Advances in knowledge discovery and data mining (Lecture Notes in Computer Science, Vol. 7819, pp. 160–172). Springer.

Cayton, L. (2008). Fast nearest neighbor retrieval for Bregman divergences. In Proceedings of the 25th International Conference on Machine Learning (pp. 112–119). ACM.

Cer, D., Yang, Y., Kong, S., Hua, N., Limtiaco, N., John, R. S., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C., Sung, Y., Strope, B., & Kurzweil, R. (2018). Universal sentence encoder for English. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (pp. 169–174). ACL.

Chang, C.-C., & Lin, C.-J. (2011). LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2(3), 1–27. https://doi.org/10.1145/1961189.1961199

Chang, W. C., Dai, L., & Xu, T. (2025). Machine-learning approaches to clinical risk prediction: multi-scale temporal alignment in electronic health records. arXiv preprint arXiv:2511.21561.

Chazelle, B. (2008). Technical perspective: Finding a good neighbor, near and fast. Communications of the ACM, 51(1), 115. https://doi.org/10.1145/1327452.1327492

FOR AUTHOR USE ONLY

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785–794). ACM.

Cherif, W. (2018). Optimization of k-NN algorithm by clustering and reliability coefficients: Application of cancer diagnosis. Procedia Computer Science, 127, 293–299. https://doi.org/10.1016/j.procs.2018.01.125

Cortes, C., & Vapnik, V. (1995). Support-vector networks. Machine Learning, 20(3), 273–297. https://doi.org/10.1007/BF00994018

Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. IEEE Transactions on Information Theory, 13(1), 21–27. https://doi.org/10.1109/TIT.1967.1053964

Cristianini, N., & Shawe-Taylor, J. (2000). An introduction to support vector machines and other kernel-based learning methods. Cambridge University Press.

Deep learning based medical image compression using cross attention learning and wavelet transform. (2025).

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (pp. 4171–4186). Association for Computational Linguistics.

Deza, M. M., & Deza, E. (2009). Encyclopedia of Distances. Springer. https://doi.org/10.1007/978-3-642-00234-2

Digital medicine and the curse of dimensionality. (2021).

Dolatshah, M., Hadian, A., & Minaei-Bidgoli, B. (2015). Ball-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces. arXiv. https://arxiv.org/abs/1511.00628

Ester, M., Kriegel, H.-P., Sander, J., & Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (pp. 226–231). AAAI Press.

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. Annals of Statistics, 29(5), 1189–1232.

FOR AUTHOR USE ONLY

Guimarães, J. E., Nadas, R., & Jorio, A. (2025). Noise Reduction in Nano-Raman Spectroscopy Using Principal Component Analysis.

Gupta, P., & Seth, D. (2022). Comparative analysis and feature importance of machine learning and deep learning for heart-disease prediction. Indonesian Journal of Electrical Engineering and Computer Science, 29(1), 451.

Hadifar, A., Sterckx, L., Demeester, T., & Develder, C. (2019). A self-training approach for short text clustering. In Proceedings of the 4th Workshop on Representation Learning for NLP (pp. 194–199). Association for Computational Linguistics.

Hamerly, G., & Elkan, C. (2003). Learning the k in k-means. In Advances in Neural Information Processing Systems (Vol. 16). MIT Press.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., & Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585(7825), 357–362.

Hasan, M. K., Alam, M. A., Das, D., Hossain, E., & Hasan, M. (2020). Diabetes prediction using ensembling of different machine learning classifiers. IEEE Access, 8, 76516–76530. https://doi.org/10.1109/ACCESS.2020.2989857

Hastie, T., Tibshirani, R., & Friedman, J. (2009). The elements of statistical learning: Data mining, inference, and prediction (2nd ed.). Springer. https://doi.org/10.1007/978-0-387-84858-7

Hearst, M. A., Dumais, S. T., Osuna, E., Platt, J., & Scholkopf, B. (1998). Support vector machines. IEEE Intelligent Systems and Their Applications, 13(4), 18–28. https://doi.org/10.1109/5254.708428

Hsu, C.-W., Chang, C.-C., & Lin, C.-J. (2016). A practical guide to support vector classification. https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf

Ikotun, A. M., Ezugwu, A. E., Abualigah, L., Abuhaija, B., & Heming, J. (2023). K-means clustering algorithms: A comprehensive review, variants, and applications in the era of big data. Information Sciences, 622, 178–210. https://doi.org/10.1016/j.ins.2022.11.139

Indyk, P., & Motwani, R. (1998). Approximate nearest neighbors: Towards removing the curse of dimensionality. In Proceedings of the 30th Annual ACM Symposium on Theory of Computing (pp. 604–613). ACM.

Jain, A. K. (2010). Data clustering: 50 years beyond k-means. Pattern Recognition Letters, 31(8), 651–666.

FOR AUTHOR USE ONLY

Jevin, M. J., Jayant, H., Sanjay, R., Hemasai, V., & Venkatasrinivas, P. V. (2023). Heart-disease identification method using machine-learning classification in e-healthcare. Heart Disease, 10(3).

Johnsmith88. Heart Disease Dataset [Data set]. Kaggle. https://www.kaggle.com/datasets/johnsmith88/heart-disease-dataset/data

Johnson, J., Douze, M., & Jégou, H. (2021). Billion-scale similarity search with GPUs. IEEE Transactions on Big Data, 7(3), 535–547.

Jolliffe, I. T. (2002). Principal Component Analysis (2nd ed.). Springer-Verlag.

Jolliffe, I. T., & Cadima, J. (2016). Principal Component Analysis: A Review and Recent Developments. Philosophical Transactions of the Royal Society A, 374(2065).

Jurafsky, D., & Martin, J. H. (2023). Speech and language processing (3rd ed., draft). Stanford University. https://web.stanford.edu/~jurafsky/slp3/

Kasim, S., Malek, S., Tang, J., Kiew, X. N., Cheen, S., Liew, B., Sajidon, N., Ezman, R., & Shariff, R. (2025). Multiclass leukemia cell classification using hybrid deep learning and machine learning with CNN-based feature extraction.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., & Liu, T.-Y. (2017). LightGBM: A highly efficient gradient-boosting decision tree. In Advances in Neural Information Processing Systems (Vol. 30, pp. 3146–3154). Curran Associates.

Khan, I. K., Daud, H., Zainuddin, N., & Sokkalingam, R. (2025). Standardizing reference data in gap statistic for selection of optimal number of cluster in k-means algorithm. Alexandria Engineering Journal, 118, 246–260. https://doi.org/10.1016/j.aej.2025.01.034

Kleinberg, J. (1997). Two algorithms for nearest-neighbor search in high dimensions. In Proceedings of the 29th Annual ACM Symposium on Theory of Computing (pp. 599–608). ACM.

Kurniawan, M. F., & Megawaty, D. A. (2025). Comparison of logistic regression, random forest, support vector machine and k-nearest neighbor algorithms in diabetes prediction. Journal of Applied Informatics and Computing, 9(5), 2154–2162.

Liu, H., Chen, J., Dy, J., & Fu, Y. (2023). Transforming complex problems into k-means solutions. IEEE Transactions on Pattern Analysis and Machine Intelligence, 45(7), 9149–9168. https://doi.org/10.1109/TPAMI.2023.3237667

FOR AUTHOR USE ONLY

Liu, T., Moore, A. W., & Gray, A. (2006). New algorithms for efficient high-dimensional nonparametric classification. Journal of Machine Learning Research, 7, 1135–1158.

Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. In Advances in Neural Information Processing Systems (Vol. 30, pp. 4765–4774). Curran Associates.

Mahesh, T. R., Dhilip Kumar, V., Vinoth Kumar, V., Asghar, J., Geman, O., Arulkumaran, G., & Arun, N. (2022). AdaBoost ensemble methods using K-fold cross-validation for survivability with the early detection of heart disease. Computational Intelligence and Neuroscience, 2022, Article 9005278.

Malek, N. H. A., Yaacob, W. F. W., Wah, Y. B., Nasir, S. A. M., Shaadan, N., & Indratno, S. W. (2023). Comparison of ensemble hybrid sampling with bagging and boosting machine learning approaches for imbalanced data. Indonesian Journal of Electrical Engineering and Computer Science, 29(1), 598.

Malkov, Y. A., & Yashunin, D. A. (2020). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, 42(4), 824–836.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to information retrieval. Cambridge University Press.

McInnes, L., Healy, J., & Astels, S. (2017). hdbscan: Hierarchical density based clustering. Journal of Open Source Software, 2(11), 205. https://doi.org/10.21105/joss.00205

Medjahed, S. A., Ait Saadi, T., & Benyettou, A. (2013). Breast cancer diagnosis by using k-nearest neighbor with different distances and classification rules. International Journal of Computer Applications, 62(1), 1–5.

Moore, A., & Gray, A. (2003). New algorithms for efficient high-dimensional non-parametric classification. In Advances in Neural Information Processing Systems (Vol. 16). MIT Press.

Mu, J., & Viswanath, P. (2018). All-but-the-top: Simple and effective postprocessing for word representations. In Proceedings of the International Conference on Learning Representations. https://openreview.net/

Nickparvar, M. (n.d.). Brain tumor MRI dataset. Kaggle. Retrieved June 5, 2026, from https://www.kaggle.com/datasets/masoudnickparvar/brain-tumor-mri-dataset

FOR AUTHOR USE ONLY

Nielsen, F., Piro, P., & Barlaud, M. (2009). Tailored Bregman ball trees for effective nearest neighbors. In Proceedings of the 25th European Workshop on Computational Geometry (pp. 29–32).

Obi, J. C. (2023). A comparative study of several classification metrics and their performances on data. World Journal of Advanced Engineering Technology and Sciences, 8(1), 308–314.

Onkamo, P., Väänänen, S., Karvonen, M., & Tuomilehto, J. (1999). Worldwide increase in incidence of Type I diabetes: The analysis of the data on published incidence trends. Diabetologia, 42(12), 1395–1403. https://doi.org/10.1007/s001250051309

Parkhi, O. M., Vedaldi, A., & Zisserman, A. (2015). Deep face recognition. In Proceedings of the British Machine Vision Conference. BMVA Press.

Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 2(11), 559–572.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O., Blondel M., Prettenhofer P., Weiss R., Dubourg V., Vanderplas J., Passos A., Cournapeau D., Brucher M., Perrot M., & Duchesnay E. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830.

Pichaivel, M., Anbumani, G., Theivendren, P., & Gopal, M. (2022). An overview of brain tumor.

Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A., & Gulin, A. (2018). CatBoost: Unbiased boosting with categorical features. In Advances in Neural Information Processing Systems (Vol. 31, pp. 6638–6648). Curran Associates.

Quinlan, J. R. (1986). Induction of decision trees. Machine Learning, 1(1), 81–106. https://doi.org/10.1007/BF00116251

Rajaguru, H., & Chakravarthy, S. R. (2019). Analysis of decision tree and k-nearest neighbor algorithm in the classification of breast cancer. Asian Pacific

FOR AUTHOR USE ONLY

Journal of Cancer Prevention, 20(12), 3777–3781. https://doi.org/10.31557/APJCP.2019.20.12.3777

Rakotondrasoa, H. M., Bucher, M., & Sinayskiy, I. (2023). Quantitative comparison of nearest neighbor search algorithms. arXiv. https://arxiv.org/abs/2307.05235

Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (pp. 3982–3992). Association for Computational Linguistics.

Ringnér, M. (2008). What is principal component analysis? Nature Biotechnology, 26, 303–304.

Salton, G., & McGill, M. J. (1983). Introduction to modern information retrieval. McGraw-Hill.

Schlegel, A. (2017). Image Compression with Principal Component Analysis.

Schroff, F., Kalenichenko, D., & Philbin, J. (2015). FaceNet: A unified embedding for face recognition and clustering. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 815–823). IEEE. https://doi.org/10.1109/CVPR.2015.7298682

Schubert, E., Sander, J., Ester, M., Kriegel, H.-P., & Xu, X. (2017). DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. ACM Transactions on Database Systems, 42(3), Article 19. https://doi.org/10.1145/3068335

scikit-learn. (n.d.). Documentation: PCA, IncrementalPCA, KernelPCA. https://scikit-learn.org/

Shlens, J. (2014). A Tutorial on Principal Component Analysis. arXiv.

Sinaga, K. P., & Yang, M.-S. (2020). Unsupervised k-means clustering algorithm. IEEE Access, 8, 80716–80727. https://doi.org/10.1109/ACCESS.2020.2988796

Sokolova, M., & Lapalme, G. (2009). A system analysis of performance measures for classification tasks. Information Processing & Management, 45(4), 427–437. https://doi.org/10.1016/j.ipm.2009.03.002

Srinivasan, S., Gunasekaran, S., Mathivanan, S. K., Jayagopal, P., & Dalu, G. T. (2023). An active-learning machine technique-based prediction of

FOR AUTHOR USE ONLY

cardiovascular heart disease from the UCI repository database. Scientific Reports, 13(1), 13588.

Van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., & Yu, T. (2014). scikit-image: Image processing in Python. PeerJ, 2, e453.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. In Advances in Neural Information Processing Systems (Vol. 30). Curran Associates.

Wan, W., & Lee, H. J. (2020). Deep feature representation and ball-tree for face sketch recognition. International Journal of System Assurance Engineering and Management, 11, 818–823. https://doi.org/10.1007/s13198-019-00882-x

Wang, F., & Liu, H. (2011). The hubness phenomenon in high-dimensional spaces. In Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM.

Wolberg, W. H., Street, W. N., & Mangasarian, O. L. (1995). Breast cancer Wisconsin (diagnostic) dataset. UCI Machine Learning Repository.

Wu, X., & Kumar, V. (2009). The top ten algorithms in data mining. Chapman & Hall/CRC.

Xu, Y., Liu, P., & Gong, M. (2023). Discovering new intents with deep aligned clustering. Proceedings of the AAAI Conference on Artificial Intelligence, 37(11), 13637–13645. https://doi.org/10.1609/aaai.v37i11.26674

Yashu, V., & Kukreja, V. (2025). Hybrid deep learning and machine learning for brain tumor detection: CNN meets random forest. Proceedings of the 6th International Conference (IEEE).

Yfantis, V., Wagner, A., & Ruskowski, M. (2025). Federated k-means clustering via dual decomposition-based distributed optimization. Franklin Open, 10, Article 100204. https://doi.org/10.1016/j.fropen.2025.100204

Zhang, D., Xu, R., & Lin, Z. (2021). Supporting clustering with contrastive learning. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (pp. 5419–5430). Association for Computational Linguistics.

Zhang, S. (2022). Challenges in kNN classification. IEEE Transactions on Knowledge and Data Engineering, 34(10), 4663–4675.

FOR AUTHOR USE ONLY

Zhang, S., Deng, Z., Cheng, D., Zong, M., & Zhu, X. (2016). Efficient kNN classification algorithm for big data. Neurocomputing, 195, 143–148.

Zhang, S., Li, X., Zong, M., Zhu, X., & Cheng, D. (2017). Learning k for kNN classification. ACM Transactions on Intelligent Systems and Technology, 8(3), 1–19.

Zhang, S., Li, X., Zong, M., Zhu, X., & Wang, R. (2018). Efficient kNN classification with different numbers of nearest neighbors. IEEE Transactions on Neural Networks and Learning Systems, 29(5), 1774–1785.

Zheng, J., Lin, D., Gao, Z., Wang, S., He, M., & Fan, J. (2020). Deep learning-assisted efficient AdaBoost algorithm for breast-cancer detection and early diagnosis. IEEE Access, 8, 96946–96954.

Zubair, M., Iqbal, M. A., Shil, A., Chowdhury, M. J. M., Moni, M. A., & Sarker, I. H. (2024). An improved k-means clustering algorithm towards an efficient data-driven modeling. Annals of Data Science, 11(5), 1525–1544. https://doi.org/10.1007/s40745-022-00428-2

FOR AUTHOR USE ONLY

Acknowledgement of AI-Assisted Editing:

Generative AI (OpenAI ChatGPT) was used solely to assist with language editing, stylistic refinement, consistency checking, and the preparation and enhancement of selected figures and tables. The author independently reviewed, verified, and approved all content and assumes full responsibility for the final manuscript.



FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY



Buy your books fast and straightforward online - at one of world's fastest growing online book stores! Environmentally sound due to Print-on-Demand technologies.

Buy your books online at
www.morebooks.shop

Kaufen Sie Ihre Bücher schnell und unkompliziert online – auf einer der am schnellsten wachsenden Buchhandelsplattformen weltweit! Dank Print-On-Demand umwelt- und ressourcenschonend produzi ert.

Bücher schneller online kaufen
www.morebooks.shop

info@omniscriptum.com
www.omniscriptum.com

OMNIScriptum

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY