Report - SVM and Logistic Regression with Kernel Methods on Wine Dataset

Giulia Demme
Statistical Methods for Machine Learning

September 1, 2025

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.

Contents

1 Introduction

This project is based on the implementation from scratch of two classic binary classification algorithms: Support Vector Machines and Logistic Regression. The report will cover the following topics:

- Preprocessing of the dataset
- Implementation of the models
- Extension of the models to a kernelized form
- Hyperparameter tuning via 5-fold cross-validation
- Analysis of the results

2 Data Exploration and Preprocessing

Before training the models, a series of preprocessing steps was performed to assess the dataset characteristics and prepare it for learning. These preprocessing steps were implemented as dedicated functions within the file `preprocessing.py`.

2.1 Dataset Overview

For this project, the dataset used is the result of the combination of two datasets. The two datasets are related to red and white variants of the Portuguese "Vinho Verde" wine. The resulting dataset consists of 6497 samples and 12 columns, including various physicochemical measurements (e.g. acidity, density, alcohol) and a quality score representing the sensory evaluation of the wine. In the function `preprocessing`the dataset was first loaded (function `dataset_loading`) in a Pandas Dataframe, then the target was extracted from the last column (function `features_target_sep`).

2.2 Preprocessing Steps

An initial statistical analysis was performed to assess the distribution of the features and the presence of missing values. No missing values were found in any column.

The features presented varying scales and distributions.

2.2.1 Target Variable and Binarization

Firstly, the target variable quality, ranging from 3 to 9, was imbalanced and clearly skewed towards 5, 6, and 7.

```
quality     count
6           2836
5           2138
7           1079
4           216
8           193
3           30
9           5
```

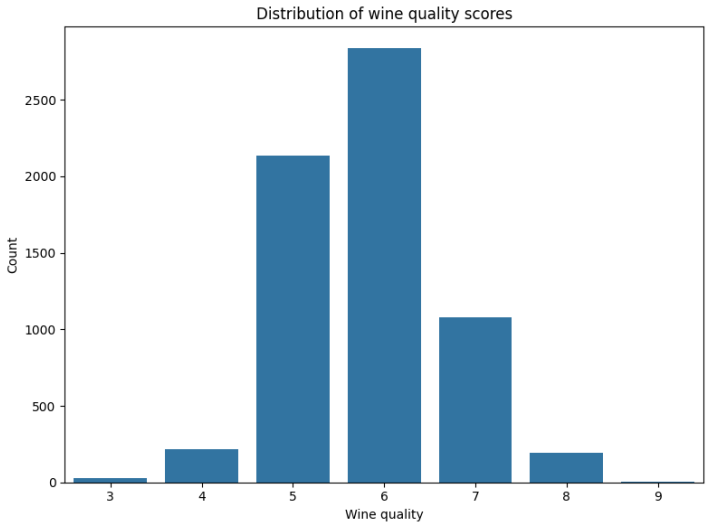


Figure 1: Distribution of wine quality scores in the dataset.

To facilitate binary classification for both Support Vector Machines and Logistic Regression, the labels were binarized differently in each model, directly inside the implementation.

The binarization strategy was based on a threshold decision: wines with quality ≥ 6 were considered "high quality" (positive class), and the rest were "low quality" (negative class).

The resulting class distribution was imbalanced, with approximately 63% positive and 37% negative samples.

```
Original dataset distribution:
  Class False: 2384 samples (36.69%)
  Class True: 4113 samples (63.31%)
```

This motivated the use of a stratified 80-20 train-test split to preserve class proportions in both training and test sets. The function `train_test_split` contains a custom stratified splitting procedure. Samples are divided into two groups according to the quality threshold. Each group is then split into training and test subsets according to the specified proportion (`train_size`), while preserving the original class balance. Optional shuffling ensures randomness in the partitioning. The result is the following:

```
Training set distribution:
  Class False: 1907 samples (36.69%)
  Class True: 3290 samples (63.31%)

Test set distribution:
  Class False: 477 samples (36.69%)
  Class True: 823 samples (63.31%)
```

2.2.2 Feature Scaling

Given the varying magnitudes and ranges of the features (e.g., residual sugar spanned from 0.6 to 65.8 while pH ranged from 2.72 to 4.01), standardization was considered beneficial, especially for SVM which is sensitive to the scale of the features.

Standardization was implemented in the function `standardize`, were the mean and standard deviation were computed from the training set, and then applied to both the training and test sets. After standardization, the mean of the standardized training features was approximately zero with unit variance, as expected. The test features retain a similar distribution, though with minor deviations due to sample differences.

Table 1: Mean and standard deviation of selected features before standardization

Feature	Mean	Std Dev
Alcohol	10.49	1.22
Density	0.9947	0.0029
Residual sugar	5.44	4.76
Volatile acidity	0.339	0.145
Citric acid	0.318	0.162

Table 2: Mean and standard deviation of selected features after standardization

Feature	Mean	Std Dev
Alcohol	≈ 0	1.00
Density	≈ 0	1.00
Residual sugar	≈ 0	1.00
Volatile acidity	≈ 0	1.00
Citric acid	≈ 0	1.00

2.2.3 Correlation Analysis

To gain insights into the relationships between the features and the target variable `quality`, a Pearson correlation matrix was computed and visualized through a heatmap, as shown in Figure 2. This matrix highlights linear dependencies between features, with correlation values ranging from -1 (perfect negative correlation) to $+1$ (perfect positive correlation).

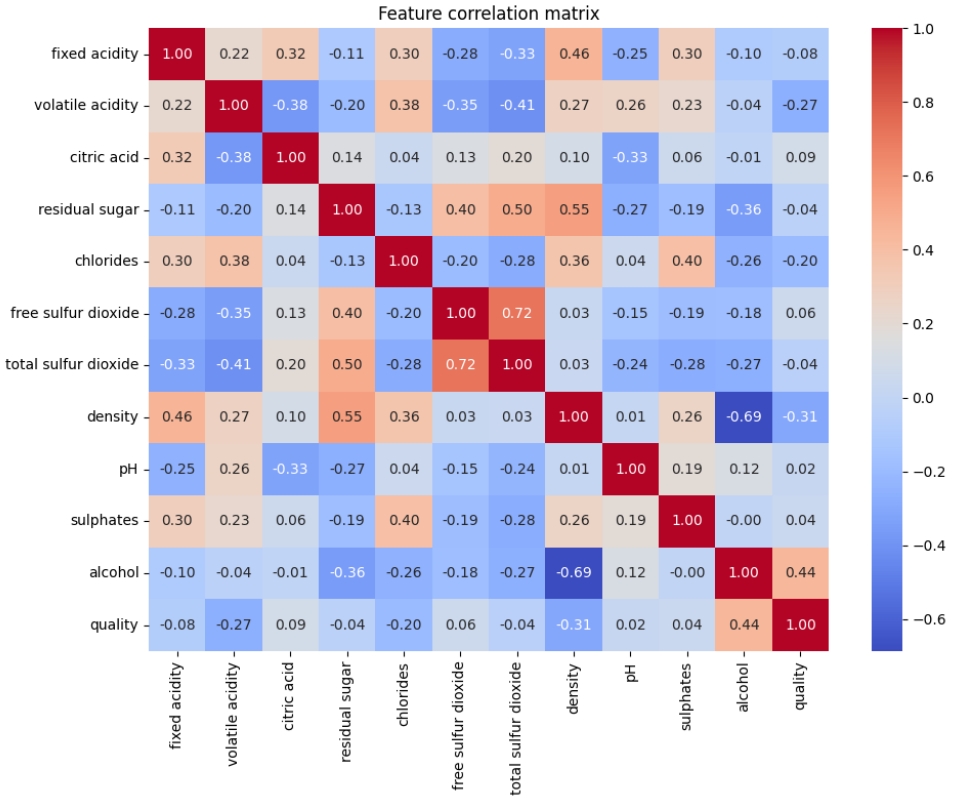


Figure 2: Heatmap of the feature correlation matrix. Strong positive correlations are shown in red, while strong negative correlations appear in blue.

Among the input variables, `alcohol` shows the strongest positive correlation with wine quality ($\approx +0.44$), indicating that higher alcohol levels are generally associated with better wines. On the contrary, `volatile acidity` is negatively correlated (≈ -0.27), suggesting that high levels may negatively affect perceived quality. Most other features exhibit weak or negligible correlation with the target.

The matrix also reveals strong correlations between some input features themselves, such as `free sulfur dioxide` and `total sulfur dioxide` ($\approx +0.72$), and between `density` and `alcohol` (≈ -0.69). These findings are important for identifying multicollinearity, which can affect model performance, especially in linear models without regularization.

2.2.4 Summary

In summary, the preprocessing phase ensured that the dataset was clean, properly scaled, and suitable for binary classification. Stratified sampling preserved the original class imbalance, avoiding biased training or evaluation splits. Finally, feature standardization was crucial to guarantee numerical stability and improve convergence. These steps established a solid foundation for the subsequent model implementation and analysis.

3 Implementation of the models

Since the assignment required the implementation of both linear and kernelized versions of the two models, the code was designed in a modular way. Two classes were created (SMO and LogRegr), each supporting both linear and non-linear formulations specifying the desired kernel type through the hyperparameter `kernel` $\in$ {'linear', 'gaussian'} at initialization.

3.1 Kernel Methods

Kernel methods provide a powerful way to extend linear models to non-linear decision boundaries. While linear models are restricted to hyperplanes, kernelized models adapt to more complex structures and typically achieve higher accuracy when the data is not linearly separable. The central idea is to replace the explicit feature transformation with a kernel function $k(x_i, x_j)$ that computes the similarity between two data points in an implicit high-dimensional feature space. This so-called "kernel trick" allows handling complex, non-linear patterns in the data without explicitly computing the high-dimensional mapping.

In this work, the Gaussian (or Radial Basis Function, RBF) kernel has been employed, defined as

$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right),$$

where σ controls the width of the kernel and therefore the smoothness of the resulting decision boundary. Small values of σ lead to highly flexible models that capture local variations in the data, while larger values yield smoother and more global decision boundaries.

3.2 Support Vector Machine

In binary classification, a SVM Classifier aims at finding the "best" hyperplane that could maximize the margin between the two classes, meaning that the distance between the hyperplane and the nearest data points on each side is the largest.

There are many methods to find the optimal weight vector and one particularly common one is Sequential Minimal Optimization (SMO). The **Sequential Minimal Optimization (SMO)** algorithm was chosen for its ability to efficiently handle both **linear** and **kernelized Support Vector Machines** without modifying the main implementation. By iteratively optimizing

pairs of Lagrange multipliers, SMO reduces the quadratic programming problem to minimal-size subproblems.

A Support Vector Machine computes a linear classifier of the form $f(x) = w^T x + b$. This can also be expressed using inner products as $f(x) = \sum_{i=1}^{m} \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b$ where we can substitute a kernel $K(x^{(i)}, x)$ in place of the inner product.

Kernelized SVM In the Support Vector Machine, kernelization was achieved by replacing the inner product with the Gaussian kernel function. Within the `SMO2` class, the kernel is implemented as:

```
def gaussian_kernel(self, x1, x2):
    return np.exp(-(np.linalg.norm(x1 - x2, 2)) ** 2 / (2 * self.sigma ** 2))
```

This kernel function is called during the optimization to compute the similarity between pairs of training points. Predictions are obtained as weighted combinations of kernel evaluations between the test points and the support vectors:

```
def prediction(self, x):
    return (self.alphas * self.y) @ self.kernel_func(self.X, x) + self.b
```

This formulation enables the SVM to construct non-linear separating surfaces without requiring an explicit feature mapping.

3.2.1 SMO algorithm

The SMO algorithm gives an efficient way of solving the dual problem of the (regularized) support vector machine optimization problem. Specifically, the objective is:

$$\begin{aligned} \max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \\ \text{subject to} \quad & 0 \le \alpha_i \le C, \quad i = 1, \ldots, m, \\ & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \end{aligned}$$

The KKT conditions can be used to check for convergence to the optimal point. For this problem the KKT conditions are

$$\begin{aligned} \alpha_i = 0 \quad &\Rightarrow \quad y^{(i)} \left(w^T x^{(i)} + b \right) \ge 1 \\ \alpha_i = C \quad &\Rightarrow \quad y^{(i)} \left(w^T x^{(i)} + b \right) \le 1 \\ 0 < \alpha_i < C \quad &\Rightarrow \quad y^{(i)} \left(w^T x^{(i)} + b \right) = 1 \end{aligned}$$

In other words, any α_i's that satisfy these properties for all i will be an optimal solution to the optimization problem given above. The SMO algorithm iterates until all these conditions are satisfied (to within a certain tolerance) thereby ensuring convergence.

The SMO algorithm selects two α parameters, α_i and α_j and optimizes the objective value jointly for both these α's. Finally it adjusts the b parameter based on the new α's. This process is repeated until the α's converge.

3.2.2 Implementation

The implementation of the Sequential Minimal Optimization algorithm is encapsulated in a dedicated Python class `SMO`. This class provides a unified framework for training both linear and kernelized SVMs. During initialization, the user can specify the penalty parameter C, the kernel type (linear or Gaussian/RBF), the tolerance level, and the maximum number of iterations.

At the beginning, all multipliers are initialized to zero and the error cache $E_i = f(x_i) - y_i$ is computed for each training point. During training, the algorithm alternates between scanning the full dataset and focusing only on samples whose multipliers lie strictly between 0 and C.

Selecting α Parameters Much of the full SMO algorithm is dedicated to heuristics for choosing which α_i and α_j to optimize so as to maximize the objective function as much as possible.

The method `examine_example` implements a simplified heuristic for choosing the α's. For each candidate α_j, the algorithm checks KKT violations. If violated, a second index α_i is chosen. A common heuristic is to maximize $|E_i - E_j|$, so that the update has a higher chance of improving the objective. If this fails, the algorithm falls back to scanning non-bound multipliers or the full set.

Update of α_i and α_j Given two selected multipliers, the algorithm computes bounds L and H for α_j depending on whether $y_i = y_j$, and the parameter

$$\eta = K(x_i, x_i) + K(x_j, x_j) - 2K(x_i, x_j).$$

If $\eta > 0$, the new α_j is updated as

$$\alpha_j^{\text{new}} = \alpha_j + \frac{y_j(E_i - E_j)}{\eta},$$

then clipped to the interval $[L, H]$. The other multiplier follows as

$$\alpha_i^{\text{new}} = \alpha_i + y_i y_j (\alpha_j - \alpha_j^{\text{new}}).$$

Numerical tolerances are applied to avoid instability near 0 or C.

Bias update After updating the multipliers, the bias term b is recomputed. Two candidates are obtained:

$$b_1 = b - E_i - y_i(\alpha_i^{\text{new}} - \alpha_i)K(x_i, x_i) - y_j(\alpha_j^{\text{new}} - \alpha_j)K(x_i, x_j),$$

$$b_2 = b - E_j - y_i(\alpha_i^{\text{new}} - \alpha_i)K(x_i, x_j) - y_j(\alpha_j^{\text{new}} - \alpha_j)K(x_j, x_j).$$

If α_i^{new} or α_j^{new} lies strictly between 0 and C, the corresponding bias is chosen; otherwise, b is set to $(b_1 + b_2)/2$.

Error cache and stopping The error cache is updated incrementally after each step, avoiding full recomputation. The outer loop alternates between exhaustive scans and restricted scans until no significant changes occur or the maximum number of iterations is reached. At each iteration, hinge loss and training accuracy are recorded, enabling convergence monitoring and later visualization.

3.3 Logistic Regression

Logistic Regression predicts the probability that a given input belongs to a certain class, using the sigmoid function, which maps any real-valued number into a range between 0 and 1.

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

In the implementation, the sigmoid function is defined as follows:

```
def sigmoid(self, z):
    return 1.0 / (1.0 + np.exp(-z))
```

Here, z is the logit

$$z = \mathbf{w}^\top \mathbf{x} + b$$

where $\mathbf{x} \in \mathbb{R}^{12}$ is the input feature vector, $\mathbf{w} \in \mathbb{R}^{12}$ are the weights, and $b \in \mathbb{R}$ is the bias term.

```
self.weights = np.zeros(X.shape[1])
self.sigmoid(X @ self.weights)
```

Logistic Regression uses a loss function called Log Loss, which penalizes incorrect predictions. The objective is to find the weights and bias which minimize this loss. This can be achieved using gradient descent, which iteratively updates the parameters in the direction opposite to the gradient of the loss.

$$\mathcal{L}(\mathbf{w}, b) = -\frac{1}{n} \sum_{i=1}^{n} \Big[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \Big]$$

3.3.1 Kernelized Logistic Regression Implementation

To achieve modularity and avoid code duplication, the implementation of Logistic Regression in this project supports both linear and Gaussian (RBF) kernels, separating the kernel computation from the gradient descent routine. The implementation is inside the class `KernelizedLogRegr`.

Modularization Strategy The class distinguishes between the kernel type using the `self.kernel` attribute. Depending on this attribute, the feature transformation is computed differently:

- **Linear kernel:** For the linear case, the input matrix X is augmented with an intercept column so that the bias term b can be absorbed into the weight vector

  $$\theta = [b, w_1, w_2, \ldots, w_d]^\top,$$

  allowing to write the model compactly as

  $$\hat{y} = \sigma(\mathbf{X}'\theta).$$

  This approach ensures that the gradient descent update automatically handles both the weights and the bias in a unified manner. This is implemented in the `_add_intercept()` method.

  ```
  def _add_intercept(self, X):
      intercept = np.ones((X.shape[0], 1))
      return np.concatenate((intercept, X), axis=1)
  ```

- **Gaussian kernel:** For the RBF kernel, the `_gaussian_kernel()` method computes the full kernel matrix between the training points. This returns an $n \times n$ matrix, where n is the number of training samples during the fit phase, a $m \times n$ matrix, where m is the number of test samples during the predict phase.

Both transformations are accessed via the `_compute_features()` method, which internally selects the correct kernel based on the `self.kernel` attribute. This allows the `fit()` and `predict()` routines to be written generically, without needing separate implementations for each kernel type.

Training After computing the feature matrix Z, gradient descent is performed using the standard logistic regression loss. The gradient update formula is the same for both kernels:

$$\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \eta \frac{1}{n} Z^\top (\sigma(Z\boldsymbol{\alpha}) - \mathbf{y})$$

where:

- $\boldsymbol{\alpha}$ is the vector of model parameters (weights). In the linear case, this includes the bias term; in the kernel case, it corresponds to the coefficients of the kernel expansion.
- η is the learning rate, controlling the step size of each gradient descent iteration. A small η ensures stable convergence, while a large η can speed up training but may cause divergence.
- n is the number of training samples, included to normalize the gradient.
- Z is the feature matrix:
  - For a linear kernel, $Z = [\mathbf{1}, X]$ is the input matrix augmented with a column of ones for the bias.
  - For a Gaussian (RBF) kernel, Z is the kernel matrix K with $K_{ij} = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right)$.
- $\sigma(\cdot)$ is the sigmoid function, applied element-wise to the vector $Z\boldsymbol{\alpha}$, producing the predicted probabilities for each training sample.
- $\mathbf{y}$ is the vector of target labels, mapped to $\{0, 1\}$.
- The term $Z^\top(\sigma(Z\boldsymbol{\alpha}) - \mathbf{y})$ represents the gradient of the logistic loss with respect to the parameters.

This is implemented as follows:

```
preds = self.sigmoid(X @ self.weights)
gradient = X.T @ (preds - y) / len(y)
self.weights -= self.lr * gradient
```

and it is repeated `n_iters` times or when convergence is reached. The change in the loss function is monitored between consecutive iterations. The training process stops early if the absolute difference in loss falls below a specified threshold `tol`.

Prediction The `predict()` method also uses `_compute_features()` to transform any new input, maintaining the same modular structure. This ensures that linear and Gaussian kernels can be used interchangeably with minimal changes to the training and prediction code.

4 Hyperparameter Tuning

Machine learning models often rely on hyperparameters, which are not learned directly from the data but must be set before training. The choice of hyperparameters strongly affects model performance and generalization ability. Therefore, systematic hyperparameter tuning is a crucial step of the training pipeline.

4.1 Grid Search

A common strategy to tune hyperparameters is *grid search*, where a finite set of candidate values is defined for each hyperparameter and all possible combinations are evaluated. The grid is implemented in the fuction `parameter_grid`, which takes in input the dictionary of hyperparameters (where each key is a hyperparameter and each value is a list of candidate values) and returns in output all combinations as dictionaries, ready to be passed to the model constructor. The function efficiently considers the fact that the `sigma` parameter is not needed for linear kernel.

4.2 Nested Cross-Validation

To avoid overfitting to a single train-validation split, hyperparameter evaluation is performed using `nested cross-validation`. In nested CV, the data is split into outer folds. For each outer split, the training portion undergoes an inner cross-validation loop to select the best hyperparameters (via grid search). Then, the model is retrained on the full outer training set with the selected parameters and evaluated on the outer test set. This procedure ensures an unbiased estimate of generalization error.

In the code, a 5-fold nested cross-validation routine was implemented as follows:

- The outer loop splits the dataset into k_{outer} folds, in this case 5. The split in folds is done by a function `kfold_indices`, which makes a stratified split maintaining the original class distribution across folds.
- Each training set is split into k_{inner} folds. In this project, due to the size of the dataset, the number of inner folds has been chosen to be 5 as well.
- For each hyperparameter combination generated by `parameter_grid` and for each inner fold, the model is trained on the training part and tested on the validation part.
- The combination achieving the lowest average validation error is selected.
- The model is retrained with the best hyperparameters on the entire outer training set and evaluated on the outer test set.
- Test errors across outer folds are averaged to obtain the final performance estimate.

4.3 Hyperparameters

In this project two models have been implemented: SVM and Logistic Regression. However, due to computational constraints, nested cross-validation was applied only to the Logistic Regression model. For this model, a grid of hyperparameters was defined. If not properly tuned, they may lead to *underfitting* (an overly simple model that cannot capture the structure of the data) or *overfitting* (an overly complex model that fits noise in the training set, resulting in poor generalization).

Listing 1: Parameter grid for Logistic Regression

```
logregr_param_grid = {
    "learning_rate": [1e-4, 1e-3, 1e-2, 1e-1, 1],
    "kernel": ['linear', 'gaussian'],
    "sigma": [0.1, 0.5, 1., 5., 10.]        # only for gaussian kernel
}
```

The `learning_rate` controls the step size in the gradient descent optimization: too small values can lead to slow convergence, while too large values may cause divergence or oscillations. The `kernel` can be either `linear` or `gaussian`, with the latter allowing non-linear decision boundaries. For the Gaussian kernel, the parameter `sigma` regulates the spread of the radial basis function: small values of `sigma` produce very localized decision regions (risk of overfitting), while large values produce smoother, more global decision boundaries (risk of underfitting).

4.4 Evaluation Metrics

In this project, several evaluation metrics were used to assess the performance of the implemented models. Each is implemented in a dedicated function within the file `evaluation.py`.

- Accuracy measures the overall proportion of correctly classified instances.
- Precision measures the proportion of true positive predictions among all instances classified as positive, which is useful to understand the model's reliability when predicting a certain class.
- Recall measures the proportion of true positives correctly identified out of all actual positive instances, highlighting the model's ability to detect positive cases.
- The F1-score is the harmonic mean of precision and recall, balancing the trade-off between these two metrics, especially important in imbalanced datasets.
- Finally, the confusion matrix gives a detailed breakdown of true positives, true negatives, false positives, and false negatives, allowing an in-depth inspection of the types of errors the model makes.

Together, these metrics give a general picture of models performances enabling to better understand underfitting and overfitting tendencies.

5 Results and Analysis

5.1 Logistic Regression

Through nested cross-validation, the best configuration for Logistic Regression was found to be:

`{'learning_rate': 10, 'sigma': 0.5, 'kernel': 'gaussian'}`

This setup achieved the following results:

- Training accuracy: ≈ 0.96
- Test accuracy: ≈ 0.81

The training accuracy is quite high (≈ 0.96), while the test accuracy is lower (≈ 0.81). This gap indicates that the model does capture patterns from the training data effectively, but it also shows signs of overfitting: the decision boundary learned during training does not perfectly generalize to unseen samples. However, the difference is not extreme.

The outer cross-validation errors (0.17–0.21) are consistent across folds, which indicates that the model has stable behavior and good generalization. This stability reduces the risk of high variance.

Precision (87.3%) is higher than recall (83.4%), showing that the model tends to produce fewer false positives, but some true positives are missed. The F1-score of 0.840 confirms a good balance between precision and recall.

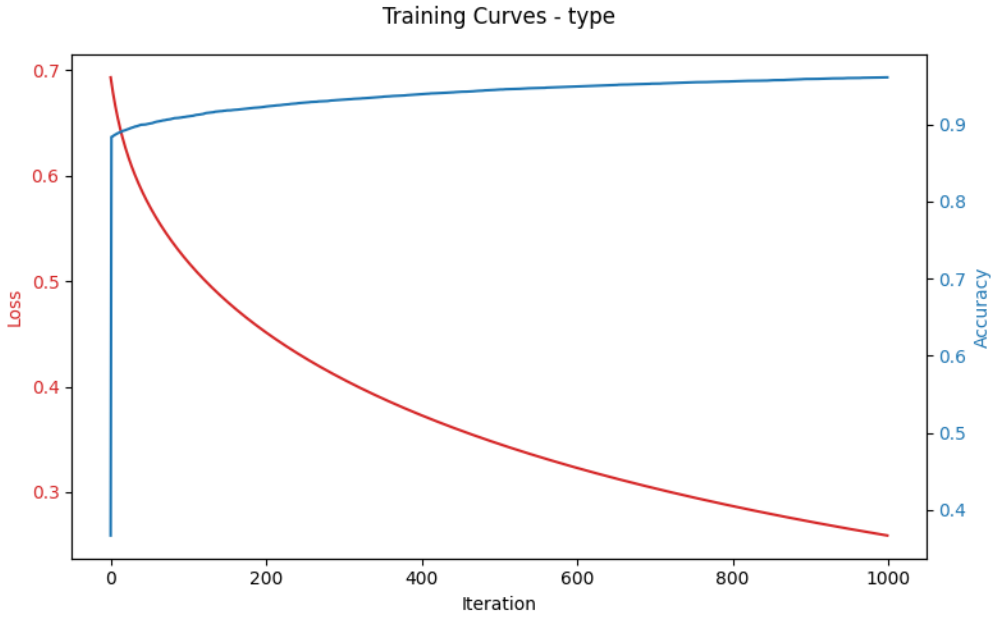


Figure 3: Loss and accuracy curves of best parameters on model LogRegr

Linear vs. Gaussian To assess the impact of kernelization, Linear Logistic Regression was also evaluated using manual tests across different learning rates.

For very small learning rates (e.g., 0.001), the model converged slowly and achieved moderate performance, with a test accuracy around 0.72. The best compromise between training and test accuracy was obtained with a learning rate of 0.5, achieving a training accuracy of 0.73, test accuracy of 0.75, and an F1-score of 0.81. Further increasing the learning rate beyond 1 caused a decrease in performance, indicatingunderfitting, with both training and test accuracy dropping below 0.70.

These results suggest that, even with the optimal learning rate, the linear model is still limited in capturing non-linear patterns in the dataset, as reflected by its lower performance compared to the Gaussian kernel configuration.

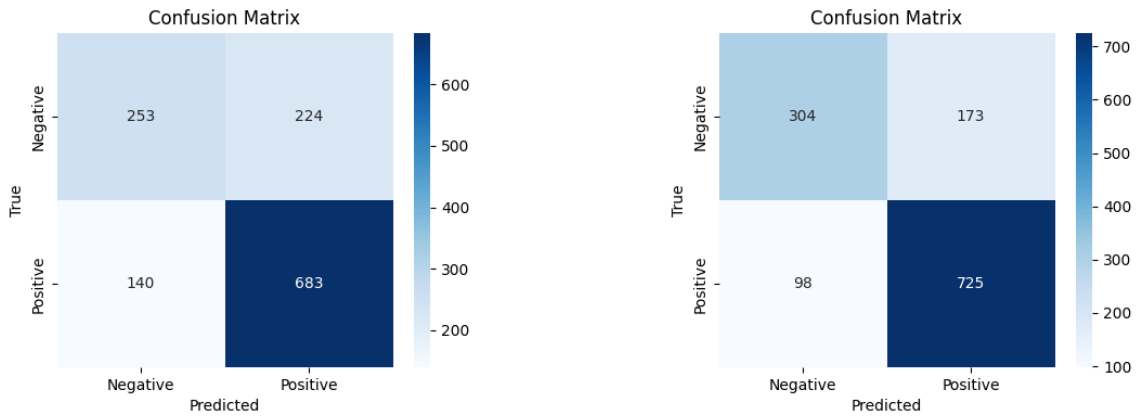
This comparison demonstrates that kernel methods provide tangible improvements when the dataset contains non-linear patterns.

5.2 SVM

Differently from Logistic Regression, SVM was evaluated using a standard train/test split. Since this model has got some hyperparameters to tune as well, they were "tuned" by hand.

Listing 2: Parameter grid for SVM

```
svm_param_grid = {
    "C": [1e-3, 1e-2, 1e-1, 1, 10, 100],
    "kernel": ['linear', 'gaussian'],
    "sigma": [0.1, 0.5, 1., 5., 10.]        # only for gaussian kernel
```



Linear Kernel Gaussian Kernel

Figure 4: Confusion matrices for Logistic Regression with Linear and Gaussian kernels.

}

The parameter C controls the regularization strength: smaller values of C mean wider margin, so higher tolerance, while bigger values of C mean narrower margin, so potential overfitting.

The optimal value of C often differs between a Linear and a Gaussian SVM. In Linear SVM, since the model's capacity is limited, higher values of C are preferred. In Gaussian SVM, since the kernel maps the data into a higher-dimensional space, increasing flexibility, it increases the risk of overfitting when C is too large. In Gaussian SVM C needs to be tuned together with the sigma parameter.

SVM was trained twice: once with Linear kernel, once with Gaussian kernel.

Linear The linear SVM model was trained and evaluated using several values of the regularization parameter C, ranging from very small ($C = 0.0001$) to very large ($C = 100$). Extremely low values of C resulted in severe underfitting: the model predicted almost exclusively the positive class, yielding high recall but poor precision and overall accuracy. Conversely, very high values of C caused overfitting: the model attempted to satisfy the margin constraints too strictly, leading to poor generalization on the test set.

Since initial experiments suggested that the optimal range for C lay between 0.1 and 1, new evaluations were conducted for $C = 0.3$, 0.5, and 0.8 to refine this search. At the end, the best compromise was found to be $C = 0.5$.

This setup achieved a test accuracy of approximately 0.76, a balanced precision of 0.80, and a recall of 0.83, resulting in the highest F1-score among the intermediate values. The confusion matrix confirmed that the model correctly recognized both positive and negative classes without a substantial bias toward either.

C	Train Acc	Test Acc	Precision	Recall	F1-score
0.1	0.737	0.727	0.793	0.769	0.781
0.5	**0.737**	**0.760**	**0.801**	**0.826**	**0.813**
1.0	0.747	0.728	0.781	0.793	0.787

Table 3: Linear SVM performance metrics for selected values of the regularization parameter C.

Gaussian The hyperparameter tuning on SVM with Gaussian kernel was based on the tuning of the regularization parameter C and the kernel width σ. As in the linear case, extreme values

were first tested to observe the overall behavior. For instance, with very small values such as $C = 0.1$ and $\sigma = 0.1$, the model exhibited strong underfitting, reaching only about 63% accuracy on both train and test sets. In this configuration, recall was maximized (equal to 1.0), but precision was extremely low, leading to poor generalization. On the opposite side, with large C values (e.g., $C = 5$, $\sigma = 1$), the model achieved very high training accuracy (above 94%) but only a limited improvement in test accuracy, showing a tendency toward overfitting.

Better results were reached keeping $\sigma = 1$ and varying C between 1 and 5. In this range, test accuracy and F1-score improved significantly. The best compromise was obtained with $C = 3$, achieving a test accuracy slightly above 80% and the highest F1-scores (around 0.85). Increasing C further to 4 or 5 increased training performance but reduced generalization capability, confirming an overfitting effect.

Table 4 reports the most relevant results for selected parameter choices.

C	σ	Train Accuracy	Test Accuracy	Precision	F1-score
1	1	0.879	0.788	0.806	0.840
3	**1**	**0.930**	**0.803**	**0.823**	**0.850**
5	1	0.946	0.799	0.823	0.846

Table 4: Results of the Gaussian kernel SVM for different values of C with $\sigma = 1$.

Linear vs. Gaussian The linear model is characterized by a simpler decision boundary, which makes it more sensitive to the choice of the regularization parameter C. The best trade-off for the linear SVM was found for intermediate values of C (around 0.5), with a test accuracy of approximately 75–76% and a balanced F1-score around 0.81.

On the other hand, the Gaussian kernel introduced additional flexibility through the parameter σ, allowing the model to capture more complex decision boundaries. In this case, the most effective configurations were obtained with $\sigma = 1$ and intermediate values of C (2 or 3). These settings provided the highest test accuracy (slightly above 80%) and F1-scores around 0.85, showing a clear improvement over the linear model in terms of generalization.

Overall, the Gaussian kernel achieved better performance than the linear model, especially in terms of recall and F1-score, at the cost of higher computational complexity.