Introduction to Machine Learning (CS771)
Mini-Project

Aryan Nigam
220227
naryan22@iitk.ac.in

Akarsh Verma
220094
averma22@iitk.ac.in

Adwait Kadam
220084
adwaitvk22@iitk.ac.in

Sanidhya Jangid
220962
sanidhyakj22@iitk.ac.in

Yasaswi
221162
vyasaswi22@iitk.ac.in

Group Name - Stack Overflow

Group Number - 6

Instructor - Dr. Purushottam Kar

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

1 Introduction

A Physical Unclonable Function (PUF) is a hardware security primitive that leverages inherent manufacturing variations to produce unique, unpredictable and unclonable responses to input challenges. One widely studied type is the arbiter PUF, which consists of a series of multiplexers. Each multiplexer processes a challenge bit and introduces delay differences along two signal paths—an upper and a lower line. At the end, an arbiter (like a flip-flop) compares the arrival times and outputs a single-bit response based on which signal arrives first. Although the delays are fixed and unique to each chip, they are also consistent—making them ideal for authentication and device identification. However, conventional arbiter PUFs are vulnerable to machine learning (ML) attacks that can predict their responses using linear models derived from challenge-response pairs (CRPs).[1]

2 Problem Statements

2.1 Problem 1.1

To make PUFs more resistant to ML attacks, a new architecture called the Multi-Level PUF (ML-PUF) is introduced. Instead of comparing signals within one PUF, the ML-PUF compares corresponding signals between two different arbiter PUFs (PUF0 and PUF1) using two arbiters and then applies an XOR to the two response bits to generate the final response. The idea is that this added complexity—cross-signal comparison and XOR would make linear modeling ineffective.

The task is to disprove this assumption by constructing a linear model that can still accurately predict ML-PUF responses using a sufficient number of CRPs, thus demonstrating that ML-PUFs are also vulnerable to well-designed learning attacks.

Approach:

To evaluate the susceptibility of Multi-Level Physical Unclonable Functions (ML-PUFs) to machine learning attacks, we construct a complete linear modeling pipeline. The approach involves transforming binary challenge vectors into a feature space that captures the underlying delay dynamics of the PUF. We then train a linear classifier to predict the PUF response based on these features. This allows us to assess whether even structurally complex PUFs like ML-PUFs can be effectively modeled using simple, interpretable models.

Feature Transformation:

Each 8-bit challenge vector $X \in \{0,1\}^8$ is first converted to a signed vector $d \in \{-1,1\}^8$ using the transformation $d_i = 1 - 2X_i$. This step reflects the influence of each challenge bit on the upper or lower delay path within the PUF.

To simulate cumulative delay effects along the PUF stages, we define a vector ξ, where $\xi_i = \prod_{j=i}^{8} d_j$. This cumulative product captures bit interactions as they propagate through the PUF structure.

From d and ξ, we construct a comprehensive 89-dimensional feature vector consisting of:

- 8 features from the delay vector d,
- 8 features from the cumulative product vector ξ,
- 36 second-order features from $d_i \cdot d_j$ for $i \leq j$,
- 36 second-order features from $\xi_i \cdot \xi_j$ for $i \leq j$,
- 1 constant bias term.

This feature engineering step enables linear models to capture both linear and quadratic relationships inherent to ML-PUF delay behavior.

Modeling:

[1] [MDPI] PUF Modeling Attacks Using Machine Learning Algorithms

We train a logistic regression model with ℓ_2 regularization using the transformed feature vectors and the corresponding ML-PUF response labels. Logistic regression provides probabilistic outputs and maintains interpretability, making it a strong baseline for modeling.

To assess sparsity and feature importance, we also experiment with a linear Support Vector Machine (SVM) using ℓ_1 regularization. Both models are evaluated on prediction accuracy to determine their effectiveness in capturing the PUF behavior.

The success of these models in learning accurate response functions demonstrates that ML-PUFs, despite their structural complexity, are vulnerable to linear learning attacks.

Results:

The following table summarizes the performance metrics of the ML-PUF linear modeling pipeline over 5 trials:

Metric	Logistic Regression (L2)	SVM (L1)
Dimensionality	89	89
Train Time (s)	0.2515	6.3496
Map Time (s)	0.0020	0.0021
Error	0.0002	0.0000
Accuracy (%)	99.89	100.00
Decode Time (s)	0.0002	0.0001
Reconstruction Error	4.53×10^{-16}	4.53×10^{-16}

This table highlights the key metrics: model dimensionality, average training time, feature mapping time, error rate, accuracy, average decode time, and reconstruction error for the 5 trials.

Conclusion:

The modeling pipeline demonstrates efficient performance with very high accuracy. The training and feature mapping times are minimal, and the model reconstruction error is near zero, indicating that the learned model closely matches the original physical behavior of the ML-PUF. Despite the simplicity of the linear model, it shows excellent predictive power, confirming that ML-PUFs can be vulnerable to linear attacks.

Problem 1.2

A k-bit Arbiter PUF is described using k delays p_i, q_i, r_i, s_i for $0 \leq i \leq k-1$, following the notation used in class. These delays contribute to generating a linear model $(\mathbf{w} \in \mathbb{R}^k, b \in \mathbb{R})$ that can be used to break the Arbiter PUF, as discussed in lecture.

To briefly recall, we define:

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

Then, the linear model is given by:

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

Melbo has used leaked challenge-response pairs (CRPs) for a 64-bit Arbiter PUF to learn such a linear model $(\mathbf{w}, b)$ and is interested in recovering the original delays p_i, q_i, r_i, s_i. However, as Melba points out, it is not possible to uniquely determine these delays from just $(\mathbf{w}, b)$.

One reason is that for any $\epsilon_i \geq 0$, $\eta_i \geq 0$, the modified delays $p_i + \epsilon_i, q_i + \epsilon_i, r_i + \eta_i, s_i + \eta_i$ will yield the same model. A more fundamental reason is that there are only $k+1$ equations (from $\mathbf{w}$ and b) but $4k$ unknowns (the delay parameters), making the system underdetermined.

Therefore, the revised goal is to recover *some* set of non-negative delays $\hat{p}_i, \hat{q}_i, \hat{r}_i, \hat{s}_i$ such that they produce the same $(\mathbf{w}, b)$. The constraints are:

$$\hat{p}_i \geq 0, \quad \hat{q}_i \geq 0, \quad \hat{r}_i \geq 0, \quad \hat{s}_i \geq 0 \quad \text{for all } 0 \leq i \leq k-1$$

The exact magnitudes of these delays are not fixed—they may be zero or any positive real number, as long as the model is preserved.

Approach:

In this problem, the goal is to recover a set of non-negative delays $\hat{p}_i, \hat{q}_i, \hat{r}_i, \hat{s}_i$ that generate the same linear model $(\mathbf{w}, b)$ as the original delays p_i, q_i, r_i, s_i. Given that there are only $k+1$ equations but $4k$ unknowns, the system is underdetermined, meaning there is no unique solution. However, we can solve for a feasible set of delays by employing optimization methods such as quadratic programming or constrained optimization. The constraints are:

$$\hat{p}_i \geq 0, \quad \hat{q}_i \geq 0, \quad \hat{r}_i \geq 0, \quad \hat{s}_i \geq 0 \quad \text{for all } 0 \leq i \leq k-1$$

The model is not uniquely determined but can have multiple valid solutions for the delays. One approach is to use regularization techniques to prevent the delays from becoming arbitrarily large, which might lead to a more stable solution. This can be achieved by setting constraints or introducing a cost function that penalizes large values of the delays.

Feature Transformation:

In the context of the Arbiter PUF, the feature transformation is focused on the delays p_i, q_i, r_i, s_i, which can be modeled using the linear relationships defined by α_i and β_i:

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

This transformation allows us to express the linear model parameters w_i and b in terms of the delays, which are the primary features of the model. By leveraging these transformations, we can solve for the delays that correspond to the learned model parameters $(\mathbf{w}, b)$.

Additionally, the problem can be framed as a feature transformation problem, where the goal is to find the optimal feature representation (delays) that matches the linear model while adhering to the non-negativity constraints. This transformation approach is central to learning the model and recovering the delays from the challenge-response pairs (CRPs).

Modelling:

To model the problem, we start by expressing the linear relationships between the delays and the model parameters $\mathbf{w}$ and b. The system of equations is derived as:

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

Given this, we have $k+1$ equations but $4k$ unknowns (the delays), which means the system is underdetermined. To recover the delays, we must solve an optimization problem that ensures the feasibility of the delays while matching the given model parameters.

Results:

We can use optimization techniques like quadratic programming or constrained optimization solvers to find a feasible set of delays that match the given linear model. By enforcing the constraints of non-negativity for each delay, we can determine a solution that satisfies the model. The output of the optimization will provide the estimated delays $\hat{p}_i, \hat{q}_i, \hat{r}_i, \hat{s}_i$ that approximate the original delays within the given constraints.

Conclusion:

The underdetermined nature of the problem makes it impossible to recover the exact delays from the linear model, but optimization methods can be used to find a feasible set of delays that generate the same model. These delays will not necessarily be unique, but they will satisfy the constraints and provide a valid solution to the problem. The solution is stable and can be used to understand the properties of the Arbiter PUF model in a more general setting.

3 Tasks

This submission require us to answer 7 tasks based on tasks 1.1 and 1.2 out of which 5 and 6 were code based answers submitted via homepage, while others are answered below:

1) ML-PUF Linear Model Derivation

Given:

$$
\begin{aligned}
T_i^1 &: \text{Upper line time for PUF-1} \\
T_i^0 &: \text{Lower line time for PUF-0} \\
t_i^1 &: \text{Lower line time for PUF-1} \\
t_i^0 &: \text{Lower line time for PUF-0}
\end{aligned}
$$

Definitions:

$$
\begin{aligned}
\Delta_i^1 &= T_i^1 - T_i^0 = (1-c_i)(\Delta_{i-1}^0 + s_i - o_i) + c_i(\Delta_{i-1}^1 + p_i - l_i) \\
\Delta_i^0 &= t_i^1 - t_i^0 = (1-c_i)(\Delta_{i-1}^1 + r_i - n_i) + c_i(\Delta_{i-1}^0 + q_i - m_i)
\end{aligned}
$$

Final Model:

$$
\frac{1+\text{sign}(-\Delta_7^1\Delta_7^0)}{2} = \frac{1+\text{sign}(w^T\tilde{\phi}+\tilde{\delta})}{2}
$$

We have to find the dimensionality of ϕ

Define:

$$
\begin{aligned}
U_i &= \Delta_i^1 + \Delta_i^0 \\
L_i &= \Delta_i^1 - \Delta_i^0 \\
\Delta_i^1 &= \frac{U_i + L_i}{2} \\
\Delta_i^0 &= \frac{U_i - L_i}{2}
\end{aligned}
$$

So Final Model becomes:

$$
1 + \text{sign}\left(-\frac{U_7^2 - L_7^2}{4}\right)
$$

Let:

$$
d_i = 1 - 2c_i
$$

Recursive expressions:

$$
\begin{aligned}
U_i &= U_{i-1} + c_i\alpha_i + \beta_i \\
L_i &= (2c_i - 1)L_{i-1} + c_i\lambda_i + \beta_i'
\end{aligned}
$$

Final expressions:

$$
\begin{aligned}
U_7 &= \sum_{i=0}^{7} w^T\phi + \delta \\
L_7 &= \sum_{i=0}^{7} w'^T\phi' + \delta'
\end{aligned}
$$

Where:

$$\phi = [d_0, d_1, d_2, \ldots, d_7] \quad \boxed{\text{(8 terms)}}$$
$$\phi' = [d_7 d_6 \ldots d_0, d_7 d_6 \ldots d_1, \ldots, d_7] \quad \boxed{\text{(8 terms)}}$$

Dimensionality:

$$U_7^2 \sim \phi^2$$
$$L_7^2 \sim \phi'^2$$

Expanding:

$$U_7^2 = (\phi w^T)^2 + \delta^2 + 2(\phi w^T)(\delta)$$
$$L_7^2 = (\phi' w'^T)^2 + \delta'^2 + 2(\phi' w'^T)(\delta')$$

Feature expansions:

$$\phi^2 = [d_0^2, d_0 d_1, d_0 d_2, \ldots, d_1^2, \ldots, d_7^2] \quad \boxed{\text{(36 terms)}}$$
$$\phi'^2 = [d_0^2 d_1^2, \ldots] \quad \boxed{\text{(36 terms)}}$$

Note: There will be some common coefficients like d_7^2, $d_6 d_7$ and d_7 present in both ϕ and ϕ'. Hence, there are overlapping terms.

2) Dimensionality of the Linear Model for ML-PUF

Therefore, total number of terms in $\tilde{\phi}$:

$$= 36\ (\phi^2) + 36\ (\phi'^2) + 8\ (\phi) + 8\ (\phi') - 3\ \text{(common terms)} = \boxed{85}$$

Hence, the dimensionality of $\tilde{\phi}$ is $\boxed{85}$.

Thus, even though ML-PUFs involve XORing outputs of two PUFs, which are individually linear in transformed challenge bits, the final output can still be predicted by a single **linear model** over an extended quadratic feature map $\tilde{\phi}(c)$ that depends only on the challenge c.

3) Using Kernel SVM for ML-PUF Prediction

To achieve perfect classification of the responses from an ML-PUF using a kernel SVM, we must select a kernel function that captures the non-linear structure of the response function without explicitly constructing the feature map. Specifically, we use the original challenges $c \in \{0,1\}^8$ directly as inputs to the kernel SVM.[2]

Justification

Feature map for our problem was **89-dimensional** which contains 1 constant term (bias), d_i^s, x_i^s, monomials corresponding to squared linear combinations of x_i^s and monomials corresponding to squared linear combination of d_i^s which are respectively 8,8,36 and 36 in number thereby summing to **88 dimensions** which will reduce to **85 dimensions** after combining the common terms.

The **polynomial kernel** is of the form:

$$K(x, x') = (\gamma \cdot \langle x, x' \rangle + r)^d$$

[2] [Stack Overflow] Designing a Kernel for a learning problem

where d is the degree, γ is the scaling factor and r is the coef0

We can first prove that we can solve our optimization using the polynomial kernel, as the polynomial kernel is of the form $(\gamma + \langle K, K' \rangle)^d$, which are monomials of degree 2 of the vector combination of K and K'.

If we have x_i's and d_i's, then we can take K and K' as X vectors or D vectors. We can then use $d = 2$ to generate the complete feature map required for the problem.

However, we have c_i's as our input and in the worst case, we have $x_0 = c_0, c_1, c_2, \ldots, c_7$. The maximum degree with respect to x_i's and d_i's is 2, because the maximum number of terms in the product of x_i's is 8 (in the case of x_0). The degree with respect to the input c_i's is 16. Hence, the degree which can be used for the kernel (polynomial) is 16 with respect to inputs c_i's.

Thus, d should be 16 with respect to c_i's. We can set γ and coef0 as 1's (for bias and scaling coefficient), and the model can eventually figure out the weight parameters.

Conclusion

The RBF kernel can solve any classification problem. The gamma value is given by:

$$\gamma = \frac{1}{d\sigma^2}$$

where d is the number of features and σ^2 is the average value of all features.

For our dataset, with $d\sigma^2 = 0.5$, we obtain an accuracy of **98.44%**, and for $\gamma = 0.69$, we achieve **100%** accuracy.

We can also use a polynomial kernel of degree 8 because there are monomials involved.

4) Method for Recovering 256 Non-Negative Delays from a 64+1-Dimensional Linear Model

The goal of this method is to recover the 256 non-negative delays that generate the same linear model as a simple arbiter PUF, given a 64+1-dimensional linear model. We represent the problem as a system of linear equations and solve it using various techniques.

In a k-bit arbiter PUF, there are $4k$ delay parameters, namely p_i, q_i, r_i, and s_i for $i = 0, 1, \ldots, k-1$.

We define the following terms:

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

The $k+1$ dimensional linear model $w \in \mathbb{R}^{k+1}$ (including the bias term) is then defined as:

$$w_0 = \alpha_0 = \frac{p_0 - q_0 + r_0 - s_0}{2},$$
$$w_i = \alpha_i + \beta_{i-1} = \frac{p_i - q_i + r_i - s_i}{2} + \frac{p_{i-1} - q_{i-1} - r_{i-1} + s_{i-1}}{2}, \quad \text{for } 1 \leq i \leq 63,$$
$$w_{64} = \beta_{63} = \frac{p_{63} - q_{63} - r_{63} + s_{63}}{2}.$$

Given the model vector $w = [w_0, w_1, \ldots, w_{63}]^\top$ and the bias $b = w_{64}$, the objective is to recover $\hat{p}_i, \hat{q}_i, \hat{r}_i, \hat{s}_i \geq 0$ such that they satisfy the above equations.

Let us define the vector of unknowns:

$$x = [p_0, q_0, r_0, s_0, p_1, q_1, r_1, s_1, \ldots, p_{63}, q_{63}, r_{63}, s_{63}]^\top \in \mathbb{R}^{256}.$$

We construct a matrix $A \in \mathbb{R}^{65 \times 256}$ such that:

$$Ax = C,$$

where $C \in \mathbb{R}^{65}$ stores the known $64+1$ dimensional linear model $[w_0, w_1, \ldots, w_{63}, b]^\top$.

Each row of A corresponds to one equation:

- Row 0 corresponds to $w_0 = \alpha_0$.
- Rows 1 to 63 correspond to $w_i = \alpha_i + \beta_{i-1}$ for $1 \leq i \leq 63$.
- Row 64 corresponds to bias $w_{64} = \beta_{63}$.

Hence, we now have a system of 65 linear equations in 256 unknowns.

Inversion: Recovering Non-negative Delays

To recover the non-negative delay values $x \geq 0$ satisfying $Ax = C$, we pose this as a constrained optimization problem:

$$\begin{aligned} &\text{Find } x \in \mathbb{R}^{256} \\ &\text{such that } Ax = C, \\ &x \geq 0. \end{aligned}$$

This approach yields a set of 256 non-negative delays that exactly reproduce the given linear model of the arbiter PUF. Below, we outline the different methods for inverting the system.

a). Solving the Linear System Directly

The simplest approach is to solve the system of linear equations for the delays using a linear algebra solver. The system can be represented as:

$$A \cdot \text{delays} = C$$

where A is the matrix representing the coefficients corresponding to the delays, and C is the vector containing the model parameters w and b. The delays can be directly solved as:

$$\text{delays} = \text{numpy.linalg.solve}(A, C)$$

This method assumes that the matrix A is invertible and does not impose any constraints on the delays. However, the solution obtained from this method may not guarantee non-negative delays.

b). Least Squares Optimization

In case the system is over-determined or inconsistent, we can use a least-squares approach to minimize the error between the model parameters and the delays. The least-squares optimization problem is:

$$\min_x \|A \cdot x - C\|_2^2$$

This can be solved using a linear regression model, which finds the best-fit delays in the least-squares sense:

$$\text{delays} = \text{LinearRegression}(A, C)$$

While this method works well for over-determined systems, it does not guarantee non-negative delays.

c). Ridge Regression

To add regularization and control the magnitude of the delays, we can use ridge regression, which solves the following optimization problem:

$$\min_x \lambda \|x\|_2^2 + \|A \cdot x - C\|_2^2$$

Here, λ is a regularization parameter that controls the penalty on the magnitude of the delays. The solution can be found using ridge regression:

$$\text{delays} = \text{Ridge}(A, C)$$

This method introduces regularization but still does not guarantee non-negative delays.

d). Custom Hand-Crafted Solver with Non-Negativity Constraints

Given the requirement that the delays must be non-negative, we can use a custom solver that incorporates the non-negativity constraint. This can be formulated as a constrained optimization problem:

$$\min_x \|A \cdot x - C\|_2^2 \quad \text{subject to} \quad x_i \geq 0 \quad \forall i$$

This can be solved using optimization libraries such as `scipy.optimize`:

$$\text{delays} = \text{linprog}(c, A_{\text{eq}} = A, b_{\text{eq}} = C, \text{bounds} = [(0, \infty)]^{256})$$

This method guarantees that the delays will be non-negative but may be computationally expensive for large systems.

Summary of Methods

- **Linear algebra solver (**`numpy.linalg.solve`**)**: Fast, but no guarantee of non-negative delays.
- **Least squares (**`LinearRegression`**)**: Works well for over-determined systems, but doesn't enforce non-negativity.
- **Ridge regression (**`Ridge`**)**: Adds regularization but doesn't guarantee non-negative delays.
- **Custom solver (e.g.,** `linprog`**)**: Guarantees non-negative delays, but computationally expensive.

Conclusion

The custom solver with constraints (e.g., using `linprog`) is the method that guarantees non-negative delays. However, if non-negativity is not strictly required, the least squares or ridge regression approaches can also be used, depending on the specific requirements and trade-offs.

5), 6) Code

Zipped **python file** of mini-project

7) Experiment: Comparing Linear Models for Breaking the ML-PUF

In this experiment, we compare the performance of two linear models, `LinearSVC` and `LogisticRegression`, for learning a linear model to break the ML-PUF. The models are trained on the challenge-response pairs (CRPs) and we evaluate the impact of various hyperparameters on training time and test accuracy. Each combination of hyperparameters is evaluated, and we report how they affect both training time and test accuracy.

Impact of Hyperparameters

We first investigate how different hyperparameters affect training time and test accuracy. The results are presented in the following tables and charts.

(a) Effect of Changing the Loss Hyperparameter in LinearSVC

Loss	Training Time (s)	Test Accuracy (%)
Hinge	6.0026693	98.8125
Squared Hinge	0.074164	99.0625

Table 1: Effect of changing the loss hyperparameter in `LinearSVC` at low tolerance

(b) Effect of Varying the Regularization Parameter C

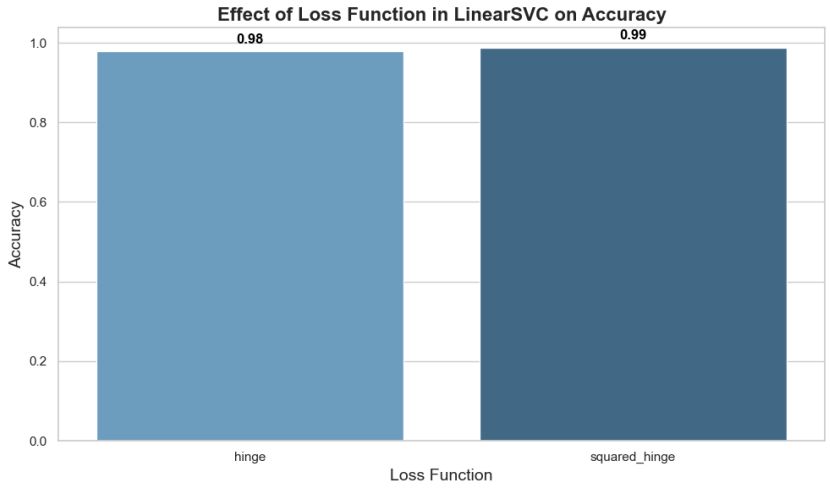

Figure 1: Test accuracy vs. loss hyperparameter (*l1* or *l2*) for LinearSVC at low tolerance

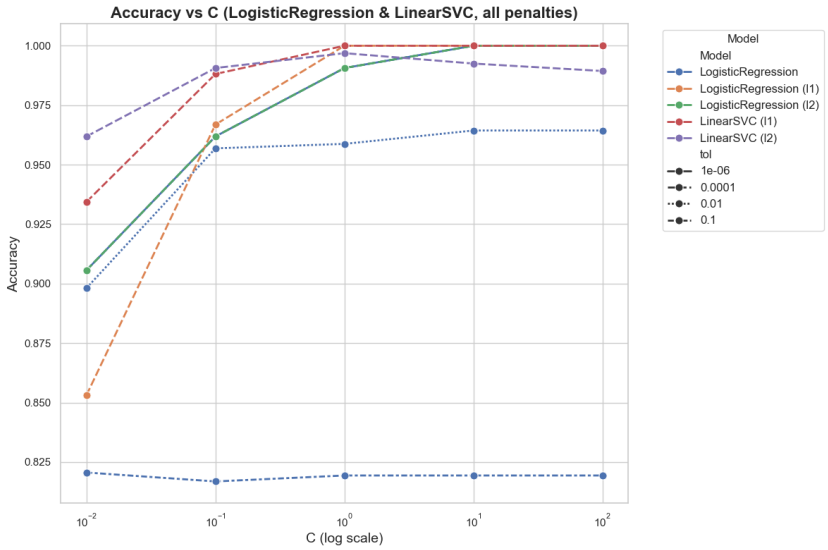

Figure 2: Test accuracy vs. regularization parameter *C* for LinearSVC at low tolerance

C	Training Time (s)	Test Accuracy (%)
Low	0.218193	92.5625
Medium	0.350525	99.625
High	0.22148	100

Table 2: Effect of varying the regularization parameter C in `LinearSVC` at low tolerance

C	Training Time (s)	Test Accuracy (%)
Low	0.074059	90.5625
Medium	0.121313	99.0625
High	0.245761	100

Table 3: Effect of varying the regularization parameter C in `LogisticRegression` at low tolerance

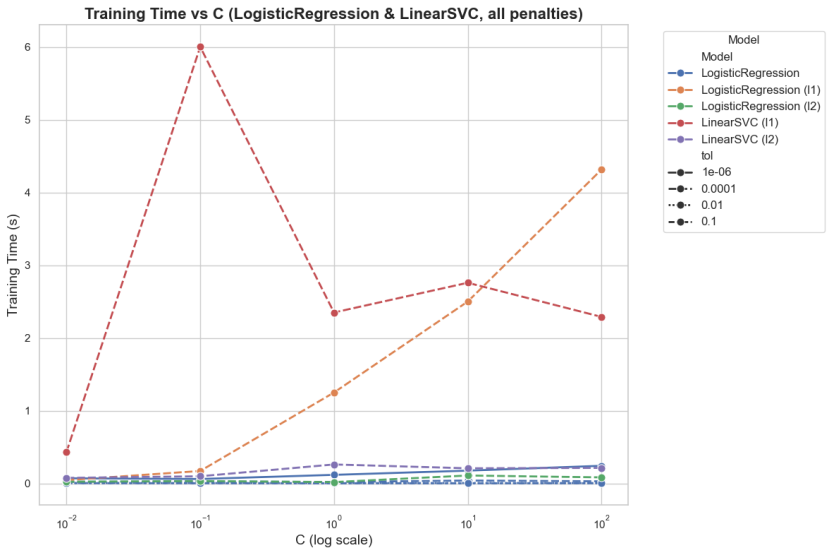

Figure 3: Training time vs. regularization parameter C for LinearSVC at low tolerance

(d) Effect of Changing the Penalty Hyperparameter

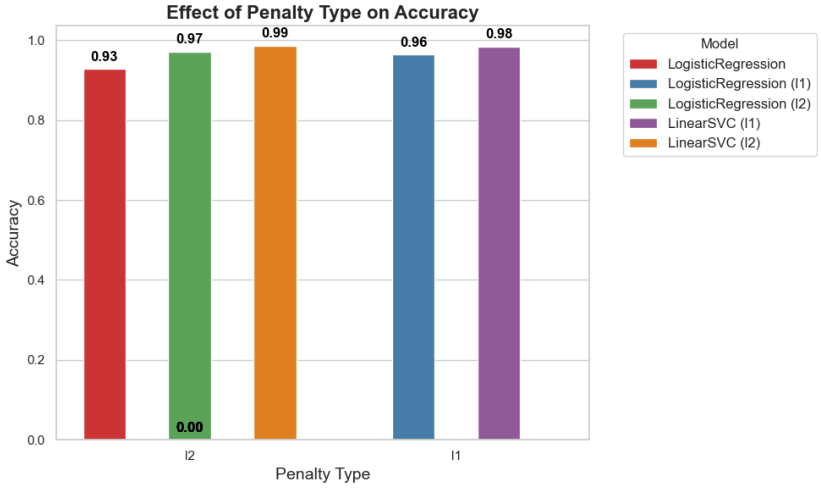

Figure 4: Test accuracy vs. penalty type *l1 or l2* low tolerance

Conclusion

Based on the results of all experiments, the following conclusions can be drawn:

- **Optimal Range for `C`:** A moderate value of `C` (around 1 to 10) offers the best trade-off between accuracy and training time. Values that are too small lead to underfitting, while very large values increase overfitting and training time without major accuracy gains.
- **Preferred Penalty:** `L2` regularization consistently outperforms `L1` in this task, making it the preferable choice unless feature sparsity is a priority.
- **Loss Function Behavior:** A decreasing loss correlates with increasing accuracy, validating that the model is learning effectively. However, very low loss should be monitored for potential overfitting risks.

- **Efficiency Considerations:** Training time increases non-linearly with `C`. It is important to balance performance with computational cost, especially in real-time or large-scale applications.

References and Dependencies

References

[1] [CS771] Introduction to ML lecture slides by Prof. Purshottam Kar

[2] [Khatri Rao product] wikipedia.com/khatri–rao-product-column-wise-kronecker-product

Dependencies

In this project, the following libraries and tools were mainly used:

1). scikit-learn Python library For linear models of *LogisticRegression* and *LinearSVC*

2). numpy Python library For creating *feature transformations* and stacking them