CS771 : Introduction to Machine Learning

Mini Project

Group Name: Passive Regression

Manya Gupta
220632
IIT Kanpur
manyag22@iitk.ac.in

Akshita Agarwal
220108
IIT Kanpur
akshita22@iitk.ac.in

Harshita
220445
IIT Kanpur
harshi22@iitk.ac.in

Vaneesha S Kumar
221166
IIT Kanpur
vaneesha22@iitk.ac.in

Nischay Patel
220721
IIT Kanpur
nischayp22@iitk.ac.in

Sikha Vamsi
221058
IIT Kanpur
svamsi22@iitk.ac.in

Part 1

Solution :

ML-PUF Linear Model Derivation

t_i^u is the (unknown) time at which the upper signal leaves the i-th PUF. t_i^l is the time at which the lower signal leaves the i-th PUF.
The answer is 0 if $t_{31}^l < t_{31}^u$ and 1 otherwise.

For stage $k = 0, 1, \ldots, n$, let:

- P_k: deterministic processing delay,
- S_k: upper-branch service delay,
- R_k: lower-branch service delay,
- $c_k \in \{0, 1\}$: the challenge bits of k^{th} multiplexer.

Define cumulative delays

$$t_k^u = (1 - c_k)(t_{k-1}^u + P_k) + c_k(t_{k-1}^l + S_k), \tag{1}$$

$$t_k^l = (1 - c_k)(t_{k-1}^l + R_k) + c_k(t_{k-1}^u + R_k), \tag{2}$$

with initial conditions $t_{-1}^u = t_{-1}^l = 0$, absorbing any base delays into P_0, S_0, R_0.

Base case ($n = 0$)

Directly,

$$t_0^u = (1 - c_0)P_0 + c_0 S_0 = P_0 + c_0(S_0 - P_0), \tag{3}$$

$$t_0^l = (1 - c_0)R_0 + c_0 R_0 = R_0. \tag{4}$$

First iteration ($n = 1$)

Using (3) and (4):

$$t_1^u = (1 - c_1)(t_0^u + P_1) + c_1(t_0^l + S_1) = P_0 + P_1 + (S_0 - P_0)c_0 + (R_0 + S_1 - P_0 - P_1)c_1, \tag{5}$$

$$t_1^l = (1 - c_1)(t_0^l + R_1) + c_1(t_0^u + R_1) = R_0 + R_1 + (P_0 + c_0(S_0 - P_0) - R_0)c_1. \tag{6}$$

Second iteration ($n = 2$)

Plugging t_1^u and t_1^l into the recursion and collecting like terms yields:

$$\begin{aligned} t_2^u = (P_0 + P_1 + P_2) + (-P_0 + S_0)\, c_0 + (-P_0 - P_1 + Q_0 + S_1)\, c_1 \\ + (-P_0 - P_1 - P_2 + Q_0 + Q_1 + S_2)\, c_2 + (P_0 - Q_0 + R_0 - S_0)\, c_0 c_1 \\ + (P_0 - Q_0 + R_0 - S_0)\, c_0 c_2 + (2P_0 + P_1 - 2Q_0 - Q_1 + R_1 - S_1)\, c_1 c_2 \\ + (-2P_0 + 2Q_0 - 2R_0 + 2S_0)\, c_0 c_1 c_2. \end{aligned}$$

Similarly,

$$\begin{aligned} t_2^l = (R_0 + R_1 + R_2) + (P_0 - R_0)\, c_1 + (P_0 + P_1 - R_0 - R_1)\, c_2 \\ + (-P_0 + S_0 - R_0 + Q_0)\, c_0 c_1 + (-P_0 + S_0 - R_0 + Q_0)\, c_0 c_2 \\ + (-2P_0 + 2S_0 - 2R_0 + 2Q_0)\, c_0 c_1 c_2. \end{aligned}$$

Each of t_n^{1u}, t_n^{1l}, t_n^{0u}, t_n^{1l} can be represented by a single vectorized expression:

$$t_n = W_n^T X_n + b_n, \tag{7}$$

where:

- X_n is the column vector of all Boolean monomials in $(1, c_0, c_1, \ldots, c_n)$, namely
$$X_n = [1, c_0, c_1, \ldots, c_n, c_0c_1, \ldots, c_0c_1 \cdots c_n]^T,$$
of dimension $2^{n+1} \times 1$.
- W_n is the matching column vector of coefficients (linear combinations of P_k, S_k, R_k), also of dimension $2^{n+1} \times 1$. (W_n is distinct for each of the four signals depending on the delay terms)
- b_n is the scalar bias term.

Base case in vector form

For $n = 0$, let

$$X_0 = [1, c_0]^T, \quad W_0^u = [P_0, S_0 - P_0]^T, \quad W_0^l = [R_0, 0]^T, \quad b_0 = 0.$$

Inductive step

Assume (7) holds at stage n. Then

$$t_{n+1} = (1 - c_{n+1})(W_n^T X_n + b_n + P_{n+1}) + c_{n+1}(W_n'^T X_n + b_n' + S_{n+1}),$$

where W_n', b_n' denote the "other" branch. Expanding:

$$t_{n+1} = W_n^T X_n + b_n + P_{n+1} + c_{n+1}\big((W_n' - W_n)^T X_n + (b_n' - b_n) + (S_{n+1} - P_{n+1})\big).$$

Hence the updates:

$$W_{n+1} = \begin{bmatrix} W_n \\ W_n' - W_n \end{bmatrix}, \quad b_{n+1} = b_n + P_{n+1}, \quad X_{n+1} = \begin{bmatrix} X_n \\ c_{n+1} X_n \end{bmatrix}.$$

For final signal output: $n = 7$

After seven stages, we obtain

$$t_7 = W_7^T X_7 + b_7,$$

where

$$X_7 = [c_0, c_1, \ldots, c_7,\ c_0c_1, \ldots, c_0c_1 \cdots c_7]^T, \tag{8}$$

$$W_7 = [w_0, w_1, \ldots, w_{01}, w_{02}, \ldots, w_{012}, \ldots, w_{01234567}]^T, \tag{9}$$

$$b_7 = \sum_{i=0}^{7} K_i. \text{(where } K_i \text{ is } Q_i \text{ for lower signal and } P_i \text{ for upper signal)} \tag{10}$$

Here X_7 and W_7 each have dimension $2^8 - 1 = 255$. Each coefficient w_I (for multi-index I) is an explicit linear combination of the delays P_i, S_i, R_i (e.g. $w_0 = \sum_{i=0}^{7} P_i$, $w_k = (S_k - P_k) + \sum_{i<k}(R_i - S_i)$, etc.).

Now for notationl simplicity let us denote the following:

$$W_u^1 = \text{Weight Vector for Upper Signal of PUF1} \tag{11}$$

$$W_l^1 = \text{Weight Vector for Lower Signal of PUF1} \tag{12}$$

$$W_u^0 = \text{Weight Vector for Upper Signal of PUF0} \tag{13}$$

$$W_l^0 = \text{Weight Vector for Lower Signal of PUF0} \tag{14}$$

$$\tag{15}$$

Now we take the signum of the difference of $W_u^0 X_7 + b_7^{0u}$ and $W_u^1 X_7 + b_7^{1u}$ and the signum of the difference of $W_l^0 X_7 + b_7^{0l}$ and $W_l^1 X_7 + b_7^{1l}$ and denote it as the following:

$$Response1 = \text{sgn}(W_u^0 X_7 + b_7^{0u} - W_u^1 X_7 - b_7^{1u}) \tag{16}$$

$$Response0 = \text{sgn}(W_l^0 X_7 + b_7^{0l} - W_l^1 X_7 - b_7^{1l}) \tag{17}$$

Let $w, v \in \mathbb{R}^n$ be fixed weight vectors, and let $x_i \in \mathbb{R}^n$ for $i = 1, \ldots, 255$ be input vectors. Define:

$$\begin{aligned} y_i &= \text{sgn}(w^\top x_i), \\ z_i &= \text{sgn}(v^\top x_i), \\ s_i &= \text{sgn}((w^\top x_i)(v^\top x_i)) = y_i \cdot z_i. \end{aligned}$$

Each of $y = [y_1, \ldots, y_{255}]$, $z = [z_1, \ldots, z_{255}]$, and $s = [s_1, \ldots, s_{255}]$ lies in $\mathbb{R}^{255}$. Again for notational convenience:

$$\begin{aligned} W_{R1} &= W_u^0 - W_u^1, \\ W_{R2} &= W_l^0 - W_l^1 \\ b_{R1} &= b_7^{0u} - b_7^{1u} \\ b_{R2} &= b_7^{0l} - b_7^{1l} \end{aligned}$$

Now applying XOR for both the PUFs:

$$r(c) = \frac{1 + (-1)^{2+1} \cdot \text{sgn}(W_{R1}^\top X_7 + b_{R1}) \cdot (W_{R2}^\top X_7 + b_{R2})}{2} \tag{18}$$

$$r(c) = \frac{1 + (-1)^{2+1} \cdot \text{sgn}((w_{R1}^1 x^1 + \cdots + w_{R1}^{255} x^{255} + b_{R1}) \cdot (w_{R2}^1 x^1 + \cdots + w_{R2}^{255} x^{255} + b_{R2}))}{2} \tag{19}$$

$$\tag{20}$$

Since $x_i^2 = 1$ for all i($x_i \in \{\pm 1\}$), then

$$(w^{R1T} x)(w^{R2T} x) = \sum_i w_i^{R1} w_i^{R2} x_i + \sum_{i \neq j} w_i^{R1} w_j^{R2} x_i x_j$$

Let the challenge vector be $\mathbf{x} \in \{-1, +1\}^{255}$, and let the two PUFs have weight vectors $\mathbf{w}_{R1}, \mathbf{w}_{R2} \in \mathbb{R}^n$ and biases $b_{R1}, b_{R2} \in \mathbb{R}$.

The response of the XOR PUF is defined as:

$$r(c) = \frac{1 - \text{sgn}(f(c))}{2}$$

where

$$f(c) = (\mathbf{w}_{R1}^\top \mathbf{x} + b_{R1}) \cdot (\mathbf{w}_{R2}^\top \mathbf{x} + b_{R2})$$

Expanding this expression:

$$\begin{aligned} f(c) &= \left(\sum_{i=1}^{n} w_{R1,i} x_i + b_{R1} \right) \cdot \left(\sum_{j=1}^{n} w_{R2,j} x_j + b_{R2} \right) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} w_{R1,i} w_{R2,j} x_i x_j + b_{R2} \sum_{i=1}^{n} w_{R1,i} x_i + b_{R1} \sum_{j=1}^{n} w_{R2,j} x_j + b_{R1} b_{R2} \end{aligned}$$

This is a quadratic function in $\mathbf{x}$, plus linear and constant terms.

To linearize this expression, define the feature map:

$$\phi(c) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ \vdots \\ x_n x_n \\ x_1 \\ \vdots \\ x_n \\ 1 \end{bmatrix} \in \mathbb{R}^{255^2 + 255 + 1}$$

Ignore diagonal terms as they are 1,

$$\phi(c) = \begin{bmatrix} x_1 x_2 \\ x_1 x_3 \\ \vdots \\ x_{n-1} x_n \\ x_1 \\ \vdots \\ x_n \\ 1 \end{bmatrix} \in \mathbb{R}^{\frac{255(254)}{2} + 255 + 1}$$

Define the weight vector $\mathbf{W} \in \mathbb{R}^{\frac{n(n-1)}{2} + n + 1}$ such that:

- The coefficient for $x_i x_j$ is $w_{R1,i} w_{R2,j}$
- The coefficient for x_i is $b_{R2} w_{R1,i} + b_{R1} w_{R2,i}$
- The constant term is $b_{R1} b_{R2} = b$

Then the function becomes:

$$f(c) = \mathbf{W}^\top \phi(c) + b$$

$$\boxed{r(c) = \frac{1 + \text{sgn}(\mathbf{W}^\top (-\phi(c)) + b)}{2}}$$

This expresses the XOR PUF as a linear function in the transformed feature space.

Part 2

Solution :

Dimensionality of the Feature Space

From the previous part,

$$\phi(c) = \begin{bmatrix} x_1x_2 \\ x_1x_3 \\ \vdots \\ x_{n-1}x_n \\ x_1 \\ \vdots \\ x_n \\ 1 \end{bmatrix} \in \mathbb{R}^{\frac{255(254)}{2}+255+1}$$

Therefore, dimensionality of feature space is **32,641**

Part 3

Solution:

Kernel SVM for Perfect Classification

We wish to use a kernel Support Vector Machine (SVM) to achieve perfect classification on the original 8-bit challenges $c \in \{0,1\}^8$ without applying any explicit feature transformations (such as converting bits to ± 1 or calculating cumulative products). The goal is to identify a suitable kernel function $K(c, c')$ and its parameters such that the SVM can perfectly separate the classes for any possible labeling of the challenge-response pairs.

Justification and Kernel Selection

The input space of challenges is finite, containing $2^8 = 256$ unique points. To guarantee perfect classification for any potential labeling of these points, the kernel must implicitly map the data into a feature space where the classes are linearly separable. The most straightforward way to ensure this for a finite dataset is to use a kernel that maps each distinct input point to its own unique, ideally orthogonal, dimension. This behavior is characteristic of the Dirac delta kernel, defined as:

$$K(c, c') = \delta_{c,c'} = \begin{cases} 1 & \text{if } c = c' \\ 0 & \text{if } c \neq c' \end{cases}$$

While the Dirac delta kernel itself isn't a standard option in most SVM libraries, we can approximate its behavior using standard kernels.

Let's consider the Radial Basis Function (RBF) kernel:

$$K(c, c') = \exp(-\gamma \|c - c'\|^2)$$

For binary input vectors $c, c' \in \{0,1\}^8$, the squared Euclidean distance $\|c - c'\|^2$ is equivalent to the Hamming distance, $H(c, c')$, which is the count of differing bits between c and c'.

- If $c = c'$, then $H(c, c') = 0$, and $K(c, c') = \exp(0) = 1$.
- If $c \neq c'$, then $H(c, c') \geq 1$. The kernel value becomes $K(c, c') = \exp(-\gamma H(c, c'))$.

By choosing a sufficiently large value for the hyperparameter γ, we can make the kernel value extremely close to zero whenever $c \neq c'$. For instance, even for the smallest possible Hamming distance of 1, the kernel value is $e^{-\gamma}$. If γ is large (e.g., $\gamma > 15$), then $e^{-\gamma}$ becomes negligible ($< 10^{-6}$), effectively approximating the $K(c, c') = 0$ condition for $c \neq c'$.

Therefore, the RBF kernel with a large γ closely mimics the Dirac delta kernel. In the feature space induced by such an RBF kernel, each unique 8-bit challenge c is mapped to a vector that is nearly orthogonal to the vectors corresponding to all other challenges $c' \neq c$. This ensures that any

arbitrary labeling of the 256 points becomes linearly separable, allowing the SVM to achieve perfect classification on the training data.

Suggested Kernel and Parameters

- **Kernel Type:** Radial Basis Function (RBF).
- **Kernel Parameter (γ - gamma):** A very large positive value. The exact value is not critical as long as it is large enough to ensure $\exp(-\gamma) \approx 0$. A value such as $\gamma = 20$ or $\gamma = 50$ would likely suffice, effectively isolating each data point in the feature space. Other parameters like degree or coef0 are not applicable to the RBF kernel.

This theoretical choice guarantees the possibility of perfect classification using a kernel SVM directly on the original 8-bit challenges, as required by the question which asks only for theoretical calculations.

Part 4

Solution:

Delay Recovery by Inverting a Simple Arbiter PUF Model

This section outlines the method to recover 256 non-negative delay values from a given 65-dimensional linear model $\mathbf{w} = (w\_0, ..., w\_63, b)$, corresponding to a simple arbiter PUF. The recovery process mirrors the model-generation mechanism and effectively inverts it by solving a constrained linear system.

Step 1: Forward Model - From Delays to Linear Model Parameters

Let $N = 64$ be the number of stages in the arbiter PUF.

1. Delay Vectors

Each stage i contributes four physical delays:

$$(p_i, q_i, r_i, s_i) \quad \text{for } i = 0, 1, ..., N-1$$

These values represent delay components along different paths in the signal race.

2. Intermediate Parameters

From these delays, we define:

$$D_i = p_i - q_i$$
$$E_i = r_i - s_i$$

We then define:

$$\alpha_i = \frac{1}{2}(D_i + E_i) = \frac{1}{2}(p_i - q_i + r_i - s_i)$$
$$\beta_i = \frac{1}{2}(D_i - E_i) = \frac{1}{2}(p_i - q_i - r_i + s_i)$$

3. Mapping to Model Parameters (w, b)

Using the definitions of α_i and β_i, we obtain the model vector:

$$w_0 = \alpha_0$$
$$w_i = \alpha_i + \beta_{i-1} \quad \text{for } 1 \leq i \leq N-1$$
$$b = \beta_{N-1}$$

This chain shows how the 256 delays influence the 65 linear model parameters.

Step 2: Inversion Method – Recovering Delays from the Model

Given $(w_0, ..., w_{63}, b)$, we aim to recover non-negative delays p_i, q_i, r_i, s_i such that they regenerate the same linear model.

1. Recovering α_i and $beta_i$

We reverse the equations:

$$\begin{aligned} \alpha_0 &= w_0 \\ \beta_{i-1} &= w_i - \alpha_i \quad \text{for } 1 \leq i \leq N-1 \\ \beta_{N-1} &= b \end{aligned}$$

From these, all α_i and β_i can be computed iteratively.

2. Recovering D_i and E_i

Once α_i and β_i are known, we compute:

$$\begin{aligned} D_i &= \alpha_i + \beta_i = p_i - q_i \\ E_i &= \alpha_i - \beta_i = r_i - s_i \end{aligned}$$

3. Recovering Non-Negative Delays

We now determine p_i, q_i, r_i, s_i such that all are non-negative and satisfy:

$$\begin{aligned} p_i - q_i &= D_i \\ r_i - s_i &= E_i \end{aligned}$$

We enforce non-negativity by selecting:

$$\begin{aligned} p_i &= \max(D_i, 0), \quad q_i = \max(-D_i, 0) \\ r_i &= \max(E_i, 0), \quad s_i = \max(-E_i, 0) \end{aligned}$$

This guarantees:

- $p_i, q_i, r_i, s_i \geq 0$
- $p_i - q_i = D_i$ and $r_i - s_i = E_i$
- The original linear model (w, b) is preserved

Step 3: Summary of System Interpretation

This process provides an explicit inverse mapping from the 65-dimensional model back to a valid 256-dimensional set of non-negative delays:

- The model is implicitly a linear system, but not expressed in matrix form.
- The inversion exploits the structure of the model, bypassing direct matrix inversion.
- The solution is not necessarily unique, but the chosen one guarantees physical plausibility (non-negative delays).

Step 4: Advantages of This Method

- **Deterministic**: Always gives a valid solution.
- **Efficient**: No need for solving a large linear system or running optimization solvers.
- **Non-Negativity Guarantee**: All output delays are non-negative by construction.

Part 7

Solution:

Hyperparameter Experiments for ML-PUF Model

In this section, we report the outcomes of experiments comparing `sklearn.linear_model.LogisticRegression` (LGR) and `sklearn.svm.LinearSVC` (SVC) when used with the 66-dimensional feature map (`my_map`) derived from 11 base features, which achieved 100% test accuracy. We investigate the effect of various hyperparameters on test accuracy and training time, as required. All experiments were performed using the provided training (6400 samples) and testing (1600 samples) datasets after applying `StandardScaler` to the features.

Experiment 1: Effect of Regularization Strength (C)

We varied the regularization parameter C over several orders of magnitude for both classifiers, keeping other parameters at reasonable defaults (LGR: solver='liblinear', penalty='l2'; SVC: loss='squared_hinge', penalty='l2', dual='auto').

Table 1: Effect of Regularization Strength (C) on Test Accuracy and Training Time (66D Features)

Classifier	C	Test Accuracy (%)	Training Time (s)
LogisticRegression	0.01	89.38	0.0897
LogisticRegression	0.1	96.62	0.0758
LogisticRegression	1.0	97.62	0.4425
LogisticRegression	10.0	100.00	0.5789
LogisticRegression	100.0	100.00	0.9550
LogisticRegression	1000.0	100.00	0.6986
LinearSVC	0.01	94.75	0.0732
LinearSVC	0.1	97.62	0.0934
LinearSVC	1.0	100.00	1.3087
LinearSVC	10.0	100.00	8.0779
LinearSVC	100.0	100.00	4.7969
LinearSVC	1000.0	100.00	4.0336

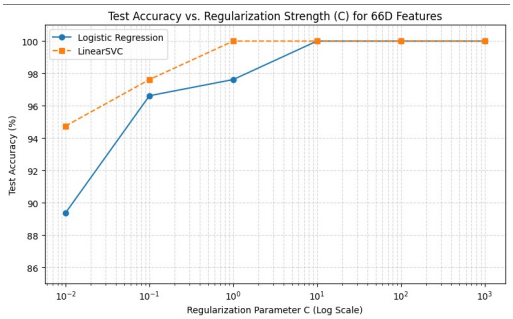

Figure 1: Test Accuracy vs. C

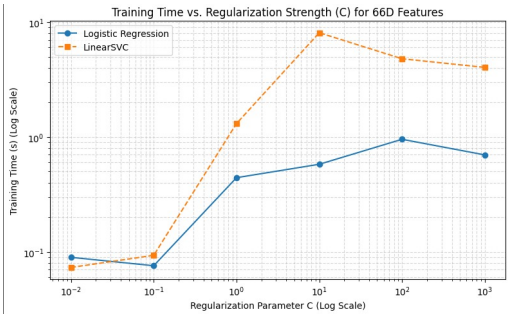

Figure 2: Training Time vs. C (Log-Log Scale)

Analysis: As shown in Table 1 and Figure 1, both classifiers achieve 100% test accuracy, but at different minimum C values. Logistic Regression requires $C \geq 10.0$, while LinearSVC achieves perfection already at $C = 1.0$. For very low C (strong regularization), accuracy drops significantly for both. Figure 2 shows that training time generally increases with C. LinearSVC shows a notable spike in training time at $C = 10.0$, whereas Logistic Regression's time increases more moderately. At the minimum C required for 100% accuracy, LinearSVC (at C=1.0, 1.3s) is slower than Logistic Regression (at C=10.0, 0.6s).

Experiment 2: Effect of Regularization Penalty (L1 vs L2)

We compared L1 and L2 penalties for both classifiers, using a fixed C value known to achieve 100% accuracy with L2 (C=10.0 for LGR, C=1.0 for SVC). For L1, compatible solvers/settings were used ('liblinear' for LGR; 'squared_hinge' loss, dual=False for SVC).

Table 2: Effect of Penalty Type on Test Accuracy and Training Time (66D Features)

Classifier	C	Penalty	Solver/Loss	Test Accuracy (%)	Training Time (s)
LogisticRegression	10.0	L2	liblinear	100.00	0.1998
LogisticRegression	10.0	L1	liblinear	100.00	6.2475
LinearSVC	1.0	L2	squared_hinge	100.00	0.8918
LinearSVC	1.0	L1	squared_hinge	100.00	6.4413

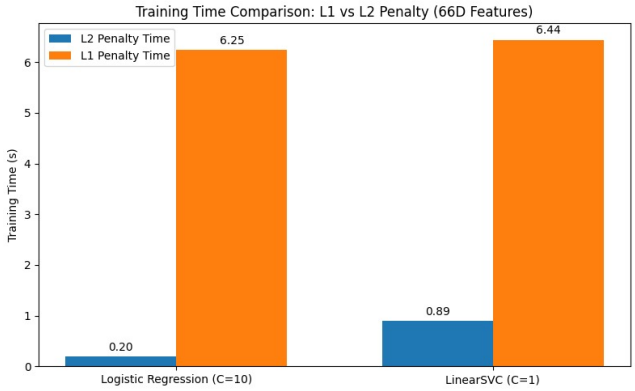

Figure 3: Training Time Comparison for L1 vs L2 Penalty

Analysis: Table 2 shows that both L1 and L2 penalties achieve 100% test accuracy for the selected C values. However, Figure 3 clearly illustrates that the L1 penalty results in substantially longer training times (over 6 seconds) compared to the L2 penalty (under 1 second) for both Logistic Regression and LinearSVC on this dataset and feature set. L1 regularization promotes sparsity, but this came at a significant computational cost here without improving accuracy.

Experiment 3: Effect of Loss Function (LinearSVC only)

We compared the 'hinge' and 'squared_hinge' loss functions for LinearSVC, using $C = 1.0$ and the L2 penalty.

Table 3: Effect of Loss Function (LinearSVC, C=1.0, L2 Penalty)

Loss Function	Test Accuracy (%)	Training Time (s)
hinge	98.25	0.9097
squared_hinge	100.00	0.9642

Analysis: As seen in Table 3, the 'squared_hinge' loss was necessary to achieve 100% accuracy with LinearSVC at $C = 1.0$. The standard 'hinge' loss resulted in slightly lower accuracy (98.25%). Training times were comparable.

Experiment 4: Effect of Tolerance (tol)

We varied the convergence tolerance (`tol`) for both models, keeping other parameters at values known to yield 100% accuracy (C=10 for LGR, C=1 for SVC, L2 penalty).

Table 4: Effect of Tolerance (tol) on Test Accuracy and Training Time

Classifier	C	Tolerance	Test Accuracy (%)	Training Time (s)
LogisticRegression	10.0	1e-3	100.00	0.2341
LogisticRegression	10.0	1e-4	100.00	0.1991
LogisticRegression	10.0	1e-5	100.00	0.2491
LogisticRegression	10.0	1e-6	100.00	0.3099
LinearSVC	1.0	1e-3	100.00	0.6401
LinearSVC	1.0	1e-4	100.00	0.7061
LinearSVC	1.0	1e-5	100.00	0.7118
LinearSVC	1.0	1e-6	100.00	0.7213

Analysis: Table 4 shows that within the tested range, the tolerance value had no impact on the final test accuracy, which remained at 100%. There was a slight tendency for training time to increase as the tolerance became stricter (smaller values), as expected, but the effect was minor.

Summary

Both Logistic Regression and LinearSVC were able to achieve 100% test accuracy on this ML-PUF dataset using the 66-dimensional feature map. LinearSVC reached perfect accuracy at a lower regularization parameter ($C = 1.0$) than Logistic Regression ($C = 10.0$), although its training time increased more sharply with C. The L1 penalty, while achieving 100% accuracy, was significantly slower than L2. For LinearSVC, the 'squared_hinge' loss was necessary for optimal accuracy. Convergence tolerance had minimal impact in the tested range. Based on these results, either classifier with appropriate hyperparameters (e.g., LinearSVC with C=1.0, squared_hinge loss, L2 penalty) provides an effective linear model for this problem.