CS771 | Introduction to Machine Learning
Assignment 1 - Report

Jyotiprakash Nayak
210478
`jpnayak21@iitk.ac.in`

Shikhar Singh
210973
`shikhars21@iitk.ac.in`

Devendra Jangir
210328
`devendraj21@iitk.ac.in`

Mohit Dhakad
210624
`mohitd21@iitk.ac.in`

Abstract

In this assignment, we demonstrate the vulnerability of Melbo's Cross Connection PUF (`COCO-PUF`) to linear models, providing a detailed mathematical derivation of a mapping function that breaks the security of `COCO-PUF`. Subsequently, we implement and analyze the linear model using `sklearn's LinearSVC` and `LogisticRegression`, exploring the impact of hyperparameters on training time and test accuracy to validate the susceptibility of `COCO-PUF` to linear modeling techniques.

1 Cross Connection PUF

1.1 Overview

An arbiter PUF is a chain of k multiplexers, each of which either swaps the lines or keeps them intact, depending on what is the challenge bit fed into that multiplexer. The multiplexers each have delays which are hard to replicate but consistent.

Let t^u, t^l respectively denote the time for the upper and lower signals to reach the finish line and let $\delta := t^u - t^l$ denote the difference in the timings. Note that δ can be negative or positive. Assume that $t^u = t^l$ never happens i.e., δ is never exactly zero. If the upper signal reaches the finish line first i.e. $\delta < 0$, the response is 0 else if the lower signal reaches first i.e. $\delta > 0$, the response is 1

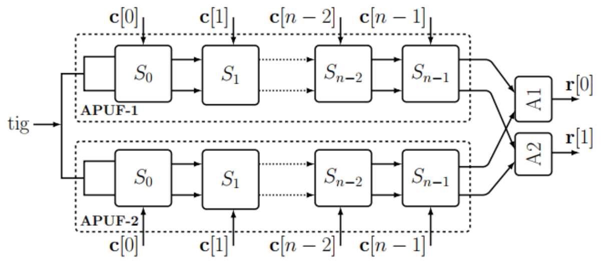


Figure 1: COCO puf

Cross Connection PUF (`COCO-PUF` for short) uses 2 arbiter PUFs– a working PUF and a reference PUF, as well as a secret threshold value $\tau > 0$. Given a challenge, it is fed into both the working

PUF and reference PUF and the timings for the upper and lower signals for both PUFs are measured. Let δ_w, δ_r be the difference in timings experienced for the two PUFs on the same challenge. The response to this challenge is 0 if $|\delta_w - \delta_r| \leq \tau$ and the response is 1 if $|\delta_w - \delta_r| > \tau$ where $\tau > 0$ is the secret threshold value.

Our job is to prove that it is not very difficult for a linear machine learning model to can predict the responses if given a few thousand challenge-response pairs.

1.2 Given Data

We have been provided with data from a COCO-PUF with 32-bit challenges. The training set consists of 40,000 Challenge-Response Pairs (CRPs) and the test set consists of 10,000 CRPs. Each CRP includes a 32-bit challenge and 2-bit responses (corresponding to Response0 and Response1). Each row in the provided training and testing files contains 34 numbers, where each number is either 0 or 1. The first 32 numbers represent the challenge, while the last two numbers represent the two responses.

Our task is to create a new feature vector from the 32-bit challenge to learn two linear models: one that predicts Response0 with high accuracy and another that predicts Response1 with high accuracy. It's important to note that unlike the arbiter PUF case, where the new feature vector had the same dimensionality as the challenges (32-dimensional for 32-dimensional challenges), the new feature vector for a COCO-PUF may need to have a different dimensionality than the original challenges.

We may create validation sets out of this data, using techniques such as held-out validation or k-fold cross-validation, to ensure our models generalize well to unseen data. By properly encoding the challenges and training linear models on these new feature vectors, we aim to achieve highly accurate predictions for both Response0 and Response1.

2 Tasks

2.1 Task 1 : Mathematical Derivation

To show derivation of a linear model to predict the arrival time of the upper signal in an arbiter PUF and the responses for a COCO-PUF, including specifying the required dimensionality for the models in each case.

2.1.1 Derivation of a Linear Model for Arbiter PUF's upper signal

In a simple Arbiter PUF, we aim to predict the time it takes for the upper signal to reach the finish line, given a challenge vector $\mathbf{c} \in \{0,1\}^{32}$. Let t_i^u denote the time the upper signal reaches the i-th arbiter, and t_i^l denote the time the lower signal reaches the i-th arbiter. The time difference between the two signals at the i-th arbiter is $\Delta_i = t_i^u - t_i^l$.

The time for the upper signal to reach the finish line can be expressed as:

$$t_i^u = t_0^u - \sum_{j=1}^{i} C_j \Delta_{j-1} + \sum_{j=1}^{i} (P_j - C_j(P_j - S_j))$$

Expanding Δ_i, we have:

$$\Delta_i = d_i \gamma_i + \beta_i$$

Substituting Δ_i into the expression for t_i^u, we get:

$$t_i^u = t_0^u - \sum_{j=1}^{i} C_j (d_{j-1}\gamma_{j-1} + \beta_{j-1}) + \sum_{j=1}^{i} (P_j - C_j(P_j - S_j))$$

Rewriting this expression:

$$t_i^u = t_0^u - \frac{1}{2} \sum_{j=1}^{i} d_j \gamma_j + \text{terms related to } \beta_j + \text{offset terms}$$

Thus, we can simplify t_i^u to:

$$t_i^u = -\frac{1}{2}\sum_{j=1}^{i} d_j\gamma_j + \sum_{j=1}^{i}(p_j + s_j) + \text{constant terms}$$

To model t_i^u using a linear model, we define the feature map $\phi(\mathbf{c})$ and weight vector W as follows. The feature vector $\phi(\mathbf{c})$ consists of the γ_i terms and β_i terms:

$$\phi(\mathbf{c}) = \begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_{32} \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{32} \end{pmatrix}$$

The dimension of the feature vector $\phi(\mathbf{c})$ is $D = 64$, where there are 32 γ_i terms and 32 β_i terms.

The weight vector W and the bias term b can be determined such that:

$$W^\top\phi(\mathbf{c}) + b = t_u(\mathbf{c})$$

In this way, the linear model for predicting the time t_i^u in an Arbiter PUF is constructed.

2.1.2 Dimension of the Feature Vector for Arbiter PUF

To predict the time $t_u(\mathbf{c})$ it takes for the upper signal to reach the finish line in an Arbiter PUF, we use a feature vector $\phi(\mathbf{c})$. This vector is constructed from the challenge vector $\mathbf{c} \in \{0,1\}^{32}$ and captures information needed for prediction.

The feature vector $\phi(\mathbf{c})$ consists of two types of features:

- $\gamma_i = p_i - s_i$, where p_i and s_i are determined by the challenge bit c_i for $i = 1, 2, \ldots, 32$.
- β_i, which is a constant bias term for each arbiter.

The feature vector is given by:

$$\phi(\mathbf{c}) = \begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_{32} \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{32} \end{pmatrix}$$

Since there are 32 γ_i terms and 32 β_i terms, the dimension D of the feature vector is:

$$D = 32 + 32 = 64$$

This 64-dimensional feature vector is used in a linear model to predict the upper signal time $t_u(\mathbf{c})$.

2.1.3 Derivation of a linear model for COCO PUF

Let $\Delta(n-1)$ be the delay difference of top and bottom paths after the final switch for a given n-bit challenge $\mathbf{c}$, where $\mathbf{c}$ follows a bipolar encoding. The sign of $\Delta(n-1)$ determines the response for a given challenge, as in the case of the COCOPUF.

Let p_{i+1} and r_{i+1} be two delay components in the top part of $(i+1)$-th stage, and q_{i+1} and s_{i+1} be the other two delay components in the bottom part of $(i+1)$-th stage . Let $\delta_t(i)$ and $\delta_b(i)$ be the propagation delays of the trigger signal along the top and bottom paths to the input of the $(i+1)$-th stage of COCOPUF, respectively. For a given challenge $\mathbf{c} \in \{+1, -1\}$, propagation delay of the trigger signal at the output of the $(i+1)$-th stage is given by:

$$\delta_t(i+1) = \frac{1+c[i+1]}{2}(\delta_t(i) + p_{i+1}) + \frac{1-c[i+1]}{2}(\delta_t(i) + r_{i+1})$$
$$\delta_b(i+1) = \frac{1+c[i+1]}{2}(\delta_b(i) + q_{i+1}) + \frac{1-c[i+1]}{2}(\delta_b(i) + s_{i+1})$$

Then delay difference between top and bottom paths can be estimated as:

$$\Delta(i+1) = \delta_t(i+1) - \delta_b(i+1) = \frac{1+c[i+1]}{2}(\Delta(i) + p_{i+1} - q_{i+1}) + \frac{1-c[i+1]}{2}(\Delta(i) + r_{i+1} - s_{i+1}) \tag{1}$$
$$\Delta(i+1) = \Delta(i) + \alpha_{i+1}c[i+1] + \beta_{i+1}, \text{where} \tag{2}$$

$$\alpha_{i+1} = \frac{p_{i+1} - r_{i+1} - q_{i+1} + s_{i+1}}{2}, \quad \text{and} \quad \beta_{i+1} = \frac{p_{i+1} + r_{i+1} - q_{i+1} - s_{i+1}}{2}.$$

Let us assume that $\Delta(-1) = 0$, and then the delay difference $\Delta(n-1)$ after the final stage can be:

$$\Delta(n-1) = \sum_{i=0}^{n-1} \alpha_i c[i] + \sum_{i=0}^{n-1} \beta_i = \sum_{i=0}^{n} w[i]\Phi[i] = \langle \mathbf{w}, \mathbf{\Phi} \rangle, \quad \text{where} \tag{3}$$

$$\mathbf{w}[n] = (\beta_{n-1} + \cdots + \beta_0) \quad \text{and} \quad \mathbf{w}[i] = \alpha_i \quad \text{for} \quad i = 0, \ldots, n-1,$$

where the feature vector $\mathbf{\Phi}$ is the parity vector of challenge c defined as:

$$\mathbf{\Phi}[n] = 1, \quad \text{and} \quad \mathbf{\Phi}[i] = \prod_{j=i}^{n-1} c[j] \quad \text{for} \quad i = 0, \ldots, n-1.$$

2.1.4 Dimension of the feature vector for COCO PUF

In the provided equations, the feature vector Φ is an extension of the previously discussed vector, defined for an n-stage DAPUF. Here, Φ is a $2n$-bit feature vector where $\Phi = (c, p)$, combining challenge bits c and a derived vector p. Specifically, Φ includes both the challenge vector

$$c = [c[0], c[1], \ldots, c[n-1]]$$

and the vector p derived from the challenge as follows:

$$p[n-1] = 1, \quad \text{and} \quad p[i] = \prod_{j=i}^{n-2} c[j] \quad \text{for } 0 \leq i \leq n-2.$$

Given $n = 32$, the Φ vector consists of the original 32-bit challenge vector c and an additional 32-bit vector p. Thus, Φ is a 64-dimensional vector when $n = 32$.

The weight vector w in this context is also a $2n$-bit vector, denoted as $w = (x_t, y_t)$, which aligns with the dimensionality of Φ. For $n = 32$, w is also a 64-dimensional vector. The relationship between the delay differences Δ_t and Δ_b and the weight vector w and feature vector Φ is captured by the inner product $\langle w, \Phi \rangle$.

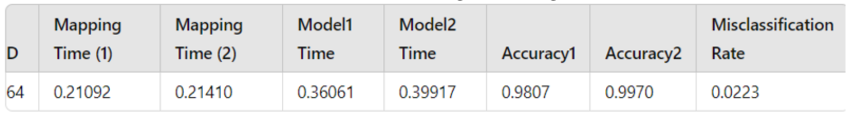
In summary, when $n = 32$:

- The dimensionality of Φ is 64.
- The dimensionality of w is 64.

2.2 Task 2 : Results Of Code

The final results of the code are as follows:

Table 1: Results for Logistic Regression



D	Mapping Time (1)	Mapping Time (2)	Model1 Time	Model2 Time	Accuracy1	Accuracy2	Misclassification Rate
64	0.21092	0.21410	0.36061	0.39917	0.9807	0.9970	0.0223

The python code for `my_fit()` function we used is as follows:

```python
def my_fit(X_train, y0_train, y1_train):
    hyperparameters = {
        'C': 30,
        'max_iter': 200,
        'penalty': 'l2',
        'tol': 0.01
    }
    global model
    model = LogisticRegression(**hyperparameters)
    global model1
    model1 = LogisticRegression(**hyperparameters)

    model.fit(my_map(X_train), y0_train)
    model1.fit(my_map(X_train), y1_train)

    w0 = model.coef_
    b0 = model.intercept_

    w1 = model1.coef_
    b1 = model1.intercept_

    return w0, b0, w1, b1
```

The python code for `my_map()` function we used is as follows:

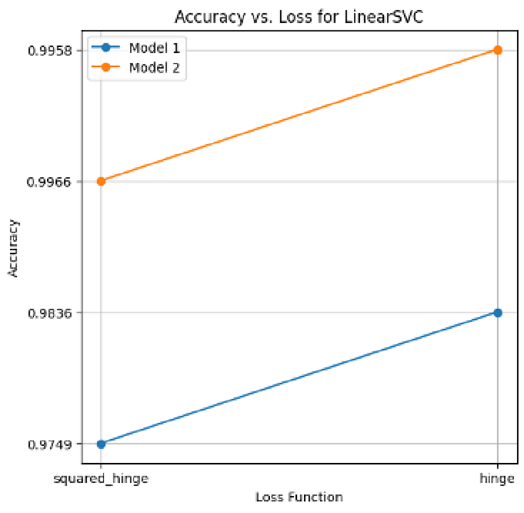
```python
def my_map(X):
    num_rows, num_cols = X.shape

    feat = np.zeros((num_rows, 2 * num_cols))

    d = 1 - 2 * X
    for k in range(num_rows):
        feat[k, :num_cols] = np.cumprod(d[k, ::-1])[::-1]

    feat[:, num_cols:] = d

    return feat
```

2.3 Task 3 : Varying Hyperparameters

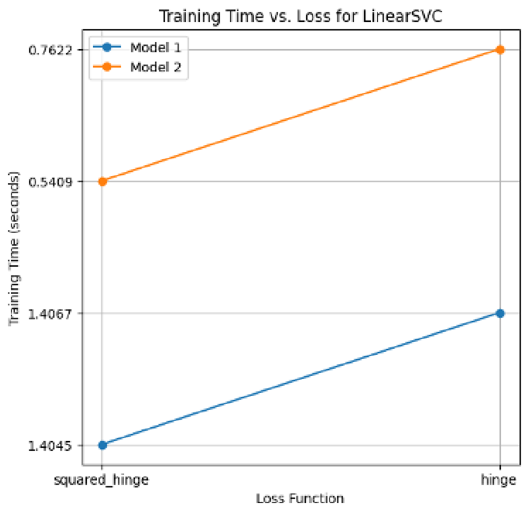
Report outcomes of the following experiments with both the `sklearn.svm.LinearSVC` and `sklearn.linear_model.LogisticRegression` methods when used to learn the linear model. In particular, report how various `hyperparameters` affected training time and test accuracy using tables and/or charts with both `LinearSVC` and `LogisticRegression` methods.

2.3.1 Changing the loss hyperparameter in LinearSVC (hinge vs squared hinge)

u



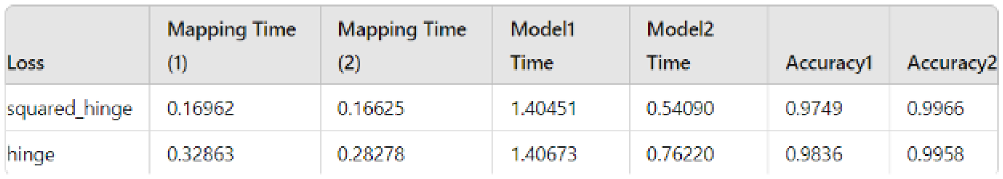
(a) Accuracy Comparison



(b) Time Comparison

Figure 2: Changing the `loss` hyperparameter in LinearSVC

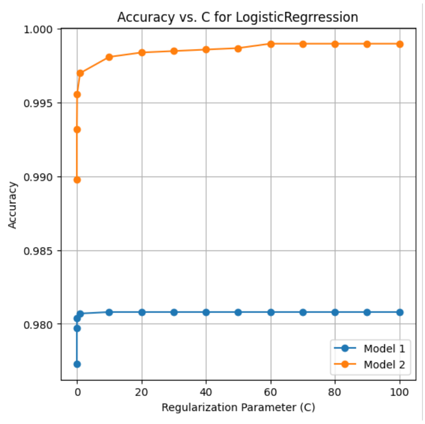
Table 2: Changing the `loss` hyperparameter in LinearSVC



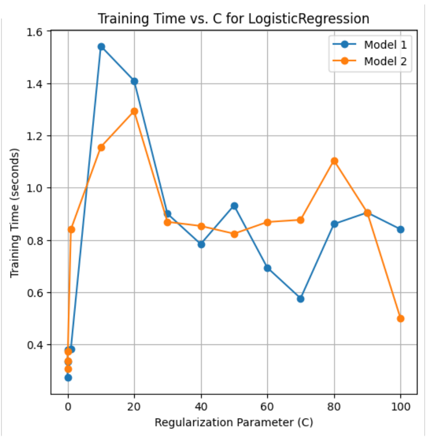
Loss	Mapping Time (1)	Mapping Time (2)	Model1 Time	Model2 Time	Accuracy1	Accuracy2
squared_hinge	0.16962	0.16625	1.40451	0.54090	0.9749	0.9966
hinge	0.32863	0.28278	1.40673	0.76220	0.9836	0.9958

2.3.2 Setting C in LinearSVC and LogisticRegression to high/low/medium values.

For LogisticRegression:



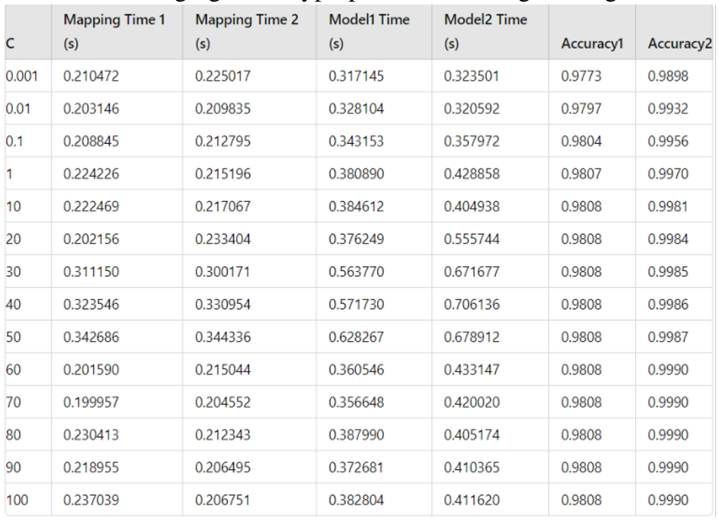
(a) Test Accuracy Comparison



(b) Training Time Comparison

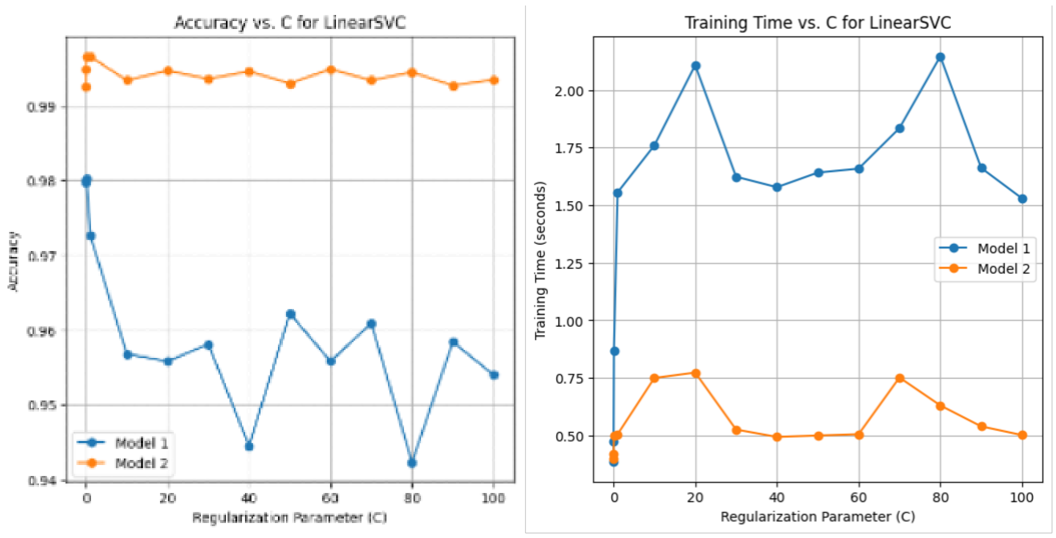
Figure 3: Changing the `C` hyperparameter in LogisticRegression

Table 3: Changing the `C` hyperparameter in LogisticRegression



C	Mapping Time 1 (s)	Mapping Time 2 (s)	Model1 Time (s)	Model2 Time (s)	Accuracy1	Accuracy2
0.001	0.210472	0.225017	0.317145	0.323501	0.9773	0.9898
0.01	0.203146	0.209835	0.328104	0.320592	0.9797	0.9932
0.1	0.208845	0.212795	0.343153	0.357972	0.9804	0.9956
1	0.224226	0.215196	0.380890	0.428858	0.9807	0.9970
10	0.222469	0.217067	0.384612	0.404938	0.9808	0.9981
20	0.202156	0.233404	0.376249	0.555744	0.9808	0.9984
30	0.311150	0.300171	0.563770	0.671677	0.9808	0.9985
40	0.323546	0.330954	0.571730	0.706136	0.9808	0.9986
50	0.342686	0.344336	0.628267	0.678912	0.9808	0.9987
60	0.201590	0.215044	0.360546	0.433147	0.9808	0.9990
70	0.199957	0.204552	0.356648	0.420020	0.9808	0.9990
80	0.230413	0.212343	0.387990	0.405174	0.9808	0.9990
90	0.218955	0.206495	0.372681	0.410365	0.9808	0.9990
100	0.237039	0.206751	0.382804	0.411620	0.9808	0.9990

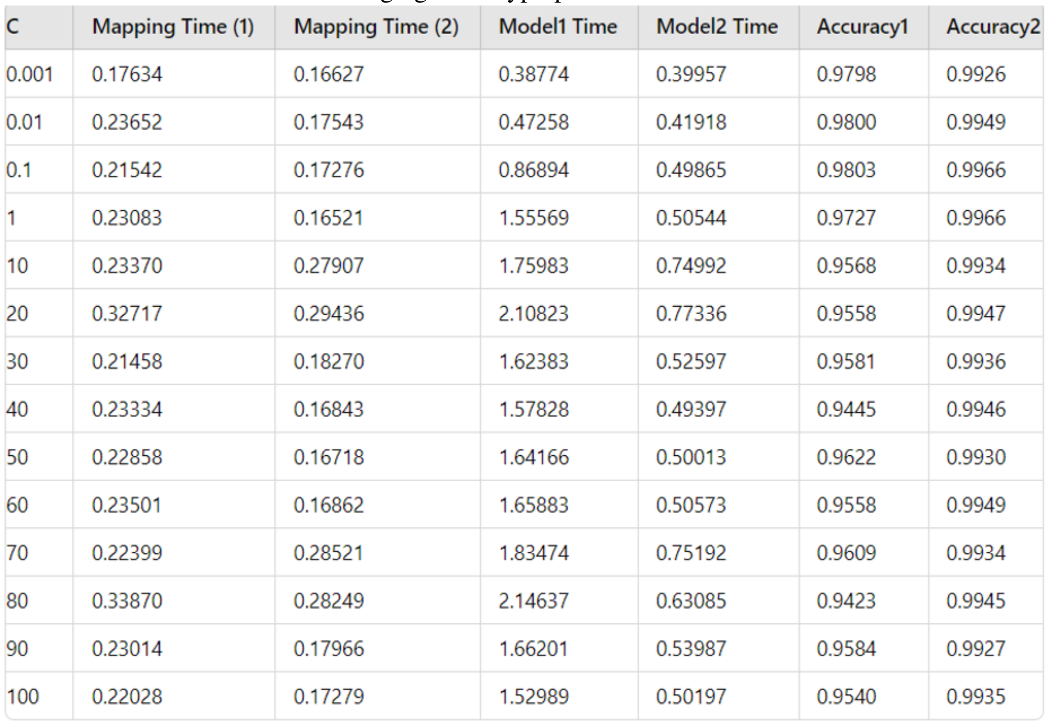
For LinearSVC:



(a) Test Accuracy Comparison (b) Training Time Comparison

Figure 4: Changing the `C` hyperparameter in LinearSVC

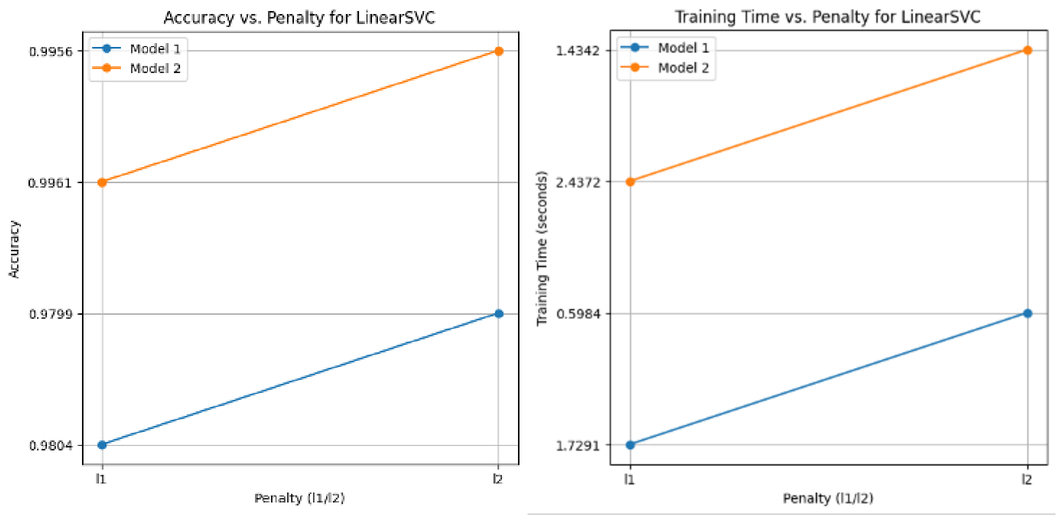
Table 4: Changing the `C` hyperparameter in LinearSVC



C	Mapping Time (1)	Mapping Time (2)	Model1 Time	Model2 Time	Accuracy1	Accuracy2
0.001	0.17634	0.16627	0.38774	0.39957	0.9798	0.9926
0.01	0.23652	0.17543	0.47258	0.41918	0.9800	0.9949
0.1	0.21542	0.17276	0.86894	0.49865	0.9803	0.9966
1	0.23083	0.16521	1.55569	0.50544	0.9727	0.9966
10	0.23370	0.27907	1.75983	0.74992	0.9568	0.9934
20	0.32717	0.29436	2.10823	0.77336	0.9558	0.9947
30	0.21458	0.18270	1.62383	0.52597	0.9581	0.9936
40	0.23334	0.16843	1.57828	0.49397	0.9445	0.9946
50	0.22858	0.16718	1.64166	0.50013	0.9622	0.9930
60	0.23501	0.16862	1.65883	0.50573	0.9558	0.9949
70	0.22399	0.28521	1.83474	0.75192	0.9609	0.9934
80	0.33870	0.28249	2.14637	0.63085	0.9423	0.9945
90	0.23014	0.17966	1.66201	0.53987	0.9584	0.9927
100	0.22028	0.17279	1.52989	0.50197	0.9540	0.9935

2.3.3 Changing the penalty (regularization) hyperparameter in LinearSVC and LogisticRegression (l2 vs l1)

For LinearSVC:



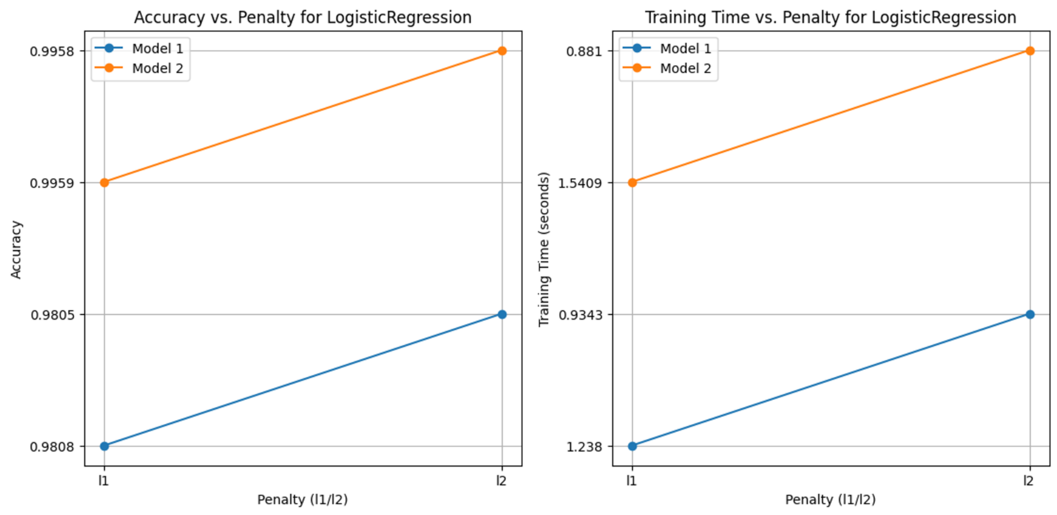
(a) Test Accuracy Comparison (b) Training Time Comparison

Figure 5: Changing the `penalty` hyperparameter in Linear SVC

Table 5: Changing the `penalty` hyperparameter in LinearSVC

Penalty	Mapping Time (1)	Mapping Time (2)	Model1 Time	Model2 Time	Accuracy1	Accuracy2
l1	0.16913	0.18403	1.72908	2.43719	0.9804	0.9961
l2	0.26144	0.23411	0.59841	1.43423	0.9799	0.9956

For LogisticRegression:



(a) Test Accuracy Comparison (b) Training Time Comparison

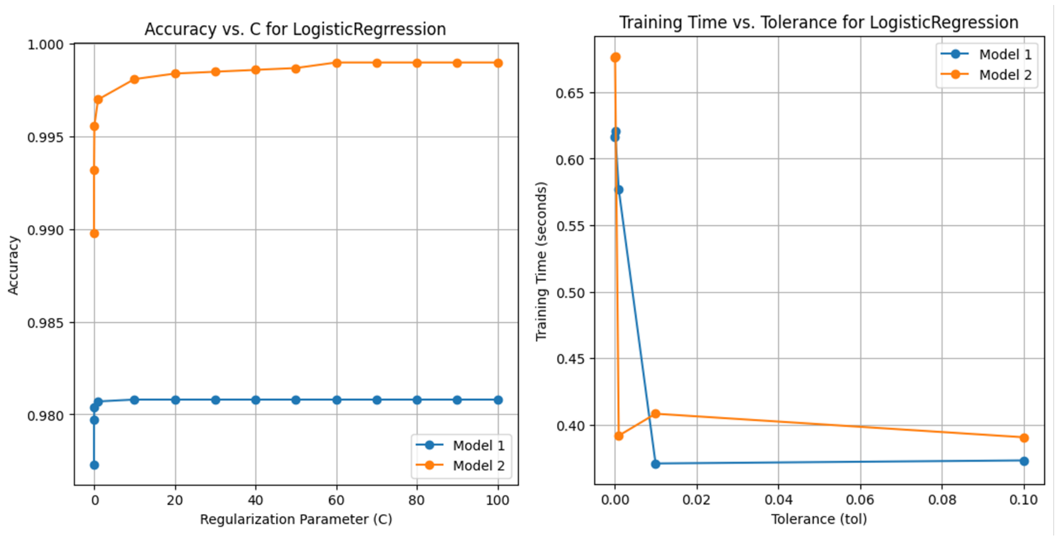
Figure 6: Changing the `penalty` hyperparameter in LogisticRegression

Table 6: Changing the `penalty` hyperparameter in LogisticRegression

Penalty	Mapping Time 1 (s)	Mapping Time 2 (s)	Model1 Time (s)	Model2 Time (s)	Accuracy1	Accuracy2
l1	0.194489	0.223757	1.279192	1.719912	0.9807	0.9965
l2	0.248569	0.181451	0.851164	0.774840	0.9805	0.9958

2.3.4 Changing tol in LinearSVC and LogisticRegression to high/low/medium values.

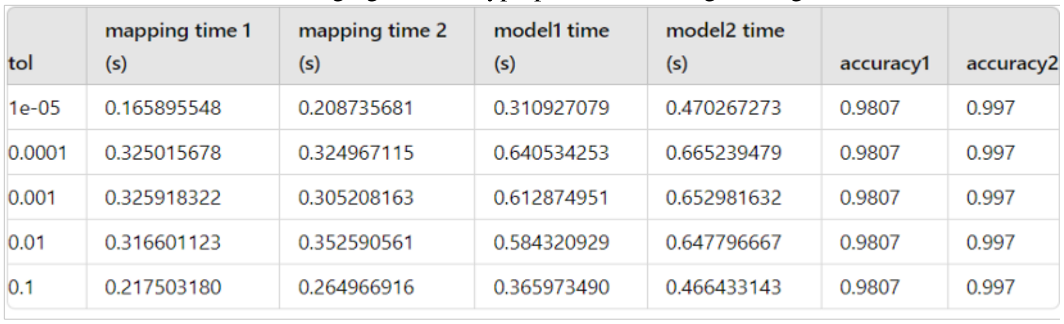
For LogisticRegression:



(a) Test Accuracy Comparison (b) Training Time Comparison

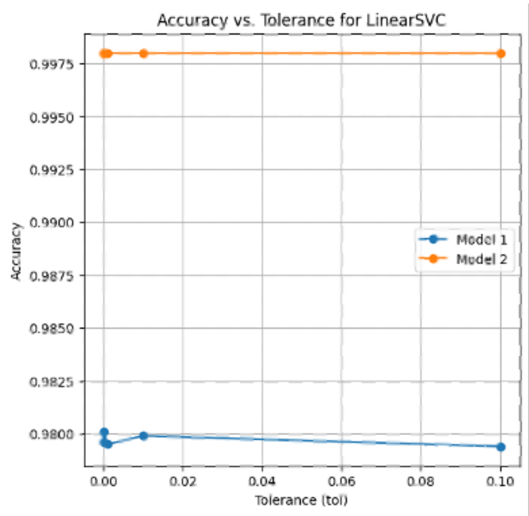
Figure 7: Changing the `tol` hyperparameter in LogisticRegression

Table 7: Changing the `tol` hyperparameter in LogisticRegression

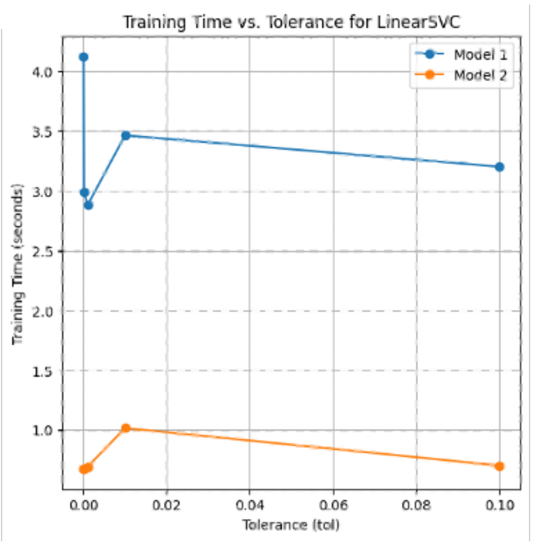


tol	mapping time 1 (s)	mapping time 2 (s)	model1 time (s)	model2 time (s)	accuracy1	accuracy2
1e-05	0.165895548	0.208735681	0.310927079	0.470267273	0.9807	0.997
0.0001	0.325015678	0.324967115	0.640534253	0.665239479	0.9807	0.997
0.001	0.325918322	0.305208163	0.612874951	0.652981632	0.9807	0.997
0.01	0.316601123	0.352590561	0.584320929	0.647796667	0.9807	0.997
0.1	0.217503180	0.264966916	0.365973490	0.466433143	0.9807	0.997

For LinearSVC:



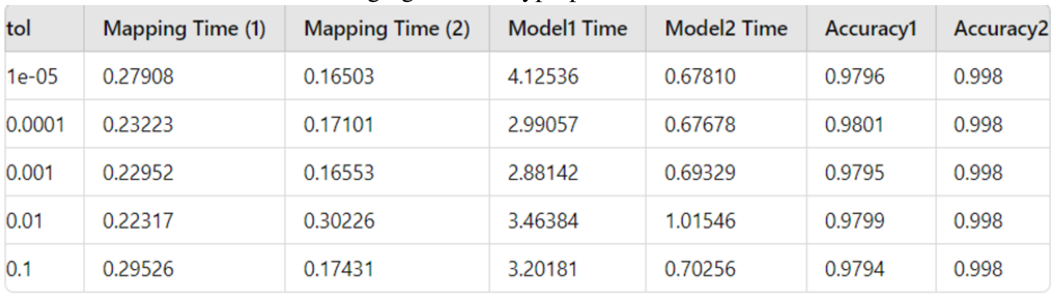
(a) Test Accuracy Comparison



(b) Training Time Comparison

Figure 8: Changing the `tol` hyperparameter in LinearSVC

Table 8: Changing the `tol` hyperparameter in LinearSVC



tol	Mapping Time (1)	Mapping Time (2)	Model1 Time	Model2 Time	Accuracy1	Accuracy2
1e-05	0.27908	0.16503	4.12536	0.67810	0.9796	0.998
0.0001	0.23223	0.17101	2.99057	0.67678	0.9801	0.998
0.001	0.22952	0.16553	2.88142	0.69329	0.9795	0.998
0.01	0.22317	0.30226	3.46384	1.01546	0.9799	0.998
0.1	0.29526	0.17431	3.20181	0.70256	0.9794	0.998