CSS771 Mini Project Report

Team Members:

Mantavya Upadhyay (220627)
Anjali Rathore (220156)
Akshat Chouksey (220099)
Manvi Verma (220631)
Shreyansh (221031)
Pravallika Mudunuru (220814)

1 Solving a Multi-level PUF

Physical Unclonable Functions (PUFs) use tiny variations from manufacturing to provide hardware-based security. A basic arbiter PUF sets up a timing race controlled by a challenge input, with the output depending on which signal wins. However, these simple PUFs can be broken using linear machine learning models. The Multi-level PUF (ML-PUF) adds complexity by using two separate PUFs, cross-comparing their paths, and taking the XOR of two intermediate results to produce the final output. The task here is to show that even this more complex ML-PUF design is still vulnerable to machine learning. By finding the right way to transform the 8-bit challenges, we aim to train a linear model that can predict the final response with high accuracy.

Question 1

Question:

Give a detailed mathematical derivation (as given in the lecture slides) how a single linear model can predict the responses of an ML-PUF. Specifically, give an explicit map

$$\tilde{\phi} : \{0,1\}^8 \to \mathbb{R}^D$$

and a corresponding linear model

$$\tilde{\mathbf{W}} \in \mathbb{R}^D, \quad \tilde{b} \in \mathbb{R}$$

that predicts the responses, i.e., for all CRPs $c \in \{0,1\}^8$, we have

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

where $r(c)$ is the response of the ML-PUF on the challenge c. Note that $\tilde{\mathbf{W}}, \tilde{b}$ may depend on the PUF-specific constants such as delays in the multiplexers. However, the map $\tilde{\phi}(c)$ must depend only on the challenge c.

Given the Multi-Level PUF (ML-PUF) setup as described, we aim to show that a linear model can indeed predict its responses. Below, we systematically construct the mathematical formulation of the ML-PUF and derive a feature map $\vec{\phi} : \{0,1\}^8 \to \mathbb{R}^D$ along with a linear weight vector $\vec{W} \in \mathbb{R}^D$ and bias $b \in \mathbb{R}$ that recovers the ML-PUF response.

Arbiter PUF Basics

Each stage i of a regular arbiter PUF introduces conditional delays depending on the challenge bit C_i. The signal delay equations are:

$$t_i^u = (1 - C_i)(t_{i-1}^u + p_i) + C_i(t_{i-1}^\ell + s_i),$$
$$t_i^\ell = (1 - C_i)(t_{i-1}^\ell + q_i) + C_i(t_{i-1}^u + r_i).$$

Likewise, for a second PUF instance (say PUF1), we write:

$$T_i^u = (1 - C_i)(T_{i-1}^u + P_i) + C_i(T_{i-1}^\ell + S_i),$$
$$T_i^\ell = (1 - C_i)(T_{i-1}^\ell + Q_i) + C_i(T_{i-1}^u + R_i).$$

Let us define the differences:

$$\Delta_i = t_i^u - T_i^u, \qquad \delta_i = t_i^\ell - T_i^\ell.$$

Recursive Formulation for ML-PUF

For ML-PUF, the key recursive relations for delay differences are:

$$\begin{aligned}\Delta_i &= \frac{1+d_i}{2}(\Delta_{i-1}+p_i-P_i)+\frac{1-d_i}{2}(\delta_{i-1}+s_i-S_i),\\ \delta_i &= \frac{1+d_i}{2}(\delta_{i-1}+q_i-Q_i)+\frac{1-d_i}{2}(\Delta_{i-1}+r_i-R_i),\end{aligned}$$

where $d_i = 1-2C_i \in \{\pm1\}$.
Expanding and simplifying:

$$\begin{aligned}2\Delta_i &= \Delta_{i-1}+\delta_{i-1}+(p_i-P_i+s_i-S_i)+d_i(\Delta_{i-1}-\delta_{i-1}+p_i-P_i-s_i+S_i),\\ 2\delta_i &= \Delta_{i-1}+\delta_{i-1}+(q_i-Q_i+r_i-R_i)+d_i(\delta_{i-1}-\Delta_{i-1}+q_i-Q_i-r_i+R_i).\end{aligned}$$

Define parameters:

$$\begin{aligned}\alpha_i &= \frac{p_i-P_i-s_i+S_i}{2}, \quad \beta_i = \frac{p_i-P_i+s_i-S_i}{2},\\ A_i &= \frac{q_i-Q_i-r_i+R_i}{2}, \quad B_i = \frac{q_i-Q_i+r_i-R_i}{2}.\end{aligned}$$

So the updates become:

$$\begin{aligned}\Delta_i &= d_i\left(\frac{\Delta_{i-1}-\delta_{i-1}}{2}+\alpha_i\right)+\beta_i+\frac{\Delta_{i-1}+\delta_{i-1}}{2},\\ \delta_i &= d_i\left(\frac{\delta_{i-1}-\Delta_{i-1}}{2}+A_i\right)+B_i+\frac{\Delta_{i-1}+\delta_{i-1}}{2}.\end{aligned}$$

Sum and Difference Derivations

We start by computing:

$$\begin{aligned}\Delta_i+\delta_i &= d_i\left(\alpha_i+A_i\right)+(\beta_i+B_i)+\Delta_{i-1}+\delta_{i-1},\\ \Delta_i-\delta_i &= d_i(\Delta_{i-1}-\delta_{i-1}+\alpha_i-A_i)+(\beta_i-B_i).\end{aligned}$$

$\mathbf{Sum}_i = \Delta_i+\delta_i$,
$\mathbf{Diff}_i = \Delta_i-\delta_i$.

Expressing Sum and Difference as Linear Forms

$$\begin{aligned}\mathrm{Sum}_i &= d_i(\alpha_i+A_i)+(\beta_i+B_i)+\mathrm{Sum}_{i-1},\\ \mathrm{Diff}_i &= d_i(\mathrm{Diff}_{i-1}+\alpha_i-A_i)+(\beta_i-B_i).\end{aligned}$$

Sum_i is a linear model and Diff_i model is suffix sum model discussed in class. Define the vectors of challenge-dependent signs:

$$x = [d_0 \quad d_1 \quad \dots \quad d_{n-1}]^T \quad \text{and} \quad y = \left[\prod_{i=0}^{n-1} d_i \quad \prod_{i=1}^{n-1} d_i \quad \dots \quad \prod_{i=n-1}^{n-1} d_i\right]^T$$

We can then express the accumulated sum across stages as

$$\text{Sum}_{n-1} = w^T x + b,$$

and the final stage difference as

$$\text{Diff}_{n-1} = W^T y + B.$$

Here $w \in \mathbb{R}^n$, $W \in \mathbb{R}^n$, and b, B absorb the PUF-specific delay constants.

Linear Model for Final Response

The upper and lower arbiter comparisons at the final stage obey:

$$\begin{aligned} 2\Delta_{n-1} &= w^T x + W^T y + b + B, \\ 2\delta_{n-1} &= w^T x - W^T y + b - B. \end{aligned}$$

Thus, their product simplifies to

$$\begin{aligned} 4\Delta_{n-1} \cdot \delta_{n-1} &= (w^T x + b)^2 - (W^T y + B)^2 \\ &= ((w \otimes w)^T (x \otimes x) + 2b(w^T x)) - ((W \otimes W)^T (y \otimes y) + 2B(W^T y)) + (b^2 - B^2) \\ &= (w \otimes w)^T (x \otimes x) - (W \otimes W)^T (y \otimes y) + 2b(w^T x) - 2B(W^T y) + (b^2 - B^2) \\ &= \sum_{0 \le i < j \le n-1} (w_i w_j x_i x_j - W_i W_j y_i y_j) + 2b \sum_{i=0}^{n-1} w_i x_i - 2B \sum_{i=0}^{n-1} W_i y_i + \text{const.} \end{aligned}$$

Thus, omitting the pure constant term, define

$$\phi(x, y) = \begin{bmatrix} (x \otimes x)_{i<j} \\ x \\ y \end{bmatrix}, \quad \theta = \begin{bmatrix} w_i w_j \quad (i < j) \\ 2bw_i \\ -2BW_i \end{bmatrix},$$

so that

$$4\Delta_{n-1} \cdot \delta_{n-1} = \theta^T \phi(x, y) + \text{const.}$$

where $(x \otimes x)_{i<j}$ (and likewise $(y \otimes y)_{i<j}$) denotes taking only the strictly upper-triangular entries of the Khatri–Rao product.

Therefore, the ML-PUF response

$$r(c) = \frac{1 + \text{sign}(\Delta_{n-1} \cdot \delta_{n-1})}{2}$$

can be written as a **linear function** over a lifted feature space that includes all quadratic monomials $x_i x_j$, i.e., there exists a mapping

$$\tilde{\varphi}(c) = \{x_i x_j : 1 \le i \le j \le D\}$$

and corresponding weights $\widetilde{W}$ and bias $\widetilde{b}$, such that

$$r(c) = \frac{1 + \text{sign}(\widetilde{W}^T \widetilde{\varphi}(c) + \widetilde{b})}{2}.$$

Linear Prediction

Let $\widetilde{W} \in \mathbb{R}^{93}$ and $\widetilde{b} \in \mathbb{R}$ be trained weights and bias that depend on the PUF instance (the internal delays). Then the final response is given by

$$\hat{r}(c) = \frac{1 + \text{sign}(\widetilde{W}^T \widetilde{\varphi}(c) + \widetilde{b})}{2}.$$

This completes the proof that an ML-PUF remains linearly predictable given enough challenge–response pairs.

Question 2

Question:
What dimensionality $\widetilde{D}$ does the linear model need to have to predict the response for an ML-PUF? Given calculations showing how you arrived at that dimensionality. The dimensionality should be stated clearly and separately in your report, and not be implicit or hidden away in some calculations.
Solution:
The linear model requires a $\widetilde{D} = 99$ to predict the response for an ML-PUF. The calculations leading to this conclusion are stated below:

We define the map $\widetilde{\varphi} : \{0,1\}^8 \to \mathbb{R}^{99}$ as follows:

- Convert $c \in \{0,1\}^8$ to $d_i = 1 - 2c_i \in \{\pm 1\}$ for $i = 0, \dots, 7$.
- Form adjacent pairwise features $d_i d_{i+1}$ for $i = 0, \dots, 6$ (total 7 features).
- Concatenate all features into a vector

$$\mathbf{v} = [d_0, d_1, \dots, d_7, d_0 d_1, d_1 d_2, \dots, d_6 d_7] \in \mathbb{R}^{15}.$$

- Compute all unique cross terms $v_i v_j$ for $i < j$, giving:

$$\binom{15}{2} = 105 \text{ cross terms.}$$

- Remove 27 known redundant features due to overlapping interactions. These typically include:
  - Redundant combinations such as $(d_i)(d_{i+1})$, $(d_i)(d_i d_{i+1})$, or $(d_i d_{i+1})(d_{i+1} d_{i+2})$.
- Final feature count:

$$8 \text{ (from } d_i) + 7 \text{ (from } d_i d_{i+1}) + (105 - 27) = 93 \text{ features.}$$

Question 3

Question:
Suppose we wish to use a kernel SVM to solve the problem instead of creating our own feature map. That is, we wish to use the original challenges $c \in \{0,1\}^8$ as input to a kernel SVM (i.e., without doing things to the features like converting challenge bits to ± 1 bits and taking cumulative products etc). What kernel should we use so that we get perfect classification? Justify your answer with calculations and give suggestions for kernel type (RBF, poly, Matern etc) as well as kernel parameters (gamma, degree, coef etc). Note that you do not have to submit code or experimental results for this part — theoretical calculations are sufficient.

Solution: To determine the appropriate kernel for modeling the ML-PUF using an SVM with original challenges $c \in \{0,1\}^8$ as input, we need to analyze the complexity of the relationship between challenges and responses established in Question 1.

1.3.1 Analysis of the ML-PUF Response Function

From Question 1, we derived that the ML-PUF response depends on the sign of $\Delta_{n-1} \cdot \delta_{n-1}$, which can be expressed as:

$$\Delta_{n-1:d_0=1} = \frac{1}{4}\left[(w^T x + w'^T y + b)^2 - (kd_{n-1} + k')^2\right] \tag{1}$$

Where:

$$\begin{aligned}
&x = [d_0, d_1, \ldots, d_7]^T \text{ with } d_i = 1 - 2c_i \in \{-1, 1\} \\
&y = [d_0 d_1, d_0 d_2, \ldots, d_6 d_7]^T \\
&w, w', k, k', b \text{ are PUF-specific constants}
\end{aligned}$$

Expanding the squared term $(w^T x + w'^T y + b)^2$:

$$\begin{aligned}
(w^T x + w'^T y + b)^2 &= (w^T x)^2 + (w'^T y)^2 + b^2 + 2(w^T x)(w'^T y) + 2b(w^T x) + 2b(w'^T y) \\
&= \sum_{i,j} w_i w_j x_i x_j + \sum_{i,j} w'_i w'_j y_i y_j + b^2 + 2\sum_{i,j} w_i w'_j x_i y_j \\
&\quad + 2b \sum_i w_i x_i + 2b \sum_i w'_i y_i
\end{aligned} \tag{2}$$

Since $y_i = d_i d_{i+1}$, the expansion contains several types of terms:

- Linear terms: $x_i = d_i$ (first-order)
- Quadratic terms: $x_i x_j = d_i d_j$ and $y_i = d_i d_{i+1}$ (second-order)
- Cubic terms: $x_i y_j = d_i \cdot d_j d_{j+1}$ (third-order)
- Quartic (biquadratic) terms: $y_i y_j = d_i d_{i+1} \cdot d_j d_{j+1}$ (fourth-order)

Additionally, the term $(kd_{n-1} + k')^2$ expands to $k^2 + (k')^2 + 2kk'd_{n-1}$, which includes linear terms in d_{n-1}.

1.3.2 Kernel Selection

Given the complexity of the ML-PUF response function, which includes terms up to the fourth degree, we need a kernel that can capture these higher-order interactions. The appropriate choice is a **polynomial kernel of degree 4**:

$$K(x, y) = (\gamma \langle x, y \rangle + r)^4 \tag{3}$$

Where:

- γ is a scaling parameter
- r is a constant term (coef0)
- The degree is set to 4 to capture all interaction terms up to quartic order

1.3.3 Parameter Justification

For optimal performance with the ML-PUF problem:

- $d = 4$ (degree): Required to capture the highest-order (quartic) interactions in the ML-PUF response function.
- $\gamma = 1$ (gamma): This gives equal weight to the dot product term.
- $r = 1$ (coef0): Including this term ensures that lower-order interactions (linear, quadratic, cubic) are also represented in the feature space.

1.3.4 Mathematical Justification

When expanded, the polynomial kernel with $d = 4$ implicitly maps the input space to a feature space containing all monomials of the original features up to degree 4:

$$(\gamma\langle x, y\rangle + r)^4 = \sum_{i=0}^{4} \binom{4}{i} \gamma^i r^{4-i} \langle x, y\rangle^i \tag{4}$$

This expansion includes:

- $\gamma^4\langle x, y\rangle^4$: Captures quartic (4th-order) interactions
- $4\gamma^3 r\langle x, y\rangle^3$: Captures cubic (3rd-order) interactions
- $6\gamma^2 r^2\langle x, y\rangle^2$: Captures quadratic (2nd-order) interactions
- $4\gamma r^3\langle x, y\rangle$: Captures linear (1st-order) terms
- r^4: Constant bias term

This covers all the types of interactions present in the ML-PUF response function that we identified in our analysis.

1.3.5 Alternative Kernel Considerations

- **RBF Kernel** ($K(x, y) = \exp(-\gamma\|x-y\|^2)$): While theoretically capable of approximating any continuous function, it would require careful tuning of γ and might not be as directly aligned with the polynomial structure of the ML-PUF function.
- **Lower-degree Polynomial Kernels**: A degree-2 polynomial kernel would be insufficient as it cannot capture the cubic and quartic terms. A degree-3 kernel would miss the quartic interactions.
- **Higher-degree Polynomial Kernels ($d > 4$)**: Would introduce unnecessary complexity and potential overfitting, as our analysis shows that terms of degree higher than 4 are not present in the ML-PUF response function.

1.3.6 Conclusion

Based on our detailed analysis of the ML-PUF response function, we recommend a polynomial kernel of degree 4 with parameters $\gamma = 1$ and $r = 1$. This kernel configuration will implicitly map the original binary challenges to a feature space that contains all the necessary interaction terms (up to fourth-order) required to accurately model the ML-PUF responses without requiring explicit feature engineering.

2 Delay Recovery by Inverting an Arbiter PUF

This problem focuses on recovering delay values in a 64-bit Arbiter PUF by reversing a linear model built from challenge-response pairs. Since the system has only 65 equations but 256 unknown delay values, it's underdetermined—meaning we can't find a unique solution. Instead, the goal is to find a set of 256 non-negative delay values that still match the given model, which is defined by a weight vector $w \in \mathbb{R}^{64}$ and a bias term $b \in \mathbb{R}$. There are given 10 such models (each with 65 real numbers), and for each one, there is a need to compute a valid set of delays that fits the model. This task involves solving underdetermined systems while keeping all delays non-negative, which is useful for understanding how machine learning can still break Arbiter PUFs by modeling their internal delays.

Question 4

Question:

Outline a method which can take a 64 + 1-dimensional linear model corresponding to a simple arbiter PUF (unrelated to the ML-PUF in the above parts) and produce 256 non-negative delays that generate the same linear model. This method should show how the model generation process of taking 256 delays and converting them to a 64+1-dimensional linear model can be represented as a system of 65 linear equations and then showing how to invert this system to recover 256 non-negative delays that generate the same linear model. This could be done, for example, by posing it as an (constrained) optimization problem or other ways.

Solutions:

A Simple Arbiter PUF is modeled as a linear function:

$$f(c) = \text{sign}(w^T \cdot \phi(c))$$

Where:

- $c \in \{0,1\}^{64}$ is the challenge
- $\phi(c) \in \mathbb{R}^{65}$ is the feature vector
- $w \in \mathbb{R}^{65}$ is the weight vector (including bias)

The output is either $+1$ or -1, based on the sign of delay difference.
Each Arbiter PUF has 64 stages, each with 4 delays:

$$\{p_i, q_i, r_i, s_i\}, \quad i = 0 \dots 63$$

There are a total of $4 \times 64 = 256$ delays.
Define the cumulative delay difference as:

$$\Delta(c) = \sum_{i=0}^{63} (\alpha_i + \beta_{i-1}) \cdot \phi_i(c) + \beta_{63}$$

Where:

$$\alpha_i = \frac{1}{2}(p_i + q_i - r_i - s_i) \quad \text{and} \quad \beta_i = \frac{1}{2}(p_i - q_i + r_i - s_i)$$

Linear Model

The weight vector is given by:

$$w_i = \alpha_i + \beta_{i-1}, \quad i = 0 \dots 63, \quad w_{64} = \beta_{63}$$

Assuming $\beta_{-1} = 0$, we get:

$$\begin{aligned} w_0 &= \alpha_0 \\ w_1 &= \alpha_1 + \beta_0 \\ w_2 &= \alpha_2 + \beta_1 \\ &\vdots \\ w_{63} &= \alpha_{63} + \beta_{62} \\ w_{64} &= \beta_{63} \end{aligned}$$

Inverse Problem

Given $w \in \mathbb{R}^{65}$, recover delays $\{p_i, q_i, r_i, s_i\}_{i=0}^{63}$.
Step 1: Recover α and β
Assume:

$$\beta_i = 0 \text{ for } i = 0 \dots 62, \quad \beta_{63} = w_{64}$$

Then:

$$\alpha_i = \begin{cases} w_i, & i = 0 \dots 62 \\ w_{63} - \beta_{63}, & i = 63 \end{cases}$$

Step 2: Solve for Delays
Using:

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

Assume:

$$q_i = s_i = 0$$

Then the system simplifies to:

$$\alpha_i = \frac{1}{2}(p_i - r_i) \quad \beta_i = \frac{1}{2}(p_i + r_i)$$

Solving:

$$p_i = \alpha_i + \beta_i \quad r_i = \beta_i - \alpha_i \quad q_i = 0 \quad s_i = 0$$

Step 3: Ensure Non-negativity
Since physical delays cannot be negative, apply a shift:

$$t_1 = \max(0, -\min(p)) \quad t_2 = \max(0, -\min(r))$$

Update delays:

$$p_i \leftarrow p_i + t_1 \quad q_i \leftarrow q_i + t_1 = t_1 \quad r_i \leftarrow r_i + t_2 \quad s_i \leftarrow s_i + t_2 = t_2$$

This ensures $p_i, q_i, r_i, s_i \geq 0$ for all i.
Finally, **Given:**

$$w \in \mathbb{R}^{65}$$

Choose:

$$\beta_i = 0 \text{ for } i = 0 \ldots 62, \quad \beta_{63} = w_{64}$$

Then:

$$\alpha_i = w_i \text{ for } i = 0 \ldots 63$$

Compute delays:

$$p_i = \alpha_i + \beta_i$$
$$r_i = \alpha_i - \beta_i$$
$$q_i = s_i = 0$$

Apply shift t_1, t_2 to ensure all delays are non-negative.

Question 7

Question:
Report outcomes of experiments with both the sklearn.svm.LinearSVC and sklearn.linear_model. LogisticRegression methods when used to learn the linear model for problem 1.1 (breaking the ML-PUF). In particular, report how various hyperparameters affected training time and test accuracy using tables and/or charts. Report these experiments with both LinearSVC and LogisticRegression methods even if your own submission uses just one of these methods or some totally different linear model learning method (e.g. RidgeClassifier). In particular, you must report how at least 2 of the following affect training time and test accuracy: (a) changing the loss hyperparameter in LinearSVC (hinge vs squared hinge), (b) setting C in LinearSVC and LogisticRegression to high/low/medium values, (c) changing tol in LinearSVC and LogisticRegression to high/low/medium values, (d) changing the penalty (regularization) hyperparameter in LinearSVC and LogisticRegression (l2 vs l1).

Solution:
We performed experiments using both `sklearn.svm.LinearSVC` and `sklearn.linear_model.LogisticRegression` to learn linear models for breaking the ML-PUF, as discussed in Problem 1.1. The goal was to understand how hyperparameters influence training time and test accuracy.

We examined the impact of the following hyperparameters:

- **loss** (LinearSVC): 'hinge' vs 'squared_hinge'
- **C**: Inverse regularization strength set to 0.01 (high regularization), 1 (medium), and 100 (low regularization)
- **tol**: Tolerance for stopping criteria set to 1e-2, 1e-4, and 1e-6
- **penalty**: 'l2' vs 'l1' (only supported with certain solvers)

The results are summarized in the following tables.

Table 1: LinearSVC: Effect of `loss` and `C` on Accuracy and Training Time

Loss	C	Accuracy (%)	Training Time (s)
hinge	N/A	100.0	0.25
squared_hinge	N/A	100.0	1.02

Table 2: Effect of `C` on Accuracy and Training Time

Model	C	Accuracy (%)	Training Time (s)
LinearSVC	0.01	90.81	0.45
LinearSVC	1	100.0	0.59
LinearSVC	100	100.0	0.57
LogisticRegression	0.01	92.13	0.35
LogisticRegression	1	100.0	0.62
LogisticRegression	100	100.0	0.75

Table 3: Effect of `tol` on Accuracy and Training Time (`C=1`)

Model	tol	Accuracy (%)	Training Time (s)
LinearSVC	10^{-2}	100	0.47
LinearSVC	10^{-4}	100	0.50
LinearSVC	10^{-6}	100	0.54
LogisticRegression	10^{-2}	100	0.49
LogisticRegression	10^{-4}	100	0.46
LogisticRegression	10^{-6}	100	0.69

Table 4: Effect of `penalty` on Accuracy and Training Time (LinearSVC, C=1, tol=10^{-4})

Penalty	Accuracy (%)	Training Time (s)
l1	100.0	1.45
l2	100.0	0.49

Table 5: Effect of `penalty` on Accuracy (LogisticRegression)

Penalty	Accuracy (%)
l2	100.0
l1	100.0

From these experiments, we observed the following:

- `Squared_hinge` loss in `LinearSVC` resulted in longer training times compared to `hinge`, but accuracy was identical (100%). Hinge loss is significantly faster, with training times 3-34× shorter than squaredhinge. Both loss functions achieve perfect accuracy on the test set when C is greater then equals to 1.0. Squaredhinge with L1 penalty frequently fails to converge with tight tolerance (1e-6).
- Increasing the value of `C` generally improved accuracy and slightly increased training time. Both models achieved 100% accuracy for `C>=1` .
- Decreasing `tol` improved precision but also led to longer training. For practical purposes, `tol = 1e-4` gave optimal balance for logistic regression.
- Both `l1` and `l2` penalties achieved 100% accuracy in this experiment. However, `l2` was significantly faster than `l1` for LinearSVC.