SVM and Logistic Regression for Wine-Quality Prediction

Riccardo Nazzari

55926A

Statistical Methods for Machine Learning

Academic Year: 2024/2025

Declaration

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.

Firm: Riccardo Nazzari

Date: october 2025

Abstract

Supervised learning algorithms such as Support Vector Machines (SVM) and Logistic Regression are fundamental pillars of machine learning classification. This project presents an implementation and evaluation of these models, developed from scratch using Python and standard libraries. The study addresses a binary classification task on the Wine Quality dataset, exploring both linear and non-linear decision boundaries through the implementation of kernel methods, specifically the Gaussian and Polynomial kernels.

A rigorous evaluation methodology was adopted, centered on a nested K-fold cross-validation procedure to ensure an unbiased estimate of model performance while performing robust hyperparameter tuning via grid search. Key data preprocessing steps, including logarithmic transformation for outlier mitigation and SMOTE for handling class imbalance, were systematically applied within the validation loops to prevent data leakage.

The results demonstrate that while linear models provide a solid performance baseline, the improvements from kernelized models are varied. The SVM with a Gaussian kernel stands out as the definitive top-performing model, achieving a mean `F1 score` of `79.16`% with high stability across folds. The analysis further delves into model behavior by comparing learning curves, investigating the statistical properties of commonly misclassified examples, and examining precision-recall trade-offs.

This work empirically validates the effectiveness of kernel methods, demonstrating that the choice of kernel and underlying algorithm is crucial for performance gains.

Contents

1 Data Exploration and Preprocessing

1.1 Exploration of the Dataset

Wine quality Dataset is available at UCI's website [2]. It is a collection of wines, where each is described by 11 features representing its physicochemical properties, and one variable that rates its quality on a scale from 1 to 10. The dataset is provided separately for red and white wines, but for the purpose of this project, the two parts are merged together. It is therefore possible to identify the **feature set** (composed by the 11 properties of wines) and **label set** (composed by the `quality` of wines).

Using `.info()` method it results that the dataset contains 6497 entries and there are no missing values. Furthermore, feature set is cast in type `float64`, while label set is in `int64`: although this ensures that no data type conversion is necessary for the feature set, a rescaling of `quality` is required in order to perform binary classification. To achieve this, `quality` is set to +1 if its value is ≥ 6, and to -1 otherwise.

1.1.1 Number of labels

By using the `.value_counts()` method, it is possible to observe the number of samples in each class. The results show that the number of positive labels `(4113)` is almost twice that of the negative labels `(2384)`, making the dataset imbalanced. This characteristic could potentially affect the performance of learning algorithms, and a data augmentation technique may be required to address it.

1.1.2 Distribution of the feature set

The distribution of the feature set was analyzed using the `.describe()` method. Table 1 is obtainted through this procedure. For simplicity, `quality` column and the `count` row have been removed from the table. The rows '25%', '50%', and '75%' correspond to the first quartile (Q_1), second quartile (Q_2), and third quartile (Q_3).

Feature	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**
fixed acidity	7.22	1.30	3.80	6.40	7.00	7.70	15.90
volatile acidity	0.34	0.16	0.08	0.23	0.29	0.40	1.58
citric acid	0.32	0.15	0.00	0.25	0.31	0.39	1.66
residual sugar	5.44	4.76	0.60	1.80	3.00	8.10	65.80
chlorides	0.06	0.04	0.01	0.04	0.05	0.07	0.61
free sulfur dioxide	30.53	17.75	1.00	17.00	29.00	41.00	289.00
total sulfur dioxide	115.74	56.52	6.00	77.00	118.00	156.00	440.00
density	0.99	0.00	0.99	0.99	0.99	1.00	1.04
pH	3.22	0.16	2.72	3.11	3.21	3.32	4.01
sulphates	0.53	0.15	0.22	0.43	0.51	0.60	2.00
alcohol	10.49	1.19	8.00	9.50	10.30	11.30	14.90

Table 1: Descriptive statistics for all dataset features

These tables highlight the presence of potential outliers in the dataset. A clear indication is provided by the large difference between the third quartile and the maximum value for certain features. In particular, `fixed acidity`, `residual sugar`, `free sulfur dioxide`, and `total sulfur dioxide` exhibit this behavior. This also indicates that the distributions of several features are non-symmetrical, confirming the need for appropriate preprocessing techniques. Furthermore, the features are expressed on different scales, each with its own range of values. Therefore, a standardization procedure is required before the training phase to ensure that all variables contribute comparably to the models.

1.1.3 Number of Outliers

In order to determine the number of outliers in the dataset, the interquartile range (IQR) method was applied. The interquartile range is defined as:

$$\mathrm{IQR} = \mathrm{Q}_3 - \mathrm{Q}_1.$$

Basing on this formula, the lower and upper bounds for the detection of outliers are given by:

$$\text{Lower Bound} = \mathrm{Q}_1 - 1.5 \times \mathrm{IQR},$$

$$\text{Upper Bound} = \mathrm{Q}_3 + 1.5 \times \mathrm{IQR}.$$

Any observation lying outside these bounds is considered an outlier. The number of such outliers identified in the dataset is reported in Table 2.

column	number of outliers
fixed acidity	357
volatile acidity	377
citric acid	509
residual sugar	118
chlorides	286
free sulfur dioxide	62
total sulfur dioxide	10
density	3
pH	73
sulphates	191
alcohol	3
quality	0

Table 2: Number of outliers for feature

1.1.4 Correlation

Correlation quantifies the strength and direction of the linear relationship between variables. Its values range within $[-1, +1]$, where:

- -1 indicates a perfect negative correlation (inverse linear relationship);
- $+1$ indicates a perfect positive correlation (direct linear relationship);
- 0 indicates the absence of a linear correlation.

The absolute value of the coefficient reflects the strength of the relationship. The `.corr()` method was employed to compute the correlation matrix among all features. The resulting heatmap is shown in Figure 1. The analysis highlights the following points:

- The correlation between `quality` and `alcohol` is `0.39`, representing the strongest association with the target variable. This suggests that `alcohol` should be considered a primary predictive feature. Furthermore, the strongest negative correlations of `quality` are with `volatile acidity` and `density`, both of `-0.27`;
- The correlation between `free sulfur dioxide` and `total sulfur dioxide` reaches `0.72`, the highest value in the dataset. Given their close relationship, one of these features may be removed to mitigate redundancy;
- Additional positive correlations include:
  - `density` and `residual sugar` (`0.55`);
  - `total sulfur dioxide` and `residual sugar` (`0.50`).
- A strong negative correlation is observed between `alcohol` and `density` (`-0.69`).

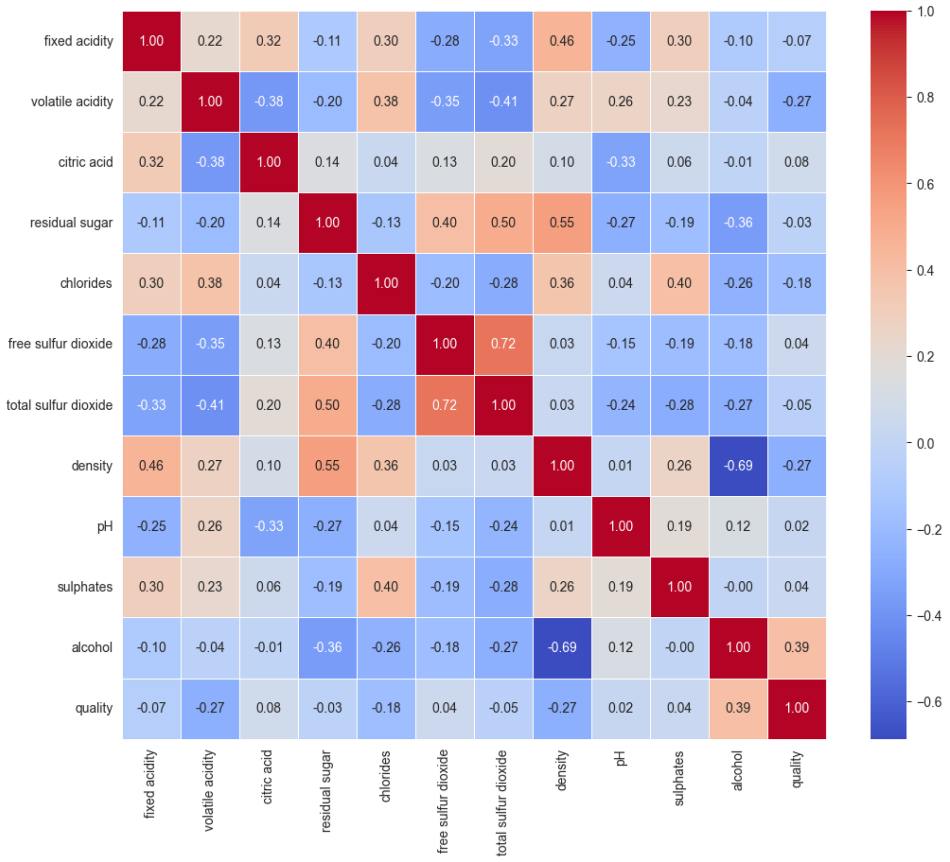

Figure 1: Correlation matrix

1.2 Preprocessing

1.2.1 Outliers treatment

Any treatment of outliers should consider both their quantity and whether it is truly necessary to remove them.

Considering Table 2, it was risky to apply techniques such as **trimming**, which removes all outliers: in the worst case, considering `citric acid`, up to 7.8% of the data would be lost. This was not advisable because dataset is already relatively small and unbalanced.

While removing outliers was not reasonable, other techniques such as **capping**, which replaces outliers with the values of the bounds, could be used to reduce their distance from the main distribution. However, it is important to note that those outliers may represent distinctive characteristics of particular wines: this means that adapting their values to the distribution could make the prediction harder - as the unique features that distinguish these wines would be diminished. Following these reasons, it has been decided to operate with a **logarithmic transformation** of the feature, in order to reduce the distance of those outliers - but without removing them.

1.2.2 Correlation Removal

Basing on the correlation matrix in Figure 1 and the previous considerations, the feature `total sulfur dioxide` was removed from the dataset. This decision was made to reduce redundancy, as it shows a strong positive correlation with `free sulfur dioxide` (`0.72`), as well as a notable correlation with `residual sugar` (`0.50`).

1.2.3 Data Augmentation

As observed in Section 1.1.1, the dataset is unbalanced due to the significant difference between the number of positive (+1) and negative (-1) labels. To mitigate potential bias during model training, it was applied a data augmentation procedure to balance the two classes. Specifically, the

Synthetic Minority Oversampling TEchnique (**SMOTE**), provided by the `imbalanced-learn` library, was employed to generate synthetic samples for the minority class to balance the distribution.

1.2.4 Synthetic Minority Oversampling TEchnique (SMOTE)

Unlike simple oversampling techniques that merely duplicate existing data, SMOTE generates new, synthetic examples by interpolating between minority class instances. This enriches the training set with novel information, allowing the model to learn a more robust decision boundary. It is crucial to note that this procedure is applied exclusively to the training data within each fold of the cross-validation to prevent data leakage. The generation process for a new sample follows a simple scheme:

1. A minority class sample $\boldsymbol{x}_i$ is randomly selected;
2. One of its k nearest neighbors $\boldsymbol{x}_{nn}$ is also chosen from the minority class;
3. A new synthetic sample $\boldsymbol{x}_{new}$ is created by generating a point along the line segment connecting the two samples:
$$\boldsymbol{x}_{new} = \boldsymbol{x}_i + \delta \cdot (\boldsymbol{x}_{nn} - \boldsymbol{x}_i)$$
where $\delta \sim U(0,1)$ is a random value from a uniform distribution.

This method effectively expands the decision region for the minority class, helping to mitigate the classifier's bias towards the majority class. A detailed description of the SMOTE algorithm is provided in the original publication by Chawla et al. [1].

1.3 Standardization

To standardize the data, **Z-score Normalization** technique was employed. It can be computed as:

$$X_{scaled} = \frac{X_{original} - \mu}{\sigma} \tag{1}$$

where μ is the mean and σ is the standard deviation. This approach was necessary to guarantee that no single feature dominates the model's learning process simply due to its larger values, also ensuring equally contributes for each example because of their same scale of features. Furthermore, standardization enhance the convergence of algorithms based on gradient descent, useful for the type of models that are going to be employed.

1.4 Split of the dataset

To train and test the performance of a model, one approach is given by dividing the dataset in 70% for training and 30% for testing. This is done by accomplishing several steps that avoid any form of data leakage:

1. shuffle data;
2. partitioning of the dataset;
3. separate feature and label sets.

If data augmentation is required, it has to be applied after partitioning, in order to guarantee that synthetic data are employed only for training. This is a critical step, as they cannot be used for testing as it would cause data leakage.

However, because it was necessary to perform a tuning of hyperparameters, this project employed a different approach. In particular, **nested k-fold cross validation** was used for both tuning and evaluation of performance. In such case, there is a different split of the dataset.

1.4.1 Nested K-fold cross validation

Nested k-fold cross-validation is a robust procedure for hyperparameter tuning and model evaluation. It employs two nested loops to prevent data leakage, ensuring that the final performance estimate is unbiased by the hyperparameter selection process. The structure is as follows:

- **Outer loop (Model performance evaluation)**: the dataset is first divided into K outer folds, $S_1, ..., S_K$. In each iteration of the outer loop, one fold (S_i) is held out as the final test set for that iteration, while the remaining K-1 folds ($S_{-i} = S \backslash S_i$); are passed to the inner loop as a complete dataset for training and hyperparameter selection;
- **Inner Loop (Hyperparameter Tuning)**: a separate J-fold cross-validation is performed exclusively on the data provided by the outer loop (S_{-i}). For each combination of hyperparameters in the grid search space ($\theta \in \Theta$) the model is trained J times on J-1 inner folds and validated on the remaining inner fold. The average validation score across these J folds is calculated for the hyperparameter set θ;
- **Final step of an outer iteration**: after the inner loop has completed, the set of hyperparameters (θ^*) that achieved the best average validation score is selected. A new model is then trained using these optimal hyperparameters (θ^*) on the entire outer training set (S_{-i}). Finally, this model's performance is evaluated on the outer test set (S_i), which was kept completely separate until this point.

This process is repeated K times. The final performance of the model is reported as the average of the K scores obtained from the outer test sets. This provides a reliable estimate of how well the entire model selection procedure is expected to perform on new, unseen data.

Algorithm 1: Nested K-fold Cross-Validation

```
parameter: Dataset $S$, outer folds $K$, inner folds $J$, hyperparameter grid $\Theta$
Split $S$ into $K$ outer folds: $S_1, \ldots, S_K$
Initialize list of outer fold errors: $E = []$
for $i = 1, \ldots, K$ do
    // Outer Loop
    outer train set $\rightarrow S_{-i} = S \setminus S_i$
    outer test set $\rightarrow S_i$
    Initialize dictionary of hyperparameter scores: $Scores = \{\}$
    for $\theta \in \Theta$ do
        // Inner Loop: Hyperparameter Tuning on $S_{-i}$
        Split $S_{-i}$ into $J$ inner folds: $S'_1, \ldots, S'_J$
        Initialize list of inner fold validation errors: $E_{inner} = []$
        for $j = 1, \ldots, J$ do
            inner train set $\rightarrow S'_{-j} = S_{-i} \setminus S'_j$
            Set inner validation set $\rightarrow = S'_j$
            Train model $A_\theta$ on $S'_{-j}$ to get predictor $h_{\theta,j}$
            Compute validation error $\varepsilon_{val} = \text{error}(h_{\theta,j}, S'_j)$
            Append $\varepsilon_{val}$ to $E_{inner}$
        end
        $Scores[\theta] = \text{average}(E_{inner})$
    end
    Find best hyperparameters $\theta^* = \arg\min_{\theta \in \Theta} Scores[\theta]$
    // Final training and evaluation for this outer fold
    Train final model $A_{\theta^*}$ on the entire $S_{-i}$ to get predictor $h_i$
    Compute outer test error $\varepsilon_i = \text{error}(h_i, S_i)$
    Append $\varepsilon_i$ to $E$
end
output : Final performance estimate: average($E$)
```

Algorithm 1 shows the pseudocode of a nested K-fold cross validation. This approach allows for measuring the generalization risk of the learning algorithm after hyperparameter tuning. Furthermore, it is suitable for computing various metrics such as `Accuracy`, `Precision`, `Recall`, and `F1 score`. In this project, the mean of each metric is reported along with its standard deviation, which helps to assess the stability of the model's performance across different folds.

The implementation of the nested K-fold cross-validation is in `classes/nested_cv`. To prevent any data leakage, `preprocessing` and `data augmentation` were applied strictly within each fold of the cross-validation loops, only after the training and test sets have been separated. This ensures that the model does not implicitly learn statistical properties from the test data. Moreover, a `stratification` method was employed to maintain the original class distribution in both the training and test splits. Finally, the implementation was designed to iterate over multiple learning algorithms, allowing for a comprehensive comparison of different models. Reproducibility of experiments was guaranteed by specifing `random_state_cv` variable: for this project, value was set to `42`. The number of inner folds was set to `4`, while the number of outer folds was set to `5`.

2 SVM and LR Implementation

2.1 Support Vector Machines

Support Vector Machines (SVMs) are a powerful and versatile class of supervised machine learning algorithms used for both classification and regression tasks. Their objective is to determine the optimal hyperplane that separates data points belonging to different classes. In the case of a linearly separable dataset, the hyperplane is chosen to maximize the margin between the closest points of each class, known as *support vectors*. In this setting, SVMs aim to solve:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \frac{1}{2}\left|\left|\boldsymbol{w}\right|\right|^2, \quad \text{s.t. } y_t\boldsymbol{w}^\top\boldsymbol{x}_t \geq 1 \ \ \forall t = 1, ..., m. \tag{2}$$

SVMs that follow Equation 2 are referred to as **hard**-SVMs, as they do not allow any misclassifications. This formulation is suitable when the data are truly linearly separable, a condition that is often not met in practice.

To handle cases where the data are not perfectly separable, a more general formulation is used:

$$\begin{aligned} &\min_{(\boldsymbol{w},\boldsymbol{\xi})\in\mathbb{R}^{d+m}} \quad \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{m}\sum_{t=1}^{m}\boldsymbol{\xi}_t \\ &\text{s.t.} \quad y_t\boldsymbol{w}^\top\boldsymbol{x}_t \geq 1 - \boldsymbol{\xi}_t \quad t = 1, \ldots, m \\ &\boldsymbol{\xi}_t \geq 0 \quad t = 1, \ldots, m. \end{aligned} \tag{3}$$

where λ is a regularization parameter and $\xi_1, \ldots, \xi_m$ are slack variables that relax the margin constraint. This variant is known as the **soft**-SVM, as it allows for a certain degree of misclassification.

Since it is reasonable to assume that the dataset is not linearly separable, for this project it is necessary to employ the soft-SVM approach. It is also convenient to rewrite 3 by defining $\xi_t = [1 - y_t\boldsymbol{w}^\top\boldsymbol{x}_t]_+$, which is the *hinge loss* of $h_t(\boldsymbol{w})$, obtaining:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \frac{1}{m}\sum_{t=1}^{m} h_t(\boldsymbol{w}) + \frac{\lambda}{2}\left|\left|\boldsymbol{w}\right|\right|^2. \tag{4}$$

To resolve Problem 4, a popular approach is found by applying Online Gradient Descent (OGD), given by the fact that $\ell_t(\boldsymbol{w}) = h_t(\boldsymbol{w}) + \frac{\lambda}{2}\left|\left|\boldsymbol{w}\right|\right|^2$ is a strongly convex function and, so, it is possible to minimize it by taking the opposite direction of the gradient with a proper step size. The Implementation of SVM is based on the Pseudocode 2, which is also called Primal estimated sub-gradient solver for SVM (Pegasos). For each iteration, it is performed a random draw of a sample from the training set (Pegasos operates with a Stochastic Gradient Descent (SGD)) and updates are performed accordingly by the prediction made by the model. The step size, defined as $\frac{1}{\lambda t}$, decreases as the number of iterations t increases.

Algorithm 2: SGD for Solving Soft-SVM

```
goal       : Solve Problem 4
parameter: $T$, $\lambda$
initialize : $\boldsymbol{\theta}^{(1)} = \mathbf{0}$
for $t = 1, \ldots, T$ do
    Let $\boldsymbol{w}^{(t)} = \frac{1}{\lambda t}\boldsymbol{\theta}^{(t)}$
    Choose $i$ uniformly at random from $[m]$
    if ($y_i \langle \boldsymbol{w}^{(t)}, \boldsymbol{x}_i \rangle < 1$) then
        Set $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + y_i\boldsymbol{x}_i$
    end
    else
        Set $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)}$
    end
end
output     : $\overline{\boldsymbol{w}} = \frac{1}{T}\sum_{t=1}^{T}\boldsymbol{w}^{(t)}$
```

The Pegasos algorithm was implemented in `models/svm.py`. While bias is not explicity written in the pseudo-code, it has been explicitly incorporated in the implementation by augmenting the feature vectors:

$$\mathbf{X} \leftarrow [\mathbf{X}, 1], \quad \boldsymbol{w} \leftarrow [\boldsymbol{w}, b].$$

For SVM, there are only two hyperparameters:

- `lambda_reg`, used for regularization;
- `n_iters`, the number of iterations on training process.

2.1.1 Hyperparameter tuning and performance

Hyperparameter	values
lambda_reg	[0.001, 0.02, 0.04, 0.08, 0.1]
n_iters	[20000, 25000, 30000]

Table 3: Grid search for SVM

Metric	Training (%)	Test (%)
Accuracy	72.69 ± 0.26	71.91 ± 1.00
Precision	73.94 ± 0.32	82.84 ± 0.68
Recall	70.09 ± 0.49	70.17 ± 1.62
F1-Score	71.96 ± 0.30	75.97 ± 1.05

Table 4: Performance of SVM on nested K-fold cross validation

Using nested K-fold cross validation with `K = 5`, setting `random_state = 42` for model and employing the grid search reported in Table 3, it turns out that value of `lambda_reg` is optimum as `0.02`, while `n_iters` is varying from `20000` and `25000`: this suggests that the model converges relatively quickly and does not benefit from an excessive number of training steps. Mean and standard deviation of results for each outer fold of nested K-fold cross validation are reported in Table 4.

2.2 Logistic Regression

Logistic Regression (LR) is a widely used model for binary classification problems, where the goal is to estimate the probability of an event or a property. Specifically, the output of a predictor $g : \mathcal{X} \rightarrow \mathbb{R}$ is passed through the logistic (or sigmoid) function $\sigma : \mathbb{R} \rightarrow (0, 1)$, defined as:

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

Since the model estimates probabilities, the logarithmic loss function (also known as cross-entropy loss) is employed:

$$\ell(y, \hat{y}) = \mathbb{I}\{y = +1\} \log_2 \frac{1}{\hat{y}} + \mathbb{I}\{y = -1\} \log_2 \frac{1}{1 - \hat{y}},$$

where $\hat{y} = \sigma(z)$. This expression can be rewritten in a more compact form as:

$$\log_2 \left(1 + e^{-yg(\boldsymbol{x})}\right), \tag{5}$$

which is commonly referred to as the **logistic loss**.

By considering a linear model $g(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$, the training objective becomes minimizing the logistic loss with an additional regularization term to prevent overfitting:

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

where λ is the regularization parameter. Optimization is typically performed using gradient descent:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t \nabla \ell_t(\boldsymbol{w}_t),$$

which leads to the following weight update rule:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta_t \frac{\sigma(-y_t \boldsymbol{w}^\top \boldsymbol{x}_t) y_t \boldsymbol{x}_t}{\ln 2} - \eta_t \lambda \boldsymbol{w}_t. \tag{6}$$

Equation 6 is used to iteratively minimize the loss. The Logistic Regression model is often trained using stochastic gradient descent and the implementation (which is reported in `models/lr.py`) follows this approach, also adopting the update rule of Equation 6. The set of hyperparameters is the same as for the SVM model, with the addition of `learning_rate` which controls the step size of the gradient descent updates.

2.2.1 Hyperparameter tuning and performance

Using nested K-fold cross validation with `K = 5`, setting `random_state = 42` for model and employing grid search of Table 5, it turns out that value of `learning_rate` is optimum between `0.005` and `0.01`, while `n_iters` is varying on `20000`, `22500` and `25000`; optimal `lambda_reg` is consistently found between `0.0001` and `0.01`, with value `0.005` being the most frequently selected, indicating a preference for a moderate level of regularization. Results of nested K-fold cross validation are reported in Table 6.

Hyperparameter	values
learning_rate	[0.005, 0.0075, 0.01, 0.05]
lambda_reg	[0.005, 0.001, 0.0005, 0.01, 0.0001]
n_iters	[20000, 22500, 25000]

Table 5: Grid search for LR

Metric	Training (%)	Test (%)
Accuracy	72.67 ± 0.25	72.37 ± 0.98
Precision	72.83 ± 0.72	82.00 ± 0.85
Recall	72.34 ± 1.36	72.23 ± 2.17
F1-Score	72.57 ± 0.43	76.78 ± 1.12

Table 6: Performance of LR on nested K-fold cross validation

2.3 Comparison between SVM and LR

By looking at Tables 4 and 6, performances of the two models look similar. This result is not surprising, since both methods rely on linear decision boundaries to separate classes. The difference of `accuracy` between them is minimal (`71.91`% vs `72.37`%), but it is more pronounced for the `F1 score` (`75.97`% vs `76.78`%). From this comparison, Logistic Regression shows slightly better results than SVM. However, those variations are not large enough to justify one model with respect to the other.

More interesting properties are explained by looking at `Precision` and `Recall`: while the former is slightly better for SVM (`82.84`% vs `82.00`%), the latter is notably higher for LR (`72.23`% vs `70.17`%). From those values, it is evident that SVM is a bit more 'cautious' for positive predictions, while LR tends to be 'aggressive' by being less careful on classification. Knowing that `F1 score` is the harmonic mean of those metrics, this explains why LR has better results overall.

Standard deviation allows for an assessment of stability with respect to different folds. For both models those values are relatively small, meaning that they are consistent across variety of data employed on training and testing.

3 Kernel Methods

When linear algorithms are applied to datasets that are not linearly separable, their effectiveness is inherently limited, as they fail to identify a suitable separating hyperplane. A standard approach to address this limitation is to project the data into a higher-dimensional feature space, where the classes may become linearly separable and therefore amenable to linear methods. Such projections can be formally described as functions $\phi : \mathbb{R}^d \to \mathcal{H}$, which map data points $\boldsymbol{x} \in \mathbb{R}^d$ into a higher-dimensional space $\mathcal{H}$. For instance, the *quadratic feature expansion* $\phi : \mathbb{R}^2 \to \mathbb{R}^6$ is defined as

$$\phi(\boldsymbol{x}_1, \boldsymbol{x}_2) = \left(1, \boldsymbol{x}_1^2, \boldsymbol{x}_2^2, \boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_1\boldsymbol{x}_2\right).$$

Although Polynomial feature expansions can construct such mappings, their computational complexity quickly becomes prohibitive for large training sets. Specifically, the number of generated features grows as $\Theta(d^n)$, which is Polynomial in the number of features d and exponential in the degree n. This issue can be circumvented by means of the **kernel trick**: instead of explicitly computing the mapping $\phi(\cdot)$ into the high-dimensional space, kernel functions operate directly in the input space by evaluating inner products between data points. Formally, a kernel is a function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ such that

$$K(\boldsymbol{x}, \boldsymbol{x'}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{x'}) \quad \text{for all } \boldsymbol{x}, \boldsymbol{x'} \in \mathbb{R}^d.$$

This approach is considerably more efficient than explicit feature expansion, yet it still allows linear predictors to leverage high-dimensional representations for improved performance.

Among the most widely adopted kernels are the **Polynomial** and the **Gaussian** kernels, defined respectively as

$$K_n(\boldsymbol{x}, \boldsymbol{x'}) = \left(1 + \boldsymbol{x}^\top \boldsymbol{x'}\right)^n, \tag{7}$$

$$K_\gamma(\boldsymbol{x}, \boldsymbol{x'}) = \exp\left(-\frac{1}{2\gamma}|\boldsymbol{x} - \boldsymbol{x}'|^2\right) \qquad \gamma > 0. \tag{8}$$

Implementation of kernel functions are reported in `classes/kernels.py`. While Gaussian function is common for both models, Polynomial includes optional normalization parameter. This is introduced specifically to handle numerical stability issues, such as overflow, encountered during the training of the Logistic Regression model, which proved more sensitive than the SVM.

Although the values of kernel functions can be computed on demand when needed, this approach is not properly time-efficient because operations need to be calculate each time. Therefore, training cycle takes place after the computation of a kernel matrix that contains all the possible combination of the kernel functions, enhancing the speed of the algorithm. However, note that this approach is practical only for datasets that are not too large; otherwise, the entire matrix would be too big to being stored in memory, slowing the execution of the algorithm.

3.1 SVM with Kernel Methods

To adapt the SVM for kernel functions, the hinge loss from Problem 4 is firstly expressed in terms of the high-dimensional feature map $\phi(\cdot)$. The loss for a single sample $\boldsymbol{x}_i$ becomes $\xi_i = [1 - y_i \langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle]_+$. Consequently, the SVM primal objective is formulated as:

$$\min_{\boldsymbol{w}} \frac{\lambda}{2} ||\boldsymbol{w}||^2 + \frac{1}{m}\sum_{t=1}^{m} \max(0, 1 - y_i \langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle), \tag{9}$$

where $\boldsymbol{w}$ is the weight in the feature space, which may be finite or even infinite-dimensional. This formulation is known as the **primal problem**.

While traditional methods for kernelized SVMs solve the dual optimization problem, the implemented Pegasos algorithm operates on the primal problem shown in Problem 9. Since working with the vector $\boldsymbol{w}$ is infeasible if its dimension is infinite, Pegasos leverages the Representer Theorem, which states that the optimal solution $\boldsymbol{w}$ can always be expressed as a linear combination of the training samples mapped into the feature space:

$$\boldsymbol{w} = \sum_{i=1}^{m} \alpha_i \phi(\boldsymbol{x}_i). \tag{10}$$

This property is crucial, as it allows the optimization to be performed on the coefficients α_i, whose count is equal to the number of training samples, even if $\boldsymbol{w}$ itself is infinite-dimensional. The decision function can then be rewritten entirely in terms of kernel evaluations:

$$\langle \boldsymbol{w}, \phi(\boldsymbol{x}) \rangle = \left\langle \sum_{i=1}^{m} \alpha_i \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}) \right\rangle = \sum_{i=1}^{m} \alpha_i \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}) \rangle = \sum_{i=1}^{m} \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}). \tag{11}$$

Pegasos algorithm does not update α coefficients, but an ausiliar vector $\boldsymbol{\beta}$. For each iteration t, alphas coefficients are computed as:

$$\boldsymbol{\alpha}_t = \frac{1}{\lambda t} \boldsymbol{\beta}. \tag{12}$$

Then, an example is randomly extracted from the training set and predictions are computed as:

$$f(\boldsymbol{x}_i) = \boldsymbol{\alpha}_t \boldsymbol{k}_i + \text{bias} \tag{13}$$

where $\boldsymbol{k}_i$ represents the kernel similarity between the sample $\boldsymbol{x}_i$ and every other sample in the training set. If margin constraint is violated ($y_i f(\boldsymbol{x}_i) < 1$), index of ausiliar vector $\boldsymbol{\beta}_i$ is updated by adding y_i.

SVM with kernel methods is implemented in `models/svm.py`. As described, training cycle updates an auxiliary vector which then compute $\boldsymbol{\alpha}_t$ each step. To ensure both stability and performance, final coefficient vector $\bar{\boldsymbol{\alpha}}$ is given by the mean of all $\boldsymbol{\alpha}_t$ vectors generated during training. Furthermore, bias is managed by using a different approach: an `eta_b` parameter is introduced to control its update during training process, as it is not possible to augment the feature vectors as seen on SVM. Both Polynomial and Gaussian kernels can be employed by setting the parameter kernel to either `"Gaussian"` or `"Polynomial"`. While the other hyperparameters such as `lambda_reg` and `n_iters` remain the same, `degree` and `gamma` are added for their respective kernel functions.

3.1.1 Hyperparameter tuning and performance with Gaussian kernel

Performances of SVM with Gaussian kernel have been evaluated using `K = 5` and setting `random_state = 42` for the model. By using the grid search reported in Table 7, most employed values are `gamma = 2` for 3 folds, while others achieved best results with `gamma = 2.5`; `lambda` is constantly preferred with value `0.001` and only one fold choose `eta_b = 0.01`, as well as `n_iters` which is `30000` for all folds excepts to one.

Hyperparameter	values
lambda_reg	[0.001, 0.01]
gamma	[1, 1.5, 2, 2.5, 3]
eta_b	[0.01, 0.001]
n_iters	[25000, 30000]

Table 7: Grid search for SVM with Gaussian kernel

Metric	Training (%)	Test (%)
Accuracy	78.70 ± 0.48	75.19 ± 0.41
Precision	79.99 ± 0.51	84.53 ± 0.78
Recall	76.56 ± 0.57	74.45 ± 1.01
F1 score	78.23 ± 0.50	79.16 ± 0.42

Table 8: Performance of SVM with Gaussian kernel on nested 5-fold cross validation

Table 8 reports mean of results and standard deviation obtained on nested K-fold cross validation. By making a comparison with Table 4, Gaussian kernel effectively achieves better results than linear SVM, also being more stable on metrics between various folds. The most significant improvement is seen in the `F1 score`, which reaches `79.16`% (vs `75.97`%) and only `0.42`% (vs `1.05`%) of standard deviation: this outcome confirms the benefits of kernel methods on SVM, enabling it to understand more-sophisticated relationships between data.

3.1.2 Hyperparameter tuning and performance with Polynomial kernel

Performances of SVM with Polynomial kernel have been evaluated using `k = 5` and setting `random_state = 42` for the model. From the grid search employed reported in Table 9, `degree` has always been

Hyperparameter	values
lambda_reg	[0.001, 0.01, 0.1]
degree	[1, 2, 3]
eta_b	[0.01, 0.001]
n_iters	[25000, 30000]

Table 9: Grid search for SVM with Polynomial kernel

Metric	Training (%)	Test (%)
Accuracy	75.22 ± 0.33	73.53 ± 0.98
Precision	76.39 ± 0.79	83.39 ± 0.60
Recall	73.03 ± 0.97	72.67 ± 2.06
F1 score	74.67 ± 0.32	77.64 ± 1.11

Table 10: Performance of SVM with Polynomial kernel on nested 5-fold cross validation

chosen with value `2`, while `lambda_reg` is constantly `0.1` as well as `n_iters` with `30000`. This stability on best hypeparameters between folds suggests that tuning may be optimal around those values.

Table 10 reports mean of results and standard deviation obtained on nested K-fold cross validation. As already seen on Gaussian kernel outcomes, once again results are better than classical SVM: `F1 score` is improved (`77.64`% vs `75.97`%), with a similar coefficient of standard deviation (`1.11`% vs `1.05`%). Hence, the outcomes are not as strong as with the Gaussian kernel: by making a comparison with Table 8, `F1 score` is lower (`77.64`% vs `79.16`%) as well as `Precision` and `Recall`, meaning that Gaussian kernel may be preferred for predictions on this specific dataset.

3.2 LR with Kernel Methods

Implementation of Logistic Regression with kernel methods follows a similar approach to that applied for SVM. The training part now involves updating the α_i coefficients, again based on the Representer Theorem, which defines the weight vector $\boldsymbol{w}$ as a linear combination of the training samples, weighted by the α coefficients. Consequently, the weight update rule from Equation 6 is adapted for kernel methods as:

$$\boldsymbol{\alpha}_{t+1} = \boldsymbol{\alpha}_t + \eta_t \frac{\sigma(-y_t \mathbf{k}_t^\top \boldsymbol{\alpha}_t) y_t \mathbf{k}_t}{\ln 2} - \eta_t \lambda \mathbf{K} \boldsymbol{\alpha}_t. \tag{14}$$

where $\mathbf{K}$ is the full m x m kernel matrix, and $\mathbf{k}_t$ is the t-th column of this matrix. Each element of $\mathbf{k}_t$ represents the kernel similarity between the sample $\boldsymbol{x}_t$ and every other sample in the training set. However, the update rule in Equation 14 is computationally expensive. In particular, the regularization term (the right-most part of the equation) involves a matrix-vector product between the kernel matrix $\mathbf{K}$ and the coefficient vector $\boldsymbol{\alpha}_t$. While this step is crucial for the theoretical correctness of the formula, it makes the training phase extremely time-consuming and computationally prohibitive for a complete nested K-fold cross-validation process. To mitigate this computational burden, for this project the exact regularization is substituted with a **weight decay** process. This technique involves applying a multiplicative decay factor directly to the alpha coefficients at each update step. Although this method avoids lengthy computations, the resulting performance obtained may be sub-optimal compared to using the exact update formula.

The Logistic Regression with kernel methods is implemented in `models/lr.py`. To enhance the stability of the algorithm, the learning rate is gradually reduced by dividing it by the square root of the progressive number of iterations. Both the exact and approximate update rules are available and can be selected by setting the `use_exact_reg` parameter to `True` or `False`, respectively. Furthermore, precautions have been taken to ensure numerical stability. For the calculation of the loss function, which involves an exponential term, the `np.logaddexp` function is used to avoid potential overflow issues. Moreover, to prevent gradient explosion, gradient clipping is applied: if the norm of the gradient exceeds a threshold of `1e5`, the gradient vector is rescaled.

3.2.1 Hyperparameter tuning and performance with Gaussian kernel

Performances of Logistic Regression with Gaussian kernel have been evaluated using `K = 5` and setting `random_state = 42` for the model. By employing grid search of Table 11, nested K-fold cross validation always choose `lambda_reg = 0.001`, `gamma = 3` and `learning_rate = 0.01`, while `n_iters` is `25000` for 3 folds and `30000` for the others. Although this behaviour might suggests that a greater value of `gamma` may be required, performance decreases when it gets higher. In particular, by using a grid search similar to Table 11 and employing `gamma = [3, 4, 5]`, `F1 score` becomes `76.13`% (vs `77.16`%), `Recall` decreases to `69.71`% and only `Precision` reaches an higher value with `83.95`% (vs `83.74`%). Hence, outcomes obtained from the first grid search are considered.

Hyperparameter	values
lambda_reg	[0.001, 0.01]
gamma	[1, 2, 3]
learning_rate	[0.01, 0.001]
n_iters	[25000, 30000]

Table 11: Grid search for LR with Gaussian kernel

Metric	Training (%)	Test (%)
Accuracy	74.03 ± 0.14	73.19 ± 0.66
Precision	75.51 ± 0.33	83.74 ± 0.70
Recall	71.14 ± 0.76	71.55 ± 1.38
F1 score	73.26 ± 0.29	77.16 ± 0.73

Table 12: Performance of LR with Gaussian kernel on nested K-fold cross validation

Table 12 reports mean of results and standard deviation obtained on nested K-fold cross validation. By making a comparison with classical Logistic Regression, Gaussian kernel improves the performance of the model. Althrough `Recall` tends to be worse than linear LR (`71.55`% vs `72.23`%), `Precision` is improved (`83.74`% vs `82.00`%) and this result also reflects on `F1 score`, which is slightly better overall (`77.16`% vs `76.78`%). Furthermore, standard deviation of LR with Gaussian kernel is less pronounced than for the linear version, meaning that model adapts better to different data. However, although the outcomes demonstrate an increase of performances, they are not so relevant as they were when comparing linear SVM and SVM with Gaussian kernel.

3.2.2 Hyperparameter tuning and performance with Polynomial kernel

Performances of Logistic Regression with Polynomial kernel have been evaluated using `K = 5` and setting `random_state = 42` for model. Using Grid search of Table 13, nested K-fold cross validation always choose `lambda_reg = 0.1`, while `degree` swings between `[3, 4, 5]`; `learning_rate` is chosen with value `0.0001` for `degree = 5` and `0.001` for `degree = 4` and `degree = 3`. Values of `n_iters` are both used.

Hyperparameter	values
lambda_reg	[0.01, 0.1]
degree	[3, 4, 5]
learning_rate	[0.001, 0.0001]
n_iters	[25000, 30000]

Table 13: Grid search for LR with Polynomial kernel

Metric	Training (%)	Test (%)
Accuracy	75.31 ± 0.67	73.51 ± 1.12
Precision	77.39 ± 2.64	84.43 ± 1.40
Recall	71.94 ± 4.36	71.41 ± 3.55
F1 score	74.40 ± 1.33	77.29 ± 1.59

Table 14: Performance of LR with Polynomial kernel on nested K-fold cross validation

Table 14 reports mean of results and standard deviation for nested K-fold cross validation. It is evident that performances are not so much improved with respect to linear version: `F1 score` (`77.29`% vs `76.78`%) and `Precision` (`84.43`% vs `82.00`%) are slightly better, but `Recall` has worsened (`71.41`% vs `72.23`%). Furthermore, outcomes report that values of metrics are defined with an high standard deviation, in particular `Recall` has `3.55` % of variety. Those results suggest that this model may be sensitive to the specific data in each fold.

4 Evaluation and Analysis

4.1 Comparison of performances

Table 15 reports the performance of the models obtained by using nested K-fold cross validation with `K = 5`. As already noted previously, linear SVM and Logistic Regression show similar outcomes, which is expected given that they both rely on linear decision boundaries.

The introduction of kernel methods allows the models to handle non-linearly separable data and generally contributes to improving their performance. The SVM with a Gaussian kernel stands out as the clear top performer, achieving the best results across all metrics. In particular, its `F1 score` (the most relevant metric for this imbalanced dataset) reached `79.16`% with a standard deviation of only `0.42`%, making it the preferable model for this classification problem. The SVM with a Polynomial kernel also demonstrated a consistent improvement over its linear counterpart.

Model	Accuracy (%)	Precision (%)	Recall (%)	F1 score (%)
SVM	71.91 ± 1.00	82.84 ± 0.68	70.17 ± 1.62	75.97 ± 1.05
LR	72.37 ± 0.98	82.00 ± 0.85	72.23 ± 2.17	76.78 ± 1.12
SVM Gaussian Kernel	75.19 ± 0.41	84.53 ± 0.78	74.45 ± 1.01	79.16 ± 0.42
SVM Polynomial Kernel	73.53 ± 0.98	83.39 ± 0.60	72.67 ± 2.06	77.64 ± 1.11
LR Gaussian Kernel	73.19 ± 0.66	83.74 ± 0.70	71.55 ± 1.38	77.16 ± 0.73
LR Polynomial Kernel	73.51 ± 1.12	84.43 ± 1.40	71.41 ± 3.55	77.29 ± 1.59

Table 15: Performance of models on nested 5-fold cross validation

However, the performance gains for the kernelized Logistic Regression models are not as significant. While their results are slightly better than the linear version, the improvements are marginal and not as pronounced as those seen with the SVM. A plausible explanation for this discrepancy lies in the implementation of the regularization for the kernelized LR. As discussed in Section 3.2, a computationally efficient weight decay approximation was used instead of the exact, but more expensive, regularization term. It is therefore possible that these modest improvements could be more substantial if the theoretically exact regularization formula were to be applied.

4.1.1 Predictions on classes

Table 16 reports the behavior of the models for each class. Note that the `Support` for class +1 is 823 and for class -1 is 477 for all models: this imbalance is inherent in the dataset itself, as already found in the data exploration phase.

Model	Class	Precision (%)	Recall (%)	F1-Score (%)
SVM	+1	82.84 ± 0.68	70.17 ± 1.62	75.97 ± 1.05
	-1	59.30 ± 1.24	74.92 ± 1.24	66.19 ± 0.93
LR	+1	82.00 ± 0.85	72.23 ± 2.17	76.78 ± 1.12
	-1	60.31 ± 1.46	72.61 ± 2.11	65.85 ± 0.93
SVM Gaussian Kernel	+1	84.53 ± 0.78	74.45 ± 1.01	79.16 ± 0.42
	-1	63.44 ± 0.60	76.47 ± 1.68	69.34 ± 0.67
SVM Polynomial Kernel	+1	83.39 ± 0.60	72.67 ± 2.06	77.64 ± 1.11
	-1	61.45 ± 1.49	75.00 ± 1.48	67.53 ± 0.76
LR Gaussian Kernel	+1	83.74 ± 0.70	71.55 ± 1.38	77.16 ± 0.73
	-1	60.78 ± 0.94	76.01 ± 1.48	67.54 ± 0.68
LR Polynomial Kernel	+1	84.43 ± 1.40	71.41 ± 3.55	77.29 ± 1.59
	-1	61.14 ± 2.00	77.14 ± 3.44	68.12 ± 0.66

Table 16: Performance of models per class

As observed in Table 16, the SVM with Gaussian kernel confirms itself as the most promising model for this dataset: in fact, the `F1 score` is more balanced across classes, with `79.16`% for class

`+1` and `69.34`% for class `-1`.

A consistent pattern of behavior is found across the models regarding `Precision` and `Recall` values: each model is more confident when predicting `+1` wines, as shown by `Precision`, which is always at least `82`%. Conversely, they are better at detecting `-1` wines, as `Recall` is consistently higher for this class. This indicates that the models are effective at identifying the majority of bad wines (high `Recall` on class -1), but do so at the cost of misclassifying some good wines as bad (lower `Precision` on class -1). This behaviour could be explained by the imbalanced dataset that contains more good wines than negative ones; despite data augmentation, this difference is still crucial in the behavior of the models.

4.1.2 Training and test error

To determine the behavior of the models on learning phase, it is useful to compare training and test error. When the test error is much larger than the training error, the model is **overfitting**, because it memorizes the training data. Conversely, when both training and test errors are high, the algorithm is **underfitting**, meaning it cannot capture the complexity of the problem. The optimal case is when test error is close to training error (and both are low overall). Table 17 reports these metrics with their standard deviation across folds.

Model	Training Error	Test Error
SVM	0.27 ± 0.00	0.28 ± 0.01
LR	0.27 ± 0.00	0.28 ± 0.01
SVM Gaussian Kernel	0.21 ± 0.00	0.25 ± 0.00
SVM Polynomial Kernel	0.25 ± 0.00	0.26 ± 0.01
LR Gaussian Kernel	0.26 ± 0.00	0.27 ± 0.01
LR Polynomial Kernel	0.25 ± 0.01	0.26 ± 0.01

Table 17: Errors of models on nested k-fold cross validation

From Table 17, all training and test errors show only minimal differences, meaning that the models do not exhibit any explicit overfitting behavior. Moreover, since the values are low overall, underfitting is not occurring either. The largest difference is observed for SVM with the Gaussian kernel: this gap is a sign of its higher complexity and capacity to fit the training data, which in turn allows it to achieve the best overall test performance.

4.2 Common errors of models

By analyzing the indices of misclassified samples, it was found that the number of mistaken examples common to all models is `1047`, with `702` belonging to class +1 and `345` to class -1. This difference can - again - be explained by the imbalance in the dataset. Table 18 reports the number of errors for each model and the percentage of common mistakes with the others.

Model	Errors	Common errors (%)
SVM	1825	57.37
LR	1795	58.33
SVM Gaussian Kernel	1612	64.95
SVM Polynomial Kernel	1720	60.87
LR Gaussian Kernel	1742	60.10
LR Polynomial Kernel	1721	60.84

Table 18: Errors of models on nested k-fold cross validation

SVM with Gaussian kernel reports the lowest number of mistakes (`1612`), confirming its role as the best model for this application. Furthermore, the fact that it also has the highest percentage of common errors (`64.95`%) suggests that certain feature vectors are inherently difficult to classify, since all models tend to misclassify them.

An interesting property can be observed with the linear version of models, as they have similar number of errors and percentage of common ones, again confirming their comparably approach to the problem. The same consideration applies to the Polynomial kernel models.

Finally, these results confirm the gap between Gaussian-based models: while SVM significantly reduces the number of misclassifications when using the kernel, improvements in Logistic Regression are not as substantial. In fact, its performance is comparable to the Polynomial version.

4.2.1 Features of common errors

	Samples of Common Errors		**Other Samples**	
Feature	mean	std	mean	std
alcohol	10.04	0.93	10.58	1.22
volatile acidity	0.359	0.164	0.336	0.165

Table 19: Comparison of key feature statistics for hard-to-classify vs. easy-to-classify samples.

To understand the models' limitations, a specific analysis was conducted on the samples misclassified by all algorithms. The statistical properties of the samples corresponding to these common errors were compared directly against the remaining, correctly classified samples. To highlight the most significant differences, Table 19 presents the comparative statistics for the two features that showed the most pronounced divergence: `alcohol` and `volatile acidity`.

The analysis reveals a distinct pattern: the common errors have, on average, a significantly lower alcohol content and a much smaller standard deviation, indicating their values are concentrated in a "borderline" region. Furthermore, they exhibit a higher mean volatile acidity, a trait associated with lower quality. From data exploration and Figure 1 it was noted that `alcohol` has a positive correlation of `0.39` with `quality`, while `volatile acidity` has a negative correlation of `-0.27`. This means that when value of `alcohol` increases `quality` should increase as well, and when value of `volatile acidity` decreases, `quality` may increase. This suggests that the models' primary limitation lies in classifying wines with these ambiguous chemical profiles, where the main predictive signal is weak and accompanied by conflicting secondary signals.

4.3 Loss curves of models

Figure 2 reports the mean and standard deviation of loss curves obtainted during the nested K-fold cross validation process for different models. The plot starts from 500th iteration, because first steps of SVMs registered a high loss that would make the figure less informative.

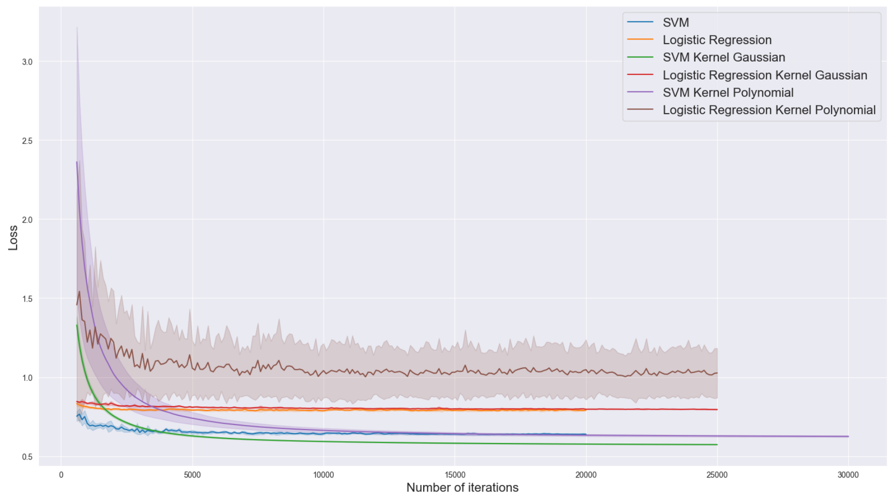

Figure 2: Mean loss curves of models from 500th iteration

The most evident fact observable are the oscillations of the loss during training process: while most of the models tend to be linear and gradually converge to lower values without any unexpected peaks, others suffers from instability that slows the decrease of the loss. In particular, Logistic Regression with Polynomial kernel has an evident problem on training: there is an irregular and noisy convergence, with the model struggling to settle on a stable, low loss value. Furthermore, its standard deviation shows that there is uncertainty in results. This instability raises questions about the training dynamics: one hypothesis is that the weight decay approximation for regularization, while computationally efficient, may not provide the same stabilizing effect as the exact gradient. This possibility is investigated in the following part.

The graph also visually confirms the superiority of the SVM with Gaussian kernel: its curve is not only one of the most stable, but it also converges to the lowest final loss value, consistent with its top-ranking F1 score performance.

4.3.1 Analysis of the Exact Regularization Term on Logistic Regression

To investigate the instability previously observed with the Logistic Regression Polynomial kernel, a separate evaluation was conducted using the theoretically exact regularization term. Due to the significant computational cost of this method, an initial hyperparameter search was performed using a small grid and a reduced number of training iterations (3000). Subsequently, the best hyperparameters identified in this search were used to conduct a final performance evaluation using a nested K-fold cross validation with the full number of iterations. Figure 3 compares the resulting learning curves of the exact regularization method and the approximated one, also evaluated with nested K-fold cross validation with best hyperparameters found previously on tuning.

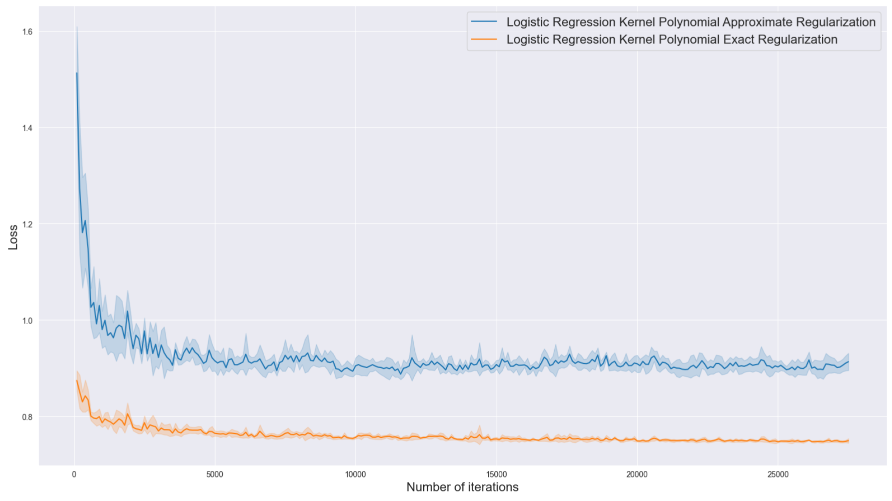

Figure 3: Logistic Regression with exact regularization and its approximation

The plot clearly demonstrates that the implementation with the exact regularization term is significantly more stable and achieves a lower final loss value compared to its approximate counterpart. This result confirms that while the weight decay approximation is computationally affordable, the use of the theoretically correct regularization technique - even with hyperparameters that were not exhaustively tuned - demonstrably surpasses its performance.

5 Conclusions

This project aimed to implement from scratch and to evaluate Support Vector Machine (SVM) and Logistic Regression classification models in both their linear and kernelized forms.

The performance evaluation revealed that while linear models provided a solid baseline, the introduction of kernel methods led to tangible, albeit varied, improvements, confirming the non-linearly separable nature of the dataset. The SVM model with a Gaussian kernel emerged as the definitive top-performing approach, achieving a mean `F1 score` of `79.16%` with a standard deviation of only `0.42%`. This result highlights its superior ability to capture complex relationships within the data while ensuring stability and strong generalization. Although the Polynomial kernel also enhanced performance compared to the linear models, it did not match the effectiveness of the Gaussian kernel for this specific problem.

A cornerstone of this project was the adoption of a nested K-fold cross-validation methodology, which ensured that the performance estimate for each model was unbiased by the hyperparameter tuning process, thereby guaranteeing the scientific validity of the comparisons. Furthermore, the application of preprocessing techniques — such as logarithmic transformation for outliers, scaling, and SMOTE to oversample the minority class — strictly within each cross-validation fold proved to be crucial in preventing any form of data leakage.

Despite the positive outcomes, it is important to acknowledge certain limitations. The inherent class imbalance of the dataset affected the entire process; even the best-performing model exhibited a performance gap between the majority (+1) and minority (-1) classes, indicating this remains a partially unresolved challenge. Furthermore, implementation challenges were encountered, particularly regarding the numerical stability of the kernelized Logistic Regression, which required specific adjustments like kernel normalization to ensure convergence. Moreover, a key methodological choice was the use of an efficient weight decay approximation for the regularization in the kernelized Logistic Regression. While this decision was necessary to manage computational costs, it is acknowledged that the theoretically exact regularization term might yield better predictive performance, as suggested by preliminary stability analysis.

In conclusion, this project successfully achieved its goal of implementing and evaluating various classification models using a robust methodology. The outcomes demonstrate that while kernel methods can enhance model performance, their effectiveness is highly dependent on the specific algorithm and kernel combination. The persistent influence of class imbalance and the trade-offs between computational efficiency and theoretical exactness in regularization remain critical considerations for future work.

References

[1] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[2] UC Irvine - Machine Learning Repository. Wine Quality Dataset. URL: `https://archive.ics.uci.edu/dataset/186/wine+quality`.