Optimization Methods for Data Science 2023-2024
Final Project Report

Lorenzo Pannacci 1948926

Part 1, Question 1: Multi-Layer Perceptron

The first task of the homework is the implementation a Multi-Layer Perceptron (MLP) neural network for the regression task of age prediction.

1.1 Implementation of forward-pass and backpropagation

The main concern in building a neural network from scratch is the implementation of the forward-pass and back-propagation operations, usually not a problem for the user since automatic differentiation tools are available in every modern machine learning library.

The **forward-pass** is the operation that propagates the input data through all the layers of the network until the output layer is reached and the prediction is generated. Each layer uses as input the result of the previous, computes the dot-product between its weight matrix and its input, sums the bias vector and then applies the activation function, producing its output. Since we are in a regression task the activation is not used in the last layer.

If we call x the input vector, $\hat{y}$ the output prediction, L the number of hidden layers, ω_i the weight matrix and β_i the bias vector of the i-th layer, Z_i the result of the dot-product at layer i plus its bias and A_i what we get after the application of the activation function $a(\cdot)$ we can summarize the forward-pass operation as:

$$Z_i = \begin{cases} x \cdot \omega_i + \beta_i & \text{if } i = 0 \\ A_{i-1} \cdot \omega_i + \beta_i & \text{otherwise} \end{cases}, \quad A_i = \begin{cases} Z_i & \text{if } i = L \\ a(Z_i) & \text{otherwise} \end{cases}, \quad \hat{y} = A_L = Z_L$$

The **backpropagation** simplifies the computation of the error gradients needed for the gradient descent by exploiting the **chain rule** for derivatives. Our loss function is the Mean Squared Error (MSE) with a L2 Regularization:

$$L(\omega, \beta) = E(\omega, \beta) + R(\omega, \beta) = \frac{1}{N} \sum_{n=1}^{N} (y_n - \hat{y}_n)^2 + \lambda \sum_{i=0}^{L} ||\omega_i||^2$$

We analyze separately the two components E (mean squared error) and R (regularization). The derivatives for E are the main issue since they depend on the prediction which itself is a function of both ω and β. Here we apply the backpropagation: we travel the network backwards using the chain rule. For simplicity we use a single sample.

For the output layer:

$$\begin{aligned}
&\frac{dE}{d\hat{y}} = \frac{dE}{dA_L} = \frac{dE}{dZ_L} = -2(y - \hat{y}) \\
&\frac{dE}{d\omega_L} = \frac{dE}{dZ_L}\frac{dZ_L}{d\omega_L} = \frac{dE}{dZ_L}\frac{d(A_{L-1} \cdot \omega_L + \beta_L)}{d\omega_L} = -2(y - \hat{y})A_{L-1} \\
&\frac{dE}{d\beta_L} = \frac{dE}{dZ_L}\frac{dZ_L}{d\beta_L} = \frac{dE}{dZ_L}\frac{d(A_{L-1} \cdot \omega_L + \beta_L)}{d\beta_L} = -2(y - \hat{y}) \\
&\frac{dE}{dA_{L-1}} = \frac{dE}{dZ_L}\frac{dZ_L}{dA_{L-1}} = \frac{dE}{dZ_L}\frac{d(A_{L-1} \cdot \omega_L + \beta_L)}{dA_{L-1}} = -2(y - \hat{y})\omega_L
\end{aligned}$$

For the hidden layers:

$$\begin{aligned}
&\frac{dE}{dZ_i} = \frac{dE}{dA_i}\frac{dA_i}{dZ_i} = \frac{dE}{dA_i}\frac{d(a(Z_i))}{dZ_i} = \frac{dE}{dA_i}a'(Z_i) \\
&\frac{dE}{d\omega_i} = \frac{dE}{dZ_i}\frac{dZ_i}{d\omega_i} = \frac{dE}{dZ_i}\frac{d(A_{i-1} \cdot \omega_i + \beta_i)}{d\omega_i} = \frac{dE}{dZ_i}A_{i-1} \\
&\frac{dE}{d\beta_i} = \frac{dE}{dZ_i}\frac{dZ_i}{d\beta_i} = \frac{dE}{dZ_i}\frac{d(A_{i-1} \cdot \omega_i + \beta_i)}{d\beta_i} = \frac{dE}{dZ_i} \\
&\frac{dE}{dA_{i-1}} = \frac{dE}{dZ_i}\frac{dZ_i}{dA_{i-1}} = \frac{dE}{dZ_i}\frac{d(A_{i-1} \cdot \omega_i + \beta_i)}{dA_{i-1}} = \frac{dE}{dZ_i}\omega_i
\end{aligned}$$

For the input layer:

$$\frac{dE}{d\omega_0} = \frac{dE}{dZ_0}\frac{dZ_0}{d\omega_0} = \frac{dE}{dZ_0}\frac{d(x \cdot \omega_0 + \beta_0)}{d\omega_0} = \frac{dE}{dZ_0}x$$
$$\frac{dE}{d\beta_0} = \frac{dE}{dZ_0}\frac{dZ_0}{d\beta_0} = \frac{dE}{dZ_0}\frac{d(x \cdot \omega_0 + \beta_0)}{d\beta_0} = \frac{dE}{dZ_0}$$

The derivatives for R are immediate: it has no influence on β and for the ω we have just a basic derivative:

$$\frac{dR}{d\beta_i} = 0, \quad \frac{dR}{d\omega_i} = 2\lambda\omega_i$$

The implemented code for the two operations is just a vectorized version of those reasonings.

1.2 Optimization routine

To approach the training of the neural network as an optimization problem we used the method *scipy.optimize.minimize()* of the Python library **SciPy**, which contains different algorithms for the minimization of scalar functions in one or more variables. Here is a breakdown of the parameters of the method and their roles in the optimization task:

- *fun*: **objective function** to be minimized. The first argument to this function is x, the vector we are optimizing on, in our case the **vector of all the weights and biases of the neural network**. Since our aim is to minimize the loss we use as objective a wrapper function that updates the parameters to those given in input and computes the loss on the training set, with predictions obtained from the **forward-pass** with the new parameters.
- *x0*: **initial guess** for the vector x. What we give here is the vectorized version of the initialized parameters of the neural network: the weights follow *Xavier/Glorot normal initialization* while biases are initialized with *zeros*.
- *method*: solver algorithm used. Our choice is *L-BFGS-B* due to **computational efficiency** reasons (it is an approximate method) and because it gives the possibility of using a **custom gradient computation function**.
- *jac*: method used for the gradient computation. In our case it is the **backpropagation**, as before with a wrapper used to update the parameters beforehand and to output a vectorized version of the gradients.
- *options*: arguments that are specific to the solver used. Those options and the values we have given to them are the maximum amount of **iterations of the algorithm** {'*maxiter*': 1000}, the **tolerance value for convergence** {'*ftol*': $1e-4$} and the maximum amount of **iterations for the linesearch** sub-routine {'*maxls*': 20}.

1.3 Hyperparameters and cross-validation

The hyperparameters of our MLP are the **structure of the network** (the amount of layers L and the amount of neurons for each layer N_l), the **term** λ that determines the importance of the regularization over the mean squared error in the loss formula and the **activation function** $a(\cdot)$, which gives non-linearity to the network.

We are constrained in the amount of hidden layers to 2, 3 or 4. We tried multiple combinations with the main criterion of following a "**pyramid shape**", meaning that we go from many neurons to few. The reason for this choice is that usually neural networks build at first many simple features that get more complex as they get mixed and merged in subsequent layers. A reverse pyramid shape would be unreasonable because few neurons force the network to compress the representation and information may be lost and therefore not used in predictions.

The two activation functions suggested by the homework (***Sigmoid*** and ***Hyperbolic tangent***) both suffer of the **vanishing gradient problem**, a phenomenon that occurs because increasing of the amount of layers makes the magnitude of the derivatives on the first layers small, slowing down or even outright stopping the training process, deteriorating performances despite the increased potential representational capabilities of the network.

Although our networks may be too shallow to present this problem we wanted to add to our hyperparameter space values that can take into account this eventuality. Usually the solution to vanishing gradient is the use of different activation functions such as *ReLU* and its variants, however here we are not able to use them since they are **not differentiable in zero** and the homework require us to use only differentiable activation functions.

We found in literature a good compromise of an function that is continuously differentiable and said to be less incline to saturation in the ***1-Swish*** function, which shape is a reasonable approximation of a *ReLU* in both function and derivative (as shown in Figure 2) and is defined as:

$$swish(x) = x * \sigma(x) = \frac{x}{1 + e^{-x}}$$

$$swish'(x) = \sigma(x) + x * \sigma(x) * (1 - \sigma(x)) = \frac{1 + e^{-x} + xe^{-x}}{(1 + e^{-x})^2}$$

In the end we discovered that the imposed limit on the amount of hidden layers avoids the presence of the vanishing gradients, however even increasing the amount of layers to just 6 there are instances of networks that with a *Sigmoid* activation end up returning always the same value but behave normally when the *Swish* activation is used.

As suggested by the tutor we have not tried to use different activation functions for each layer to avoid an exponential increase in the search space of the cross-validation.

Finally for the regularization parameter λ we selected values from across multiple magnitude orders and also a zero value, meaning that no regularization is applied. We report in Table 1 the hyperparameter space:

Hyperparameter	**Values**
Network structure	(8, 4), (16, 8), (32, 16), (64, 32), (16, 8, 4), (32, 16, 8), (64, 32, 16), (16, 8, 4, 2), (32, 16, 8, 4)
λ	0, 1e-4, 5e-4, 1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 5e-1, 1e+0
Activation function	Sigmoid, Hyperbolic tangent, Swish

Table 1: Hyperparameters for MLP k-fold cross-validation

The criterion of choice is the **highest mean validation MAPE across all folds**. The k-fold cross validation (executed with $k = 5$) gave us as best combination:$\{network\_structure : (64, 32), \quad \lambda : 0.05, \quad activation : \text{1-swish}\}$. All the requested informations are in Table 2 and 3 and the message returned in output by the optimization routine is in Figure 1.

The message of the optimizer reports a positive termination message (*success: True*) saying that the algorithm stopped due to convergence with the given tolerance (*message: CONVERGENCE*), it also reports the value of the objective function (*fun: 98.160*), the final values of the input vector and the Jacobian (*x, jac*), the number of iterations performed by the optimizer (*nit: 75*) and the number of evaluations of the objective functions and of its Jacobian (*nfev: 85, njev: 85*).

PERFORMANCE								
LOSS				MAPE				
INITIAL TRAIN	FINAL TRAIN	FINAL VAL	FINAL TEST	INITIAL TRAIN	FINAL TRAIN	FINAL VAL	FINAL TEST	**CPU TIME**
1678.542	98.160	98.982	92.782	100.204	23.203	23.224	23.435	38.429 s

Table 2: Performance table MLP

SETTINGS				
HIDDEN LAYER 1		HIDDEN LAYER 2		OTHER
NUMBER OF NEURONS	NON-LINEARITY	NUMBER OF NEURONS	NON-LINEARITY	REGULARIZATION TERM
64	1-Swish	32	1-Swish	0.05

Table 3: Settings table MLP

```
 message: CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
 success: True
  status: 0
     fun: 98.1609475592716
       x: [ 2.185e-04 -3.456e-02 ...  9.314e-01  2.326e+00]
     nit: 75
     jac: [-2.522e-03 -1.693e-02 ...  1.012e-02  3.171e-02]
    nfev: 85
    njev: 85
hess_inv: <4225x4225 LbfgsInvHessProduct with dtype=float64>
```

Figure 1: Output message of the solver for the training of the MLP with the reported hyperparameters.

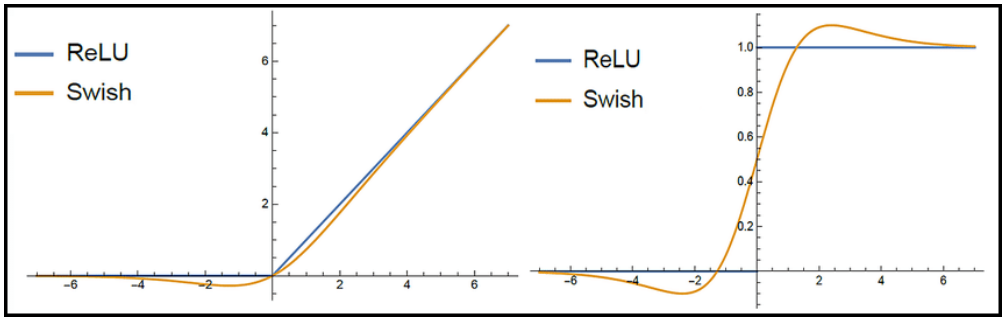

Figure 2: Comparison between 1-Swish and ReLU on the left and their derivatives on the right.

Part 2: Support Vector Machine

The second part of the homework consists in the creation and training of Support Vector Machines using two different approaches seen during the course and then use them for a multi-class classification task.

2.1 Question 2: training SVM via solving the dual quadratic problem

As seen during the lectures the training of a (*soft-margin*) SVM consists in finding the best hyperplane that separates the two classes of training data. Given the quadratic nature of the problem the **strong duality condition** holds and **the solution of the primal can be recovered from the solution of the dual**. Solving the dual instead of the primal is preferred due to computational efficiency and the possibility of using the **kernel trick**.

Given a dataset of size L of samples (x_i, y_i) and a kernel function $K(\cdot, \cdot)$ we can write the dual problem as:

$$\begin{array}{ll} \text{minimize} & (1/2)\lambda^T Q \lambda - e^T \lambda \\ \text{subject to} & y^T \lambda = 0 \\ & \mathbf{0} \leq \lambda \leq \mathbf{C} \end{array} \tag{1}$$

Where:

$$\lambda = \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_L \end{pmatrix}, \quad q_{ij} = y_i y_j K(x_i, x_j), \quad e = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ \vdots \\ y_L \end{pmatrix}, \quad \mathbf{0} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \mathbf{C} = \begin{pmatrix} C \\ \vdots \\ C \end{pmatrix},$$

To solve this problem we use the Python library **CVXOPT** and in particular its method for **solving numerically** quadratic programming problems *cvxopt.solvers.qp()*, which requires to formulate the problem in the following form:

$$\begin{array}{ll} \text{minimize} & (1/2)x^T P x + q^T x \\ \text{subject to} & Gx \preceq h \\ & Ax = b \end{array} \tag{2}$$

We can transform the problem (1) in the form of (2) by setting the components as follows:

$$x = \lambda, \quad P = Q = \begin{pmatrix} q_{11} & q_{12} & \dots & q_{1L} \\ q_{21} & q_{22} & \dots & \vdots \\ \vdots & & \ddots & \\ q_{L1} & \dots & & q_{LL} \end{pmatrix}, \quad q = -e = \begin{pmatrix} -1 \\ -1 \\ \vdots \end{pmatrix},$$

$$G = \begin{pmatrix} -I \\ I \end{pmatrix} = \begin{pmatrix} -1 & 0 & \dots & 0 \\ 0 & -1 & 0 & \dots \\ \vdots & & \ddots & \\ 0 & \dots & 0 & -1 \\ 1 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots \\ \vdots & & \ddots & \\ 0 & \dots & 0 & 1 \end{pmatrix}, \quad h = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ C \\ C \\ \vdots \\ C \end{pmatrix}, \quad A = y^T = \begin{pmatrix} y_1 & \cdots & y_L \end{pmatrix}, \quad b = 0$$

The solution we get from the solving the dual problem is λ^*, the vector of the **Lagrange multipliers**. The so-called **support vectors** are all those samples with a positive multiplier: $SV = \{i : \lambda_i^* > 0\}$.

When we use the kernel trick we can't explicitly write the separating hyperplane in the original input space but we can write the prediction made by the Support Vector Machine as:

$$\hat{y}(x) = sgn\left(\sum_{i \in SV} \lambda_i^* y_i K(x_i, x) + b^*\right), \text{ where } b^* = \frac{1}{|SV|} \sum_{i \in SV} \left(y_i - \sum_{j \in SV} \lambda_j^* y_j K(x_i, x_j)\right) \tag{3}$$

As per the **settings of the optimization routine** we tried different combinations without finding differences, if not for an increased computation time by lowering the tolerances, but without an increase in performance. For this reason everything is kept as **DEFAULT** with the exception of '*show_progress*' which is set to *False* to avoid prints.

2.2 SVM Cross-validation

We decided to use as kernel function the **Gaussian kernel**. *Question 2* require us to use the **k-fold cross-validation** to identify the best values for C, the upper bound of the inequality constraints of the dual and γ, the parameter of the kernel. We chose a hyperparameter space (Table 4) with a **wide array of magnitude orders for both variables** because we wanted to find optimal values but also identify situations of both underfitting and overfitting and smaller changes didn't provide meaningful differences.

Hyperparameter	**Values**
γ	1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e+0, 1e+1, 1e+2, 1e+3, 1e+4
C	1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e+0, 1e+1, 1e+2, 1e+3

Table 4: Hyperparameters for SVM k-fold cross-validation

The criterion of choice is the **highest mean validation accuracy across all folds**. The k-fold cross validation (executed with $k = 5$) gave us as best combination $\{C : 0.1, \gamma : 1.0\}$. All the requested informations are in Table 5.

2.3 Study of underfitting and overfitting

Interpreting the prediction formula written above (Formula 3) the γ determines the influence each support vector has on the prediction and by reversing this explanation we can say it **determines the range of influence of each support vector**. Given the nature of negative exponential of the kernel high values of γ leads to quickly decreasing outputs in function of the distance while small values bring an increasingly uniform behavior across all distances.

The *plot of train and test accuracy vs. γ with fixed C* (Figure 3) confirms that: with high γ it is likely that just the closest support vector determines the whole classification leading to overfitting, while with small γ all support vectors have an increasingly similar influence on every point, making the model always predict the same class, leading to underfitting. The plot stops to change when those two extreme behaviors are reached.

A similar discourse can be made for C: it determines the **trade-off between maximizing the margin and avoiding misclassifications on the train set**. A large C leads to a behavior similar to an *hard-SVM*, the model aims to classify correctly all the training points at the expense of a narrower margin. This causes a worse generalization ability, since the model is forced to learn the noise to correctly classify outliers (overfitting).

On the other hand a low C focuses on optimizing the margin disregarding errors, which in some datasets can lead to an overly simplistic behavior as the model is more concerned to separate large clusters of data points while ignoring meaningful patterns in other regions of the feature space. Similarly to before both those behaviors are shown in the *plot of train and test accuracy vs. C with fixed γ* (Figure 4).

2.4 Question 3: training SVM using the SMO-MVP algorithm

The task of *Question 3* is the implementation of the **Sequential Minimal Optimization** (SMO) algorithm for finding the solution of the same SVM problem as before by sequentially solving smaller sub-problems obtained by the restriction to a working set of indices. We are asked to fix the dimension of the sub-problems to $q = 2$ and use as **Working Set Selection Rule** (WSSR) the **Most Violating Pair** (MVP), which work as follows:

$$(i,j), \quad i \in I(a^k), \quad j \in J(a^k), \quad I(a^k) = argmax_{i \in R(a^k)} \left\{ -\frac{\nabla_i f(a^k)}{y_i} \right\}, \quad J(a^k) = argmin_{j \in S(a^k)} \left\{ -\frac{\nabla_j f(a^k)}{y_j} \right\}$$

$$R(a^k) = L^+(a^k) \cup U^-(a^k) \cup \{i : 0 < a_i^k < C\}, \quad S(a^k) = L^-(a^k) \cup U^+(a^k) \cup \{i : 0 < a_i^k < C\},$$

$$L^+(a^k) = \{i : a_i^k = 0, y_i = 1\}, L^-(a^k) = \{i : a_i^k = 0, y_i = -1\}, U^+(a^k) = \{i : a_i^k = C, y_i = 1\}, U^-(a^k) = \{i : a_i^k = C, y_i = -1\}$$

The kind of optimization problem we are solving here is exactly the same as in *Question 2*, with the exception of being of fixed size 2 since **everything is restricted to the components** (i, j), which is the most violating pair of the current iteration. We provide the pseudo-code of the algorithm:

Algorithm 1 SMO-MVP Algorithm

$a^0 \leftarrow 0$
$\nabla f(a^0) = -e$
$k \leftarrow 0$
while stopping criterion is not met **do**
  find most violating pair $W = \{i, j\}$
  $\overline{W} = \{1, ..., l\}/W$ fixed variables
  compute $a^* = \arg\min \left((1/2)a_W^T Q_{WW} a_W + p_W^T a_W : y_W^T a_W = -y_{\overline{W}} a_{\overline{W}}, 0 \leq a_W \leq C\right)$
  $\alpha_i^{k+1} = \begin{cases} \alpha_i^* & \text{if } i \in W \\ \alpha_i^k & \text{if } i \in \overline{W} \end{cases}$
  $\nabla f(a^{k+1}) = \nabla f(a^k) + Q_W(a^{k+1} - a^k)$
  $k \leftarrow k + 1$
end while

While in *Question 2* the problem was solved numerically due to its size ***Question 3* require us to solve the sub-problems analytically** since they are small. We decided to implement into code the algorithm that is described mathematically in paragraph 2.1 of the 1998 paper by *John C. Platt* "**Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines**", the paper that firstly proposed the SMO algorithm. The paper is available here: https://www.microsoft.com/en-us/research/uploads/prod/1998/04/sequential-minimal-optimization.pdf

Since the cross-validation was not required for this question (and it would have given the same results) we decided to use the same hyperparameters found in *Question 2*. Performances are reported on Table 5 at the end of this section. **We decided to report multiple instances with a variation of the tolerance parameter**. In the provided execution the final difference between $m(\lambda)$ and $M(\lambda)$ is -0.0096, confirming that the algorithm fulfilled the optimality condition.

The computation time significantly depends on the tolerance parameter for the optimality condition. We notice that **even for high tolerances the accuracy is identical** and the objective function extremely similar to those we get from *Question 2*, with a **reduction of the computational time of almost two magnitude orders**. Reducing the tolerance computation times increase and objective gets closer and closer to the one of *Question 2*.

We believe the time improvement would be even better if the solver were implemented in a low-level programming language instead of Python, as it would be a fairer comparison with performances in *Question 2* since the solver used by the *cvxopt.solvers.qp()* method is written in C.

2.5 Question 4: SVM for multi-class classification

The bonus *Question 4* requires to implement a strategy for **multi-class classification** using Support Vector Machines. While a single SVM supports only binary classification we can use multiple SVMs to make multi-class classification possible. This can be done by following one of two approaches:

- **One-vs-All** (OvA): each SVM implements the classification of one class against all the others together. It requires K different SVMs where K is the number of classes.
- **One-vs-One** (OvO): we create a different SVM for each possible combination of two classes. It requires $\binom{K}{2}$ different SVMs where K is the number of classes. It usually performs better but the number of required SVMs scales badly (quadratically) with the amount of classes.

Since the task asks us to consider only three classes the amount of required SVMs is the same for the two approaches ($K = 3 \Rightarrow \binom{K}{2} = 3$) and we therefore opt to use the **OvO approach**. The selected classes are those with classification numbers $\{0, 1, 2\}$ inside the original .csv file.

To train the SVMs we use the **SMO-MVP algorithm** implemented in *Question 3* as it runs much quicker and as hyperparameters for all the SVMs we use those we found in *Question 2*. All the requested informations are reported in Table 5.

	HYPERPARAMETERS			ML PERFORMANCE			OPTIMIZATION PERFORMANCE		
QUESTION	KERNEL	C	p (or γ)	TRAIN ACC.	VAL. ACC.	TEST ACC.	ITERATIONS	OBJECTIVE	CPU TIME
Q2	Gaussian	0.1	1.0	91.75%	91.625%	92%	13	Initial: 0 — Final: -141.575	1.109 s
Q3 ($\epsilon = 1e-1$)	Gaussian	0.1	1.0	91.75%	-	92%	160	Initial: 0 — Final: -140.966	0.020 s
Q3 ($\epsilon = 1e-2$)	Gaussian	0.1	1.0	91.75%	-	92%	1319	Initial: 0 — Final: -141.550	0.135 s
Q3 ($\epsilon = 1e-3$)	Gaussian	0.1	1.0	91.75%	-	92%	3188	Initial: 0 — Final: -141.575	0.340 s
Q4	Gaussian	0.1	1.0	92.9%	-	87.7%	584	-	0.070 s

Table 5: Results table SVM

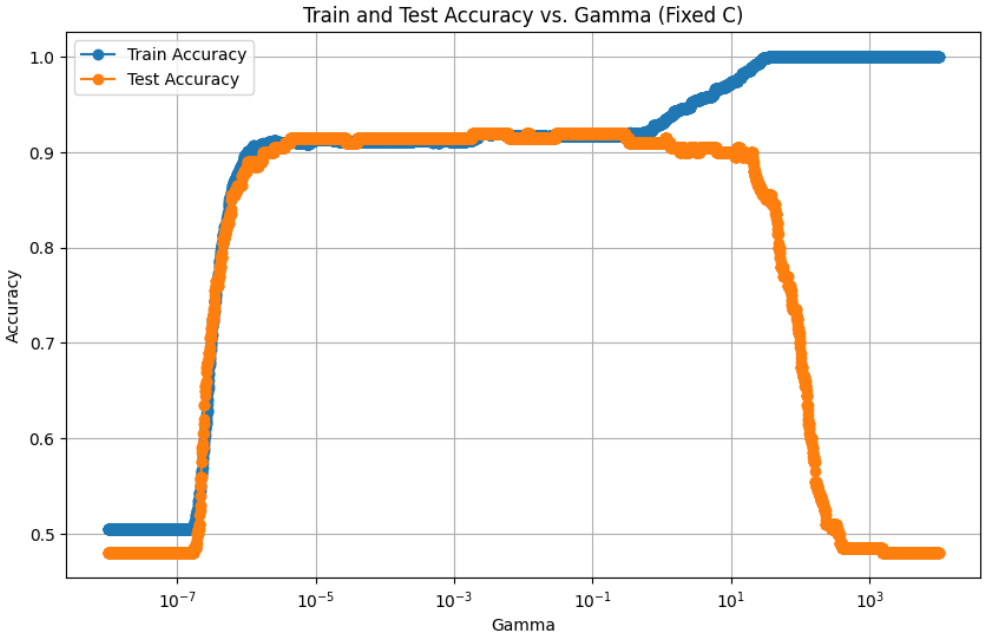

Figure 3: Plot that shows underfitting and overfitting of SVMs varying the hyperparameter γ. On the left we have underfitting and on the right overfitting.

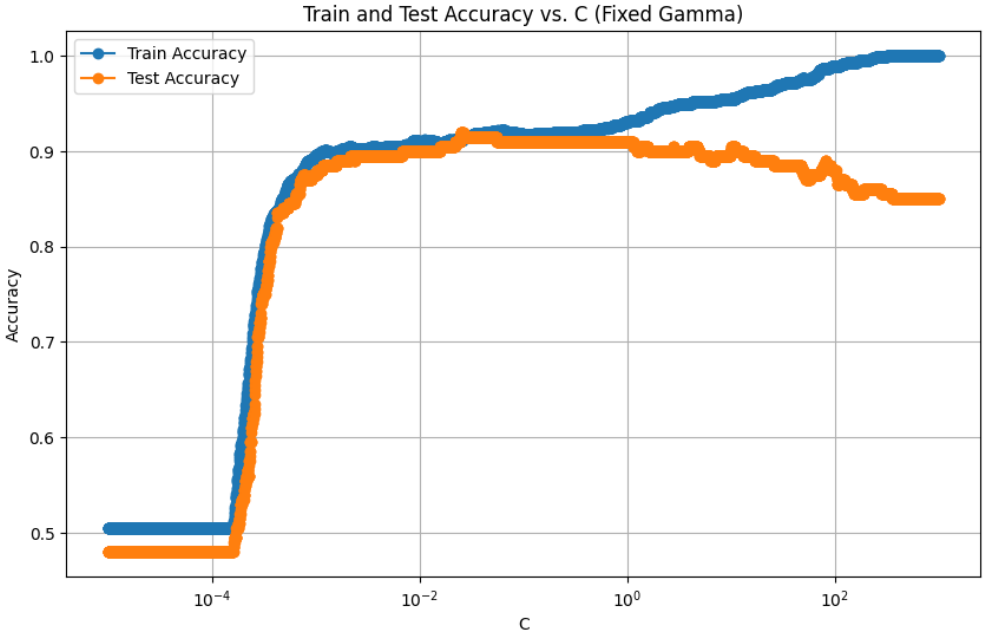

Figure 4: Plot that shows underfitting and overfitting of SVMs varying the hyperparameter C. On the left we have underfitting and on the right overfitting. The decay of test accuracy for overfitting is less severe than for Figure 3.