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

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.

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 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.

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

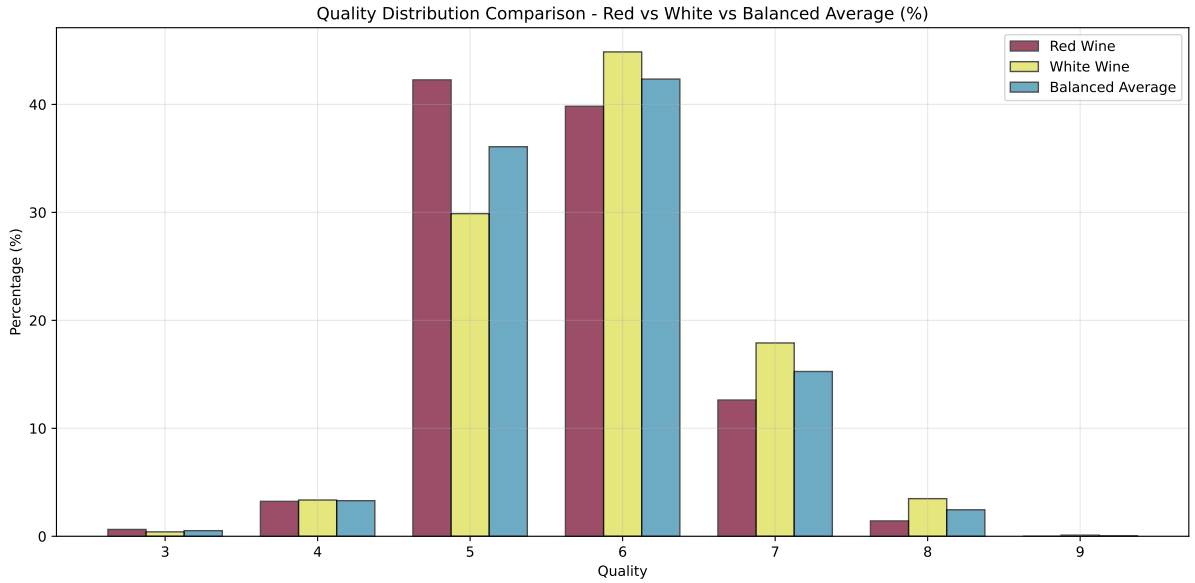

Figure 1: *Quality Distribution.*

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 $1.5 \cdot$ 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.

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.

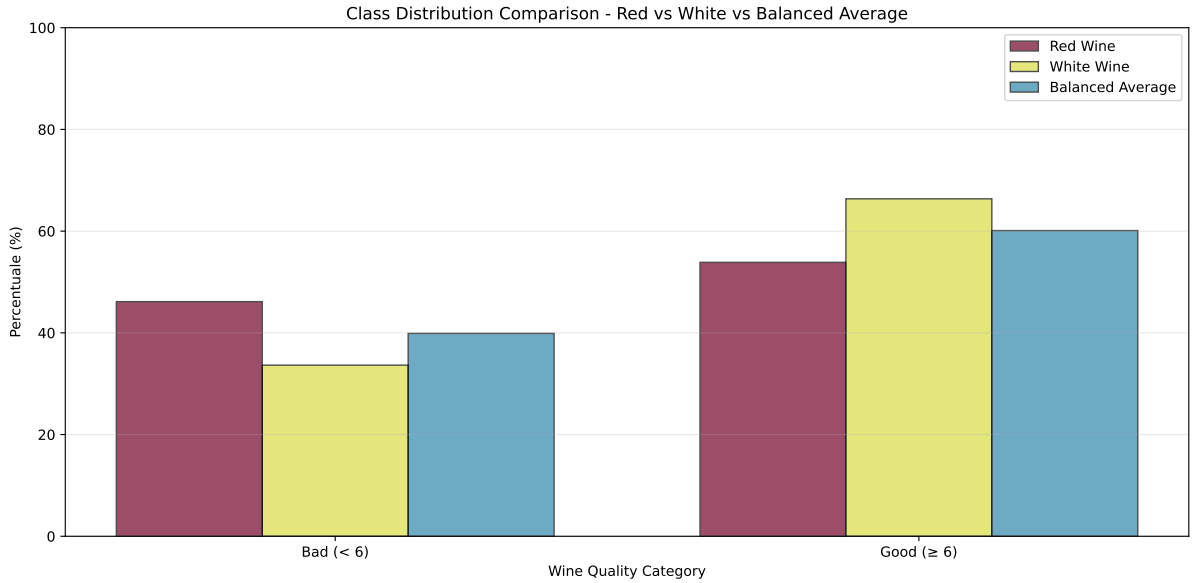

Figure 2: *Class Distribution.*

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

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

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

$$f(\mathbf{x}) = \sum_i \alpha_i \cdot K(\mathbf{x}_{sv_i}, \mathbf{x})$$

[2] [3], where the coefficients α_i 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 $h_t = \max(0, 1 - y_t \cdot \mathit{decision}) > 0$ 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:

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

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

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

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

[2] La notazione $\sum_{s \in S} y_s \alpha_s K(\mathbf{x}_s, \cdot)$ è equivalente a $\sum_i \alpha_i K(\mathbf{x}_{sv_i}, \mathbf{x})$ dove α_i incorpora già il termine $y_s \alpha_s$.

Prediction in the RKHS framework The final prediction

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

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

A configuration constant has been introduced to manage training execution times, which in some configurations can require several minutes of CPU processing. This constant can assume two values:

- `training`: executes the entire training process, including all computationally expensive phases, allowing complete experimentation with different hyperparameters
- `evaluating`: bypasses the most demanding training phases and directly loads pre-calculated results for the optimal configuration (`RANDOM_STATE = 42`, `METRIC = 'accuracy'` and `CV = 5`), allowing immediate visualization of graphs and analysis of the best results

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 linear models, a **two-phase consecutive search strategy** has been implemented, consisting of a succession of two grid searches designed to identify a local (and potentially global) optimum of hyperparameters. This approach is made possible by the significantly lower training times of linear models compared to their nonlinear counterparts. The implementation of this intensive search strategy also serves to highlight a **fundamental trade-off in machine learning**: simpler (linear) models offer the advantage of allowing much more thorough exploration of the hyperparameter space thanks to their contained computational costs, while more complex models (with polynomial kernel), despite being capable of capturing sophisticated nonlinear patterns, require considerably longer training times that necessarily limit the granularity of the search for optimal hyperparameters.

4.1.2 Support Vector Machines

Linear The search strategy was implemented through two successive phases. The initial parameter grid explored a broad hyperparameter space with:

- `n_iters`: [1000, 2000, 3000, 4000, 5000, 6000, 7000]
- `lambda_param`: [1, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5]

This first phase allowed identification of the most promising region of the parameter space. Subsequently, automatically based on the optimal results from the first grid search, a fine-tuning search was performed with a more granular grid centered on the best values:

- `n_iters`: [b - 500, b - 250, b, b + 250, b + 500]
- `lambda_param`: [b · 5, b · 3, b, b / 2, b / 5]

with the best value of the previous search as b. This coarse-to-fine approach led to the identification of optimal parameters: `n_iters` = 5750 and `lambda_param` = 0.05.

Polynomial With the polynomial kernel, instead, due to higher computational costs, a more limited parameter grid was used:

- `n_iters`: [3000, 4000, 5000]
- `lambda_param`: [1, 1e-1, 1e-2]
- `degree`: [2, 3]

The search identified as optimal configuration: `n_iters` = 4000, `lambda_param` = 0.1 and `degree` = 2. This reduction in search granularity compared to the linear case concretely illustrates the trade-off between model complexity and optimization capacity: while for the linear kernel it was possible to explore 42 combinations in the first phase and an additional 20 in the second, for the polynomial kernel we had to limit ourselves to only 18 total combinations.

4.1.3 Logistic Regression

Linear For Logistic Regression, the first parameter grid was:

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

The `n_iters` values are significantly lower compared to SVM since Logistic Regression, calibrating on each sample at every epoch, converges much more rapidly. The second fine-tuning grid search was:

- `n_iters`: [max(1, b - 5), max(1, b - 2), b, b + 2, b + 5]
- `lambda_param`: [b · 5, b, b / 2]
- `learning_rate`: [b · 5, b, b / 2]

with b as best. The two-phase strategy identified optimal parameters: `n_iters` = 20, `lambda_param` = 0.001 and `learning_rate` = 0.001. Note the use of the `max(1, ...)` function to avoid invalid iteration values when the optimal value from the first search was negative and so not valid.

Polynomial For the polynomial kernel version, the parameter grid was:

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

The optimal parameters identified were: `n_iters` = 20, `lambda_param` = 1e-05, `degree` = 3 and `learning_rate` = 0.0001. In this case as well, as with SVM, the introduction of the polynomial kernel required a reduction in search granularity, limiting to 32 total combinations compared to the 108 (27 + 45) explored in the linear version. It is interesting to note how the optimal values of `lambda_param` and `learning_rate` are significantly smaller in the polynomial version, suggesting the need for more delicate regularization and more conservative learning steps to handle the greater complexity of the nonlinear model.

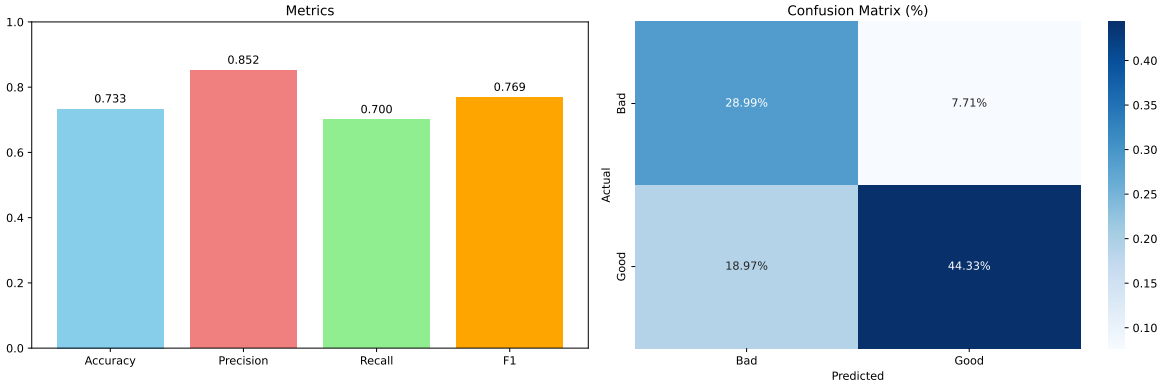

Figure 3: *Linear SVM stats.*

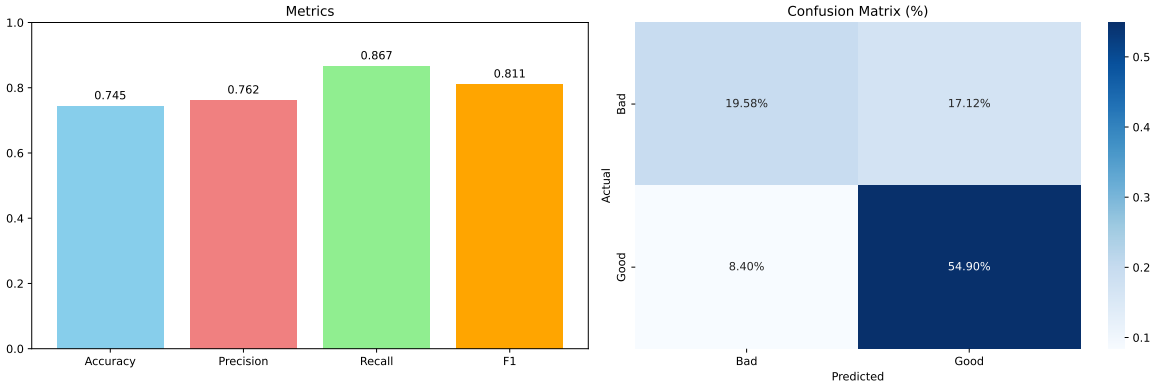

Figure 4: *Polynomial SVM stats.*

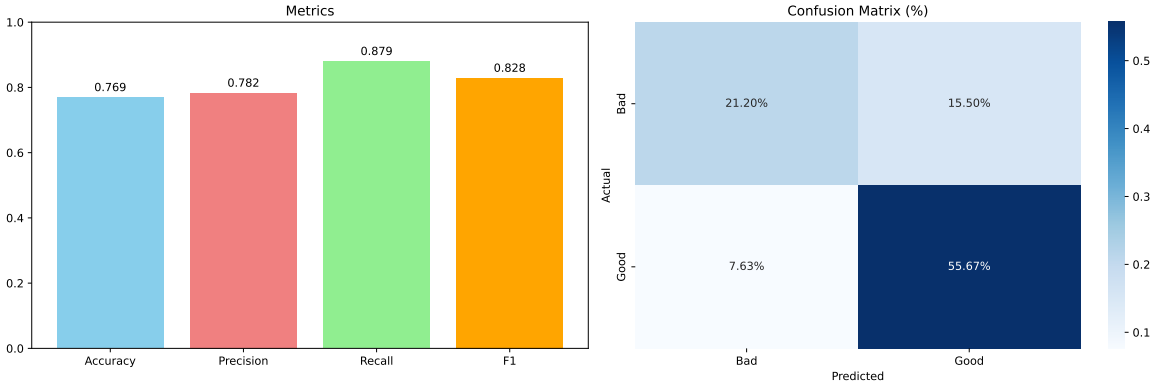

Figure 5: *Linear LR stats.*

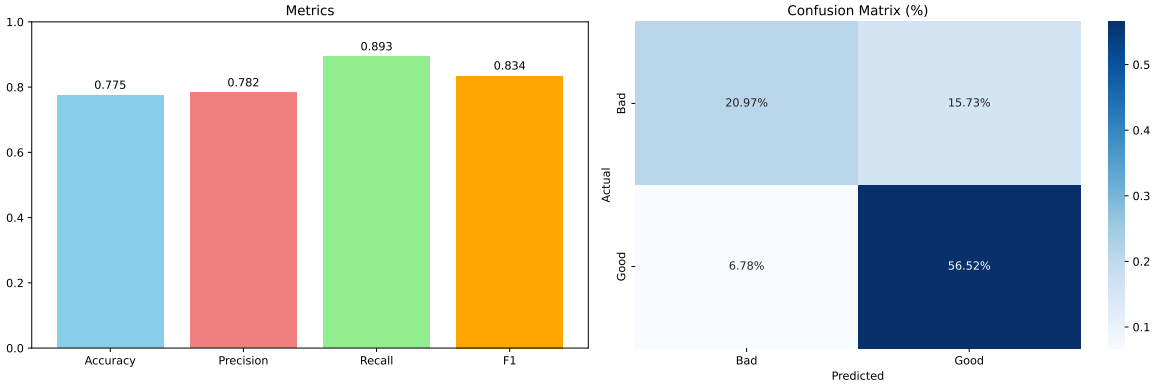

Figure 6: *Polynomial LR stats.*

4.2 Evaluation

4.2.1 Support Vector Machines

The comparative analysis between linear SVM (see Fig. 3) and SVM with polynomial kernel (see Fig. 4) reveals **significant differences** in performance on test data. The polynomial kernel (degree=2) shows overall improvement compared to the linear version, with an **accuracy** increase from 73.3% to 74.5%.

The most evident advantage of the polynomial kernel emerges in **recall**, which increases from 70.0% to 86.7%, indicating a much superior ability to correctly identify good quality wines. This translates into a substantial improvement in **F1-score** (from 76.9% to 81.1%), which balances precision and recall. However, **precision** decreases slightly (from 85.2% to 76.2%), suggesting an increase in false positives.

The confusion matrix confirms this trend: the polynomial kernel significantly reduces **false negatives** (from 18.97% to 8.40%) but increases **false positives** (from 7.71% to 17.12%). This pattern indicates that the nonlinear model is more aggressive in identifying wines as *good*, which can be advantageous in an application context where it is preferable not to miss quality wines (minimize false negatives) even at the cost of some false alarms.

The polynomial kernel therefore demonstrates its ability to capture nonlinear relationships in the data that escape the linear model, particularly in identifying the positive class.

4.2.2 Logistic Regression

The comparative analysis between linear Logistic Regression (see Fig. 5) and polynomial kernel (see Fig. 6) shows a consistent but **more contained improvement** pattern compared to

SVM. The polynomial kernel (degree=3) presents an **accuracy** increase from 76.9% to 77.5%, confirming the **advantage of nonlinear methods**.

For Logistic Regression as well, the main benefit of the polynomial kernel emerges in **recall**, which improves from 87.9% to 89.3%, indicating an even superior ability to identify good quality wines. The **F1-score** increases from 82.8% to 83.4%, while **precision** remains stable at 78.2%, suggesting better control over false positives compared to SVM.

The confusion matrix reveals a **more refined improvement**: the polynomial kernel reduces false negatives from 7.63% to 6.78% while keeping false positives substantially unchanged (from 15.50% to 15.73%).

4.2.3 Final Comparison

The comparative analysis reveals that **Logistic Regression consistently outperforms SVM** across all tested configurations. In the linear version, LR achieves 76.9% accuracy against SVM's 73.3%, while with polynomial kernel the gap persists (77.5% vs 74.5%). This systematic advantage suggests better adaptability of LR's optimization algorithm to the specific wine dataset.

Both models benefit from the introduction of polynomial kernel, but with different dynamics:

SVM: Shows the most dramatic improvement, with recall increasing from 70.0% to 86.7% (+16.7 percentage points). However, this gain comes at the cost of significant precision loss (from 85.2% to 76.2%, -9 percentage points), indicating more aggressive behavior in identifying good wines.

Logistic Regression: Presents more refined and controlled improvement, with recall moving from 87.9% to 89.3% (+1.4 percentage points) while maintaining stable precision at 78.2%. This behavior suggests superior ability to balance sensitivity and specificity.

LR demonstrates superior capability in identifying good quality wines, with consistently lower false negative rates (7.63% $\rightarrow$ 6.78%) compared to SVM (18.97% $\rightarrow$ 8.40%). SVM shows greater volatility in false positives, which increase dramatically when transitioning to polynomial kernel (7.71% $\rightarrow$ 17.12%), while LR maintains stability (15.50% $\rightarrow$ 15.73%). **Logistic Regression demonstrates greater stability** in evolution from linear to nonlinear versions, with gradual and controlled improvements, while SVM presents more abrupt performance changes.

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.