CS771 Mini-Project Report
ML-PUF Modeling and Arbiter PUF Inversion

Ekansh Bajpai
Department of Biological Sciences and Bioengineering
Indian Institute of Technology Kanpur
ekanshb22@iitk.ac.in

Hritvija Singh
Department of Material Science and Engineering
Indian Institute of Technology Kanpur
hritvijasi22@iitk.ac.in

Mohd Adil
Department of Civil Engineering
Indian Institute of Technology Kanpur
madil22@iitk.ac.in

Pulkit Dhayal
Department of Civil Engineering
Indian Institute of Technology Kanpu
pulkitd22@iitk.ac.in

V Nikhil
Department of Electrical Engineering
Indian Institute of Technology Kanpur
vnikhil22@iitk.ac.in

Abhishek Kumar
Department of Chemical Engineering
Indian Institute of Technology Kanpur
abhishkum22@iitk.ac.in

Abstract

This report explores the modeling and inversion of Physically Unclonable Functions (PUFs) using linear machine learning techniques. We begin by analyzing the structure of Multi-Level PUFs (ML-PUFs), which combine two arbiter PUFs and utilize XOR-based logic to generate more robust responses. Through careful feature engineering, we demonstrate how a linear model can be constructed to accurately predict ML-PUF responses by transforming input challenges into a higher-dimensional feature space. Additionally, we address the inverse problem of reconstructing delay parameters from a learned arbiter PUF model. We formulate this as a constrained optimization problem, recovering a valid set of non-negative delays consistent with the given linear weights. Experimental results validate the efficacy of our methods in both tasks, highlighting the surprising predictability of ML-PUFs and feasibility of reverse-engineering PUF characteristics from model parameters.

1 Mathematical Derivation of Linear Model for Predicting ML-PUF Responses

We will use a straightforward approach to solve this problem. The strategy involves leveraging the known differences in time delays for the upper and lower signals, as well as the sum of their delays. First, we will derive the difference in time delays for 8-bit input in a way similar to as outlined in the lectures, and then proceed to derive the sum of these delays. Also the notations are same as described in lectures.

1.1 Deriving the difference of Delays

Here, t_i^u and t_i^l represent the times at which the signal departs from the i^{th} MUX pair. Expressing t_i^u in terms of t_{i-1}^u, t_{i-1}^l and c_i, we obtain:

$$t_i^u = (1 - c_i) \cdot (t_{i-1}^u + p_i) + c_i \cdot (t_{i-1}^l + s_i) \tag{1}$$
$$t_i^l = (1 - c_i) \cdot (t_{i-1}^l + q_i) + c_i \cdot (t_{i-1}^u + r_i) \tag{2}$$

Thus, following the lecture slides we have, $\Delta_i = t_i^u - t_i^l$. Subtracting equations (1), (2) we get, $\Delta_i = \Delta_{i-1} \cdot d_i + \alpha_i \cdot d_i + \beta_i$, where α_i, β_i depend on constants that are indeterminable from the physical/measurable perspective and thus we call them system constants and are given by:

$$\alpha_i = \frac{(p_i - q_i + r_i - s_i)}{2}, \quad \beta_i = \frac{(p_i - q_i - r_i + s_i)}{2} + a$$

where p_i, q_i, r_i, s_i are system parameters. Also d_i is governed by the challenge bits/input (c_i's):

$$d_i = (1 - 2 \cdot c_i)$$

$$\Delta_{-1} = 0$$

Moreover, observing the recursion unfold carefully we can simplify this relation further:

$$\begin{aligned}
\Delta_0 &= \alpha_0 \cdot d_0 + \beta_0 \\
\Delta_1 &= \alpha_0 \cdot d_1 \cdot d_0 + (\alpha_1 + \beta_0) \cdot d_1 + \beta_1 \\
\Delta_2 &= \alpha_0 \cdot d_2 \cdot d_1 \cdot d_0 + (\alpha_1 + \beta_0) \cdot d_2 \cdot d_1 + (\alpha_2 + \beta_1) \cdot d_2 + \beta_2 \\
&\vdots
\end{aligned}$$

The only Δ we require is Δ_7, the last output.

$$\Delta_7 = w_0 \cdot x_0 + w_1 \cdot x_1 + \cdots + w_7 \cdot x_7 + \beta_7 = \mathbf{w}^T \cdot \mathbf{x} + b = \mathbf{W}^T \cdot \mathbf{X}$$

where $\mathbf{w}, \mathbf{x}$ are 8-dimensional vectors and $\mathbf{W}, \mathbf{X}$ are 9-dimensional, just including the bias term $W_8 = \beta_7$ and $X_8 = 1$. Each term of $\mathbf{w}, \mathbf{x}$ being:

$$b = \beta_7, \quad w_0 = \alpha_0, \quad w_i = \alpha_i + \beta_{i-1}, \quad x_i = \prod_{j=i}^{7} d_j$$

1.2 Deriving the Sum of Delays

We have,

$$t_i^u = (1 - c_i) \cdot (t_{i-1}^u + p_i) + c_i \cdot (t_{i-1}^l + s_i) \tag{1}$$
$$t_i^l = (1 - c_i) \cdot (t_{i-1}^l + q_i) + c_i \cdot (t_{i-1}^u + r_i) \tag{2}$$

Adding both the equations,

$$t_i^u + t_i^l = t_{i-1}^u + t_{i-1}^l + (p_i + q_i) + c_i \cdot (s_i + r_i - p_i - q_i)$$

Put

$$(s_i + r_i - p_i - q_i) = r_i'$$

to get,

$$t_i^u + t_i^l = t_{i-1}^u + t_{i-1}^l + (p_i + q_i) + c_i \cdot (r_i')$$

Substituting the value of $i = 7, 6, \ldots, 0$ and adding the equations:

$$\begin{aligned} t_7^u + t_7^l = (c_7 \cdot r_7') &+ (p_7 + q_7) + \\ (c_6 \cdot r_6') &+ (p_6 + q_6) + \\ \cdots &+ \\ (c_0 \cdot r_0') &+ (p_0 + q_0) \end{aligned}$$

We finally get,

$$t_7^u + t_7^l = \sum_{i=0}^{7} (c_i \cdot r'i) + \sum i = 0^7 (p_i + q_i) \tag{1}$$

where

$$r_i' = s_i + r_i - p_i - q_i$$

We also derived the difference in previous sections as:

$$t_7^u - t_7^l = w_0 \cdot x_0 + w_1 \cdot x_1 + \cdots + w_7 \cdot x_7 + \beta_7 \tag{2}$$

where

$$w_0 = \alpha_0, \quad w_i = \alpha_i + \beta_{i-1}, \quad x_i = \prod_{j=i}^{7} d_j, \quad d_i = (1 - 2 \cdot c_i)$$

and

$$\alpha_i = \frac{p_i - q_i + r_i - s_i}{2}, \quad \beta_i = \frac{p_i - q_i - r_i + s_i}{2}$$

Adding (1) and (2):

$$\begin{aligned} 2 \cdot t_7^u = (c_7 \cdot r_7') + (c_6 \cdot r_6') + \cdots + (c_0 \cdot r_0') \\ + w_0 \cdot x_0 + w_1 \cdot x_1 + \cdots + w_7 \cdot x_7 + \beta_7 \end{aligned}$$

Dividing both sides by 2:

$$t_7^u = \frac{1}{2} \left(\sum_{i=0}^{7} c_i \cdot r'i + \sum i = 0^7 w_i \cdot x_i + \beta_7 \right)$$

Rewriting $c_7 = 1 - 2c_7$ and taking c_7 common from $c_7 \cdot r_7'$ and $2 \cdot w_7 \cdot c_7$, we get:

$$t_7^u = \frac{1}{2} \left(c_7 \cdot (r'7 - 2w_7) + \sum i = 0^6 c_i \cdot r'i + \sum i = 0^7 w_i \cdot x_i + \beta_7 + w_7 \right)$$

In matrix form, this can be expressed as:

$$t_7^u = \frac{1}{2} \left(\mathbf{W}^T \cdot \phi(c) + b \right)$$

where:

$$\phi(c) = [c_7, c_6, \dots, c_0, x_0, x_1, \dots, x_6]^T$$

$$\mathbf{W} = [r'_7 - 2w_7, r'_6, \dots, r'_0, w_0, w_1, \dots, w_6]^T, \quad b = \beta_7 + w_7$$

In this solution we denote parameters for PUF0 as $A_{0,i}$, and parameters for PUF1 as $A_{1,i}$.

Linear Model for Response0:

$$t^u_{0,7} = \frac{1}{2}\left((c_7 \cdot (r_{0,7} - 2w_{0,7})) + (c_6 \cdot r_{0,6}) + \cdots + (c_0 \cdot r_{0,0}) + w_{0,0} \cdot x_0 + \cdots + w_{0,7} \cdot x_7 + \beta_{0,7}\right)$$

$$t^u_{1,7} = \frac{1}{2}\left((c_7 \cdot (r_{1,7} - 2w_{1,7})) + (c_6 \cdot r_{1,6}) + \cdots + (c_0 \cdot r_{1,0}) + w_{1,0} \cdot x_0 + \cdots + w_{1,7} \cdot x_7 + \beta_{1,7}\right)$$

Now

$$\begin{aligned} t^u_{0,7} - t^u_{1,7} = \frac{1}{2}\Big(&c_7 \cdot ((r_{0,7} - 2w_{0,7}) - (r_{1,7} - 2w_{1,7})) + \\ &c_6 \cdot (r_{0,6} - r_{1,6}) + \cdots + c_0 \cdot (r_{0,0} - r_{1,0}) + \\ &(w_{0,0} - w_{1,0}) \cdot x_0 + \cdots + (w_{0,6} - w_{1,6}) \cdot x_6 + \\ &(\beta_{0,7} + w_{0,7}) - (\beta_{1,7} + w_{1,7})\Big) \end{aligned}$$

In matrix form, this can be expressed as:

$$t^u_{0,7} - t^u_{1,7} = \frac{1}{2}\left(\tilde{W}^T \cdot \phi(c) + \tilde{b}\right)$$

Where:

$$\phi(c) = [c_7, c_6, \dots, c_0, x_0, x_1, \dots, x_6]^T$$

$$\tilde{W} = [(r_{0,7} - 2w_{0,7}) - (r_{1,7} - 2w_{1,7}),\ r_{0,6} - r_{1,6}, \dots, r_{0,0} - r_{1,0},\ w_{0,0} - w_{1,0}, \dots, w_{0,6} - w_{1,6}]^T$$

$$\tilde{b} = (\beta_{0,7} + w_{0,7}) - (\beta_{1,7} + w_{1,7})$$

Linear Model for Response1:

$$t^l_{0,7} = \frac{1}{2}\left((c_7 \cdot (r_{0,7} + 2w_{0,7})) + \cdots + (c_0 \cdot r_{0,0}) - w_{0,0} \cdot x_0 - \cdots - w_{0,7} \cdot x_7 - (\beta_{0,7} + w_{0,7})\right)$$

$$t^l_{1,7} = \frac{1}{2}\left((c_7 \cdot (r_{1,7} + 2w_{1,7})) + \cdots + (c_0 \cdot r_{1,0}) - w_{1,0} \cdot x_0 - \cdots - w_{1,7} \cdot x_7 - (\beta_{1,7} + w_{1,7})\right)$$

Now

$$\begin{aligned} t^l_{0,7} - t^l_{1,7} = \frac{1}{2}\Big(&c_7 \cdot ((r_{0,7} + 2w_{0,7}) - (r_{1,7} + 2w_{1,7})) + \\ &c_6 \cdot (r_{0,6} - r_{1,6}) + \cdots + c_0 \cdot (r_{0,0} - r_{1,0}) - \\ &(w_{0,0} - w_{1,0}) \cdot x_0 - \cdots - (w_{0,6} - w_{1,6}) \cdot x_6 - \\ &((\beta_{0,7} + w_{0,7}) - (\beta_{1,7} + w_{1,7}))\Big) \end{aligned}$$

In matrix form, this can be expressed as:

$$t^l_{0,7} - t^l_{1,7} = \frac{1}{2}\left(\tilde{W}^T \cdot \phi(c) + \tilde{b}\right)$$

Where:

$$\phi(c) = [c_7, c_6, \ldots, c_0, x_0, x_1, \ldots, x_6]^T$$

$$\tilde{W} = [(r_{0,7} + 2w_{0,7}) - (r_{1,7} + 2w_{1,7}),\ r_{0,6} - r_{1,6}, \ldots, r_{0,0} - r_{1,0},\ -(w_{0,0} - w_{1,0}), \ldots, -(w_{0,6} - w_{1,6})]^T$$

$$\tilde{b} = (\beta_{1,7} + w_{1,7}) - (\beta_{0,7} + w_{0,7})$$

We now compute the multiplication of the two expressions to model the XOR response of the ML-PUF, as per MP1.

Recall the expressions:

$$\begin{aligned} t^u_{0,7} - t^u_{1,7} = \frac{1}{2}&\left(c_7 \cdot ((r_{0,7} - 2w_{0,7}) - (r_{1,7} - 2w_{1,7})) + c_6 \cdot (r_{0,6} - r_{1,6}) + \cdots + c_0 \cdot (r_{0,0} - r_{1,0})\right. \\ &\left.+(w_{0,0} - w_{1,0}) \cdot x_0 + \cdots + (w_{0,6} - w_{1,6}) \cdot x_6 + (\beta_{0,7} + w_{0,7}) - (\beta_{1,7} + w_{1,7})\right) \end{aligned}$$

$$\begin{aligned} t^l_{0,7} - t^l_{1,7} = \frac{1}{2}&\left(c_7 \cdot ((r_{0,7} + 2w_{0,7}) - (r_{1,7} + 2w_{1,7})) + c_6 \cdot (r_{0,6} - r_{1,6}) + \cdots + c_0 \cdot (r_{0,0} - r_{1,0})\right. \\ &\left.-(w_{0,0} - w_{1,0}) \cdot x_0 - \cdots - (w_{0,6} - w_{1,6}) \cdot x_6 - ((\beta_{0,7} + w_{0,7}) - (\beta_{1,7} + w_{1,7}))\right) \end{aligned}$$

Now multiplying:

$$(t^u_{0,7} - t^u_{1,7}) \cdot (t^l_{0,7} - t^l_{1,7}) = \frac{1}{4} \cdot \left(\text{Expression}_u\right) \cdot \left(\text{Expression}_l\right)$$

Writing all terms symbolically:

$$\begin{aligned} (t^u_{0,7} - t^u_{1,7}) \cdot (t^l_{0,7} - t^l_{1,7}) = \frac{1}{4}\Bigg[&\left(\sum_{i=0}^{7} c_i \cdot \Delta r_i^{(u)} + \sum_{j=0}^{6} \Delta w_j^{(u)} \cdot x_j + b_u\right) \cdot \\ &\left(\sum_{i=0}^{7} c_i \cdot \Delta r_i^{(l)} - \sum_{j=0}^{6} \Delta w_j^{(u)} \cdot x_j - b_u\right)\Bigg] \end{aligned}$$

Where:
• $\Delta r_7^{(u)} = (r_{0,7} - 2w_{0,7}) - (r_{1,7} - 2w_{1,7})$ • $\Delta r_7^{(l)} = (r_{0,7} + 2w_{0,7}) - (r_{1,7} + 2w_{1,7})$ • $\Delta r_i^{(u)} = r_{0,i} - r_{1,i}, \quad \text{for } i = 0 \text{ to } 6$ • $\Delta r_i^{(l)} = r_{0,i} - r_{1,i}, \quad \text{for } i = 0 \text{ to } 6$ • $\Delta w_j^{(u)} = w_{0,j} - w_{1,j}$ • $b_u = (\beta_{0,7} + w_{0,7}) - (\beta_{1,7} + w_{1,7})$

This final expression models the XOR logic by taking the product of the two linear expressions.

To obtain the predicted response bit, we apply:

$$\boxed{\text{Response}(c) = \frac{1 - \text{sign}\left((t^u_{0,7} - t^u_{1,7}) \cdot (t^l_{0,7} - t^l_{1,7})\right)}{2}}$$

Therefore, the final feature map is:

$$\begin{aligned}
&(c_1c_1,\, c_1c_2, \ldots, c_7c_6,\, c_7c_7, \\
&\ x_1x_1,\, x_1x_2, \ldots, x_6x_5,\, x_6x_6, \\
&\ c_1x_1,\, c_1x_2, \ldots, c_7x_5,\, c_7x_6, \\
&\ x_1c_1,\, x_1c_2, \ldots, x_6c_6,\, x_6c_7, \\
&\ c_1.1,\, c_2.1, \ldots, c_7.1,\, x_1.1, \ldots, x_6.1, \\
&\ 1.c_1,\, 1.c_2, \ldots, 1.c_7,\, 1.x_1, \ldots, 1.x_6)
\end{aligned}$$

2 Feature Dimensionality

2.1 Challenge Encoding and Feature Map Construction

Let $c = (c_0, c_1, \ldots, c_7) \in \{0, 1\}^8$ denote the 8-bit input challenge. We first convert each bit to the domain $\{-1, +1\}$ via:

$$d_i = 1 - 2c_i, \quad \text{so that} \quad d_i \in \{-1, +1\}$$

Next, we define the *suffix product features* x_i as:

$$x_i = \prod_{j=i}^{7} d_j \qquad \text{for } i = 0, 1, \ldots, 6$$

These suffix features encode the multiplicative parity from position i to the end of the challenge vector.

We define the feature vector $\psi(c) \in \mathbb{R}^{15}$ as:

$$\psi(c) = [x_0, x_1, \ldots, x_6, d_0, d_1, \ldots, d_7]^\top$$

This forms a compact feature basis consisting of 7 suffix products and 8 individual bits, capturing both nonlinear and linear properties of the challenge.

2.2 Lifting to Higher Dimensional Space via Khatri-Rao Product

To model the XOR-ML-PUF, we lift $\psi(c)$ into a higher-dimensional space by computing the second-order products. Define the Khatri–Rao (column-wise Kronecker) product:

$$\Psi(c) = \psi(c) \otimes \psi(c)$$

This results in a vector in $\mathbb{R}^{225}$, containing all possible $\psi_i\psi_j$ terms for $0 \leq i, j < 15$.

However, because multiplication is commutative ($\psi_i\psi_j = \psi_j\psi_i$), we only retain the **upper-triangular** terms to eliminate redundancy. The upper-triangular part yields:

$$\text{dim} = \frac{15 \cdot (15 + 1)}{2} = 120$$

2.3 Reducing Redundant Terms

We prune redundant features using domain-specific knowledge:

- For all $d_i \in \{-1, +1\} \Rightarrow d_i^2 = 1$, so terms like d_id_i can be omitted. - Similarly, each $x_i \in \{-1, +1\} \Rightarrow x_i^2 = 1$, so x_ix_i terms add no new information. - Also, terms like x_ix_{i+1} and x_ix_{i+2} can be expressed linearly using existing d_i, x_i terms:

$$x_ix_{i+1} = d_i, \quad x_ix_{i+2} = d_id_{i+1}$$

- Other redundant cross-terms like c_ix_{i+1} or higher-degree terms like c_i^2 are removed similarly.

After eliminating these 15 linearly dependent terms, we obtain the final reduced feature vector:

$$\Phi(c) \in \mathbb{R}^{105}$$

Thus, we conclude:

$$\text{Dimensionality} = 105$$

3. Kernel SVM Approach

1.3 Kernel SVM for ML-PUF Classification

In the earlier sections of this project, we constructed a custom 105-dimensional feature map $\phi(c)$ using the original 8-bit challenge $c \in \{0,1\}^8$. This feature map included:

- Linear terms: $c_1, \ldots, c_8$
- Suffix products (cumulative parity): $x_1, \ldots, x_7$
- Pairwise interactions: $c_i c_j$, $x_i x_j$, $c_i x_j$, $x_i c_j$
- Constants: 1

This engineered feature map enabled perfect classification of the ML-PUF response using a linear model.

We wish to train a Support Vector Machine (SVM) directly on the original challenge vector $c \in \{0,1\}^8$, without applying any custom feature mapping (e.g., parity products or cumulative suffix products). The goal is to identify a kernel function that allows the SVM to perfectly classify responses from an Arbiter PUF or ML-PUF model.

Recommended Kernel: Polynomial Kernel

We recommend using the **polynomial kernel**, defined as:

$$K(\mathbf{c}, \mathbf{c}') = (\mathbf{c}^\top \mathbf{c}' + r)^d$$

where:

- d is the degree of the polynomial
- r is the bias term (typically $r = 1$)

Justification

The total number of possible 8-bit binary input vectors is:

$$|\{0,1\}^8| = 256$$

The total number of possible Boolean functions over these 256 inputs is:

$$2^{256}$$

However, it is a well-known fact from Boolean algebra that any Boolean function $f : \{0,1\}^8 \to \{0,1\}$ can be exactly represented as a multivariate polynomial of degree at most 8 over the binary field. Thus, using a polynomial kernel of degree 8 allows us to model **any possible labeling** of the challenge inputs.

Feature Space Dimensionality

The feature space induced by the polynomial kernel of degree d over n input dimensions has dimensionality:

$$\sum_{k=0}^{d} \binom{n+k-1}{k}$$

For $n = 8$ and $d = 8$:

$$\text{dim} = \sum_{k=0}^{8} \binom{8+k-1}{k} = \sum_{k=0}^{8} \binom{7+k}{k} = 1+8+36+120+330+792+1716+3432+6435 = 12870$$

This is more than sufficient to separate all possible labelings of the 256-point input space.

Suggested Parameters

- **Kernel type**: Polynomial
- **Degree**: $d = \boxed{8}$
- **Bias term**: $r = \boxed{1}$
- **Scaling factor (gamma)**: $\gamma = \boxed{1}$

The kernel becomes:

$$K(\mathbf{c}, \mathbf{c}') = (\mathbf{c}^\top \mathbf{c}' + 1)^8$$

Comparison with Other Kernels

Kernel	**Expressivity**	**Tuning Needed**	**Perfect Classification**
Polynomial (deg = 8)	High	Low	✓
RBF (Gaussian)	Very High	High (tune γ)	✓(possible)
Matern	Moderate	Moderate	×

Conclusion

Using a polynomial kernel of degree 8 guarantees perfect classification of any function over an 8-bit binary space. This kernel exactly captures the expressivity required to model even complex physical systems such as Arbiter or ML-PUFs. While RBF kernels can also work, they require delicate tuning. Hence, the polynomial kernel offers a theoretically sound and practically reliable choice.

3 Problem 1.2: Inverting Arbiter PUFs

Given a trained arbiter PUF linear model represented by a weight vector $w \in \mathbb{R}^{64}$ and a bias term $b \in \mathbb{R}$, we aim to reconstruct the original 256 physical delays (p_i, q_i, r_i, s_i) used to generate the linear model. This constitutes an inverse problem.

Formulating the Inverse Problem

The standard Arbiter PUF model computes the difference in delay as:

$$\Delta = \sum_{i=0}^{63} \alpha_i x_i + \sum_{i=1}^{63} \beta_i x_i + \beta_{64}$$

where $\alpha_i = \frac{p_i - q_i + r_i - s_i}{2}$ and $\beta_i = \frac{p_i - q_i - r_i + s_i}{2}$, and x_i are deterministic parity terms derived from the challenge vector.

The trained linear model essentially computes:

$$w^\top \phi(c) + b$$

This implies that we must find 256 values of p_i, q_i, r_i, s_i such that they lead to the same α_i and β_i which satisfy the trained w and b. This is a linear transformation from $\mathbb{R}^{256} \to \mathbb{R}^{65}$ (64 weights and 1 bias). Hence, this can be encoded as a sparse linear system:

$$Ax = b$$

where:

- $A \in \mathbb{R}^{65 \times 256}$ maps delays to linear model parameters
- $x \in \mathbb{R}^{256}$ is the vector of unknown delays (ordered as $p_0, q_0, r_0, s_0, \ldots, p_{63}, q_{63}, r_{63}, s_{63}$)
- $b \in \mathbb{R}^{65}$ is the known linear model (weights and bias)

Each row of A has only up to 8 non-zero entries due to the way α_i and β_i are defined, making A extremely sparse.

Solving the Inverse Problem

We now describe multiple strategies to recover the delays.

1. Least Squares

We solve the optimization problem:

$$\min_{x \in \mathbb{R}^{256}} \|Ax - b\|_2^2$$

This gives the minimum-norm solution. However, the result may contain negative delays.

2. Ridge Regression

To improve numerical stability, we regularize the solution:

$$\min_{x \in \mathbb{R}^{256}} \lambda \|x\|_2^2 + \|Ax - b\|_2^2$$

This avoids overfitting and helps in cases where A is poorly conditioned.

3. Constrained Optimization

To enforce non-negativity of delays, we solve:

$$\min_{x \geq 0} \|Ax - b\|_2^2$$

This can be done using:

- `scipy.optimize.lsq_linear` with `bounds=(0, np.inf)`
- `sklearn.linear_model.Ridge(..., positive=True)`

4. Postprocessing (Melba's Trick)

If using an unconstrained solver, we can clip negative delays after solving:

$$x_i \leftarrow \max(x_i, 0)$$

This is fast and effective when negative values are small due to noise.

5. Custom Sparse Solver (Recommended)

Given A is very sparse and small (65×256), a custom coordinate descent or projected gradient descent solver can be:

- Faster than general-purpose solvers
- More numerically stable
- Easily tuned for non-negativity constraints

A simple iterative update is:

$$x_i \leftarrow \max\left(0, x_i - \eta \cdot \frac{\partial}{\partial x_i} \|Ax - b\|_2^2\right)$$

Conclusion

We cast the problem of recovering physical delays from a linear model as solving a sparse system of 65 linear equations in 256 variables. By combining classical optimization techniques with domain-specific tricks (e.g., Melba's postprocessing), we can efficiently and accurately reconstruct non-negative delays consistent with the trained model.

4 Code

4.1 5. Linear Model Training

Solution for Part 5

*Zipped **Solution** to Assignment 1*

4.2 6. Delay Recovery Code

Solution for Part 6

*Zipped **Solution** to Assignment 1*

5 Performance Analysis for Models

(a) Performance Comparison of LinearSVC with Different Loss Functions:

Loss function	Total Features	Model Train Time (s)	Test accuracy
Hinge	105	0.5813	1.0
Squared Hinge	105	0.6372	1.0

Analysis:

From the table, we observe the following:

- **Training Time:** The model with the Hinge loss function has a shorter training time of 0.5813 seconds compared to the Squared Hinge loss function, which has a training time of 0.6372 seconds. This indicates that the Hinge loss function is computationally less expensive.
- **Test Accuracy:** Both the `hinge` and `squared hinge` loss functions achieve perfect test accuracy of **1.0**, indicating that the model can perfectly classify the test set regardless of the loss choice.

Conclusion: Given that both loss functions result in perfect classification, the hinge loss offers a small advantage in terms of computational efficiency, making it a suitable choice for this problem.

(b) Performance Comparison based on the value of C

Analysis for LinearSVC with different C values:

To assess the impact of the regularization parameter C on the performance of LinearSVC, we evaluated the model with low, medium, and high C values.

Table 1: Performance Comparison of LinearSVC with Different C Values

C Value	Total Features	Model Train Time (s)	Test Accuracy
0.01	105	1.487	0.9263
0.1	105	0.2236	1.0
1	105	1.9586	1.0
10	105	1.7525	1.0
100	105	1.6259	1.0

Training Time:

- The training time shows some fluctuation with varying values of the regularization parameter C.

- The lowest training time of **0.2236 seconds** is observed at $C = 0.1$, while higher values such as $C = 1$, $C = 10$, and $C = 100$ result in training times around **1.6 to 1.95 seconds**.
- Interestingly, a very small value of $C = 0.01$ yields a training time of **1.487 seconds**, which is higher than $C = 0.1$, possibly due to underfitting leading to slower convergence.

Test Accuracy:

- The test accuracy is **perfect (1.0)** for all values of C **except** for $C = 0.01$, which achieves only **0.9263**.
- This indicates that too much regularization (i.e., a very small C) harms the model's ability to fit the training data adequately, leading to underfitting and lower test performance.
- For $C \geq 0.1$, the model generalizes perfectly to the test data.

Conclusion:

- A moderate to high value of the regularization parameter C (e.g., 0.1 to 100) ensures both **fast training** and **perfect accuracy**.
- Extremely small values of C should be avoided due to risk of underfitting.

Analysis for LogisticRegression with different C values:

We also evaluated the performance of LogisticRegression with varying C values.

Table 2: Performance Comparison of LogisticRegression with Different C Values

C Value	Total Features	Model Train Time (s)	Test Accuracy 0
0.01	105	0.1375	1.0
0.1	105	0.2888	1.0
1	105	0.5482	1.0
10	105	0.6894	1.0
100	105	1.1192	1.0

- **Training Time:** LogisticRegression also maintains 100 accuracy regardless of C, but trains fastest at C=100, indicating that weaker regularization leads to quicker convergence due to reduced optimization constraints.

In conclusion, both LinearSVC and LogisticRegression demonstrate robustness to the choice of regularization parameter C, achieving perfect accuracy across all tested values. However, their training efficiencies respond differently to C: LinearSVC performs best with moderate regularization ($C = 1$), while LogisticRegression trains fastest with weaker regularization ($C = 100$). This highlights that, even when accuracy is unaffected, selecting an appropriate C can lead to meaningful gains in computational efficiency depending on the underlying optimization algorithm.

(c) Effect of Variation in Tolerance for LinearSVC and LogisticRegression Models

For part (c) of the analysis, we evaluate the performance of the LinearSVC and LogisticRegression models with varying values of the `tol` hyperparameter. The `tol` hyperparameter controls the tolerance for the optimization algorithm, with lower values potentially leading to longer training times but potentially improved convergence and accuracy.

LinearSVC:

- **Training Time:** The training time decreases significantly with increasing tolerance. At a strict tolerance of 10^{-6}, the training time is 3.4480 seconds, reducing to just 0.1460 seconds at tolerance 1. This behavior indicates faster convergence when the optimization is allowed to stop early with looser tolerance settings.
- **Test Accuracy:** Perfect classification accuracy (1.0) is achieved at lower tolerance values 10^{-6} and 10^{-3}. However, accuracy drops to 0.8225 at tolerance 1, showing that overly loose convergence criteria can lead to underfitting and poor generalization.

Table 3: Performance Comparison of LinearSVC and LogisticRegression with Different Tolerance

Model	Tolerance	Total Features	Model Train Time (s)	Test Accuracy
LinearSVC	10^{-6}	105	3.4480	1.0
LinearSVC	10^{-3}	105	2.1749	1.0
LinearSVC	1	105	0.1460	0.8225
LogisticRegression	10^{-6}	105	4.2326	1.0
LogisticRegression	10^{-3}	105	1.7120	1.0
LogisticRegression	1	105	0.0516	0.85

LogisticRegression:

- **Training Time:** Similar to LinearSVC, the training time drops dramatically as the tolerance increases. From 4.2326 seconds at 10^{-6} to 0.0516 seconds at tolerance 1, suggesting quick convergence under loose optimization conditions.
- **Test Accuracy:** Accuracy remains perfect (1.0) for 10^{-6} and 10^{-3}, but drops to 0.85 at tolerance 1. This again confirms that a high tolerance value (loose convergence) can lead to premature stopping and underfitting.

Conclusion:

- While increasing the tolerance greatly reduces training time for both models, it may lead to underfitting and reduced test accuracy.
- Tolerances in the range of 10^{-3} to 10^{-6} offer the best balance, achieving perfect test accuracy with acceptable training times.

(d) Effect of Variation in Penalty (Regularization) Hyperparameter for LinearSVC and LogisticRegression Models

In part (d) of the analysis, we evaluate the performance of the LinearSVC and LogisticRegression models with different penalty (regularization) hyperparameters. The penalty hyperparameter controls the type of regularization applied to the model to prevent overfitting. We compare the performance of the models using L1 and L2 regularization.

Table 4: Performance Comparison of LinearSVC and LogisticRegression with Different Penalty

Model	Penalty	Total Features	Model Train Time (s)	Test Accuracy
LinearSVC	L1	105	3.5853	1.0
LinearSVC	L2	105	0.6372	1.0
LogisticRegression	L1	105	3.4177	1.0
LogisticRegression	L2	105	0.3936	1.0

LinearSVC:

- **Training Time:** The training time for LinearSVC with L1 penalty is 3.5853 seconds, which is significantly higher than that of L2 penalty (0.6372 seconds). This indicates that L1 regularization, which induces sparsity in coefficients, takes more time to converge.
- **Test Accuracy:** Both L1 and L2 penalties yield perfect classification accuracy (1.0), indicating that either regularization technique is effective for this problem.

LogisticRegression:

- **Training Time:** LogisticRegression also shows a clear difference in training time between L1 (3.4177 seconds) and L2 (0.3936 seconds). L2 penalty leads to much faster convergence, again showing the computational efficiency of L2 regularization.
- **Test Accuracy:** Both regularization methods achieve perfect test accuracy (1.0), demonstrating robustness across regularization types for LogisticRegression as well.

Conclusion:

- Both L1 and L2 regularization result in perfect classification for LinearSVC and LogisticRegression.
- However, L2 regularization is significantly more efficient in terms of training time for both models, making it a better choice when computational cost is a concern.

Conclusion: Both L1 and L2 penalties yield perfect classification performance for LinearSVC and LogisticRegression models. The training times differ only slightly, with L1 being marginally faster in LinearSVC and L2 in LogisticRegression. In this task, either regularization choice works well, and L2 may be preferred for its consistent convergence and computational efficiency unless model sparsity is a requirement.