CS771 Introduction to Machine Learning

Lokesh Mehra
220591
lokeshm22@iitk.ac.in

Anany Dev Choudhary
220135
ananydc22@iitk.ac.in

Anirudh Kumar
220146
anirudhk22@iitk.ac.in

Problem 1

The actual response from the ML-PUF is modeled as:

$$\text{ML-PUF Response} = \text{XOR}(b_1, b_2)$$

where $b_1, b_2 \in \{0, 1\}$ are the responses from two independent Arbiter PUFs.

Let each PUF be represented by a weight vector and a bias: (u_1, a_1) and (u_2, a_2), respectively.

Given a challenge vector $c \in \{0, 1\}^8$, the response bits can be modeled as:

$$b_1 = \frac{1 + \text{sign}(u_1^\top c + a_1)}{2}, \quad b_2 = \frac{1 + \text{sign}(u_2^\top c + a_2)}{2}$$

XOR Truth Table

b_1	b_2	$\text{XOR}(b_1, b_2)$
0	0	0
0	1	1
1	0	1
1	1	0

We note that:

$$\text{sign}(u^\top c + a) = \begin{cases} -1, & \text{if } b = 0 \\ +1, & \text{if } b = 1 \end{cases}$$

Sign Product vs XOR Mapping

$\text{sign}(u_1^\top c + a_1)$	$\text{sign}(u_2^\top c + a_2)$	Product	Expected XOR Output
-1	-1	$+1$	0
-1	$+1$	-1	1
$+1$	-1	-1	1
$+1$	$+1$	$+1$	0

Therefore, we can express the XOR output as:

$$y = \frac{1 - p}{2}, \quad \text{where } p = \text{sign}(u_1^\top c + a_1) \cdot \text{sign}(u_2^\top c + a_2)$$

Using the identity:

$$\prod_{i=1}^{n} \text{sign}(x_i) = \text{sign}\left(\prod_{i=1}^{n} x_i\right),$$

we get:

$$y = \frac{1 - \text{sign}\big((u_1^\top c + a_1)(u_2^\top c + a_2)\big)}{2}$$

Hiding the Bias Term

To hide the bias term inside the model vector, we introduce an augmented representation:

$$\tilde{u}_1 = \begin{bmatrix} u_1 \\ a_1 \end{bmatrix}, \quad \tilde{u}_2 = \begin{bmatrix} u_2 \\ a_2 \end{bmatrix}, \quad \tilde{c} = \begin{bmatrix} c \\ 1 \end{bmatrix}$$

Now we consider the product:

$$(\tilde{u}_1^\top \tilde{c}) \cdot (\tilde{u}_2^\top \tilde{c})$$

We know that:

$$\tilde{u}^\top \tilde{c} = \sum_{i=1}^{9} \tilde{u}_i \cdot \tilde{c}_i = \tilde{u}_1\tilde{c}_1 + \tilde{u}_2\tilde{c}_2 + \cdots + \tilde{u}_9\tilde{c}_9$$

Hence, the product becomes:

$$(\tilde{u}_1^\top \tilde{c})(\tilde{u}_2^\top \tilde{c}) = \left(\sum_{i=1}^{9} \tilde{u}_{1,i} \cdot \tilde{c}_i\right)\left(\sum_{j=1}^{9} \tilde{u}_{2,j} \cdot \tilde{c}_j\right) = \sum_{i=1}^{9}\sum_{j=1}^{9} \tilde{u}_{1,i}\tilde{u}_{2,j}\tilde{c}_i\tilde{c}_j$$

As i,j = 0,1 ... 9, this expansion has $9 \times 9 = 81$ terms of the form:

$$\tilde{u}_{1,i}\tilde{u}_{2,j}\tilde{c}_i\tilde{c}_j \quad \text{for } i, j = 1, 2, \ldots, 9$$

$$\sum_{i=1}^{9}\sum_{j=1}^{9} \tilde{u}_{1,i}\tilde{u}_{2,j}\tilde{c}_i\tilde{c}_j = [\tilde{u}_{1,1}\tilde{u}_{2,1} \quad \tilde{u}_{1,1}\tilde{u}_{2,2} \quad \cdots \quad \tilde{u}_{1,9}\tilde{u}_{2,9}] \begin{bmatrix} \tilde{c}_1\tilde{c}_1 \\ \tilde{c}_1\tilde{c}_2 \\ \vdots \\ \tilde{c}_9\tilde{c}_9 \end{bmatrix}$$

We define

$$W^\top = [\tilde{u}_{1,1}\tilde{u}_{2,1} \quad \tilde{u}_{1,1}\tilde{u}_{2,2} \quad \cdots \quad \tilde{u}_{1,9}\tilde{u}_{2,9}], \quad \phi(\tilde{c}) = \begin{bmatrix} \tilde{c}_1\tilde{c}_1 \\ \tilde{c}_1\tilde{c}_2 \\ \vdots \\ \tilde{c}_9\tilde{c}_9 \end{bmatrix}$$

$$\text{where } W \in \mathbb{R}^{81}$$

Here, $\phi(\tilde{c})$ is a mapping from $\mathbb{R}^9 \to \mathbb{R}^{81}$. Therefore, our earlier expression becomes:

$$y = \frac{1 - \text{sign}(W^\top \phi(\tilde{c}))}{2}$$

This can be rewritten as:

$$y = \frac{1 + \text{sign}(\tilde{W}^\top \phi(\tilde{c}))}{2}, \quad \text{where } \tilde{W}^\top = -W^\top$$

Now, to work directly with the original challenge $c \in \{0,1\}^8$, we define an extended mapping:

$$\tilde{\phi}(c) = \phi(\tilde{c}) \quad \text{where } \tilde{c} = \begin{bmatrix} c \\ 1 \end{bmatrix}$$

Thus, $\tilde{\phi} : \{0,1\}^8 \to \mathbb{R}^{81}$, and we can express the final response function as:

$$y = r(c) = \frac{1 + \text{sign}(\tilde{W}^\top \tilde{\phi}(c) + \tilde{b})}{2}$$

where $\tilde{W} \in \mathbb{R}^{81}$ and the bias term $\tilde{b} = 0$

The dimension can be 64 if the bias term is not hidden,because the vectors will be of 8 dimensions.

Problem 2

As shown above, we have $\tilde{D} = 81$. The calculations are provided below in detail. After hiding the bias term, we consider the multiplication of the outputs of two PUF models, represented as:

$$(\tilde{\boldsymbol{u}}_1^\top \tilde{\boldsymbol{c}})(\tilde{\boldsymbol{u}}_2^\top \tilde{\boldsymbol{c}})$$

We expand this using the inner product formula. Recall that:

$$\tilde{\boldsymbol{u}}^\top \tilde{\boldsymbol{c}} = \sum_{i=1}^{9} \tilde{u}_i \cdot \tilde{c}_i = \tilde{u}_1 \tilde{c}_1 + \tilde{u}_2 \tilde{c}_2 + \cdots + \tilde{u}_9 \tilde{c}_9$$

Therefore, the product becomes:

$$(\tilde{\boldsymbol{u}}_1^\top \tilde{\boldsymbol{c}})(\tilde{\boldsymbol{u}}_2^\top \tilde{\boldsymbol{c}}) = \left(\sum_{i=1}^{9} \tilde{u}_{1,i} \tilde{c}_i \right) \left(\sum_{j=1}^{9} \tilde{u}_{2,j} \tilde{c}_j \right) = \sum_{i=1}^{9} \sum_{j=1}^{9} \tilde{u}_{1,i} \tilde{u}_{2,j} \tilde{c}_i \tilde{c}_j$$

Since $i, j = 1, 2, \ldots, 9$, this expansion contains $9 \times 9 = 81$ unique terms of the form:

$$\tilde{u}_{1,i} \tilde{u}_{2,j} \tilde{c}_i \tilde{c}_j \quad \text{for } i, j \in \{1, 2, \ldots, 9\}$$

This bilinear form can be visualized as a matrix-vector product:

$$\sum_{i=1}^{9} \sum_{j=1}^{9} \tilde{u}_{1,i} \tilde{u}_{2,j} \tilde{c}_i \tilde{c}_j = \begin{bmatrix} \tilde{u}_{1,1} \tilde{u}_{2,1} & \tilde{u}_{1,1} \tilde{u}_{2,2} & \cdots & \tilde{u}_{1,9} \tilde{u}_{2,9} \end{bmatrix} \begin{bmatrix} \tilde{c}_1 \tilde{c}_1 \\ \tilde{c}_1 \tilde{c}_2 \\ \vdots \\ \tilde{c}_9 \tilde{c}_9 \end{bmatrix}$$

Hence, the dimensionality of the expanded feature space for the 2-XOR PUF becomes:

$$\tilde{D} = 9 \times 9 = 81$$

Problem 3

Each individual Arbiter PUF computes a response bit that can be modeled as a linear threshold function over a non-linear transformation of the input, typically involving cumulative products of transformed challenge bits. When two such PUFs are XORed, the resulting function becomes **non-linearly separable** even in the high-dimensional space used for modeling a single PUF.

Why RBF Kernel Works Perfectly

The **RBF (Radial Basis Function) kernel** is defined as:

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

This kernel implicitly maps input vectors into an *infinite-dimensional* feature space, where it becomes possible to linearly separate any two classes, provided the kernel parameters are chosen appropriately and the data is not too noisy.

Since the XOR of two threshold functions is highly non-linear and not linearly separable in the original input space, the RBF kernel is well-suited for this task. It provides the flexibility needed to capture the XOR logic by transforming the input space such that the complex decision boundary becomes linearly separable.

Empirically, we observe that with appropriate values of γ and C, the RBF kernel achieves perfect classification accuracy on the 2-XOR PUF dataset:

$$\text{Accuracy}_{\text{RBF}} = 1.0$$

Polynomial Kernel

The **polynomial kernel** is defined as:

$$K(x, x') = (\gamma x^\top x' + \text{coef}_0)^d$$

This kernel maps input vectors into a finite-dimensional space containing monomials up to degree d.

However, the XOR function is not linearly separable, and it cannot be perfectly modeled using polynomials of low degree. Even at $d = 2$ or $d = 3$, the polynomial kernel lacks the capacity to separate the classes corresponding to the XOR of two threshold functions. Capturing XOR exactly would require a degree equal to the input length (i.e., $d = 8$), which makes the model prone to overfitting and numerically unstable.

Empirically, the best performance with a polynomial kernel is:

$$\text{Accuracy}_{\text{Poly}} \approx 0.97$$

Results

Our initial model used `(kernel="rbf", gamma="scale", C=1.0)`, which gave an accuracy of **98.5625%**. We further improved the RBF kernel using the following grid:

```
rbf_params = {
    "C": [1, 10, 100],
    "gamma": [0.01, 0.1, 1, "scale"]
}
```

The best parameters discovered were:

- `C = 1`
- `gamma = 1`

With these parameters, the RBF kernel achieved a perfect accuracy of **100%**.

Summary

Both kernels demonstrated strong performance, with the RBF kernel slightly outperforming the polynomial kernel after tuning. This highlights the effectiveness of kernel selection and hyperparameter optimization for modeling PUF response data.

Problem 4

Mathematical Formulation

Given a linear model of an Arbiter PUF represented by a 65-dimensional vector $\mathbf{w} = [w_0, w_1, \ldots, w_{63}, b]^T$, where:

- w_0 to w_{63} are the weights
- b is the bias term

we aim to recover the **256 non-negative delays** (p_i, q_i, r_i, s_i for $i = 0$ to 63) that generate this model.

Construction of the Linear System

The relationship between delays and model parameters is:

$$\mathbf{w} = A\mathbf{d}$$

where:

- $\mathbf{d} \in \mathbb{R}^{256}$ is the delay vector:

$$\mathbf{d} = [p_0, q_0, r_0, s_0, p_1, q_1, r_1, s_1, \ldots, p_{63}, q_{63}, r_{63}, s_{63}]^T$$

- $A \in \mathbb{R}^{65 \times 256}$ is the coefficient matrix

Equations for Weights (w_i)

$$\begin{aligned} w_0 &= \frac{(p_0 - q_0) + (r_0 - s_0)}{2} \\ w_i &= \frac{(p_i - q_i + r_i - s_i)}{2} + \frac{(p_{i-1} - q_{i-1} - r_{i-1} + s_{i-1})}{2} \quad \text{for } i \geq 1 \end{aligned}$$

Equation for Bias (b)

$$b = \frac{(p_{63} - q_{63} - r_{63} + s_{63})}{2}$$

Weight Equations

The weights combine current and previous stage delays:

$$\begin{aligned} w_0 &= \alpha_0 = \frac{(p_0 - q_0) + (r_0 - s_0)}{2} \\ w_i &= \alpha_i + \beta_{i-1} \\ &= \frac{(p_i - q_i + r_i - s_i)}{2} + \frac{(p_{i-1} - q_{i-1} - r_{i-1} + s_{i-1})}{2} \quad \text{for } i \geq 1 \end{aligned}$$

Matrix Construction

The coefficient matrix $A \in \mathbb{R}^{65 \times 256}$ encodes these relationships:

Weight Rows (0-63)

Algorithm 1 Matrix Construction

```
for $i = 0$ to 63 do
    $p \leftarrow 4i,\ q \leftarrow 4i + 1,\ r \leftarrow 4i + 2,\ s \leftarrow 4i + 3$
    if $i = 0$ then
        $A[i, p] \leftarrow 0.5,\ A[i, q] \leftarrow -0.5$
        $A[i, r] \leftarrow 0.5,\ A[i, s] \leftarrow -0.5$
    else
        $A[i, p] \leftarrow 0.5,\ A[i, q] \leftarrow -0.5$
        $A[i, r] \leftarrow 0.5,\ A[i, s] \leftarrow -0.5$
        $A[i, 4(i - 1)] \mathrel{+}= 0.5$ {Previous $p$}
        $A[i, 4(i - 1) + 1] \mathrel{+}= -0.5$ {Previous $q$}
        $A[i, 4(i - 1) + 2] \mathrel{+}= -0.5$ {Previous $r$}
        $A[i, 4(i - 1) + 3] \mathrel{+}= 0.5$ {Previous $s$}
    end if
end for
```

Bias Row (64)

The bias term uses only the last stage's delays:

$$b = \beta_{63} = \frac{(p_{63} - q_{63} - r_{63} + s_{63})}{2}$$

$$
\begin{aligned}
A[64, 252] &= 0.5 \quad (p_{63}) \\
A[64, 253] &= -0.5 \quad (q_{63}) \\
A[64, 254] &= -0.5 \quad (r_{63}) \\
A[64, 255] &= 0.5 \quad (s_{63})
\end{aligned}
$$

Solution Method

Projected Gradient Descent

Solve the optimization problem:

$$\min_{\mathbf{d} \geq 0} \|A\mathbf{d} - \mathbf{w}\|_2^2$$

1. Initialize $\mathbf{d}^{(0)} = \mathbf{0}$
2. Compute gradient:
$$\nabla f(\mathbf{d}) = 2A^T(A\mathbf{d} - \mathbf{w})$$
3. Update with projection:
$$\mathbf{d}^{(k+1)} = \mathcal{P}_+ \left(\mathbf{d}^{(k)} - \alpha \nabla f(\mathbf{d}^{(k)})\right)$$
where $\mathcal{P}_+(x) = \max(x, 0)$
4. Stop when $\|\mathbf{d}^{(k+1)} - \mathbf{d}^{(k)}\|_2 < 10^{-6}$

Output

The solution $\mathbf{d}$ is reshaped into four vectors:

$$
\begin{aligned}
P &= [p_0, p_1, \ldots, p_{63}] \\
Q &= [q_0, q_1, \ldots, q_{63}] \\
R &= [r_0, r_1, \ldots, r_{63}] \\
S &= [s_0, s_1, \ldots, s_{63}]
\end{aligned}
$$

Properties

- **Non-Negativity**: All delays ≥ 0
- **Convergence**: Guaranteed to local minimum

Problem 7

Table 1: Effect of Loss Function in LinearSVC on Test Accuracy and Training Time

Loss	C	Test Accuracy	Training Time (s)
hinge	1	0.92625	0.1005
squared_hinge	1	0.92	0.1712
hinge	100	0.9356	0.2562
squared_hinge	100	0.914375	0.2570

For both models, changing the loss parameter to "squared_hinge" from "hinge", decreases the test accuracy and also increases the training time.

Table 2: Effect of Regularization Parameter C on Model Performance

Model	C	Test Accuracy	Training Time (s)
LinearSVC	0.01	0.8768	0.0232
LinearSVC	1	0.92	0.2099
LinearSVC	100	0.9106	0.1832
LogisticRegression	0.01	0.7693	0.1469
LogisticRegression	1	0.92	0.0576
LogisticRegression	100	0.97	0.0494

For **LogisticRegression**, changing "**C**" from 0.01 to 1 to 100 increases test accuracy greatly and also improves training time significantly.

Table 3: Effect of Tolerance (tol) on Model Performance

Model	tol	Test Accuracy	Training Time (s)
LinearSVC	1e-1	0.9243	0.1100
LinearSVC	1e-3	0.92	0.1959
LinearSVC	1e-5	0.92	0.2604
LogisticRegression	1e-1	0.92	0.0437
LogisticRegression	1e-3	0.92	0.0388
LogisticRegression	1e-5	0.92	0.0334

As evident from the above results, by changing the "**tol**", there is no significant change in the accuracy values, but the training times are affected. For **LinearSVC** the time increases while for **LogisticRegression** it decreases.

Table 4: Effect of Regularization Penalty on Model Performance

Model	Penalty	Test Accuracy	Training Time (s)
LinearSVC	l1	0.9256	1.1413
LinearSVC	l2	0.92	0.0490
LogisticRegression	l1	0.9356	0.5592
LogisticRegression	l2	0.92	0.0338

When running Logistic Regression with penalty = "l1", the solver was set to "liblinear".

Changing the penalty paramater to "**l2**" from "**l1**", does not change the accuracy much, but the training times are significantly reduced. For **LinearSVC**, the time reduces to 3% of the previous time and for **LogisticRegression**, it comes down to 7% of its original value.