UNIVERSITY OF MILAN

Computer Science
Statistical Methods for Machine Learning

SVM & Logistic Regression for Wine-Quality Prediction

Student:
Alessandro Sarchi
ID number: 66046A
alessandro.sarchi@studenti.unimi.it

Professor:
Nicolò Cesa-Bianchi
Computer Science Department

Academic Year 2024/2025

Contents

Abstract **1**

Declaration of Authorship **1**

1 Introduction **1**

2 Dataset Analysis **1**
2.1 Exploration . . . 1
2.1.1 Initial exploration and data cleaning . . . 2
2.1.2 Data preparation and train-test split . . . 2
2.1.3 Distribution analysis . . . 2
2.1.4 Correlations and multicollinearity . . . 3
2.2 Preprocessing . . . 3
2.2.1 Categorical variable encoding . . . 3
2.2.2 Feature Engineering . . . 3
2.2.3 Scaling . . . 4
2.2.4 Data export . . . 4

3 Models Implementation **4**
3.1 Support Vector Machines . . . 4
3.1.1 First Implementation . . . 4
3.1.2 Enhancement and Polynomial Kernel . . . 5
3.2 Logistic Regression . . . 6
3.2.1 First Implementation . . . 6
3.2.2 Enhancement and Polynomial Kernel . . . 7

4 Experimental Analysis **8**
4.1 Training . . . 8
4.1.1 `grid_search_cv()` . . . 8
4.1.2 Support Vector Machines . . . 8
4.1.3 Logistic Regression . . . 10
4.2 Evaluation . . . 10
4.2.1 Support Vector Machines . . . 10
4.2.2 Logistic Regression . . . 10
4.2.3 Final Comparison . . . 13

Abstract

This project investigates the task of predicting wine quality using machine learning techniques, focusing on the **binary classification** of wines into *good* (score $\geqslant 6$) and *poor* (score < 6) categories. The analysis compares two fundamental algorithms, **Support Vector Machines (SVM)** and **Logistic Regression (LR)**, both implemented from scratch to provide a deeper understanding of their mathematical foundations. The study is based on the **Wine Quality dataset** from the UCI Machine Learning Repository, comprising over 6,500 samples of red and white wines described by 11 physicochemical features. After an extensive exploratory and preprocessing phase, including handling multicollinearity, feature engineering, and scaling, models were trained and evaluated through **cross-validation** and **grid search** hyperparameter tuning. Results show that both models benefit from polynomial kernels, with SVM exhibiting a marked improvement in recall but at the cost of precision, while Logistic Regression achieves more balanced and stable performance. Overall, **Logistic Regression outperforms SVM** in accuracy and F1-score, demonstrating greater robustness in distinguishing wine quality in this specific context.

Declaration of Authorship

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Introduction

This project addresses the problem of automatic wine quality prediction through machine learning techniques. The objective is to develop a binary classification system capable of distinguishing between *good* quality wines (score $\geqslant 6$) and *poor* quality wines (score < 6) based exclusively on measurable physicochemical properties.

The study implements and compares two fundamental algorithms: **Support Vector Machine** (SVM) and **Logistic Regression** (LR), both developed from scratch to ensure a thorough understanding of the underlying mathematical principles. The analysis uses the **Wine Quality dataset** from the UCI Machine Learning Repository[1], which includes approximately 6,500 samples of red and white wines characterized by 11 physicochemical properties (fixed acidity, volatile acidity, citric acid, residual sugars, chlorides, free and total sulfur dioxide, density, pH, sulfates, and alcohol content).

The project provides a comparative analysis of the performance of both methodological approaches, evaluating their effectiveness in wine quality classification and offering insights into the strengths and weaknesses of each algorithm in the specific context of enological analysis.

2 Dataset Analysis

2.1 Exploration

The project began with a thorough **exploratory analysis of the dataset**, a fundamental phase for understanding the nature and characteristics of the data before proceeding with algorithm

[1] dataset available here: https://archive.ics.uci.edu/dataset/186/wine+quality

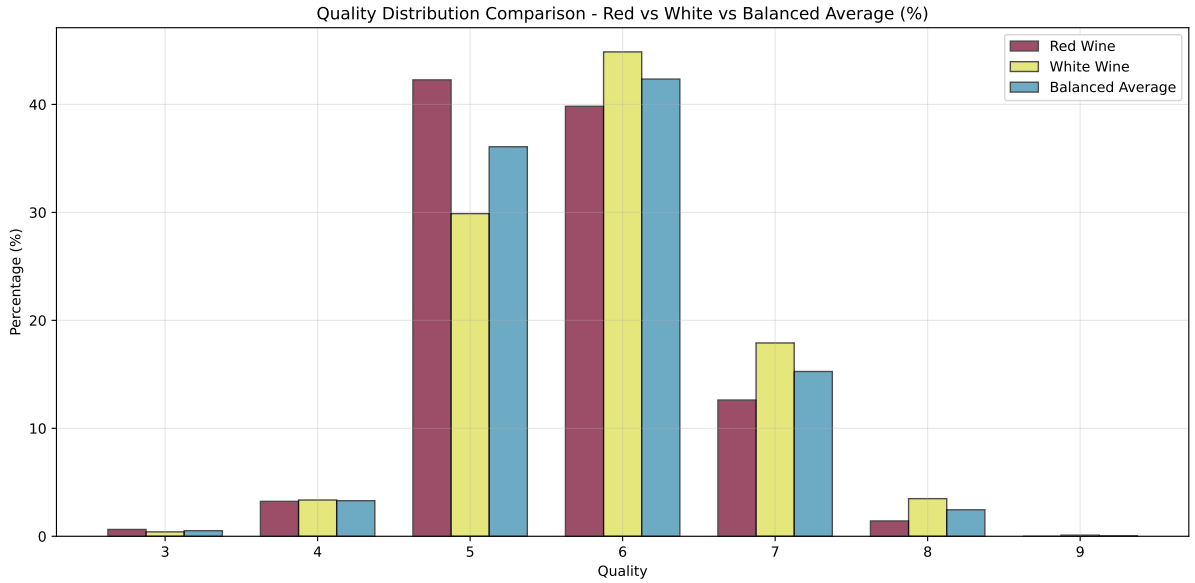

Figure 1: *Quality Distribution.*

implementation.

2.1.1 Initial exploration and data cleaning

Initially, the two datasets (red wine and white wine) were analyzed separately, examining the number of samples and main descriptive statistics for each feature, including means, medians, and percentiles. This preliminary analysis allowed us to understand the variability domain of the different physicochemical properties. Subsequently, **dataset consistency** was verified by checking for null or missing values that would have required specific treatment strategies. Fortunately, the dataset proved to be complete and free of missing values.

2.1.2 Data preparation and train-test split

The two datasets were merged to simplify subsequent analysis. The `split_train_test()` function was then implemented, inspired by interface and nomenclature from **scikit-learn** to ensure familiarity and consistency in use. This design choice was maintained constant throughout the project to facilitate usage by those with experience with that library. The function accepts as parameters the examples `X`, labels `y`, a `random_state` seed to ensure reproducibility (set to default 42 throughout the project), and the `stratify` parameter to maintain balanced the target class distribution in the train and test sets. This separation was performed immediately **to avoid any form of data leakage** and treat test data as completely new during final evaluation.

2.1.3 Distribution analysis

Analysis of the target variable distribution revealed that most wines are concentrated on scores 5 (approximately 36%) and 6 (approximately 42%). Regarding binary classification (*good* vs *bad*), the dataset presents a **slight imbalance** with 60% of wines classified as *good* and 40% as *bad*. Examination of individual feature distributions showed that most follow approximately normal distributions, although with varying degrees of skewness and kurtosis. A **significant presence of outliers** beyond the $1.5 \cdot \text{IQR}$ (InterQuartile Range) threshold emerged for almost all features. The only exception is the *alcohol* feature, which presents a distribution that is difficult to interpret.

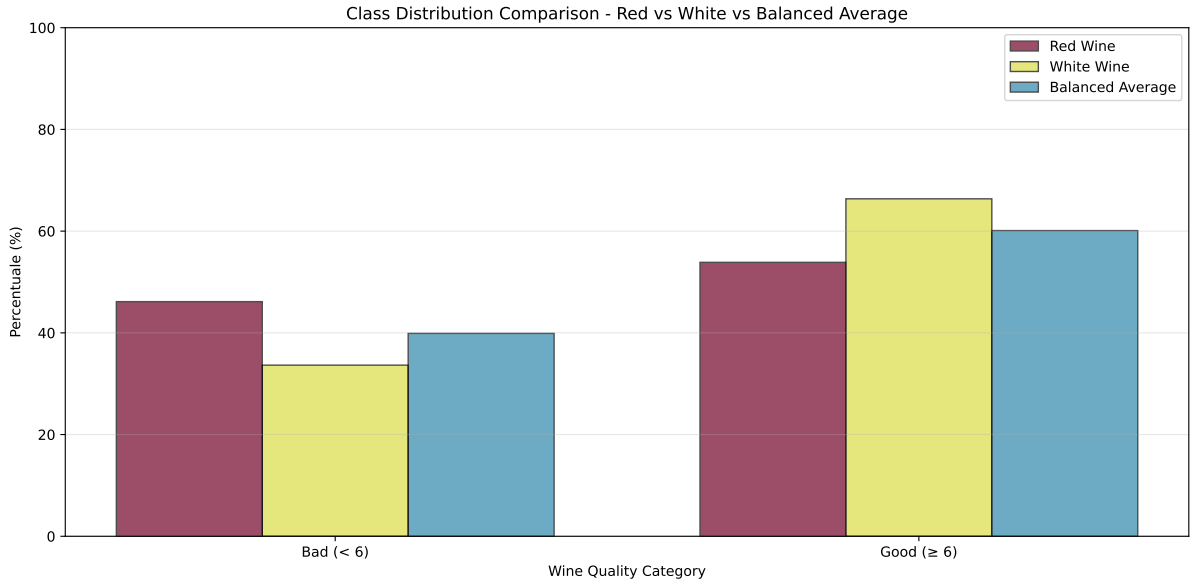

Figure 2: *Class Distribution.*

2.1.4 Correlations and multicollinearity

Correlation matrix analysis identified **potential multicollinearity problems** using a |0.6| threshold to highlight significant correlations. Two pairs of strongly correlated features emerged:

- *free sulfur dioxide* - *total sulfur dioxide*: a logically predictable correlation given the chemical nature of the variables
- *density* - *alcohol*: a less intuitive correlation for those not expert in the wine sector

This analysis provided the foundation for considering feature selection strategies in subsequent phases of the project.

2.2 Preprocessing

2.2.1 Categorical variable encoding

The first preprocessing step involved **encoding categorical features**, replacing non-numerical variables with numerical values 0 and 1 to make them compatible with machine learning algorithms.

2.2.2 Feature Engineering

Subsequently, we addressed the management of problematic features identified in the correlation analysis:

- ***Sulfur dioxide* treatment**: Given the high correlation between *free sulfur dioxide* and *total sulfur dioxide*, a new feature called *free sulfur dioxide ratio* was created that captures the ratio between these two variables, eliminating informational redundancy and providing a more compact representation of the relationship.
- **For the *density* - *alcohol* pair**, we chose to maintain both features since the correlation, while high, provides complementary information useful for predictive models. The same strategy was applied to the correlation that emerged after encoding between ***volatile acidity* and *wine type***, considering this relationship more as an advantage than a problem for the predictive capacity of the models.

2.2.3 Scaling

The `StandardScaler` class was implemented, following the same design philosophy adopted for other project components. This class operates through two main methods: `fit_transform()`: used on training data (`X_train`) to calculate mean and standard deviation of each feature and simultaneously apply the transformation `transform()`: used on test data applying parameters (mean and standard deviation) calculated exclusively on training data, thus avoiding data leakage Standardization applies the formula

$$X_{scaled} = \frac{X_{train} - \mu}{\sigma}$$

maintaining the original shape of distributions but centering each feature on mean $\mu = 0$ and standard deviation $\sigma = 1$. This process ensures that no feature has disproportionate weight due to its numerical scale.

2.2.4 Data export

At the end of preprocessing, the transformed **data was exported** to freeze the dataset state and facilitate import in subsequent notebooks dedicated to algorithm implementation and evaluation.

3 Models Implementation

Both models went through several evolutionary phases during development, with significant modifications introduced particularly in the implementation of kernel methods. Below are the main steps that characterized the evolution of each algorithm.

3.1 Support Vector Machines

3.1.1 First Implementation

The first version of the SVM class implements a **linear Support Vector Machine** for binary classification using the **PEGASOS algorithm** [1]. This approach solves the SVM optimization problem through the stochastic gradient descent method.

Class architecture The class constructor defines three fundamental parameters:

- `n_iters`: controls the number of SGD algorithm iterations
- `lambda_param`: represents the regularization coefficient that balances the trade-off between margin maximization and classification error minimization
- `random_state`: ensures experiment reproducibility

The weight vector `w`, initialized as `None`, will represent the separating hyperplane once model training is completed.

PEGASOS algorithm The `fit()` method implements the PEGASOS algorithm characterized by an adaptive learning rate defined as

$$\eta_t = \frac{1}{\lambda \cdot t}$$

where t represents the current iteration. The iterative process randomly selects a sample for each iteration and updates weights based on the local gradient with the formula

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \cdot \nabla \ell(\mathbf{x}_t)$$

The algorithm uses **hinge loss** ℓ as the loss function. For each sample, the **margin**

$$y_t(\mathbf{w}^\top \mathbf{x}_t)$$

is calculated:

- **insufficient margin** (< 1): the point violates the safety margin, activating hinge loss penalization. The weight update includes both the regularization term and correction based on the misclassified sample
- **sufficient margin** ($\geqslant 1$): the point is correctly classified with adequate margin, so the update considers only the regularization term

Prediction The `predict()` method implements the SVM decision function by applying the sign function to the dot product between input data and the trained weight vector. The decision hyperplane is defined by the equation $\mathbf{w}^\top \mathbf{x} = 0$, and the final classification depends on which side of the hyperplane the point to be classified is positioned.

3.1.2 Enhancement and Polynomial Kernel

While the original implementation was dedicated exclusively to linear classification through PEGASOS, this new version represents an evolution toward *Kernel PEGASOS*, which operates in RKHS (Reproducing Kernel Hilbert Space). The polynomial kernel

$$K(\mathbf{x}_1, \mathbf{x}_2) = (1 + \mathbf{x}_1^\top \mathbf{x}_2)^{degree}$$

implicitly defines a higher-dimensional feature space where data becomes linearly separable.

Trasformation The fundamental transformation occurs in the transition from explicit weight representation $\mathbf{w}$ (in the linear case) to implicit representation in kernel space. According to the representer theorem, the optimal solution $\mathbf{w}^*$ can be written as

$$\mathbf{w}^* = \sum_s \alpha_s y_s \mathbf{x}_s$$

[2], where S indicates the set of support vectors. In the kernel context, this translates to the decision function

$$f(\mathbf{x}) = \sum_{s \in S} \alpha_s y_s K(\mathbf{x}_s, \cdot)$$

[3], where the coefficients α_s determine the relative importance of each support vector in defining the separating hyperplane in the transformed feature space.

Kernel PEGASOS The polynomial branch faithfully implements the theoretical update rule:

$$g_{t+1} = \left(1 - \frac{1}{t}\right) g_t + \frac{y_{st}}{\lambda t} \mathbb{I}[h_{st}(g_t) > 0] K(\mathbf{x}_{st}, \cdot)$$

from [3]. This translates into code through two synchronized mechanisms:

- **Temporal decay**: all existing coefficients α_i are scaled by $\left(1 - \frac{1}{t}\right)$, corresponding to the term $\left(1 - \frac{1}{t}\right) g_t$ in the formula
- **Conditional insertion**: when the hinge loss $h_t = \max(0, 1 - y_t \cdot decision) > 0$ indicates margin violation, a new support vector is added with coefficient $\alpha = \frac{y_t}{\lambda t}$, implementing the term

$$\frac{y_{st}}{\lambda t} \mathbb{I}[h_{st}(g_t) > 0]$$

RKHS space representation The function g belonging to RKHS space $\mathcal{H}_K$ is maintained as a linear combination of kernels:

$$g = \sum_i \alpha_i y_s K(\mathbf{x}_i, \cdot)$$

This representation is fundamental because it allows working implicitly in the feature space $\phi(\mathbf{x})$ without ever computing it explicitly, exploiting the *kernel trick*. The value

$$decision = \sum_i \alpha_i y_s K(\mathbf{x}_{sv_i}, \mathbf{x}_t)$$

represents the evaluation of function g at point $\mathbf{x}_t$, determining both classification and the need for updating.

Due to the computational cost caused by the computation of g functions, it was decided to sample `averaging_step` steps, and a new parameter was added, defaulted at 1, since if one wants to use it in linear, it does not have to mind a specific value for this parameter.

Prediction in the RKHS framework The final prediction

$$\text{sgn}\left(\sum_i \alpha_i y_s K(\mathbf{x}_{sv_i}, \mathbf{x})\right)$$

reflects the theoretical structure of the algorithm, where each support vector contributes to the decision proportionally to its coefficient α_i and kernel similarity with the new point. This approach maintains the theoretical convergence guarantees of PEGASOS while extending them to the nonlinear case through RKHS theory.

3.2 Logistic Regression

3.2.1 First Implementation

The `LogisticRegression` class implements a **regularized logistic regression** model for binary probabilistic classification. This approach combines the logistic function as a probabilistic model with gradient descent optimization to maximize the regularized likelihood.

Class Architecture The class constructor defines three fundamental parameters:

- `n_iters`: controls the number of complete epochs through the dataset
- `lambda_param`: represents the regularization coefficient that balances the trade-off between data fitting and model complexity control
- `learning_rate`: determines the step size magnitude during gradient descent optimization

The weight vector `w`, initialized as a zero vector, represents the parameters of the underlying linear model that defines the decision boundary in the feature space.

Preprocessing and Bias Handling A significant feature of the implementation is the automatic bias insertion through `np.hstack([bias_column, X])`. This operation extends the feature space by adding a constant component that allows the decision hyperplane to not necessarily pass through the origin. The bias is treated as an additional weight, simplifying both mathematical notation and computational implementation.

The validation requires binary labels encoded as $\{-1, +1\}$, consistent with the theoretical formulation of logistic regression for binary classification.

Optimization Algorithm The `fit()` method implements **gradient descent** through a nested double loop. For each epoch, the algorithm sequentially processes all dataset samples, computing the regularized loss function gradient for each example and immediately updating the parameters [4].

The logistic function implementation with the formula

$$f(z) = \frac{1}{1 + e^{-z}}$$

provides the nonlinear transformation that maps real values to the interval $(0, 1)$, interpretable as probabilities. This function constitutes the core of the probabilistic model, transforming the linear combination $\mathbf{w}^\top \mathbf{x}$ into a probability estimate.

Gradient and Regularization The gradient calculation combines two distinct components: the term derived from the logistic loss and the regularization term. The expression

$$\ell_t(\mathbf{w}) = \log_2 \left(1 + e^{-y_t \mathbf{w}^\top \mathbf{x}_t}\right) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

from [4] reflects the mathematical structure of the regularized objective function, where:

- **First term**: represents the single example's contribution to the likelihood
$$\log_2 \left(1 + e^{-y_t \mathbf{w}^\top \mathbf{x}_t}\right)$$
- **Second term**: implements quadratic penalization on parameters
$$\frac{\lambda}{2}\|\mathbf{w}\|^2$$

Prediction and Decision Threshold The `predict()` method computes probabilities through the logistic function applied to the dot product between input and weights, then applies a fixed threshold at 0.5 to obtain binary classifications.

The decision boundary is implicitly defined by the equation $\mathbf{w}^\top \mathbf{x} = 0$, where points with $\sigma(\mathbf{w}^\top \mathbf{x}) \geq 0.5$ are classified as positive class $(+1)$.

The regularized approach ensures that the model maintains generalization capability even in the presence of correlated features or limited dataset sizes, following the principles of statistical learning theory for model complexity control.

3.2.2 Enhancement and Polynomial Kernel

Evolution to Polynomial Feature Expansion Here **polynomial feature expansion** were introduced, enabling the model to capture non-linear relationships in the data. Unlike the previous implementation that operated exclusively in the original feature space, this version introduces **explicit feature transformation through polynomial combinations**.

Feature Space Transformation The fundamental innovation lies in the `expand_features()` method, which implements two distinct operational modes:

- **Linear mode**: preserves the original feature space unchanged
- **Polynomial mode**: generates all possible polynomial combinations up to degree d using `combinations_with_replacement`

For polynomial expansion, the method constructs feature vectors of the form:

$$\phi(\mathbf{x}) = [1, x_1, x_2, \ldots, x_1^2, x_1 x_2, \ldots, x_1^d, x_2^d, \ldots]$$

where each term represents a monomial of degree up to `self.degree`, see [5]. This transformation maps the original n-dimensional input space to a higher-dimensional feature space containing $\binom{n+d}{d}$ features.

Bias Handling Modification The bias insertion strategy has been refined: for linear kernels, the bias column is added after feature expansion, while for polynomial kernels, the constant term 1 is automatically included during the expansion process. This eliminates redundancy and ensures consistent mathematical formulation across both modes.

Numerical Stability Improvements The implementation introduces numerical stability enhancements through `np.clip(z, -500, 500)` in the logistic function. This prevents overflow/underflow issues that can occur with large weight values or extreme feature combinations, particularly relevant when dealing with high-degree polynomial features that can produce very large values.

The enhanced architecture maintains the regularized gradient descent framework while extending the model's expressiveness through explicit polynomial feature engineering, bridging the gap between linear and non-linear classification approaches.

4 Experimental Analysis

4.1 Training

4.1.1 `grid_search_cv()`

The entire training process is based on the `grid_search_cv()` function, which systematically explores all hyperparameter combinations defined in the parameter grid. For each combination, the algorithm is evaluated using a specified number of folds (`cv` parameter) with a fixed random state (default: 42) to ensure reproducibility. The system supports four classification metrics selectable through the `scoring` parameter: accuracy, precision, recall, and f1-score. The `grid_search_cv()` relies on the `cross_val_score()` function for cross-validation implementation, which divides the dataset into k-folds, trains and evaluates the model on each train/validation combination, and returns average performance across folds to enable comparison between different hyperparameter configurations, see [6].

For both linear and poly models , a **grid search** has been implemented to identify optimal hyperparameters. Due to problems related to computation times, only degree 2 was evaluated for the polynomial ones.

4.1.2 Support Vector Machines

The linear search phase used the following parameters:

- `n_iters`: 100, 200, 500, 1000, 2000, 5000]
- `lambda_param`: [1e-1, 1e-2, 1e-3, 1e-4]

The range of iteration values starts from very few (100) to a considerable number (5000) in order to thoroughly explore how training improves and to observe convergence (see Fig. 3).

Clearly, when the number of iterations increases, there is a tendency to overfit, but this phenomenon has been mitigated by the cross-validation rounds. Regarding regularization, different orders of magnitude were simply used to see which one was most suitable.

According to theory, we should have SVM convergence as iterations increase, and indeed this behavior has also been observed empirically.

For the kernel phase (with only degree 2), we see from the learning curve (see Fig. 4) that it actually starts to overfit as iterations pass, given that the test error remains more or less similar while the train error begins to decrease. So the cross-validation rounds were not sufficient in this case.

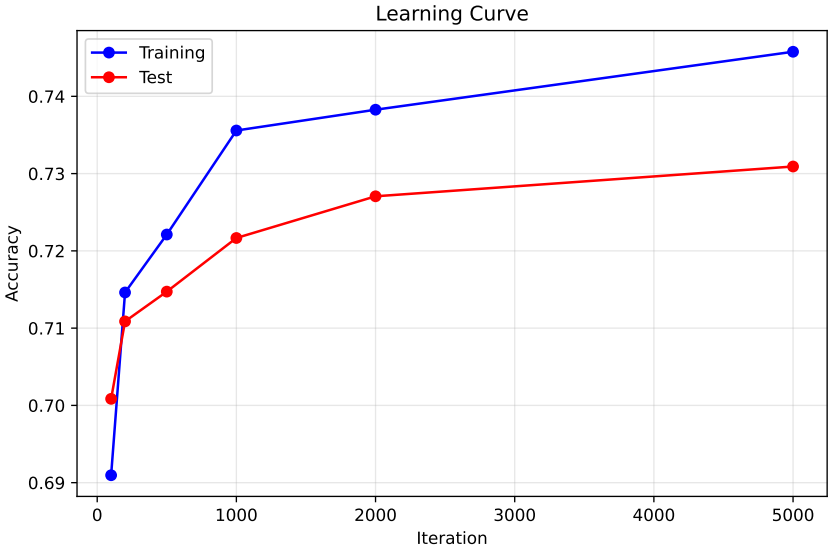

Figure 3: *Linear SVM learning curve.*

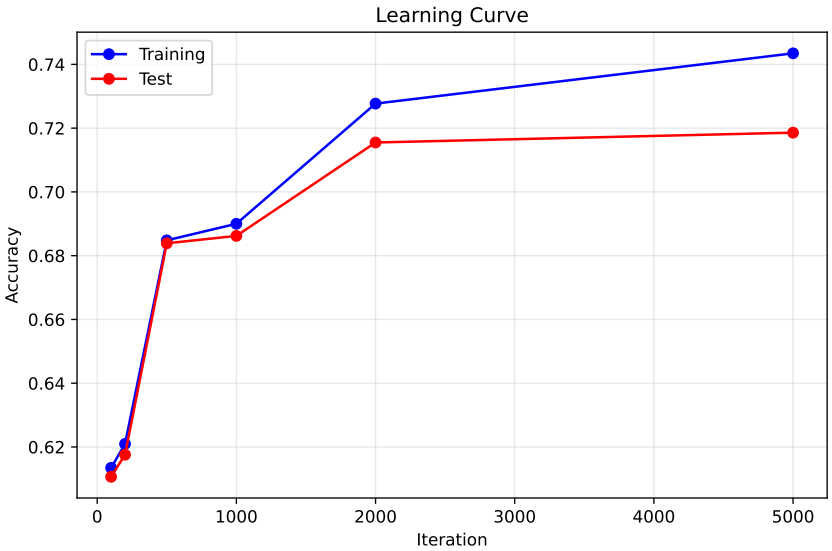

Figure 4: *Poly SVM learning curve.*

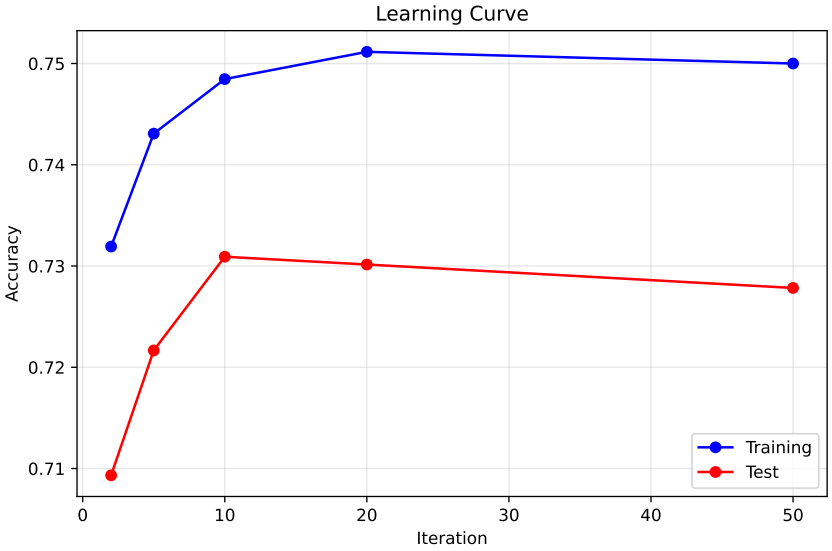

Figure 5: *Linear LR learning curve.*

4.1.3 Logistic Regression

For Logistic Regression, the first parameter grid was:

- `n_iters`: [2, 5, 10, 20, 50]
- `lambda_param`: [1e-1, 1e-2, 1e-3, 1e-4]
- `learning_rate`: [1e-1, 1e-2, 1e-3, 1e-4]

The `n_iters` values are significantly lower compared to SVM since Logistic Regression, calibrating on each sample at every epoch, converges much more rapidly (see Fig. 5). Regarding regularization and learning rate, the previous strategy was used; therefore, different orders of magnitude were explored to find the best one. Here both manage more or less to resist overfitting, following the training curve reasonably well.

4.2 Evaluation

4.2.1 Support Vector Machines

The comparative analysis between linear SVM (see Fig. 7) and SVM with polynomial kernel (see Fig. 8) reveals **significant differences** in performance on test data. The polynomial kernel apparently performs worse, and consequently, we have that the reason could be overfitting.

The only positive thing is that the precision parameter has increased, which translates to a model that has become more stringent in positive predictions. Indeed, a decrease in correct predictions of 'good' can be seen in the confusion matrix. This results in a model that, to be safe, lets some good wines slip away in exchange for fewer false positives.

4.2.2 Logistic Regression

The comparative analysis between linear Logistic Regression (see Fig. 9) and polynomial kernel (see Fig. 10) shows an improvement. We notice that indeed all parameters improve, so there isn't much to say. The model is generally better. As seen before, since there was little overfitting, the model performed well when put under test.

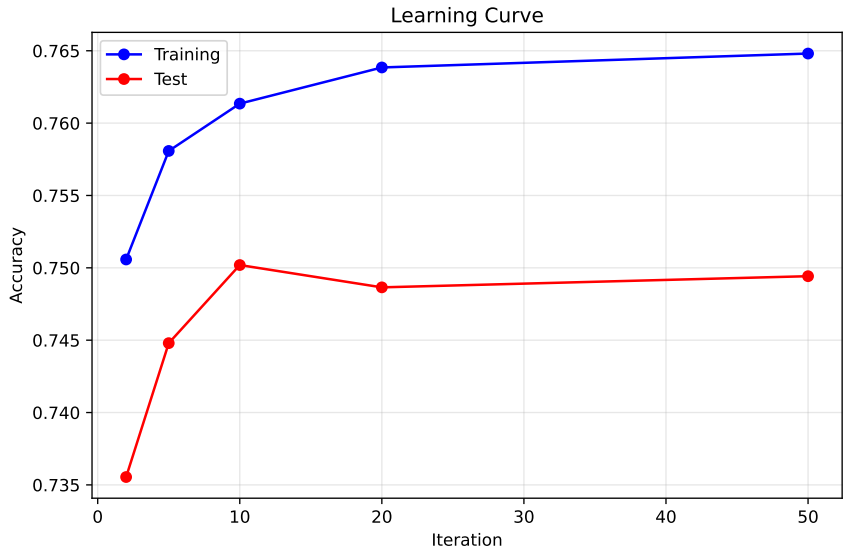

Figure 6: *Poly LR learning curve.*

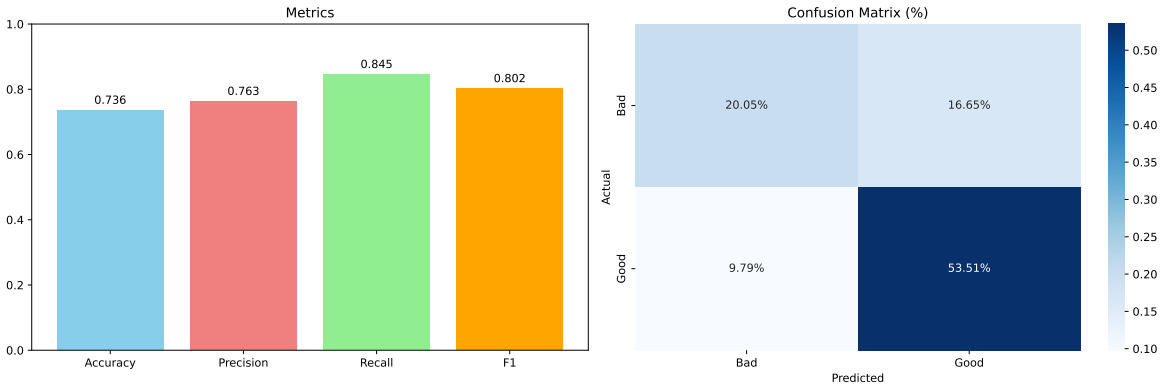

Figure 7: *Linear SVM stats.*

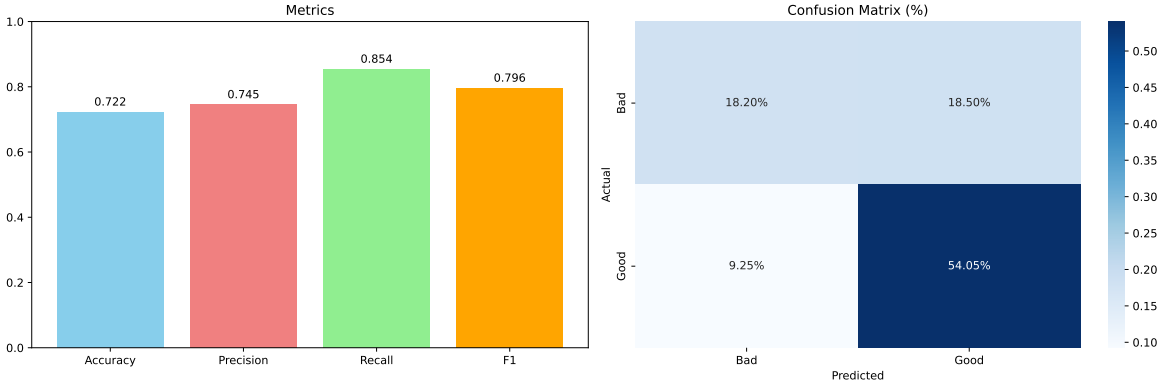

Figure 8: *Polynomial SVM stats.*

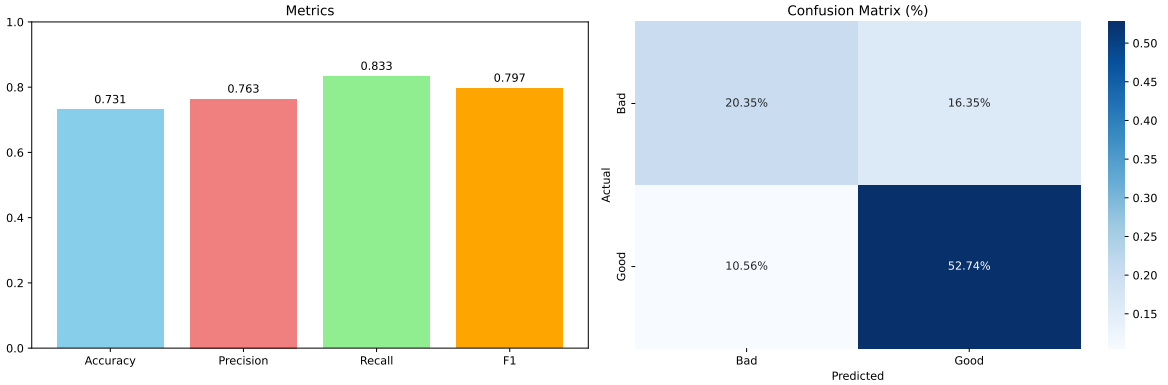

Figure 9: *Linear LR stats.*

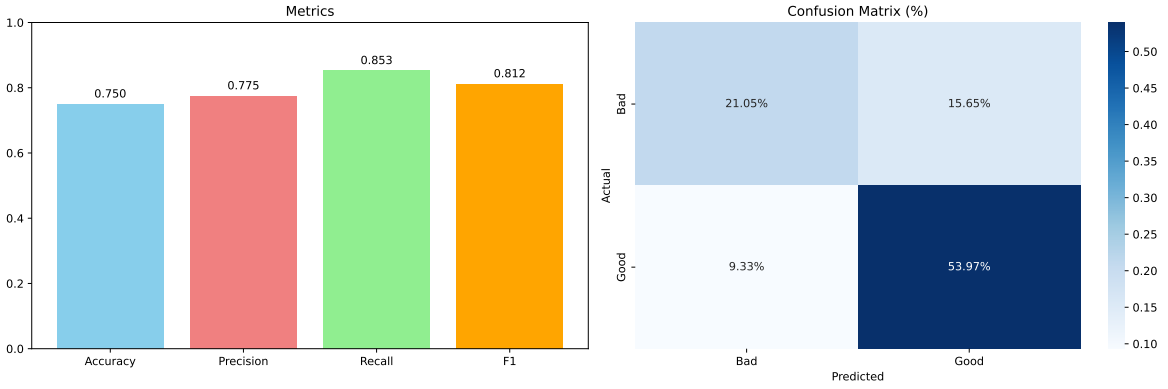

Figure 10: *Polynomial LR stats.*

4.2.3 Final Comparison

The comparative analysis reveals that **Logistic Regression outperforms SVM** across in general and in polynomial kernel comparison. In the linear version, LR achieves 73.1% accuracy against SVM's 73.6%, while with polynomial LR achieves (75.0% vs 72.2%). This results suggent general balance between model, with the LR that performs slightly better.

So the best result we obtained is a good 75.0%, which considering the unbalanced dataset (60-40) is a respectable result.

It is very important to note that this results are dictated by the random state configuration. Indeed, we choose number 1.

References

[1] Nicolò Cesa-Bianchi (2024), *12. Support Vector Machines*, p. 4.

[2] Nicolò Cesa-Bianchi (2024), *12. Support Vector Machines*, p. 3.

[3] Nicolò Cesa-Bianchi (2024), *12. Support Vector Machines*, p. 6.

[4] Nicolò Cesa-Bianchi (2023), *15. Logistic regression and surrogate loss functions*, p. 1, 2.

[5] Nicolò Cesa-Bianchi (2023), *11. Kernel functions*, p. 1.

[6] Nicolò Cesa-Bianchi (2025), *6. Hyperparameter tuning and risk estimates*, p. 4.