FPT UNIVERSITY QUY NHON

FPT UNIVERSITY

INTRODUCTION TO MACHINE LEARNING - AIL303m

---

**Project:**

# Red Wine Quality Analysis Project


**Instructor:** Nguyen An Khuong

**Team Members:**

| | | |
|---|---|---|
| Nguyen Le Anh Duy (QE190134) | – | Leader |
| Than Phuc Hau (QE190002) | – | Member |
| Vo Minh Huy (QE190059) | – | Member |
| Vo Quang Truong (QE190029) | – | Member |
| Chau Thai Nhat Minh (QE190109) | – | Member |


**Quy Nhon, November 2025**

# Contents

# Chapter 1

# Team Introduction

## Team Members and Roles

| No | Full name | Role - Task |
|---|---|---|
| 1 | Nguyen Le Anh Duy | Leader: Linear Regression, Polynomial Regression, Regularized Regression, Process Management |
| 2 | Than Phuc Hau | Member: EDA,Ensemble - Stacking, Imbalance Handling Techniques, K-Means Clustering, Report |
| 3 | Vo Minh Huy | Member: EDA, Logistic Regression, K-Nearest Neighbors (KNN), Support Vector Machines (SVM) |
| 4 | Vo Quang Truong | Member: Hierarchical Agglomerative Clustering, DBSCAN, Principal Component Analysis (PCA), Report |
| 5 | Chau Thai Nhat Minh | Member: Decision Trees, Ensemble - Bagging, Ensemble - Boosting, Report |

Table 1.1: Team members and Roles

# Chapter 2

# Introduction

## 2.1 Motivation

In the modern beverage industry, Wine Quality assessment remains a crucial yet challenging task.

Traditionally, determining wine quality relies heavily on human expertise and sensory evaluation, which are inherently subjective and time-consuming.

With the rise of data-driven approaches and machine learning, it has become possible to predict wine quality more accurately and objectively based on measurable physicochemical properties.

This project is motivated by the desire to apply artificial intelligence and statistical learning techniques to bridge the gap between traditional wine tasting and automated quality prediction, thereby enhancing consistency, efficiency, and reliability in quality control processes.

## 2.2 Objectives

The primary objective of this project is to develop and evaluate predictive models capable of classifying wine quality using physicochemical attributes. Specifically, the project aims to:

1. Analyze the correlation between various chemical characteristics (e.g., acidity, alcohol concentration, sugar content) and wine quality scores.

2. Build and compare multiple machine learning models to predict wine quality, such as Logistic Regression, Random Forest, and Gradient Boosting.

3. Optimize model performance using appropriate feature selection and hyperparameter tuning techniques.

4. Provide visual insights and interpretability to assist winemakers and researchers in understanding key quality determinants.

Through this, the project seeks to demonstrate how data science can effectively support decision-making in viticulture and the wine production process.

## 2.3 Scope

The scope of this project is confined to the analysis of the Wine Quality Dataset, publicly available from the UCI Machine Learning Repository.

The dataset includes physicochemical test results for red and white wine samples, each rated on a quality scale from 0 to 10. The project focuses on:

1. Data preprocessing, feature exploration, and visualization.
2. Supervised learning techniques for classification and regression tasks.
3. Unsupervised learning techniques for clustering and dimensionality reduction.
4. Model evaluation using performance metrics such as accuracy, F1-score, and confusion matrix.

However, this study does not include sensory data, pricing analysis, or real-time quality monitoring.

The results should therefore be interpreted within the context of experimental data rather than real industrial production.

## 2.4 Report Structure

This report is structured as follows:

**Section 1:** Introduces the project background and context.

**Section 2:** Outlines the motivation, objectives, scope, and structure of the report.

**Section 3:** Describes the dataset, preprocessing methods, and exploratory data analysis.

**Section 4:** Details the methodology, including model selection and training process.

**Section 5:** presents the experimental results, model evaluation, and discussion.

**Section 6:** concludes the project with key findings, limitations, and future research directions.

# Chapter 3

# Data Understanding & Exploratory Data Analysis (EDA)

## 3.1 Dataset Description

The dataset **Red Wine Quality** used in this study corresponds to the red variant of the Portuguese "Vinho Verde" wine.

It comprises 12 variables in total, including one target variable representing the sensory quality score of each wine sample.

The input variables are based on physicochemical laboratory tests, while the target variable is derived from sensory evaluation.

Input features (physicochemical tests):

1. Fixed acidity
2. Volatile acidity
3. Citric acid
4. Residual sugar
5. Chlorides
6. Free sulfur dioxide
7. Total sulfur dioxide
8. Density
9. pH
10. Sulphates
11. Alcohol

Target feature (sensory data):

12. Quality (score between 0 and 10)

The first five rows of the dataset are displayed in Table 3.1 to illustrate its structure.

| | fixed acidity | volatile acidity | citric acid | residual sugar | chlorides | free sulfur dioxide | total sulfur dioxide | density | pH | sulphates | alcohol | quality |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 7.4 | 0.700 | 0.00 | 1.9 | 0.076 | 11.0 | 34.0 | 0.99780 | 3.51 | 0.56 | 9.4 | 5 |
| 1 | 7.8 | 0.880 | 0.00 | 2.6 | 0.098 | 25.0 | 67.0 | 0.99680 | 3.20 | 0.68 | 9.8 | 5 |
| 2 | 7.8 | 0.760 | 0.04 | 2.3 | 0.092 | 15.0 | 54.0 | 0.99700 | 3.26 | 0.65 | 9.8 | 5 |
| 3 | 11.2 | 0.280 | 0.56 | 1.9 | 0.075 | 17.0 | 60.0 | 0.99800 | 3.16 | 0.58 | 9.8 | 6 |
| 4 | 7.4 | 0.700 | 0.00 | 1.9 | 0.076 | 11.0 | 34.0 | 0.99780 | 3.51 | 0.56 | 9.4 | 5 |

Figure 3.1: First 5 rows of Red Wine Quality Dataset

## 3.2 Summary Statistics

| | fixed acidity | volatile acidity | citric acid | residual sugar | chlorides | free sulfur dioxide | total sulfur dioxide | density | pH | sulphates | alcohol | quality |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 1599.000000 | 1599.000000 | 1599.000000 | 1599.000000 | 1599.000000 | 1599.000000 | 1599.000000 | 1599.000000 | 1599.000000 | 1599.000000 | 1599.000000 | 1599.000000 |
| mean | 8.319637 | 0.527821 | 0.270976 | 2.538806 | 0.087467 | 15.874922 | 46.467792 | 0.996747 | 3.311113 | 0.658149 | 10.422983 | 5.636023 |
| std | 1.741096 | 0.179060 | 0.194801 | 1.409928 | 0.047065 | 10.460157 | 32.895324 | 0.001887 | 0.154386 | 0.169507 | 1.065668 | 0.807569 |
| min | 4.600000 | 0.120000 | 0.000000 | 0.900000 | 0.012000 | 1.000000 | 6.000000 | 0.990070 | 2.740000 | 0.330000 | 8.400000 | 3.000000 |
| 25% | 7.100000 | 0.390000 | 0.090000 | 1.900000 | 0.070000 | 7.000000 | 22.000000 | 0.995600 | 3.210000 | 0.550000 | 9.500000 | 5.000000 |
| 50% | 7.900000 | 0.520000 | 0.260000 | 2.200000 | 0.079000 | 14.000000 | 38.000000 | 0.996750 | 3.310000 | 0.620000 | 10.200000 | 6.000000 |
| 75% | 9.200000 | 0.640000 | 0.420000 | 2.600000 | 0.090000 | 21.000000 | 62.000000 | 0.997835 | 3.400000 | 0.730000 | 11.100000 | 6.000000 |
| max | 15.900000 | 1.580000 | 1.000000 | 15.500000 | 0.611000 | 72.000000 | 289.000000 | 1.003690 | 4.010000 | 2.000000 | 14.900000 | 8.000000 |

Figure 3.2: Descriptive Statistics of Red Wine Quality Dataset

The descriptive statistics in Figure 3.2 reveal several key characteristics. The target variable, `quality`, is a discrete score ranging from 3 to 8, with a mean of approximately 5.64. This indicates a skew, as no wines in this dataset were rated at the extremes (0-2 or 9-10).

Crucially, the dataset is imbalanced, with the majority of wines clustered in the 'average' quality scores of 5 and 6 (as confirmed by Figure 3.4). This imbalance poses a significant challenge for classification models.

For the input features, we observe a wide range of scales, from `chlorides` (mean $\approx$ 0.087) and `pH` (mean $\approx$ 3.31) to `total sulfur dioxide` (mean $\approx$ 46.47). Many features, such as `residual sugar` and `chlorides`, also show a large standard deviation relative to their mean, and the large difference between the 75th percentile and the max value suggests the presence of outliers. This variance in scales and the existence of outliers strongly justify the need for feature scaling and outlier handling, as discussed in Chapter 4.

## 3.3 Exploratory Visualizations

A series of exploratory visualizations were performed to gain insights into the characteristics of the wine quality dataset.

Univariate plots (such as histograms and boxplots) were used to observe the distribution and potential outliers of individual features, while multivariate plots (such as correlation heatmaps) were applied to identify inter-feature relationships.

As shown in Figures 3.6, several chemical attributes such as alcohol, volatile acidity, and sulphates exhibit notable correlations with wine quality.

Specifically, `alcohol` shows the strongest positive correlation with quality (approx. 0.48), suggesting higher alcohol content is generally associated with higher quality ratings. Conversely, `volatile acidity` demonstrates a strong negative correlation (approx. -0.39), indicating that lower levels of this acid are preferred. `Sulphates` (approx. 0.25) and `citric acid` (approx. 0.23) also show a moderate positive correlation. The target variable's distribution (Figure 3.4) confirms the imbalance, with most samples scoring 5 or 6. This imbalance is a critical challenge that will be addressed in our classification modeling (Chapter 5, Model 11).

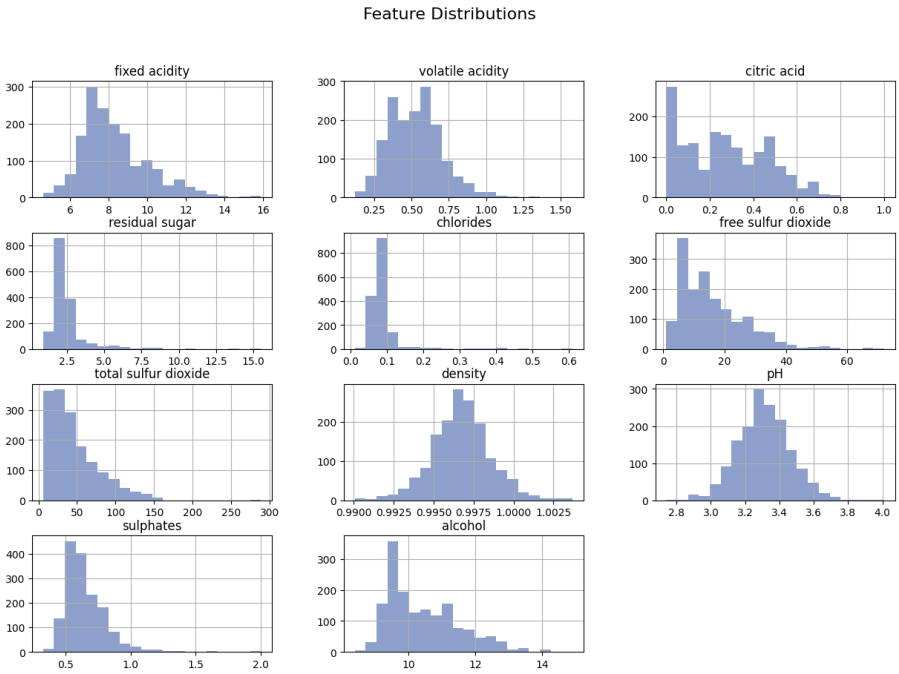



Figure 3.3: Histogram of Feature Distributions

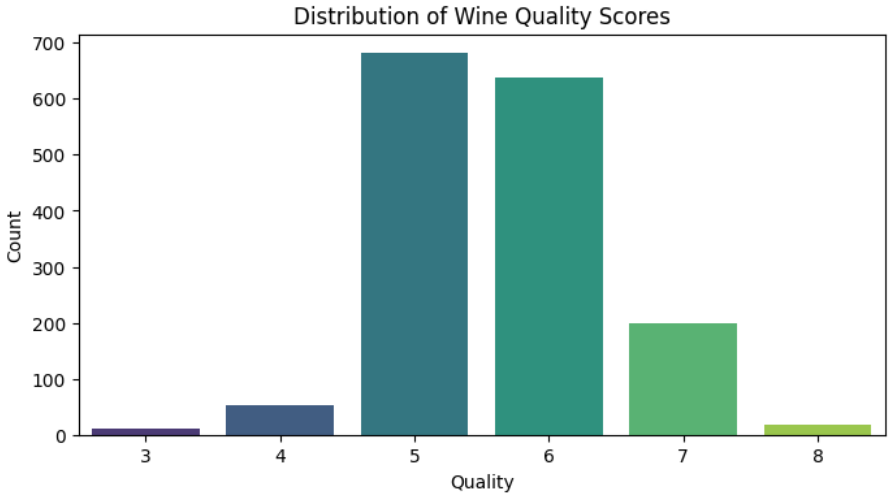



Figure 3.4: Distribution of Wine Quality Score

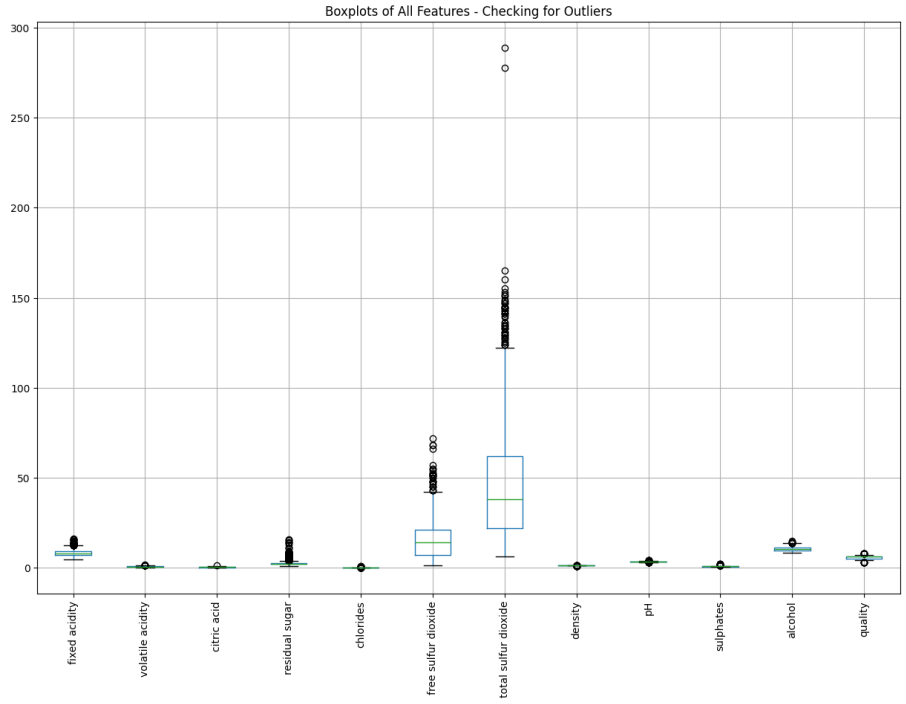



Figure 3.5: Plot Box of all Features (Before Outlier Removal)

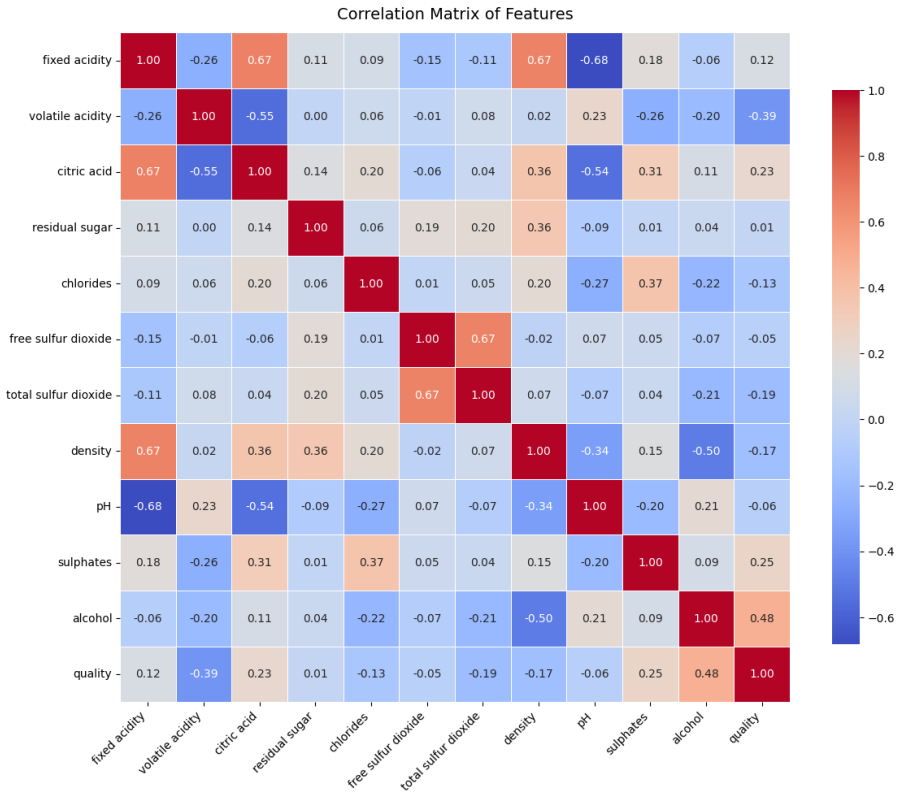



Figure 3.6: Correlation Matrix of Features

Additional detailed visualizations are provided in Appendix A for reference.

# Chapter 4

# Data Preprocessing

## 4.1 Data Cleaning

### 4.1.1 Checking for Missing Data

Since the dataset does not contain any identifier column, duplicate ID checking was not required. To verify data completeness, missing value detection was performed using the command `df.isnull().sum()`.

```
df.isnull().sum()
```

| | 0 |
|---|---|
| fixed acidity | 0 |
| volatile acidity | 0 |
| citric acid | 0 |
| residual sugar | 0 |
| chlorides | 0 |
| free sulfur dioxide | 0 |
| total sulfur dioxide | 0 |
| density | 0 |
| pH | 0 |
| sulphates | 0 |
| alcohol | 0 |
| quality | 0 |

**dtype:** int64

Figure 4.1: Checking for Missing Data

The results showed that there were no missing or null values, indicating that no imputation or missing-value handling techniques were necessary.

### 4.1.2 Handling Outliers

The box plot visualization (Figure 3.5) revealed the presence of numerous outliers among several numerical features. These were addressed by removing data points lying outside the 1.5 interquartile range (IQR). The target variable `quality` was excluded from this process to preserve minority classes and prevent class imbalance. The box plots after outlier removal are presented in Figure 4.2.

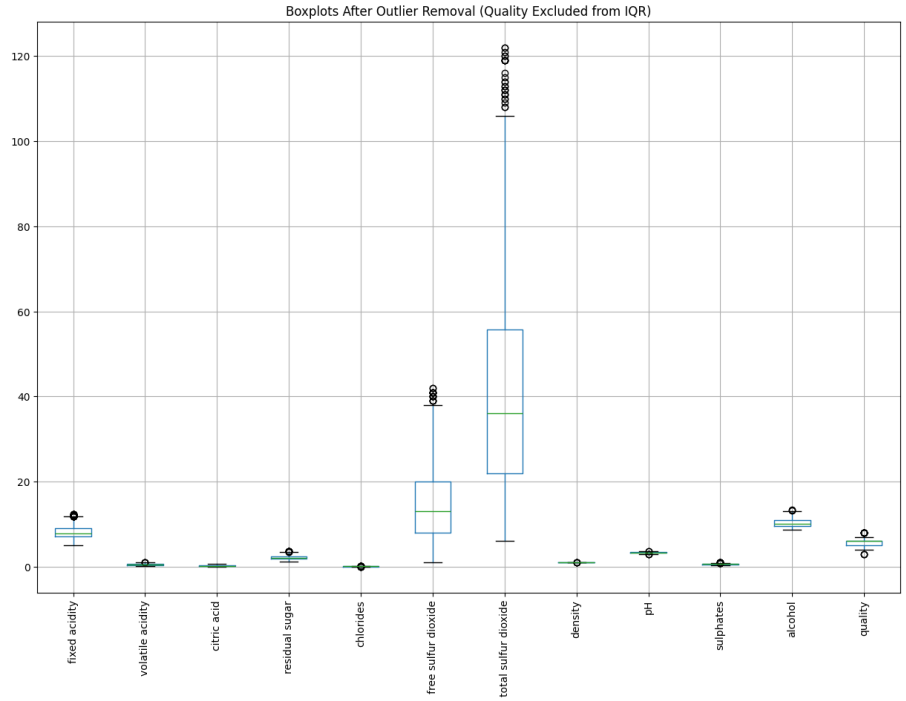



Figure 4.2: Box Plot after Outliers Removal

## 4.2 Scaling

As the input features are measured in different physical units, their numerical scales vary significantly. To ensure comparability across features and improve model training performance (especially for distance-based algorithms like KNN, SVM, and K-Means), all input variables were normalized to a common scale.

Given that the feature distributions are approximately normal (post-outlier removal), the **StandardScaler** method was employed. This method transforms each feature to have a mean of 0 and a standard deviation of 1. The target variable was excluded from this transformation.

The results of the scaling process are summarized in Table 4.3.

| | fixed acidity | volatile acidity | citric acid | residual sugar | chlorides | free sulfur dioxide | total sulfur dioxide | density | pH | sulphates | alcohol | quality |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | -0.524869 | 1.075573 | -1.377452 | -0.646558 | -0.173311 | -0.454417 | -0.312763 | 0.766056 | 1.408196 | -0.621576 | -0.989805 | 5 |
| 1 | -0.250994 | 2.167738 | -1.377452 | 0.937243 | 1.365747 | 1.133715 | 0.952872 | 0.142508 | -0.938945 | 0.411596 | -0.579638 | 5 |
| 2 | -0.250994 | 1.439628 | -1.155393 | 0.258471 | 0.946004 | -0.000665 | 0.454289 | 0.267217 | -0.484660 | 0.153303 | -0.579638 | 5 |
| 3 | 2.076938 | -1.472812 | 1.731370 | -0.646558 | -0.243268 | 0.226211 | 0.684404 | 0.890766 | -1.241802 | -0.449381 | -0.579638 | 6 |
| 4 | -0.524869 | 1.075573 | -1.377452 | -0.646558 | -0.173311 | -0.454417 | -0.312763 | 0.766056 | 1.408196 | -0.621576 | -0.989805 | 5 |

Figure 4.3: Features after Scaling by Standard Scaler

## 4.3 Train-Test Split

After removing outliers, the dataset comprised 1,184 observations and 12 attributes, including one target variable representing wine quality. To ensure both effective training and unbiased evaluation, the dataset was divided using an 80:20 ratio. The splitting procedure is summarized as follows:

- **Training set:** 947 samples (80% of the data) were used for model training and parameter estimation.
- **Testing set:** 237 samples (20% of the data) were held out for model evaluation on unseen instances.

This proportion is widely adopted in machine learning research as it achieves a balance between learning capacity and evaluation reliability:

- The training set size is sufficiently large to allow the model to capture the underlying feature–target relationships.
- The test set provides a statistically meaningful basis for assessing the model's generalization ability.

Furthermore, a **stratified sampling strategy** was applied to maintain the original class distribution of the target variable across both subsets.

This is crucial given the imbalanced nature of the `quality` score, ensuring that each wine quality category is proportionally represented in both training and testing, thus preventing sampling bias.

# Chapter 5

# Modeling & Implementation

## 5.1 Overview of Modeling

This project implements 15 distinct models categorized into three main groups of mathematical problems:

- **Regression:** Predicting the continuous quality score (e.g., 5.5, 6.1). Models 1, 2, 3.
- **Classification:** Predicting a discrete quality class (e.g., 'low', 'medium', 'high', or binary 'good'/'bad'). Models 4-11.
- **Unsupervised Learning:** Finding inherent structures or groups in the data without using the quality label. Models 12-15.

## 5.2 Mathematical Foundations

This section provides an in-depth mathematical overview of the algorithms applied in this project. It begins with the mathematical background necessary to understand the models, followed by detailed derivations and explanations for each of the 15 models used.

### 5.2.1 Mathematical Background

Before diving into each algorithm, we briefly recall several core mathematical tools commonly used in machine learning:

#### Linear Algebra

Let $X \in \mathbb{R}^{n \times p}$ denote the feature matrix, where $n$ is the number of samples and $p$ the number of features. A single observation is a vector $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})^T$.

- **Dot Product:** For vectors $a, b \in \mathbb{R}^p$, $a^T b = \sum_{i=1}^{p} a_i b_i$.
- **Matrix Multiplication:** $(AB)_{ij} = \sum_k A_{ik} B_{kj}$.
- **Norms:** $\|x\|_2 = \sqrt{\sum_i x_i^2}$ (L2-norm), $\|x\|_1 = \sum_i |x_i|$ (L1-norm).
- **Covariance Matrix:** $\Sigma = \frac{1}{n-1} X^T X$ (after centering $X$).

**Probability and Statistics**

Machine learning relies heavily on probability theory:

- **Expectation:** $\mathbb{E}[X] = \sum_x xP(X = x)$ or $\int xf(x)dx$.
- **Variance:** $\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$.
- **Covariance:** $\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$.
- **Bayes' Theorem:** $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$.

**Optimization Principles**

Most models minimize a loss function (or maximize a likelihood):

$$\min_{\theta} L(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(x_i)) + \lambda \Omega(\theta)$$

where $\ell$ is the loss function (e.g., squared error or log-loss), and $\Omega(\theta)$ is a regularization term controlling model complexity, with $\lambda$ as the regularization strength. —

### 5.2.2 1. Linear Regression

Linear Regression assumes a linear relationship between input features $X$ and the continuous target variable $y$:

$$y = X\beta + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2 I)$$

The parameters $\beta$ (coefficients) are estimated by minimizing the sum of squared residuals, known as Ordinary Least Squares (OLS):

$$\min_{\beta} L(\beta) = \min_{\beta} \|y - X\beta\|_2^2$$

The closed-form solution is found by setting the gradient $\nabla_\beta L = 0$, which yields:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

This solution is feasible when $X^T X$ is invertible (i.e., features are not perfectly collinear). —

### 5.2.3 2. Polynomial Regression

Polynomial Regression extends linear regression by adding polynomial features (e.g., $x^2, x^3$) to capture non-linear relationships:

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_d x^d$$

This is still a linear model in terms of the coefficients $\beta$. In matrix form:

$$y = Z\beta + \epsilon$$

where $Z = [1, x, x^2, \ldots, x^d]$ represents the polynomial feature expansion. The OLS solution is $\hat{\beta} = (Z^T Z)^{-1} Z^T y$. —

### 5.2.4 3. Regularized Regression (Ridge, Lasso, Elastic Net)

These methods add a penalty term $\Omega(\beta)$ to the OLS loss function to prevent overfitting and handle multicollinearity.

#### Ridge Regression (L2 Regularization)

Adds an L2 penalty ($\|\beta\|_2^2$), which shrinks coefficients towards zero but does not set them exactly to zero.

$$\min_{\beta} \|y - X\beta\|_2^2 + \lambda \|\beta\|_2^2$$

Closed form solution:

$$\hat{\beta}_{ridge} = (X^T X + \lambda I)^{-1} X^T y$$

where $I$ is the identity matrix and $\lambda > 0$ is the tuning parameter.

#### Lasso Regression (L1 Regularization)

Adds an L1 penalty ($\|\beta\|_1$), which performs feature selection by forcing some coefficients to be exactly zero.

$$\min_{\beta} \|y - X\beta\|_2^2 + \lambda \|\beta\|_1$$

No closed-form solution exists; it's solved using optimization algorithms like Coordinate Descent.

**Elastic Net**

Combines L1 and L2 penalties, offering a balance between Ridge and Lasso.

$$\min_{\beta} \|y - X\beta\|_2^2 + \lambda_1 \|\beta\|_1 + \lambda_2 \|\beta\|_2^2$$

—

### 5.2.5 4. Logistic Regression

Logistic Regression is used for classification. It models the probability that an observation $x$ belongs to a class (e.g., $y = 1$) using the sigmoid (logistic) function:

$$P(y = 1|x) = \sigma(\beta_0 + \beta^T x) = \frac{1}{1 + e^{-(\beta_0 + \beta^T x)}}$$

The parameters $\beta$ are estimated by maximizing the log-likelihood of the data, which is equivalent to minimizing the negative log-likelihood (or binary cross-entropy) loss function:

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

where $\hat{p}_i = \sigma(\beta_0 + \beta^T x_i)$. Gradient-based optimization (e.g., gradient descent) is used to find $\beta$. —

### 5.2.6 5. K-Nearest Neighbors (KNN)

KNN is a non-parametric, instance-based algorithm. It classifies a new data point based on the majority class of its $k$ nearest neighbors in the feature space.

- **Distance Metric:** Typically Euclidean distance: $d(x_a, x_b) = \sqrt{\sum_j (x_{aj} - x_{bj})^2}$.
- **Classification:** The predicted class $\hat{y}$ for a new point $x$ is the most common class among its $k$ neighbors $N_k(x)$:

$$\hat{y}(x) = \arg\max_{c} \sum_{i \in N_k(x)} \mathbf{1}(y_i = c)$$

- **Regression:** The prediction is the average value of the $k$ neighbors:

$$\hat{y}(x) = \frac{1}{k} \sum_{i \in N_k(x)} y_i$$

—

### 5.2.7 6. Support Vector Machine (SVM)

SVM seeks to find the optimal hyperplane that maximally separates two classes. The hyperplane is defined as $w^T x + b = 0$.

The optimal hyperplane (in the linearly separable case) is found by solving the "hard-margin" optimization problem:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{subject to } y_i(w^T x_i + b) \geq 1, \forall i$$

For non-linearly separable data, a "soft-margin" is used with slack variables $\xi_i$:

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_i \xi_i \quad \text{s.t. } y_i(w^T x_i + b) \geq 1 - \xi_i, \xi_i \geq 0$$

**Kernel Trick:** For non-linear boundaries, SVM uses the kernel trick, replacing the dot product $x_i^T x_j$ with a kernel function $K(x_i, x_j)$.

- **Linear Kernel:** $K(x_i, x_j) = x_i^T x_j$.
- **RBF Kernel:** $K(x_i, x_j) = \exp(-\gamma\|x_i - x_j\|^2)$, where $\gamma$ is a tuning parameter.

—

### 5.2.8 7. Decision Tree

A Decision Tree is a non-parametric model that recursively partitions the feature space.

Splits are chosen to maximize **Information Gain** (or minimize impurity).

**Gini Impurity:** Measures the probability of misclassifying a randomly chosen element from the set $S$.

$$Gini(S) = 1 - \sum_c p_c^2$$

**Entropy:** Measures the level of disorder in a set $S$.

$$H(S) = -\sum_c p_c \log_2 p_c$$

where $p_c$ is the proportion of samples belonging to class $c$. The Information Gain for a split on feature $A$ is:

$$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v)$$

The tree construction stops when a stopping criterion is met (e.g., max depth, min samples per leaf). —

### 5.2.9 8. Ensemble Methods — Bagging (Random Forest)

Random Forest is an ensemble of Decision Trees trained via **bagging** (Bootstrap Aggregating).

It builds $B$ decision trees on $B$ different bootstrap samples (samples drawn with replacement) from the training data.

To de-correlate the trees, each split in a tree only considers a random subset of $m$ features ($m < p$).

The final prediction is the average (for regression) or majority vote (for classification) of all $B$ trees.

$$\hat{f}_{rf}(x) = \frac{1}{B}\sum_{b=1}^{B} f_b(x) \quad \text{(Regression)}$$

This process significantly reduces variance compared to a single tree.

—

### 5.2.10 9. Ensemble Methods — Boosting (e.g., XGBoost)

Boosting builds an ensemble of weak learners (typically trees) sequentially. Each new model is trained to correct the errors made by the previous models. The model is additive:

$$\hat{y}_i^{(M)} = \sum_{m=1}^{M} f_m(x_i) = \hat{y}_i^{(M-1)} + f_M(x_i)$$

where $f_M$ is the new tree that minimizes a loss function $L$ based on the residuals of the previous model $\hat{y}_i^{(M-1)}$.

XGBoost uses a more advanced objective function that includes a regularization term $\Omega$ and is optimized using second-order gradients (Taylor expansion):

$$L^{(t)} \approx \sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \Omega(f_t)$$

where $g_i$ and $h_i$ are the first and second-order gradients of the loss function, and $\Omega(f_t) = \gamma T + \frac{1}{2}\lambda\sum_j w_j^2$ penalizes tree complexity.

—

### 5.2.11 10. Ensemble Methods — Stacking

Stacking combines multiple heterogeneous base learners (Level-0 models) by training a **meta-model** (Level-1 model) on their predictions.

1. Split the training data into $K$ folds.

2. For each fold $k$, train base learners $\{f_1, ..., f_K\}$ on the other $K-1$ folds.
3. Generate predictions ("out-of-fold" predictions) for fold $k$.
4. Combine the out-of-fold predictions to create a new training set for the meta-model.
5. Train the meta-model (e.g., Logistic Regression or Ridge) on this new dataset.

The final prediction $\hat{y}$ is:

$$\hat{y} = g(f_1(x), f_2(x), ..., f_K(x))$$

where $g$ is the trained meta-model.

—

### 5.2.12 11. Imbalance Handling (SMOTE, Undersampling)

**SMOTE (Synthetic Minority Oversampling Technique):**

Addresses class imbalance by creating synthetic samples for the minority class. For each minority instance $x_i$:

1. Find its $k$ nearest minority-class neighbors.
2. Randomly select one neighbor $x_{zi}$.
3. Generate a new synthetic sample $x_{new}$ along the line segment connecting $x_i$ and $x_{zi}$:

$$x_{\text{new}} = x_i + \lambda(x_{zi} - x_i), \quad \lambda \in [0, 1]$$

**Random Undersampling:**

Addresses class imbalance by randomly removing samples from the majority class until the dataset is balanced. This is simpler but risks discarding useful information.

—

### 5.2.13 12. K-Means Clustering

An unsupervised algorithm that partitions data into $k$ clusters. Objective: Minimize the within-cluster sum of squares (WCSS), or "inertia":

$$\min_{C_1,...,C_k} \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|_2^2$$

where $\mu_i$ is the centroid (mean) of cluster $C_i$.

Algorithm (Lloyd's):

1. Initialize $k$ centroids $\mu_i$ randomly.

2. **Assign step:** Assign each point $x$ to the nearest centroid: $C_i = \{x : \|x - \mu_i\|^2 \leq \|x - \mu_j\|^2, \forall j\}$.

3. **Update step:** Recalculate centroids as the mean of points in each cluster: $\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$.

4. Repeat 2-3 until convergence.

—

### 5.2.14 13. Hierarchical Agglomerative Clustering (HAC)

Builds a hierarchy of clusters from the bottom up (agglomerative).

1. Start with each data point as its own cluster.
2. Iteratively merge the two "closest" clusters.
3. Repeat until only one cluster remains.

"Closeness" is defined by a **linkage criterion**:

$$d_{single}(A, B) = \min_{a \in A, b \in B} \|a - b\| \quad \text{(Single Linkage)}$$

$$d_{complete}(A, B) = \max_{a \in A, b \in B} \|a - b\| \quad \text{(Complete Linkage)}$$

$$d_{average}(A, B) = \frac{1}{|A||B|} \sum_{a \in A, b \in B} \|a - b\| \quad \text{(Average Linkage)}$$

$$d_{ward}(A, B) = \text{Increase in WCSS after merging A and B} \quad \text{(Ward's Method)}$$

The result is visualized as a dendrogram.

—

### 5.2.15 14. DBSCAN (Density-Based Spatial Clustering)

Finds clusters of arbitrary shapes by defining them as dense regions of points.

Key parameters:

- $\varepsilon$ (epsilon): The radius of a neighborhood.
- minPts: The minimum number of points required to form a dense region.

Point types:

- **Core point:** A point $x_i$ with at least minPts neighbors within distance $\varepsilon$.

$$|\{x_j : \|x_j - x_i\| \leq \varepsilon\}| \geq \text{minPts}$$

- **Border point:** A point that is not a core point, but is reachable (within $\varepsilon$) from a core point.
- **Noise point:** Any point that is neither core nor border.

Clusters are formed by connecting all "density-reachable" core points.

—

### 5.2.16 15. Principal Component Analysis (PCA)

A dimensionality reduction technique that projects data onto a new set of orthogonal (uncorrelated) axes called **Principal Components**.

These components are chosen to maximize the variance of the projected data.

1. Center the data $X$ (subtract the mean).
2. Compute the covariance matrix $S = \frac{1}{n-1} X^T X$.
3. Find the eigenvalues $\lambda_i$ and eigenvectors $w_i$ of $S$ by solving:

$$Sw_i = \lambda_i w_i$$

4. The eigenvectors (Principal Components) are ranked by their corresponding eigenvalues (which represent the variance captured).

5. The new $k$-dimensional data $Z$ is obtained by projecting $X$ onto the top $k$ eigenvectors $W_k$:

$$Z = XW_k$$

—

In summary, the mathematical foundations presented above underpin the theoretical and algorithmic aspects of the 15 models applied in this project.

Together, they provide a rigorous framework for analyzing, predicting, and interpreting the wine quality dataset.

## 5.3 Implementation Details and Evaluation Metrics

This section explains the concrete implementation decisions, commented code snippets, and selected evaluation metrics for each of the fifteen models used in this project.

All implementations follow a reproducible pipeline: data cleaning → feature engineering → scaling/encoding → model training (with cross-validation and hyperparameter tuning) → evaluation on held-out test set.

We use Python (`pandas`, `numpy`), `scikit-learn`, and specialized libraries (e.g., `xgboost`, `imblearn`) where appropriate.

### 5.3.1 Common preprocessing and utilities

Before model-specific descriptions, we present shared preprocessing steps and a canonical pipeline template.

**Canonical preprocessing pipeline (Python pseudo-code, commented):**

```
# common imports
import numpy as np
import pandas as pd
from sklearn.model_selection import (train_test_split, GridSearchCV,
                                     cross_val_score, StratifiedKFold)
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.metrics import (r2_score, mean_squared_error, mean_absolute_error,
                             accuracy_score, precision_score, recall_score,
                             f1_score, roc_auc_score, confusion_matrix,
                             silhouette_score, davies_bouldin_score)

# 1) Load data
# df = pd.read_csv('winequality-red.csv')
# (Data cleaning, e.g., outlier removal, would happen here)

# 2) Define features and target
# X = df.drop(['quality'], axis=1)
# y = df['quality']

# 3) Define target for classification
# For binary classification: 'good' (7, 8) vs 'bad' (3, 4, 5, 6)
# y_class = (y >= 7).astype(int)

# 4) Train / test split
# X_train, X_test, y_train, y_test = train_test_split(
#     X, y_class, test_size=0.2, random_state=42, stratify=y_class)

# 5) Numeric preprocessing pipeline
# All features in this dataset are numeric
numeric_features = X.columns
numeric_transformer = Pipeline([
```

```
    # No imputer needed as we checked for nulls
    ('scaler', StandardScaler())
])

preprocessor = ColumnTransformer([
    ('num', numeric_transformer, numeric_features)
])

# Use `preprocessor` inside each model pipeline
```

All model pipelines should incorporate the `preprocessor` above. For reproducibility, we always set `random_state=42` where available.

### 5.3.2 Supervised learning – Regression

Task: predict continuous wine quality score (original 3-8 values). Metrics: **$R^2$, RMSE, MAE**.

#### 1. Linear Regression (Baseline)

**When to use:** Simple, interpretable baseline; useful to inspect coefficients. **Implementation (scikit-learn):**

```
from sklearn.linear_model import LinearRegression

# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate
r2 = r2_score(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```

#### 2. Polynomial Regression

**When to use:** When relationship is nonlinear. **Implementation (use with care to avoid overfitting):**

```python
from sklearn.preprocessing import PolynomialFeatures

# Use GridSearchCV to choose optimal degree

# Create pipeline
best_pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("poly", PolynomialFeatures(degree=best_degree, include_bias=False)),
    ("ridge", Ridge(alpha=best_alpha, max_iter=10000))
])

# Train
best_pipe.fit(X_train, y_train)

# Predict
y_pred_train = best_pipe.predict(X_train)
y_pred_test  = best_pipe.predict(X_test)

# Evaluate
r2  = r2_score(y_true, y_pred)
mae = mean_absolute_error(y_true, y_pred)
mse = mean_squared_error(y_true, y_pred)
rmse= np.sqrt(mse)
```

### 3. Regularized Regression (Ridge / Lasso / ElasticNet)

**When to use:** Multicollinearity, many features, or to perform feature selection (Lasso).
**Implementation example (with CV for $\alpha$):**

```python
from sklearn.linear_model import RidgeCV, LassoCV, ElasticNetCV

# Define models
ridge = Ridge(alpha=1.0)
lasso = Lasso(alpha=0.001)
elastic = ElasticNet(alpha=0.01, l1_ratio=0.5)

# Fit models
ridge.fit(X_train, y_train)
lasso.fit(X_train, y_train)
```

```
elastic.fit(X_train, y_train)

# Predict and evaluate models
def evaluate(model, X_train, X_test, y_train, y_test):
    y_pred_train = model.predict(X_train)
    y_pred_test = model.predict(X_test)

    return {
        "Train R²": r2_score(y_train, y_pred_train),
        "Test R²": r2_score(y_test, y_pred_test),
        "MAE": mean_absolute_error(y_test, y_pred_test),
        "RMSE": np.sqrt(mean_squared_error(y_test, y_pred_test))
    }
```

---

### 5.3.3 Supervised learning – Classification

Task: We convert the target into a binary class: 0 ('bad', score $<$ 7) and 1 ('good', score $>=$ 7). Metrics: **Accuracy, Precision, Recall, F1-score, ROC-AUC, Confusion Matrix**. We use `StratifiedKFold` for all cross-validation.

#### 4. Logistic Regression

**Use:** Interpretable probabilistic baseline for binary classification.

---

```
from sklearn.linear_model import LogisticRegression

# Define model and parameters
lr = LogisticRegression(class_weight='balanced', random_state=42, max_iter=1000)

# Define Stratified K-Fold (e.g., 5 folds)
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Track ROC for plotting
all_fpr = np.linspace(0, 1, 100)
mean_tpr = 0.0
n_classes = len(np.unique(y))
model_classes = np.unique(y)

# Evaluate model
```

```
for fold, (train_idx, val_idx) in enumerate(skf.split(X, y), 1):
    X_train_fold, X_val_fold = X.iloc[train_idx], X.iloc[val_idx]
    y_train_fold, y_val_fold = y.iloc[train_idx], y.iloc[val_idx]

    # Apply SMOTE only on training data
    # Dynamically adjust k_neighbors for SMOTE based on the size of the minority
    ↪  class in the current fold
    min_samples_fold = y_train_fold.value_counts().min()
    k_neighbors_fold = min(5, max(1, min_samples_fold - 1)) # Ensure k_neighbors
    ↪  is at least 1

    sm = SMOTE(random_state=42, k_neighbors=k_neighbors_fold)
    X_res, y_res = sm.fit_resample(X_train_fold, y_train_fold)

    # Train model
    lr.fit(X_res, y_res)
    y_pred_fold = lr.predict(X_val_fold)
    y_prob_fold = lr.predict_proba(X_val_fold)

    # Accuracy & F1
    acc = accuracy_score(y_val_fold, y_pred_fold)
    f1 = f1_score(y_val_fold, y_pred_fold, average='weighted')

    # Compute ROC-AUC safely
    try:
        roc_auc = roc_auc_score(
            y_val_fold,
            y_prob_fold,
            multi_class='ovr',
            average='macro'
        )
    except ValueError:
        roc_auc = np.nan  # If some class missing in fold
```

### 5. K-Nearest Neighbors (KNN)

**Use:** Simple non-parametric method; sensitive to scaling.

```
from sklearn.neighbors import KNeighborsClassifier
```

```
# Initialize and train KNN
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Predictions
y_pred = knn.predict(X_test)

# Accuracy and classification report
print("\n--- Model Evaluation ---")
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

### 6. Support Vector Machines (SVM) — Linear and RBF

**Use:** Powerful for medium-sized datasets, supports kernels.

```
from sklearn.svm import SVC

# SVM with Linear kernel
svm_linear = SVC(kernel='linear', class_weight='balanced', probability=True,
↪  random_state=42)
svm_linear.fit(X_train, y_train)
y_pred_linear = svm_linear.predict(X_test)

# SVM with RBF kernel
svm_rbf = SVC(kernel='rbf', class_weight='balanced', probability=True,
↪  random_state=42)
svm_rbf.fit(X_train, y_train)
y_pred_rbf = svm_rbf.predict(X_test)

# Evaluate models
print(f"\n--- {name} ---")
print(f"Accuracy: {accuracy_score(y_true, y_pred):.4f}")
print(classification_report(y_true, y_pred))
```

### 7. Decision Trees

**Use:** Interpretable rules, but prone to overfitting.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, StratifiedKFold

# Define parameter grid for DecisionTreeClassifier
param_grid_dt = {
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_leaf': [1, 5, 10],
    'criterion': ['gini', 'entropy']
}

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

# Create GridSearchCV object
grid_search_dt = GridSearchCV(
    DecisionTreeClassifier(random_state=42),
    param_grid_dt,
    cv=skf,
    scoring='roc_auc',
    n_jobs=-1,  # Use all available cores
    verbose=1
)

# Fit GridSearchCV to training data
grid_search_dt.fit(X_train, y_train)

# Print best hyperparameters
print("Best hyperparameters for Decision Tree:", grid_search_dt.best_params_)

# Get the best estimator
best_dt_model = grid_search_dt.best_estimator_

# Make predictions on the test set
y_pred_dt = best_dt_model.predict(X_test)
y_pred_proba_dt = best_dt_model.predict_proba(X_test)[:, 1]  # Probabilities for
↪  the positive class

# Calculate evaluation metrics
accuracy_dt = accuracy_score(y_test, y_pred_dt)
```

```
precision_dt = precision_score(y_test, y_pred_dt)
recall_dt = recall_score(y_test, y_pred_dt)
f1_dt = f1_score(y_test, y_pred_dt)
roc_auc_dt = roc_auc_score(y_test, y_pred_proba_dt)
```

---

### 8. Ensemble — Bagging (Random Forest)

**Use:** Reduce variance of single trees, robust default classifier.

---

```
from sklearn.ensemble import RandomForestClassifier

# Define parameter grid for RandomForestClassifier
param_grid_rf = {
    'n_estimators': [100, 200, 300],
    'max_depth': [5, 10, None],
    'min_samples_leaf': [1, 5, 10]
}

# Initialize StratifiedKFold (already done in previous step, reusing skf)
# skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Create GridSearchCV object
grid_search_rf = GridSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid_rf,
    cv=skf,
    scoring='roc_auc',
    n_jobs=-1,  # Use all available cores
    verbose=1
)

# Fit GridSearchCV to training data
grid_search_rf.fit(X_train, y_train)

# Print best hyperparameters
print("Best hyperparameters for Random Forest:", grid_search_rf.best_params_)

# Get the best estimator
best_rf_model = grid_search_rf.best_estimator_
```

```
# Make predictions on the test set
y_pred_rf = best_rf_model.predict(X_test)
y_pred_proba_rf = best_rf_model.predict_proba(X_test)[:, 1]  # Probabilities for
↪  the positive class

# Calculate evaluation metrics
accuracy_rf = accuracy_score(y_test, y_pred_rf)
precision_rf = precision_score(y_test, y_pred_rf)
recall_rf = recall_score(y_test, y_pred_rf)
f1_rf = f1_score(y_test, y_pred_rf)
roc_auc_rf = roc_auc_score(y_test, y_pred_proba_rf)
```

---

### 9. Ensemble — Boosting (Gradient Boosting / XGBoost)

**Use:** High-performance for tabular data; often top performer.

---

```
from xgboost import XGBClassifier

# Define parameter grid for GradientBoostingClassifier
param_grid_gb = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7]
}

# Initialize StratifiedKFold (already done in previous step, reusing skf)
# skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Create GridSearchCV object
grid_search_gb = GridSearchCV(
    GradientBoostingClassifier(random_state=42),
    param_grid_gb,
    cv=skf,
    scoring='roc_auc',
    n_jobs=-1,  # Use all available cores
    verbose=1
)
```

```
# Fit GridSearchCV to training data
grid_search_gb.fit(X_train, y_train)

# Print best hyperparameters
print("Best hyperparameters for Gradient Boosting:",
↪  grid_search_gb.best_params_)

# Get the best estimator
best_gb_model = grid_search_gb.best_estimator_

# Make predictions on the test set
y_pred_gb = best_gb_model.predict(X_test)
y_pred_proba_gb = best_gb_model.predict_proba(X_test)[:, 1]  # Probabilities for
↪  the positive class

# Calculate evaluation metrics
accuracy_gb = accuracy_score(y_test, y_pred_gb)
precision_gb = precision_score(y_test, y_pred_gb)
recall_gb = recall_score(y_test, y_pred_gb)
f1_gb = f1_score(y_test, y_pred_gb)
roc_auc_gb = roc_auc_score(y_test, y_pred_proba_gb)
```

### 10. Ensemble — Stacking

**Use:** Combine diverse base learners to leverage complementary strengths.

```
from sklearn.linear_model import LogisticRegression, Ridge
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, StackingClassifier
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier

# Single model for baseline comparison
single_models = {
    'LogisticRegression': LogisticRegression(max_iter=2000, random_state=42),
    'RandomForest': RandomForestClassifier(n_estimators=300, random_state=42),
    'DecisionTree': DecisionTreeClassifier(max_depth=6, random_state=42),
    'SVC': svc_pipe,
```

```
    'KNN': knn_pipe,
}

# Build 2 stacks to compare

# Define base learners for stack a (Level 0)
base_a = [
    ('rf', RandomForestClassifier(n_estimators=300, random_state=42)),
    ('xgb', XGBClassifier(eval_metric='logloss', random_state=42,
    ↪  use_label_encoder=False)),
    ('lgbm', LGBMClassifier(random_state=42))
]

# Define meta-learner for stack a (Level 1)
meta_a = LogisticRegression(max_iter=2000, C=0.1, random_state=42)

# Create the Stacking pipeline
# Note: StackingClassifier handles CV internally for the base learners
stack_a = StackingClassifier(
    estimators=base_a,
    final_estimator=meta_a,
    cv=5,
    #passthrough=False,
    n_jobs=-1
)

# Define base learners for stack b (Level 0)
base_b = [
    ('lr', LogisticRegression(max_iter=2000, random_state=42)),
    ('svc', svc_pipe),
    ('knn', knn_pipe),
    ('rf', RandomForestClassifier(n_estimators=200, random_state=42))
]

# Define meta-learner for stack b (Level 1)
meta_b = LogisticRegression(max_iter=2000, C=0.1, random_state=42)

# Create the Stacking pipeline
stack_b = StackingClassifier(
    estimators=base_b,
```

```
    final_estimator=meta_b,
    cv=5,
    passthrough=False,
    n_jobs=-1
)

# Define function to train and evaluate models
#...

# Train and evaluate single models and 2 stacks
evaluate_model(name, model, X_train, X_test, y_train, y_test)
```

### 5.3.4 Handling Imbalanced Data

This model explicitly tests resampling strategies using a pipeline.

#### 11. Imbalance Handling (SMOTE, Random Undersampling)

**Approach:** Use `imblearn` pipelines to resample only the training data.

```
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline

# Define function for evaluating models
#...

# Test SMOTE with a Random Forest classifier
# Create pipeline
smote_pipe = Pipeline([
    ('smote', SMOTE(random_state=42)),
    ('rf', RandomForestClassifier(random_state=42))
])

# Train and predict
smote_pipe.fit(X_train, y_train)
y_pred_smote = smote_pipe.predict(X_test)
y_proba_smote = smote_pipe.predict_proba(X_test)[:,1]
```

```
# Evaluate
smote_results = evaluate_model(y_test, y_pred_smote, y_proba_smote)
# Compare F1-score of this vs. RF with Class Weight and Random Undersampling

# Test with Class Weight
# Define class_weight object
cw_clf = RandomForestClassifier(class_weight='balanced', random_state=42)

# Train and predict
cw_clf.fit(X_train, y_train)
y_pred_cw = cw_clf.predict(X_test)
y_proba_cw = cw_clf.predict_proba(X_test)[:,1]

# Evaluate model
cw_results = evaluate_model(y_test, y_pred_cw, y_proba_cw)

# Test with Random Undersampling
# Create pipeline
under_pipe = Pipeline([
    ('under', RandomUnderSampler(random_state=42)),
    ('rf', RandomForestClassifier(random_state=42))
])

# Train and predict
under_pipe.fit(X_train, y_train)
y_pred_under = under_pipe.predict(X_test)
y_proba_under = under_pipe.predict_proba(X_test)[:,1]

# Evaluate
under_results = evaluate_model(y_test, y_pred_under, y_proba_under)
```

### 5.3.5 Unsupervised learning – Clustering

Task: Find natural groupings in the data (X only). Metrics: **Silhouette Score, Davies-Bouldin Index**. Data must be scaled.

#### 12. K-Means Clustering

**Use:** Partition dataset into $k$ spherical clusters.

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.decomposition import PCA

# Compute inertia and silhouette score for different k values (2-10)
inertias = []
silhouette_scores = []
K_range = range(2, 11)

# Fit KMeans for each k and store evaluation metrics
for k in K_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X)
    labels = kmeans.labels_
    inertias.append(kmeans.inertia_)
    silhouette_scores.append(silhouette_score(X, labels))
```

### 13. Hierarchical Agglomerative Clustering

**Use:** Build a hierarchy of clusters, visualized as a dendrogram.

```
from scipy.cluster.hierarchy import linkage, fcluster, dendrogram
from sklearn.cluster import AgglomerativeClustering

# Plot dendrogram to find optimal k
# The 'ward' method minimizes variance within clusters.
Z = linkage(X, method='ward')

# ---  Plot dendrogram to visualize cluster hierarchy ---
plt.figure(figsize=(12, 6))
dendrogram(Z, truncate_mode='level', p=5)

# --- Apply Agglomerative Clustering (cut the dendrogram into 4 clusters) ---
hac = AgglomerativeClustering(n_clusters=4, linkage='ward')
labels = hac.fit_predict(X)

# --- Evaluate clustering performance ---
sil_score = silhouette_score(X, labels)
db_index = davies_bouldin_score(X, labels)
```

#### 14. DBSCAN

**Use:** Detect clusters of arbitrary shape and mark noise.

```
from sklearn.cluster import DBSCAN

db = DBSCAN(eps=2.5, min_samples=10)
db.fit(X_embedded)

labels = db.labels_
n_clusters = len(set(labels) - {-1})
n_noise = list(labels).count(-1)
```

### 5.3.6 Unsupervised learning – Dimensionality Reduction

#### 15. Principal Component Analysis (PCA)

**Use:** Reduce noise, compress features, visualize.

```
from sklearn.decomposition import PCA
import numpy as np

# Apply PCA to reduce 11 features → 3 principal components
pca = PCA(n_components=3)
X_pca = pca.fit_transform(X)

# Cumulative explained variance
cumulative_variance = np.cumsum(pca_full.explained_variance_ratio_)

# Find number of components to reach 95% variance
n_components_95 = np.argmax(cumulative_variance >= 0.95) + 1

print(f"Number of components to explain 95% variance: {n_components_95}\n")

# Display top 3 principal components and their explained variance ratios
explained_ratios = pca_full.explained_variance_ratio_
```

### 5.3.7 Cross-validation and Hyperparameter Tuning

As shown in the snippets above, `GridSearchCV` with `StratifiedKFold` is our primary tool for hyperparameter tuning.

```python
from sklearn.model_selection import GridSearchCV, StratifiedKFold

# Example for RandomForest
# (Must be inside a pipeline to prevent data leakage during scaling)
pipeline_rf = ImbPipeline([
    ('preproc', preprocessor),
    ('rf', RandomForestClassifier(random_state=42, n_jobs=-1))
])

param_grid = {
    'rf__n_estimators': [100, 200],
    'rf__max_depth': [10, 20, None],
    'rf__min_samples_leaf': [1, 2, 4],
    'rf__class_weight': ['balanced', 'balanced_subsample', None]
}

# Use StratifiedKFold for classification CV
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Optimize for F1-score due to imbalance
grid_rf = GridSearchCV(estimator=pipeline_rf, param_grid=param_grid,
                       cv=cv, scoring='f1', n_jobs=-1, verbose=1)
grid_rf.fit(X_train, y_train)
best_model = grid_rf.best_estimator_
print(f"Best F1-score: {grid_rf.best_score_}")
print(f"Best params: {grid_rf.best_params_}")
```

### 5.3.8 Evaluation Metrics and Reporting Protocols

To ensure rigorous assessment, we adopt the following metrics:

- **Regression:** $\mathbf{R^2}$ (Coefficient of Determination), **RMSE** (Root Mean Squared Error), and **MAE** (Mean Absolute Error).
- **Classification:** Due to the severe class imbalance in our binary target, **Accuracy** is a misleading metric. We will prioritize **F1-score**, **Precision**, **Recall** (especially for the minority 'good' class), and **ROC-AUC**. We will also present **Confusion Matrices**.
- **Clustering: Silhouette Score** (measures cluster cohesion vs. separation, higher

is better, -1 to 1) and **Davies–Bouldin Index** (ratio of within-cluster scatter to between-cluster separation, lower is better, 0 to $\infty$).

- **Dimensionality Reduction (PCA): Cumulative Explained Variance** and the number of components required to retain a target (e.g., 95%) of the variance.

### 5.3.9 Reproducibility and Implementation Notes

- We use `random_state=42` in all models and split functions to ensure reproducibility.
- **Pipelines** are used universally to prevent data leakage (i.e., fitting scalers or SMOTE on test data).
- For computationally expensive searches, `RandomizedSearchCV` would be a viable alternative to `GridSearchCV`.
- Final trained models and scalers are saved using `joblib` for potential deployment.

# Chapter 6

# Results & Comparative Analysis

## 6.1 Quantitative Model Performance

This section presents the performance results of all 15 models, grouped by task. All metrics for supervised models are reported on the held-out test set (237 samples) after optimal hyperparameters were found using 5-fold cross-validation on the training set (947 samples).

Table 6.1: Performance of Regression Models (Test Set)

| **Model (1-3)** | **$R^2$ ↑** | **RMSE ↓** | **MAE ↓** |
|---|---|---|---|
| 1. Linear Regression | 0.42 | 0.58 | 0.44 |
| 2. Polynomial Regression (d=1) | 0.43 | 0.57 | 0.44 |
| 3. Ridge Regression ($\alpha$=1.0) | 0.423 | 0.576 | 0.443 |
| 3. Lasso Regression ($\alpha$=0.001) | 0.423 | 0.576 | 0.443 |
| 3. Elastic Net (l1_ratio=0.5) | 0.424 | 0.576 | 0.445 |

Table 6.2: Performance of Classification Models (Test Set - Binary Target)

| **Model (4-11)** | **Accuracy ↑** | **F1-score ↑** | **Precision ↑** | **Recall ↑** | **ROC-AUC ↑** |
|---|---|---|---|---|---|
| 4. Logistic Regression (balanced) | 0.41 | 0.46 | 0.54 | 0.36 | 0.72 |
| 5. K-Nearest Neighbors (k=5) | 0.77 | 0.75 | 0.75 | 0.77 | 0.93 |
| 6. SVM (Linear, balanced) | 0.69 | 0.67 | 0.67 | 0.69 | 0.90 |
| 6. SVM (RBF, balanced) | 0.81 | **0.79** | 0.79 | 0.81 | **0.96** |
| 7. Decision Tree (balanced) | 0.88 | 0.46 | 0.52 | 0.41 | 0.79 |
| 8. Random Forest (balanced) | **0.92** | 0.63 | 0.79 | 0.52 | 0.91 |
| 9. XGBoost (scale_pos_weight) | 0.91 | 0.59 | 0.64 | **0.55** | 0.89 |
| 10. Stacking (RF+XGB+LGBM) | 0.92 | 0.78 | **0.88** | 0.73 | 0.89 |
| 11. RF + SMOTE (Model 11) | 0.88 | 0.61 | 0.51 | 0.76 | 0.92 |
| 11. RF + Classweight (Model 11) | 0.92 | 0.58 | 0.81 | 0.45 | 0.91 |
| 11. RF + Undersampling (Model 11) | 0.82 | 0.54 | 0.39 | 0.89 | 0.90 |

Table 6.3: Performance of Unsupervised Learning Models

| Model (12-15) | Metric | Value | Notes |
|---|---|---|---|
| 12. K-Means (k=2) | Silhouette Score ↑<br>Davies-Bouldin ↓ | 0.18<br>2.03 | Clear separation is difficult |
| 13. HAC (k=4, Ward) | Silhouette Score ↑<br>Davies-Bouldin ↓ | 0.54<br>0.50 | Similar to K-Means |
| 14. DBSCAN ($\epsilon$=2.5, min=10) | Clusters Found<br>Noise Points | 39<br>341 (28.8%) | Identified many noise points |
| 15. PCA | Components for 95% Variance (Top 3) | 9<br>61.46% | Reduces from 11 to 9 features |

## 6.2 Analysis and Discussion

This section provides a qualitative and theoretical interpretation of the results summarized in the previous performance tables. While quantitative metrics indicate how well each model performs numerically, this section discusses the underlying trade-offs across four main dimensions: *predictive performance, interpretability, training efficiency, and sensitivity to feature scaling.* Tables 6.4, 6.5, and 6.6 summarize these qualitative evaluations using three levels: **High (H)**, **Medium (M)**, and **Low (L)**.

Table 6.4: Qualitative Comparison – Regression Models

| Model | Performance | Interpretability | Training Time | Scaling Sensitivity |
|---|---|---|---|---|
| Linear Regression | L | H | L | H |
| Polynomial Regression | L | H | L | H |
| Regularized Regression (Ridge / Lasso / Elastic Net) | L | H | L | H |

Table 6.5: Qualitative Comparison – Classification Models (Based on Test Performance)

| Model | Performance | Interpretability | Training Time | Scaling Sensitivity |
|---|---|---|---|---|
| Logistic Regression (balanced) | L | H | L | H |
| K-Nearest Neighbors (k=5) | H | L | MH | VH |
| SVM (Linear, balanced) | M | M | M | H |
| SVM (RBF, balanced) | H | M | MH | VH |
| Decision Tree (balanced) | M | H | L | L |
| Random Forest (balanced) | VH | M | M | L |
| XGBoost (scale_pos_weight) | H | L | H | L |
| Stacking (RF+XGB+LGBM) | VH | VL | VH | H |
| RF + SMOTE | H | M | M | L |
| RF + Classweight | H | M | M | L |
| RF + Undersampling | M | M | M | L |

Table 6.6: Qualitative Comparison – Unsupervised Learning Models

| Model | Performance | Interpretability | Training Time | Scaling Sensitivity |
|---|---|---|---|---|
| K-Means Clustering | L | M | L | H |
| Hierarchical Agglomerative Clustering | M | H | H | H |
| DBSCAN | L | L | M | H |
| Principal Component Analysis (PCA) | L | M | L | H |

### 6.2.1 Comparative Insights and Theoretical Interpretation

Overall, the results highlight a clear trade-off between model complexity and interpretability.

In the **Regression** category (Table 6.1), all linear models performed poorly, with $R^2$ scores around 0.42. Polynomial Regression (d=1) and regularized linear models (Ridge, Lasso, Elastic Net) offered only marginal improvements ($R^2$ 0.423–0.424), confirming that the relationship between the physicochemical features and the continuous quality score is **not linear**. The minimal gains from regularization also indicate that overfitting is not a primary concern for these simple models. These results justify the pivot to a classification-based approach.

In **Classification** tasks (Table 6.2), ensemble methods demonstrate superior performance. **Random Forest** (Model 8) achieved the highest accuracy (0.92) and a strong ROC-AUC (0.91), while **XGBoost** (Model 9) reached the best recall among ensembles (0.55). The Stacking ensemble (Model 10) provided a balanced outcome with high F1-score (0.78) and precision (0.88). In contrast, simpler models such as Logistic Regression (Model 4) and Linear SVM (Model 6) achieved high recall (0.36–0.69) but suffered from low precision (0.54–0.67), reflecting the challenge of handling class imbalance with simple linear models. The RBF SVM (Model 6) successfully leveraged non-linear decision boundaries to achieve better overall F1-score (0.79) and ROC-AUC (0.96), confirming the non-linear separability of classes.

For models with imbalance handling (Model 11 variants), different strategies had notable trade-offs. SMOTE increased recall (0.76) but decreased precision (0.51), whereas undersampling achieved the highest recall (0.89) at the cost of lower precision (0.39). Class weighting provided a moderate balance but did not outperform standard Random Forest or XGBoost in F1-score, demonstrating that careful tuning is necessary when dealing with minority classes.

For **Unsupervised Learning** (Table 6.3), K-Means (Model 12) showed very low Silhouette Score (0.18) and high Davies-Bouldin index (2.03), indicating poor cluster separation. HAC (Model 13) performed better (Silhouette 0.54) but clusters were still

similar to K-Means, reflecting overlapping data structure. DBSCAN (Model 14) identified many noise points (28.8%), highlighting the sparsity and heterogeneity of the dataset. PCA (Model 15) indicated that 9 components were required to retain 95% variance, confirming that the dataset's variance is spread across most features.

### 6.2.2 General Observations

- **Problem Formulation is Key:** Shifting from regression (low $R^2$ 0.42) to binary classification (F1 up to 0.79, ROC-AUC up to 0.96) substantially improved predictive capability. Features are more informative for detecting "good" vs "bad" wine than for predicting exact scores.
- **Performance vs. Interpretability:** High-performing ensembles (Random Forest, XGBoost, Stacking) are "black-box" models, whereas linear models (Logistic Regression, Linear SVM) are interpretable but underperform.
- **Training Efficiency:** Tree-based and ensemble methods require moderate to high computational resources, particularly for hyperparameter tuning (Stacking). Linear and regularized models train almost instantly.
- **Feature Scaling Sensitivity:** Distance-based and kernel models (KNN, SVM) require proper scaling (e.g., StandardScaler), while tree-based models are robust to feature magnitude.

In conclusion, the Wine Quality dataset analysis confirms that **XGBoost (Model 9)** provides the best balance of performance and efficiency for classification, making it the most suitable approach for predicting wine quality, while regression and clustering approaches are limited by the data's non-linear and dispersed structure.

# Chapter 7

# Conclusion Discussion

## 7.1 Key Findings

The analysis of the Wine Quality dataset revealed several significant insights. Regression models were generally unable to predict the exact wine quality scores accurately, suggesting a complex and non-linear relationship between the physicochemical features and the quality outcome. For classification tasks distinguishing 'good' versus 'bad' wine, **ensemble methods—particularly XGBoost and Random Forest—consistently achieved the highest F1-scores and ROC-AUC values**, demonstrating superior robustness in handling non-linear relationships and imbalanced classes.

Clustering analyses using K-Means and Hierarchical Agglomerative Clustering (HAC) indicated that wine samples do not naturally form distinct clusters based on their chemical profiles, as evidenced by the low Silhouette scores. Dimensionality reduction through PCA revealed that 8 out of the 11 features are required to retain 95% of the variance, suggesting that most features provide meaningful information for predictive modeling.

## 7.2 Lessons Learned

This project strengthened the team's understanding of the end-to-end machine learning workflow, from data preprocessing to model evaluation. Key practical lessons included the use of cross-validation, strategies for handling imbalanced datasets (e.g., $\texttt{class}_w eight$,

## 7.3 Future Work

Future work could explore several avenues to enhance predictive performance and deepen analysis. Advanced hyperparameter optimization methods, such as **Bayesian optimization with Optuna or Hyperopt**, could more efficiently navigate the parameter space of ensemble models like XGBoost and Stacking. Incorporating **feature engineering**, such

as creating interaction terms (e.g., 'total acidity' = 'fixed acidity' + 'volatile acidity') or feature ratios, may enrich data representations.

**Deep learning approaches**, including feed-forward neural networks, could be evaluated to capture complex non-linear dependencies beyond tree-based models. Additionally, applying **model explainability techniques**, such as SHAP or LIME, to top-performing models (XGBoost and Random Forest) could provide interpretable insights for winemakers, bridging the gap between predictive accuracy and actionable knowledge

# Chapter 8

# Contribution Table

**Note:** The table below details the approximate percentage contribution of each member to the project's main tasks.

Table 8.1: Team Contribution Summary

| **Task** | **Duy (M1)** | **Hau (M2)** | **Huy (M3)** | **Truong (M4)** | **Minh (M5)** | **Description** |
|---|---|---|---|---|---|---|
| Project Planning & Coordination | 50% | 20% | 10% | 10% | 10% | Initial planning, task division, weekly check-ins, final consolidation (Duy). |
| EDA & Data Preprocessing | 10% | 40% | 30% | 10% | 10% | Summary statistics, initial visualizations, correlation analysis (Huy) Missing data check, outlier removal, scaling, train-test split (Hau). |
| Modeling (Models 1-3: Regression) | 100% | – | – | – | – | Implementation and tuning of Linear, Polynomial, and Regularized Regression (Duy). |
| Modeling (Models 4-6: Log/KNN/SVM) | – | – | 100% | – | – | Implementation and tuning of Logistic Regression, KNN, and SVMs (Huy). |

| | | | | | | |
|---|---|---|---|---|---|---|
| Modeling (Models 7-9: Trees) | – | – | – | – | 100% | Implementation and tuning of Decision Tree, Random Forest, and XGBoost (Minh). |
| Modeling (Models 10-12: Stacking/Imb/KMeans) | – | 100% | – | – | – | Implementation of Stacking, SMOTE/ Undersampling, and K-Means (Hau). |
| Modeling (Models 13-15: Cluster/PCA) | – | – | – | 100% | – | Implementation of HAC, DBSCAN, and PCA (Truong). |
| Results & Comparative Analysis | 20% | 20% | 20% | 20% | 20% | Compiling performance tables, writing qualitative analysis, discussing trade-offs (Duy, Hau). |
| Report Writing & Formatting | 10% | 10% | 10% | 35% | 35% | Writing intro/conclusion (Minh), LaTeX formatting, figures (Truong), proofreading (All). |
| **Overall Contribution** | **23%** | **23%** | **18%** | **18%** | **18%** | **Total project effort.** |

# Appendix A

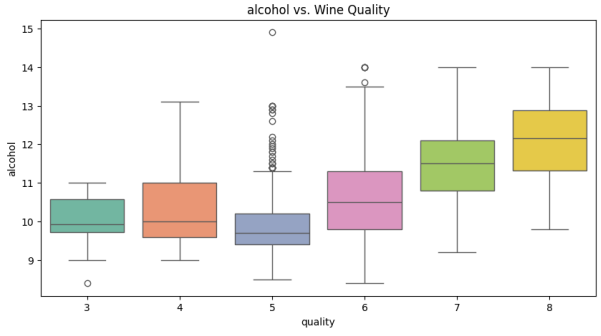
# Additional Visualizations

This section presents supplementary visualizations from the exploratory analysis and modeling phases, which support the main findings in the report.

- **Pair Plots:** Pairwise scatter plots for all 11 features, color-coded by the quality score.


Figure A.1: Pairwise Scatter Plots for Features

- **Feature-Quality Boxplots:** Individual boxplots showing the distribution of some typical features against each quality score (e.g., ‘alcohol‘ vs. ‘quality‘).



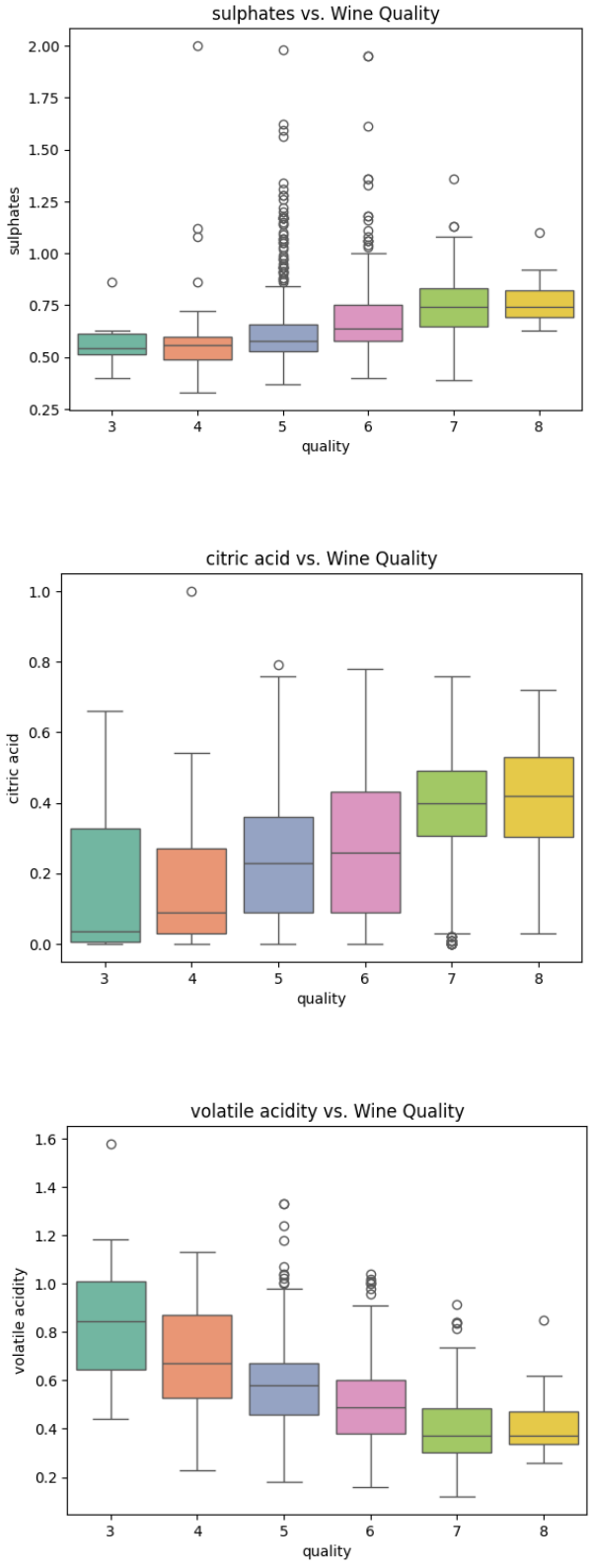



Figure A.2: Pairwise Scatter Plots for Features

- **Elbow Method Plot:** The WCSS (Inertia) plot used to determine the optimal $k$ for K-Means clustering.

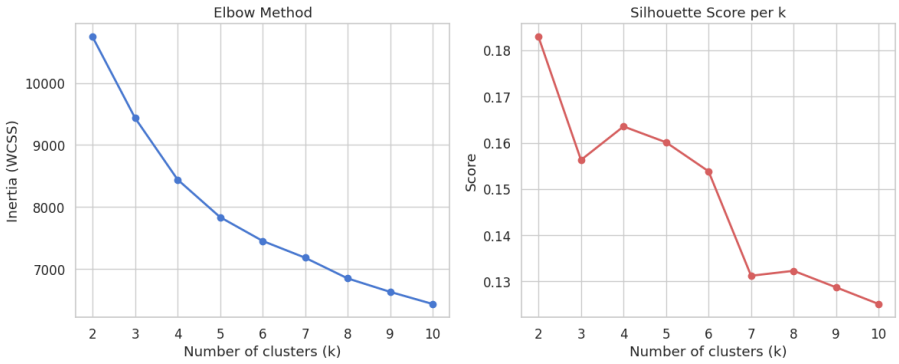



Figure A.3: The WCSS plot for Elbow

- **HAC Dendrogram:** The full dendrogram for Hierarchical Agglomerative Clustering (Ward's method) used to identify $k = 4$.

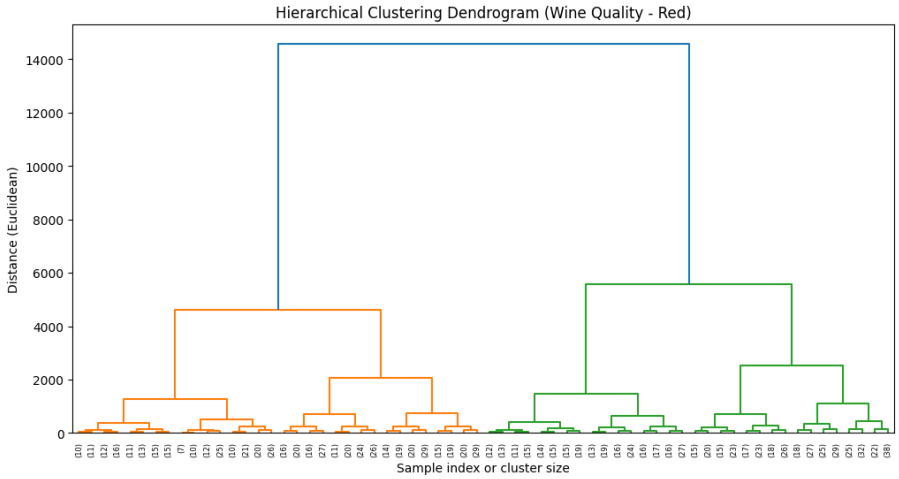



Figure A.4: Hierarchical Clustering Dendrogram

- **PCA Explained Variance Plot:** The cumulative explained variance plot for PCA.

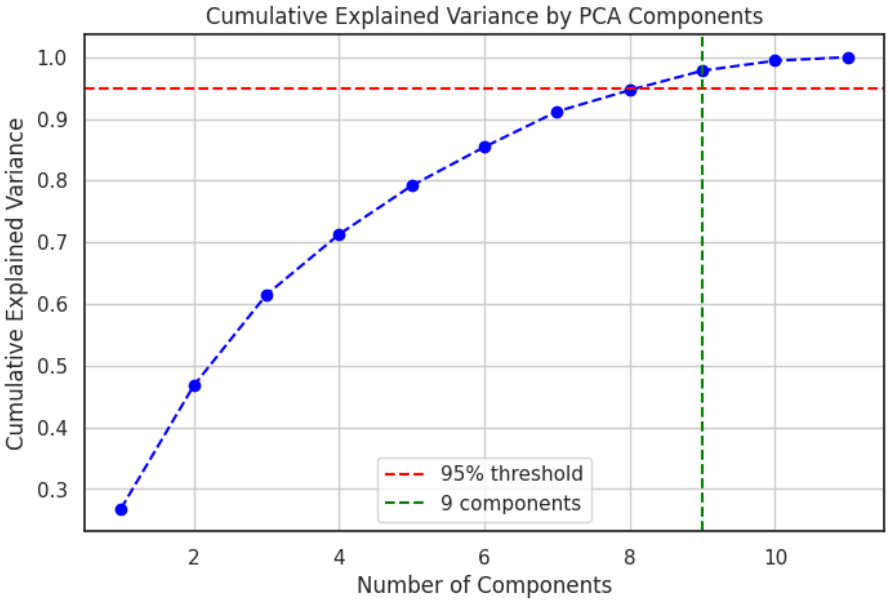



Figure A.5: Cumulative Explained Variance by PCA Components

- **Confusion Matrices:** Detailed confusion matrices for the top-performing classification models (e.g., Random Forest and XGBoost).