Mini-Project 1: CS973 ML for CyberSecurity
Group: ML Panchayat

Honey Das
233560013
honeyd23@iitk.ac.in

Mohammed Jawed
233560019
mjawed23@iitk.ac.in

Nikita Shetty
233560023
narayans23@iitk.ac.in

Ritika Chaurasia
233560029
ritikac23@iitk.ac.in

S Sathya Kumar
23157064
subbarao23@iitk.ac.in

1 Solution to Question 1:

For a Ring Oscillator(RO), the O/P is connected back as input to the first NOT gate. Therefore, after 5 inversions, O/P will flip. This means that if the value of O/P is 0 at t_0, then it will flip to 1 after $t_0 = \delta_0^0 + \delta_1^1 + \delta_2^0 + \delta_3^1 + \delta_4^0$ seconds. Similarly, it will flip back to 0 after $t_1 = \delta_0^1 + \delta_1^0 + \delta_2^1 + \delta_3^0 + \delta_4^1$ seconds.

Consider a simple XORRO PUF with two XORROs. Each of the XORROs has R config bits as input. The simple XORRO PUF returns a 1, if the upper XORRO has a higher frequency and returns a 0, if the lower XORRO has a higher frequency. The frequency of a XORRO is defined as

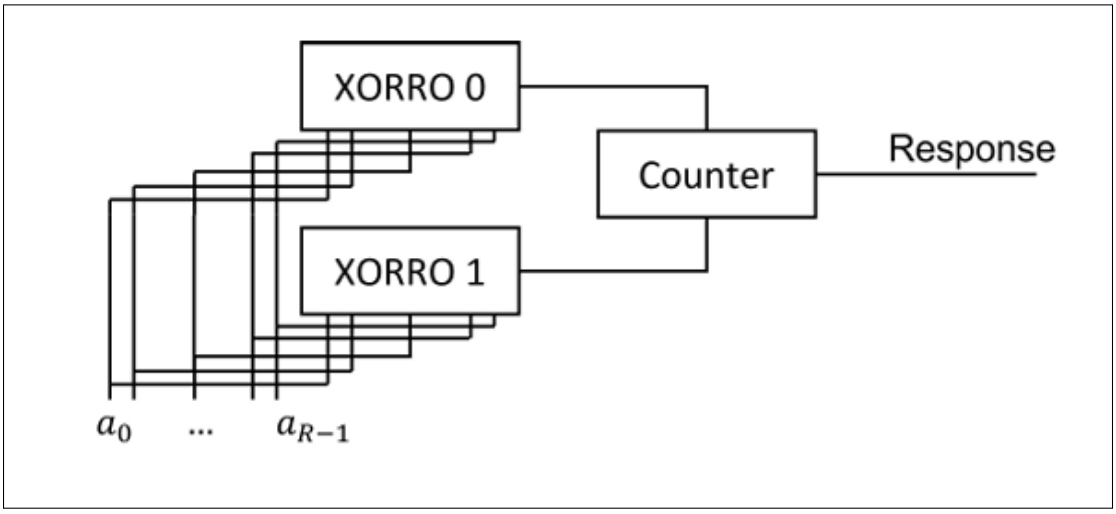

Figure 1: A Simple XORRO PUF using 2 XORROs

Preprint. Under review.

$f = \frac{1}{t_0+t_1}$

Therefore, frequencies of $XORRO_0$ and $XORRO_1$ can be respectively written as:

$f_0 = \frac{1}{t_0^0+t_1^0} \qquad and \qquad f_1 = \frac{1}{t_0^1+t_1^1}$

Consider $\delta_{00}, \delta_{01}, \delta_{10}, \delta_{11}$ are the delays based on input and configuration bits.

Therefore, total time t_0 and t_1 for a $XORRO$ will be:

$t_0 = \delta^0_{0a0} + \delta^0_{1a_0} + \delta^0_{0a_1} + \delta^0_{1a_1} + ... + \delta^0_{0a_{63}} + \delta^0_{1a_{63}}$ and

$t_1 = \delta^1_{0a0} + \delta^1_{1a_0} + \delta^1_{0a_1} + \delta^1_{1a_1} + ... + \delta^1_{0a_{63}} + \delta^1_{1a_{63}}$

where, $\delta^0_{0a_0}$ is the delay for input 0 and configuration bit a_0
$\delta^0_{1a_0}$ is the delay for input 1 and configuration bit a_0 and so on.

Sum of delays t_0 and t_1 for $i^{th} XORRO$ can be respectively written as:

$t_0^i = \sum_{j=0}^{63} \delta^{i,j}_{0a_j} \qquad and \qquad t_1^i = \sum_{j=0}^{63} \delta^{i,j}_{1a_j}$

Generalizing for R config bits: $t_0^i = \sum_{j=0}^{R-1} \delta^{i,j}_{0a_j} \qquad and \qquad t_1^i = \sum_{j=0}^{R-1} \delta^{i,j}_{1a_j}$

where $i \in \{0, 1\}$ and $j \in \{0, R-1\}$

Total time delay for $XORRO_0$ can be therefore written as:

$t_0^0 + t_1^0 = \sum_{j=0}^{R-1} (\delta^{0,j}_{0a_j} + \delta^{0,j}_{1a_j})$

Total time delay for $XORRO_1$ can be therefore written as:

$t_0^1 + t_1^1 = \sum_{j=0}^{R-1} (\delta^{1,j}_{0a_j} + \delta^{1,j}_{1a_j})$

Consider Δ as the difference of time periods between $XORRO_1$ and $XORRO_0$:

$\Delta = (t_0^1 + t_1^1) - (t_0^0 + t_1^0) = \frac{1}{f_1} - \frac{1}{f_0}$

$\therefore$ This will give us output y of the form

$y = \frac{1+\text{sign}(\triangle)}{2}$

Since, if $f_0 > f_1 \; then \; \triangle > 0$ and if $f_1 > f_0 \; then \; \triangle < 0$

Substituting $t_0^1, t_1^1, t_0^0, t_1^0$ from above we get:

$\Delta = \sum_{j=0}^{63} (\delta^{1,j}_{0a_j} + \delta^{1,j}_{1a_j}) - \sum_{j=0}^{R-1} (\delta^{0,j}_{0a_j} + \delta^{0,j}_{1a_j})$
We see that:

$\delta^0_{0a_0} = a_0(\delta^0_{01}) + (1 - a_0)(\delta^0_{00})$

$\delta^0_{1a_0} = a_0(\delta^0_{11}) + (1 - a_0)(\delta^0_{10})$

$\delta^0_{0a_0} + \delta^0_{1a_0} = a_0(\delta^0_{01} + \delta^0_{11}) + (1 - a_0)(\delta^0_{00} + \delta^0_{10})$

$\delta^1_{0a_0} + \delta^1_{1a_0} = a_0(\delta^1_{01} + \delta^1_{11}) + (1 - a_0)(\delta^1_{00} + \delta^1_{10})$

This gives us a pattern:

$\sum_{j=0}^{R-1}(\delta^0_{0a_j} + \delta^0_{1a_j}) = \sum_{j=0}^{R-1}((a_j)(\delta^0_{01} + \delta^0_{11}) + (1 - a_j)(\delta^0_{00} + \delta^0_{10}))$

Similarly, we have

$\sum_{j=0}^{R-1}(\delta^1_{0a_j} + \delta^1_{1a_j}) = \sum_{j=0}^{R-1}((a_j)(\delta^1_{01} + \delta^1_{11}) + (1 - a_j)(\delta^1_{00} + \delta^1_{10}))$

Now, we can write:

$\Delta = \sum_{j=0}^{R-1}(a_j)(\delta^1_{01} + \delta^1_{11}) + (1 - a_j)(\delta^1_{00} + \delta^1_{10})) - (a_j)(\delta^0_{01} + \delta^0_{11}) - (1 - a_j)(\delta^0_{00} + \delta^0_{10})$

$\Delta = \sum_{j=0}^{R-1}(a_j)(\delta^1_{01} + \delta^1_{11} - \delta^0_{01} - \delta^0_{11}) + (1 - a_j)(\delta^1_{00} + \delta^1_{10} - \delta^0_{00} - \delta^0_{10})$

$\Delta = \sum_{j=0}^{R-1}(a_j)(\delta^1_{01} + \delta^1_{11} - \delta^1_{00} - \delta^1_{10} + \delta^0_{00} + \delta^0_{10} - \delta^0_{01} - \delta^0_{11}) + (\delta^1_{00} + \delta^1_{10} - \delta^0_{00} - \delta^0_{10})$

This can be written of the form

$\Delta j = \sum_{j=0}^{R-1}(a_j)(\alpha_j) + (\beta_j)$

$\Delta j = \Delta_{j-1} + (a_j)(\alpha_j) + (\beta_j)$

$\Delta_0 = (a_0)(\alpha_0) + (\beta_0)$

$\Delta_1 = (a_1)(\alpha_1) + (\beta_1) + (a_0)(\alpha_0) + (\beta_0)$
$\Delta_2 = (a_2)(\alpha_2) + (\beta_2) + (a_1)(\alpha_1) + (\beta_1) + (a_0)(\alpha_0) + (\beta_0)$
:
:
$\Delta_{63} = (a_{63})(\alpha_{63}) + (\beta_{63}) + ... + (a_2)(\alpha_2) + (\beta_2) + (a_1)(\alpha_1) + (\beta_1) + (a_0)(\alpha_0) + (\beta_0)$

In case of 64 config bits:
if $\Delta_{63} < 0$, then the output is 1
if $\Delta_{63} > 0$, then the output is 0

For R_{th} response

$\Delta_R = (W^T)(x) + (b_R)$
$where \quad x_j = a_j \quad and \quad W_j = x_j$

$$\text{W} = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ : \\ \alpha_{R-1} \end{bmatrix} \quad x = \begin{bmatrix} a_0 \\ a_1 \\ : \\ a_{R-1} \end{bmatrix} \quad \text{b} = \begin{bmatrix} \beta_0 \\ \beta_0 + \beta_1 \\ : \\ \beta_0 + \beta_1 + \beta_{R-1} \end{bmatrix}$$

$\therefore y = (1 + sign(\Delta))/2$

or,

$y = (1 + sign(W^T\phi(c) + b))/2$

where $\phi(c) = x$

Thus, a simple XORRO PUF can be cracked using the above linear model.

2 Solution to Question 2:

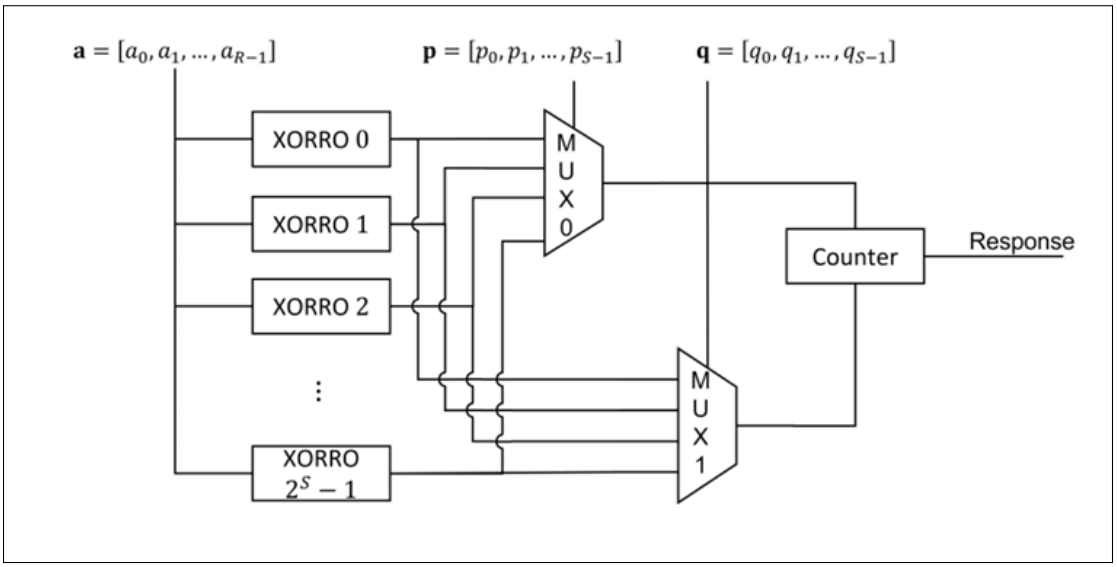

Figure 2: An Advanced XORRO PUF using 2^S XORROs

In case of the advanced XORRO PUF, we have two multiplexers that chooses the 2 XORROs. For each of these XORROs, we shall use the linear model from the previous question to solve for each XORRO.

Possible ways of selecting XORROs from the first half = 2^S

Hence possible ways of selecting XORROs from the first half = $2^S - 1$

As first S bits are not same as the second S bits,

Total possible combinations = $2^S(2^S - 1)$

Therefore, number of models will be $2^S(2^S - 1)$

For MUX_0, the select bits p = $[p_0 p_1 p_2 ... p_{S-1}]$

For MUX_1, the select bits q = $[q_0 q_1 q_2 ... q_{S-1}]$

We know that time difference can be written as:

$\Delta = (W^T)(x) + (b)$

Therefore, time difference $p_{th}XORRO$ selected by MUX_0 and $q_{th}XORRO$ selected by MUX_1 will be:

$\Delta_{p,q} = W_{p,q}^T x + b_{p,q}$

where

$$W_{p,q}^T = \begin{bmatrix} \alpha_0^{p,q} \\ \alpha_1^{p,q} \\ \vdots \\ \alpha_{R-1}^{p,q} \end{bmatrix} \quad b_{p,q} = \begin{bmatrix} \beta_0^{p,q} \\ \beta_0^{p,q} + \beta_1^{p,q} \\ \vdots \\ \beta_0^{p,q} + \beta_1^{p,q} + \beta_{R-1}^{p,q} \end{bmatrix}$$

And response

$y = (1 + sign(W_{p,q}^T x + b_{p,q}))/2$

From Solution to Question 1 above, we can now write:

$\alpha_j^{p,q} = \delta_{01}^q + \delta_{11}^q - \delta_{00}^q - \delta_{10}^q + \delta_{00}^p + \delta_{10}^p - \delta_{01}^p - \delta_{11}^p$
and

$\alpha_j^{q,p} = \delta_{01}^p + \delta_{11}^p - \delta_{00}^p - \delta_{10}^p + \delta_{00}^q + \delta_{10}^q - \delta_{01}^q - \delta_{11}^q$
This shows that $\alpha_j^{p,q} = -\alpha_j^{q,p}$

$\therefore W_{p,q}^T = -W_{p,q}^T$

Similarly,

$\beta_j^{p,q} = \delta_{00}^q + \delta_{10}^q - \delta_{00}^p - \delta_{10}^p$

$\beta_j^{q,p} = \delta_{00}^p + \delta_{10}^p - \delta_{00}^q - \delta_{10}^q$
This shows that $\beta_j^{pq} = -\beta_j^{qp}$

$\therefore b_{pq} = -b_{pq}$

$\Delta_{p,q} = W_{p,q}^T x + b_{p,q}$

$= (-1)(-W_{p,q}^T x - b_{p,q})$

$= -\Delta_{q,p}$
For $p > q$, $y = (1 + sign(W_{pq}^T x + b_{pq}))/2$
For $q > p$, $y = (1 + sign(-(W_{pq}^T x + b_{pq})))/2$

$\therefore$ Only half of the models are needed

$\implies M = 2^S(2^S - 1)/2 = s^{S-1}(2^S - 1)$

3 Solution to Question 3:

3.1 Language and Libraries

Q3 is written in Python(submit.py) and utilizes the scikit-learn library for machine learning. The same has been validated using provided Google Colab script. The specific implementation employs Logistic Regression with the configuration:

$LogisticRegression(max\_iter = 10000, solver =' liblinear', C = 10, tol = 1e - 4, penalty =' l2')$

3.2 Hyperparameters

The hyperparameters used include:

- max_iter=10000: Allows the optimization algorithm to run for a maximum of 10,000 iterations, ensuring adequate time for convergence.
- solver='liblinear': Suitable for small datasets and efficient for logistic regression tasks.
- C=10: Provides moderate regularization, balancing model complexity and fitting the training data.
- tol=1e-4: Tolerance level for convergence that helps determine when to stop the optimization process.
- penalty='l2': Indicates the use of L2 regularization to discourage overly complex models.

3.3 Summary of Methods

3.3.1 my_fit()

The my_fit() method processes the Challenge-Response Pairs (CRPs) for the Advanced XORRO PUF, extracting relevant selection bits to generate unique model keys. It groups the training data by these keys and trains a Logistic Regression model for each configuration of selected XORROs, ensuring a total of M = 120 models are learned based on the number of selection bits.

3.3.2 my_fit() flow control

1. Start
2. Initialize `xor_data_map` as an empty dictionary.
3. For each row in `Z_train`:
   - Calculate `x` and `y` from selection bits.
   - Compute `model_key = 16 * min(x, y) + max(x, y)`. [Each pair of XORRO selections (identified by x and y) needs a unique identifier for indexing into a collection of trained models. This allows the program to correctly access the model corresponding to any given selection of XORROs. Given that both x and y can take values from 0 to 15, the keys generated will range from 0 to 255 (i.e., 16×15+15), allowing for efficient mapping of configurations to model indices.
   - If `model_key` is not in `xor_data_map`: Initialize `xor_data_map[model_key]` as an empty list.
   - Create a copy of the current row.
   - If `x > y`, flip the response (last element of the row).[a data preprocessing step aimed at maintaining consistent labeling of the output responses based on the selected XORROs. This helps prevent confusion during model training by ensuring that the model learns from consistently defined input-output mappings.]
   - Append the modified row to `xor_data_map[model_key]`.
4. Convert lists in `xor_data_map` to NumPy arrays.
5. Initialize `trained_models` as an empty dictionary.
6. For each combination of `i` and `j` (where `i < j` from 0 to 15):
   - Compute `model_key = 16 * i + j`.
   - If `model_key` is in `xor_data_map`:
     - Initialize a `LinearSVC` model.
     - Fit the model on the data associated with `model_key`.
     - Store the trained model in `trained_models` using `model_key`.
7. Return `trained_models`.
8. End

3.3.3 my_predict()

The my_predict() method takes the test data and utilizes the trained models from my_fit() to predict responses based on the provided challenges. It generates the corresponding model keys for the test inputs, retrieves the appropriate trained model, and returns the predicted outputs, ensuring accurate classification based on the learned configurations.

3.3.4 my_predict() flow control

1. Start
2. Initialize `X_pred` as a zero array of length equal to the number of rows in `X_tst`.
3. For each row in `X_tst`:
   - Calculate `x` and `y` from selection bits.
   - Compute `model_key = 16 * min(x, y) + max(x, y)`.
   - If `model_key` is not in `models`: Continue to the next iteration.
   - Reshape the first 64 features of the current row for prediction.
   - Use the corresponding model to predict the response based on the reshaped features.
   - If `x > y`, flip the prediction.
   - Store the prediction in `X_pred`.
4. Return `X_pred`.
5. End

3.4 Results

The following are the results we obtained after using the optimized Logistic Regression classifier model with the best-chosen hyperparameters on the provided Google Colab script.

	Train Time (s)	Test Time (s)	Model Size (bytes)	Accuracy (%)
secret_test.dat secret_train.dat	0.93	5.92	88156	94.87
test.dat and train.dat	0.91	6.02	88156	94.84

4 Solution to Question 4: Reporting the Outcome of Various Hyperparameters

In this section, we report the outcomes of the effect of various hyperparameters by comparing the performance of LinearSVC and Logistic Regression. Below, we include a data table of hyperparameters along with corresponding charts for clear visualization.

During the experiments with various hyperparameters, we chose to vary only the specific hyperparameter value under test while keeping all other hyperparameters at their default values (not explicitly mentioned in this report).

We also experimented with RidgeClassifier using various hyperparameters to compare its performance with LinearSVC and Logistic Regression. However, we chose not to include those results in the report, as it would be too cumbersome for the data table and chart. Also, we observed the training time was longer and the model had less accuracy for this method.

So for clarity, we focused solely on the results from LinearSVC and Logistic Regression.

Notably, for Logistic Regression with the hyperparameter penalty='l1', we specified solver='liblinear' as the default since the default solver solver='lbfgs' does not support the l1 penalty.

4.1 Tables and Graphs

4.1.1 4a. Changing the loss hyperparameter in LinearSVC (hinge vs squared hinge)

Table 1: Loss Hyperparameter

Loss Hyperparameter	Training Time (seconds)	Accuracy (%)
hinge	7.18	93.97
squared hinge	1.39	94.75

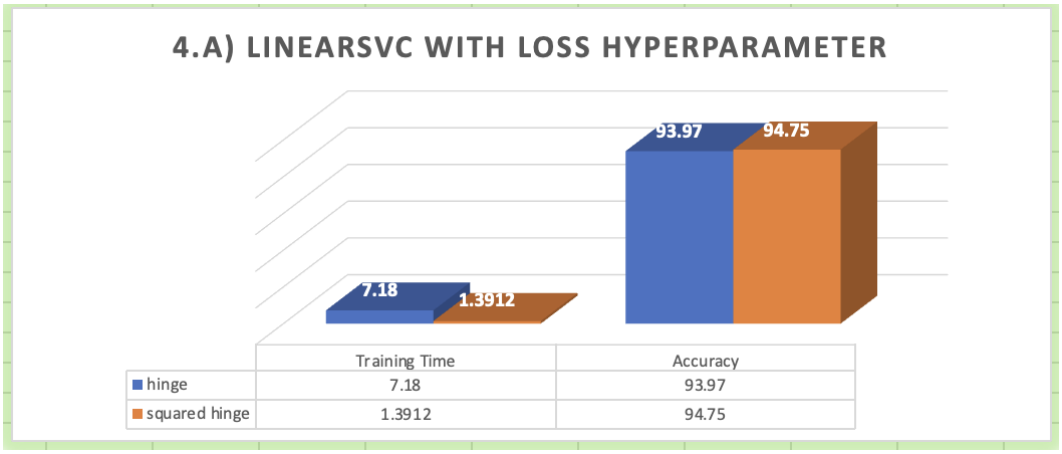

Figure 3: LinearSVC with Loss Hyperparameter

4.1.2 4b. setting C hyperparameter in LinearSVC and LogisticRegression to high/low/medium values)

Table 2: C Hyperparameter

C Hyperparameter	Training Time (seconds)		Accuracy (%)	
	LinearSVC	Logistic Regression	LinearSVC	Logistic Regression
Low (C=0.01)	1.03	1.66	90.39	82.52
Medium (C=1)	0.99	2.12	94.74	93.91
High (C=10)	1.28	3.04	94.45	94.94

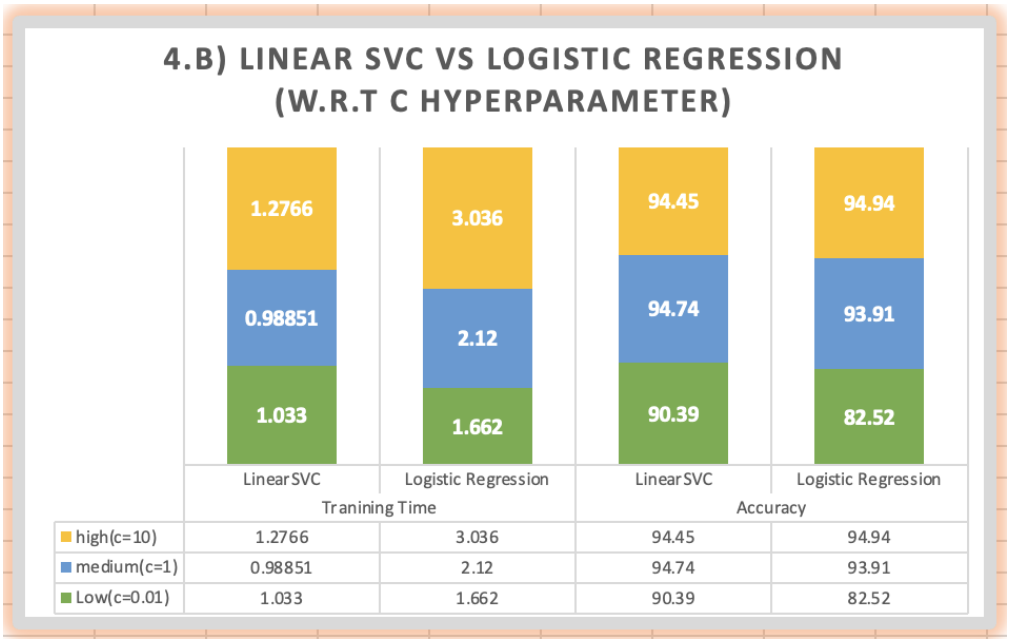

Figure 4: Linear SVC vs Logistic Regression

4.1.3 4c. Changing the tol hyperparameter in LinearSVC and Logistic Regression to high, low, and medium values)

Table 3: tol Hyperparameter

tol Hyperparameter	Training Time (seconds)		Accuracy (%)	
	LinearSVC	Logistic Regression	LinearSVC	Logistic Regression
Low (tol=1e-10)	1.07	3.02	94.74	93.91
Medium (tol=1e-4)	1.14	2.56	94.74	93.94
High (tol=1e1)	0.85	1.23	61.71	46.29

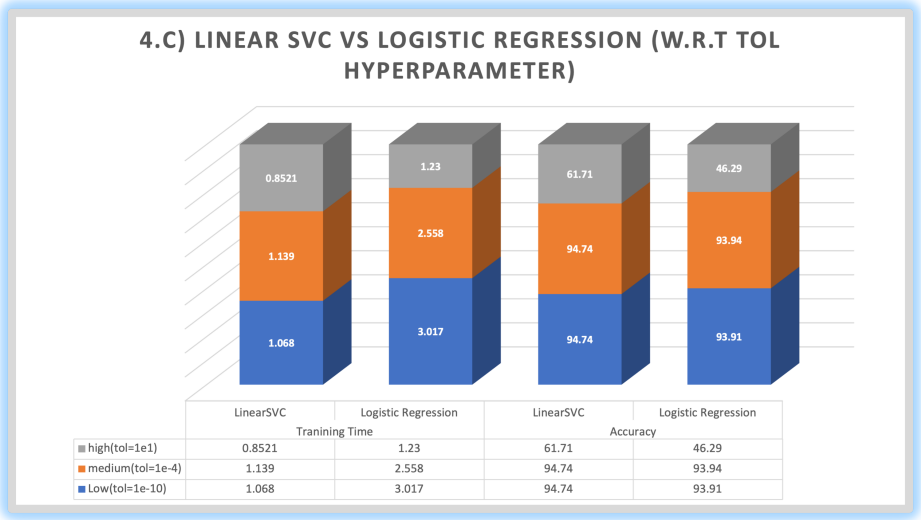

Figure 5: Linear SVC vs Logistic Regression (w.r.t. tol Hyperparameter)

4.1.4 4d. Changing the penalty (regularization) hyperparameter in LinearSVC and Logistic Regression (l2 vs l1)

Table 4: Penalty Hyperparameter

Penalty	Training Time (seconds)		Accuracy (%)	
	LinearSVC	Logistic Regression	LinearSVC	Logistic Regression
l1	11.29	2.12	94.59	93.53
l2	1.32	2.15	94.75	93.94

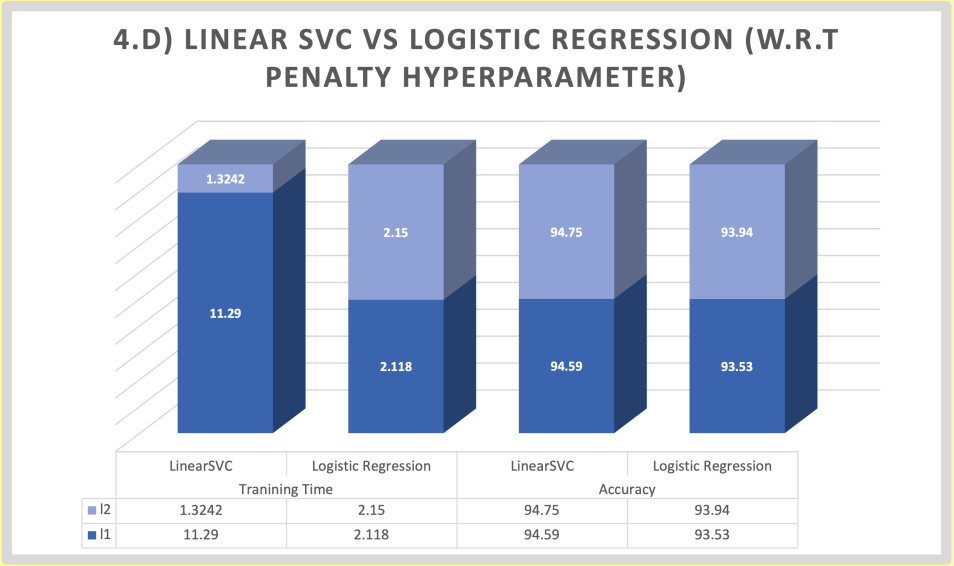

Figure 6: LinearSVC vs Logistic Regression (Penalty Hyperparameters)

5 References

1. CS973 - Machine Learning for Cyber Security, Week 3 - 4 Live Session and Slides by Prof. Roop Aparajita Subhra Purushottam
2. IIT Kharagpur workshop slides on PUFs
3. Research gate articles on side channel attacks on PUFs
4. `https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html`
5. `https://scikit-learn.org/1.5/modules/generated/sklearn.linear_model.LogisticRegression.html`
6. `https://scikit-learn.org/1.5/modules/generated/sklearn.linear_model.RidgeClassifier.html`
7. `https://www.geeksforgeeks.org/support-vector-machine-algorithm/`
8. `https://www.linkedin.com/pulse/linearsvc-classification-kirolos-fawzy-z5wof/`