Indian Institute of Technology Kanpur
CS771 Introduction to Machine Learning

Instructor: Purushottam Kar
Date: April 19, 2025
Total: 150 marks

ASSIGNMENT 1

1 What should I submit, where should I submit and by when?

Your submission for this mini-project will be one PDF (.pdf) file and one ZIP (.zip) file. Instructions on how to prepare and submit these files are given below.

Mini-project Package:
`https://www.cse.iitk.ac.in/users/purushot/courses/ml/2024-25-w/material/projects/mp1.zip`
Deadline for all submissions: May 03, 2025, 9:59PM IST
Code Validation Script: `https://colab.research.google.com/drive/1agYL6z-HNe5UwAl0rhWHCyrixcTkSNcq?usp=sharing`
Code Submission: `https://forms.gle/FbumwPvpzegoyKe96`
Report Submission: on Gradescope
There is no provision for "late submission" for this mini-project

1.1 How to submit the PDF report file

1. The PDF file must be submitted using Gradescope in the *group submission mode*.
2. Note that unregistered auditors cannot make submissions to this mini-project.
3. Make only one submission per mini-project group on Gradescope, not one submission per student. Gradescope allows you to submit in groups - please use this feature to make a group submission.
4. **Ensure that you validate your submission files on Google Colab before making your submission** (validation details below). Submissions that fail to work with our automatic judge since they were not validated will incur penalties.
5. Link all group members in your group submission. If you miss out on a group member while submitting, Gradescope will think that person never submitted anything.
6. You may overwrite your group's submission as many times as you want before the deadline (submitting again on Gradescope simply overwrites the old submission).
7. Do not submit Microsoft Word or text files. Prepare your report in PDF using the style file we have provided (instructions on formatting given later).

1.2 How to submit the code ZIP file

1. Your ZIP file should contain a single Python (.py) file and nothing else. The reason we are asking you to ZIP that single Python file is so that you can password protect the ZIP file. Doing this safeguards you since even after you upload your ZIP file to your website,

no one can download that ZIP file and see your solution (since you will tell the password only to the instructor). If you upload a naked Python file to your website, someone else may guess the location where you have uploaded your file and steal it and you may get charged with plagiarism later.

2. We do not care what you name your ZIP file but the (single) Python file sitting inside the ZIP file must be named "submit.py". There should be no sub-directories inside the ZIP file – just a single file. We will look for a single Python (.py) file called "submit.py" inside the ZIP file and delete everything else present inside the ZIP file.

3. Do not submit Jupyter notebooks or files in other languages such as C/C++/R/Julia/Matlab/Java. We will use an automated judge to evaluate your code which will not run code in other formats or other languages (submissions in other languages will get a zero score).

4. Password protect your ZIP file using a password with 8-10 characters. Use only alphanumeric characters (a-z A-Z 0-9) in your password. Do not use special characters, punctuation marks, whitespaces etc in your password. Specify the file name properly in the Google form.

5. Remember, your file is not under attack from hackers with access to supercomputers. This is just an added security measure so that even if someone guesses your submission URL, they cannot see your code immediately. A length 10 alphanumeric password (that does not use dictionary phrases and is generated randomly e.g. 2x4kPh02V9) provides you with more than 55 bits of security. It would take more than 1 million years to go through all $> 2^{55}$ combinations at 1K combinations per second.

6. Make sure that the ZIP file does indeed unzip when used with that password (try
   `unzip -P your-password file.zip`
   on Linux platforms).

7. Upload the password protected ZIP file to your IITK (CC or CSE) website (for CC, log on to `webhome.cc.iitk.ac.in`, for CSE, log on to `turing.cse.iitk.ac.in`).

8. Fill in the following Google form to tell us the exact path to the file as well as the password `https://forms.gle/FbumwPvpzegoyKe96`

9. **Do not host your ZIP submission file on file-sharing services like Dropbox or Google drive. Host it on IITK servers only**. We will autodownload your submissions and GitHub, Dropbox and Google Drive servers often send us an HTML page (instead of your submission) when we try to download your file. Thus, it is best to host your code submission file locally on IITK servers.

10. While filling in the form, you have to provide us with the password to your ZIP file in a designated area. Write just the password in that area. For example, do not write "Password: helloworld" in that area if your password is "helloworld". Instead, simply write "helloworld" (without the quotes) in that area. Remember that your password should contain only alphabets and numerals, no spaces, special or punctuation characters.

11. While filling the form, give the complete URL to the file, not just to the directory that contains that file. The URL should contain the filename as well.

    (a) Example of a proper URL:
    `https://web.cse.iitk.ac.in/users/purushot/mlproj/my_submit.zip`

   (b) Example of an improper URL (file name missing):
   `https://web.cse.iitk.ac.in/users/purushot/mlproj/`
   (c) Example of an improper URL (incomplete path):
   `https://web.cse.iitk.ac.in/users/purushot/`

12. We will use an automated script to download all your files. If your URL is malformed or incomplete, or if you have hosted the file outside IITK in a manner that is difficult to download, then your group may lose marks.

13. Make sure you fill-in the Google form with your file link before the deadline. We will close the form at the deadline.

14. Make sure that your ZIP file is actually available at the link at the time of the deadline. We will run a script to automatically download these files after the deadline is over. If your file is missing, we will treat this as a blank submission.

15. We will entertain no submissions over email, Piazza etc. All submissions must take place before the stipulated deadline over the Gradescope and the Google form. The PDF file must be submitted on Gradescope at or before the deadline and the ZIP file must be available at the link specified on the Google form at or before the deadline.

Problem 1.1 (Multi-level PUF). Melbo realized something interesting about arbiter PUFs that could potentially lead to a more secure PUF that is resistant to ML attacks. Recall that 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. At the finish line resides an arbiter (usually a flip-flop) which decides which signal reached first, the upper signal or the lower signal. The arbiter then uses this decision to generate the response. Although it is possible for a simple linear model to predict whether the upper signal will reach the finish line first or the lower signal (as we have seen in class), it seems that it is not so simple for that model to predict the time taken by the upper signal to reach the finish line, or predict the time taken by the lower signal to reach the finish line. Armed with this realization, Melbo created a new PUF variant and gave it the name **Multi-level PUF** or **ML-PUF** for short.

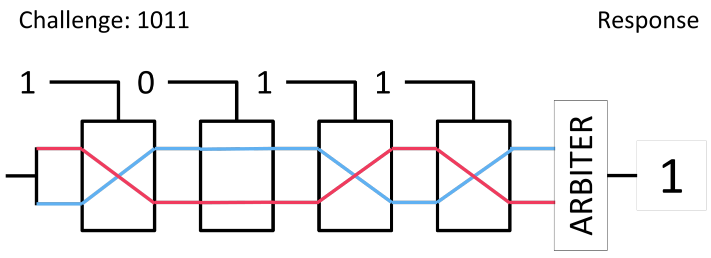

Figure 1: A simple arbiter PUF with 4 multiplexers

An ML-PUF uses 2 arbiter PUFs, say PUF0 and PUF1 – each PUF has its own set of multiplexers with possibly different delays. Given a challenge, it is fed into both the PUFs. However, the way responses are generated is different. Instead of the lower and upper signals of PUF0 competing with each other, Melbo makes the lower signal from PUF0 compete with the lower signal from PUF1 using an arbiter called Arbiter0 to generate a response called Response0. If the signal from PUF0 reaches first, Response0 is 0 else if the signal from PUF1 reaches first, Response0 is 1. Melbo also makes the upper signal from PUF0 compete with the upper signal from PUF1 using a second arbiter called Arbiter1 to generate a response called Response1. If the signal from PUF0 reaches first, Response1 is 0 else if the signal from PUF1 reaches first, Response1 is 1. The XOR of Response0 and Response1 is the final response. More precisely, if both Response0 and Response1 are the same bit, then the final response is 0 else if Response0 and Response1 are different then the final response is 1. On each challenge, the ML-PUF generates two responses and then takes their XOR to generate a single final response.

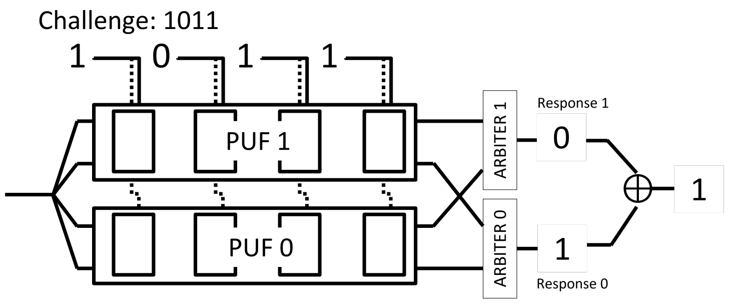

Figure 2: A ML-PUF with 4-bit challenges and 2-bit responses

Melbo thinks that these cross connections and the additional XOR operation should make it difficult for a linear model to can predict the responses if given a few thousand challenge-response pairs. Your job is to prove Melbo wrong! You will do this by showing that there do exist linear models that can perfectly predict the responses of an ML-PUF and these linear model can be estimated fairly accurately if given enough challenge-response pairs (CRPs).

Your Data. We have provided you with data from an ML-PUF with 8-bit challenges. The training set consists of 6400 CRPs and the test set consists of 1600 CRPs. For each CRP, an 8-bit challenge and a 1-bit response (corresponding to the XOR of Response0 and Response1) is provided. Each row in the training and testing files provided to you contain 9 numbers, each number being 0 or 1. The first 8 numbers correspond to the challenge and the last number corresponds to the response. If you wish, you may create (held out/k-fold) validation sets out of this data in any way you like. Recall that if we properly encode the challenge as an 8-dim vector, then for any arbiter PUF, there exists a linear model that always predicts the correct response for that arbiter PUF. Your job is to create a new feature vector from the 8-bit challenge so that you can learn a linear model to predict the XOR of Response0 and Response1 very accurately. However, note that unlike the arbiter PUF case where the new feature vector was also 8-dimensional for 8-dimensional challenges, for an ML-PUF, the new feature vector may have a dimensionality different from that of the challenges.

(100 marks)

Problem 1.2 (Delay Recovery by Inverting an Arbiter PUF). A k-bit Arbiter PUF is described using k-delays $p_i, q_i, r_i, s_i, 0 \leq i \leq (k-1)$ using notation from the class slides. These delays generate a linear model ($\mathbf{w} \in \mathbb{R}^k, b \in \mathbb{R}$) that will break this Arbiter PUF using the derivation discussed in class. To recapitulate, if we define $\alpha_i \triangleq (p_i - q_i + r_i - s_i)/2$ and $\beta_i \triangleq (p_i - q_i - r_i + s_i)/2$, then we can get the model as $w_0 = \alpha_0$, $w_i = \alpha_i + \beta_{i-1}$ for $1 \leq i \leq k-1$ and $b = \beta_{k-1}$.

Melbo has used leaked CRPs for a 64-bit arbiter PUF (unrelated to the PUF used in the ML-PUF question above) to learn a linear model ($\mathbf{w} \in \mathbb{R}^{64}, b \in \mathbb{R}$) and is curious if the delays $p_i, q_i, r_i, s_i, 0 \leq i \leq 63$ can be back-calculated using only $\mathbf{w}, b$. Melba advises Melbo that it is not possible to uniquely recover the delays using just the model. Melba gives several justifications that help Melbo realize why this is the case – for example, for any $\epsilon_i \geq 0, \eta_i \geq 0, 0 \leq i \leq 63$, the model generated by the delays $p_i, q_i, r_i, s_i, 0 \leq i \leq 63$ is identical to the model generated by the delays $p_i + \epsilon_i, q_i + \epsilon_i, r_i + \eta_i, s_i + \eta_i, 0 \leq i \leq 63$. A more profound reason why the delays cannot be uniquely recovered is that the model ($\mathbf{w} \in \mathbb{R}^{64}, b \in \mathbb{R}$) is generated using a system of 65 linear equations that use the delays $p_i, q_i, r_i, s_i, 0 \leq i \leq 63$ as input. However, whereas there are 256 delay parameters to be recovered, only 65 equations are available.

Using Melba's advice, Melbo revises the goal – to not necessarily recover the actual delays of the PUF. Instead, any set of delays that generate the same linear model would do. Given a linear model ($\mathbf{w} \in \mathbb{R}^{64}, b \in \mathbb{R}$), Melbo wishes to find out 256 delays that are non-negative real numbers $\hat{p}_i \geq 0, \hat{q}_i \geq 0, \hat{r}_i \geq 0, \hat{s}_i \geq 0, 0 \leq i \leq 63$ so that these delays generate the exact same linear model $\mathbf{w}, b$. Note that the delays must be non-negative numbers but their magnitude is not restricted i.e., the delays are allowed to be zero or even a large positive real number.

Your Data. We have provided you with 10 linear models, each represented as a list of 65 real numbers that may be negative or positive or zero but will always lie between -1 and +1. The first 64 real numbers in each list represent a model vector $\mathbf{w} \in \mathbb{R}^{64}$ and the last real number in the list represents a bias term $b \in \mathbb{R}$. Your job is to take each linear model and return 256 non-negative real numbers that, if interpreted as arbiter PUF delays, result in the same linear model. A distinct list of 256 delays is needed for each linear model.

(50 marks)

Your Task. The following enumerates 7 parts to the question. Parts 1, 2, 3, 4, 7 need to be answered in the PDF file containing your report. Parts 5, 6 needs to be answered in the Python file. **Please note that Problem 1.1 and 1.2 need to be answered together i.e. the same PDF file should contain solutions to theoretical answers to ML-PUF and Delay Recovery problems and the same Python file should contain code answers to both problems. Thus, submit a single PDF file and a single Python file.**

1. 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 $\tilde{\phi} : \{0,1\}^8 \to \mathbb{R}^{\tilde{D}}$ and a corresponding linear model $\tilde{\mathbf{W}} \in \mathbb{R}^{\tilde{D}}, \tilde{b} \in \mathbb{R}$ that predicts the responses i.e. for all CRPs $\mathbf{c} \in \{0,1\}^8$, we have $\frac{1+\text{sign}(\tilde{\mathbf{W}}^\top \tilde{\phi}(\mathbf{c})+\tilde{b})}{2} = r(\mathbf{c})$ where $r(\mathbf{c})$ is the response of the ML-PUF on the challenge $\mathbf{c}$. Note that $\tilde{\mathbf{W}}, \tilde{b}$ may depend on the PUF-specific constants such as delays in the multiplexers. However, the map $\tilde{\phi}(\mathbf{c})$ must depend only on $\mathbf{c}$ (and perhaps universal constants such as $2, \sqrt{2}$ etc). The map $\tilde{\phi}$ must not use PUF-specific constants such as delays. (20 marks)

2. What dimensionality $\tilde{D}$ 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. (5 marks)

3. Suppose we wish to use a kernel SVM to solve the problem instead of creating our own feature map. Thus, we wish to use the original challenges $\mathbf{c} \in \{0,1\}^8$ as input to a kernel SVM (i.e., without doing things to the features like converting challenge bits to $+1, -1$ 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. (10 marks)

4. 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 – see hints below. (20 marks)

5. Write code to solve the ML-PUF problem by learning a linear model $\mathbf{W}, b$ using the training data that predict the response for the ML-PUF. You are allowed to use any linear classifier formulation available in the sklearn library to learn the linear model (i.e. you need not write a solver yourself). For instance, you may use LinearSVC, LogisticRegression, RidgeClassifier etc. However, the use of non-linear models is not allowed. Submit code for your chosen method in `submit.py`. Note that your code will need to implement at least 2 methods namely

   (a) `my_map()` that should take test challenges and map each one of them to a $\tilde{D}$-dimensional feature vector.

   (b) `my_fit()` that should take train CRPs and learn the linear model $\tilde{\mathbf{W}}, \tilde{b}$ by invoking an sklearn routine (see validation code on Google Colab for clarification). It is likely that `my_fit()` will internally call `my_map()`.

Note that your learnt model must be one vector and one bias term (if you do not wish to use a bias term, set it to 0). The use of non-linear models such as decision trees, random forests, nearest neighbors, neural networks, kernel SVMs etc is not allowed. To implement the map ϕ, you may find the Khatri-Rao product useful (`https://en.wikipedia.org/wiki/Khatri%E2%80%93Rao_product#Column-wise_Kronecker_product`). The KR-product is implemented in the Scipy package as `scipy.linalg.khatri_rao` which you can freely use without penalty (use of any other Scipy routine will incur penalties). However, use of the KR product is not compulsory and you may choose to not use it. Although the KR-product is define in a column-wise manner, you may find the row-wise version more useful by transposing the matrix before applying the KR-product. (55 marks)

6. Write code to solve the Arbiter PUF inversion problem to recover delays. You are allowed to use any standard solver in numpy, sklearn such as solvers for linear system of equations, least squares, ridge regression etc. However, as the hint below indicates, you might find it faster and more accurate to write a simple solver yourself for this question. The use of non-linear models is not allowed for this part (and not required either). Submit code for your chosen method in `submit.py`. Note that your code will need to implement at least one method namely

   (a) `my_decode()` that should take a single 65-dimensional linear model (last dimension interpreted as bias) as input and return four 64-dimensional vectors as output that represent the delays that your code has inferred for the model given as input (see validation code on Google Colab for clarification).

   Note that your output must be four 64-dimensonal vectors that must contain only non-negative entries (zeros are allowed). The use of non-linear models such as decision trees, random forests, nearest neighbors, neural networks, kernel SVMs etc is not allowed for this part (and not required either). (30 marks)
   Note that all three methods my_map(), my_fit(), my_decode() must be provided in the same submit.py file and not in separate files.

7. 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)

   You may of course perform and report all the above experiments and/or additional experiments not mentioned above (e.g. changing the `solver`, `max_iter` etc) but reporting at least 2 of the above experiments is mandatory. You do not need to submit code for these experiments – just report your findings in the PDF file. Your submitted code should only include your final method (e.g. learning the linear model using LinearSVC or recovering the delays using ridge regression) with hyperparameter settings that you found to work the best. No hyperparameter reporting required for problem 1.2 (inverting the arbiter PUF to recover delays) (10 marks)

Parts 1, 2, 3, 4, 7 need to be answered in the PDF file containing your report. Parts 5, 6 needs to be answered in the Python file.

Hint for Part 1. First show how a linear model can predict the time it takes for the upper signal to reach the finish line for a simple arbiter PUF (similarly for the lower signal). Using this, show how a linear model can predict Response0 and a (different) linear model can predict Response 1. Then apply the trick used to handle XOR-PUFs to complete the problem.

Hint for Part 3. A system of equations $A\mathbf{x} = \mathbf{b}$ can be inverted in many ways

1. Solving the linear system $A, \mathbf{b}$ using a linear algebra solver e.g. `numpy.linalg.solve`
2. Solving least squares $\min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|_2^2$ e.g. `sklearn.linear_model.LinearRegression`
3. Solving ridge-regression $\min_{\mathbf{x}} \lambda \cdot \|\mathbf{x}\|_2^2 + \|A\mathbf{x} - \mathbf{b}\|_2^2$ e.g. `sklearn.linear_model.Ridge`
4. Custom hand-crafted solver

Note that we need to ensure that the delays are non-negative that can be done in several ways e.g. postprocessing if the solver returns delays that take negative values (might find Melba's words useful here) or using positivity constraints allowed by sklearn solvers for least squares or ridge regression. However, note that the system of linear equations in this case is very sparse here (if encoded properly, $A \in \mathbb{R}^{65\times 256}$ but for each row in A, less than 10 entries are non-zero). Thus, a custom hand-crafted solver written in plain Python might be faster and be more numerically stable than using general-purpose solvers available in libraries.

Evaluation Measures and Marking Scheme for Problem 1.1. We created two ML-PUFs – a public one and a secret one. Both used 8-bit challenges but the multiplexer delays in the secret ML-PUF are different than those in the public ML-PUF. Thus, a model that is able to predict the public ML-PUF responses is expected to do poorly at predicting the responses for the secret ML-PUF and vice versa. The train/test sets we have provided you with the mini-project package were created using CRPs from the public ML-PUF. We similarly created a secret train and secret test set using CRPs from the secret ML-PUF.

We will use your `my_fit()` method to train on our secret train set and use the learnt model $\mathbf{W}, b$ to make predictions on our secret test set. Given a test challenge $\tilde{\mathbf{x}}_t$, we will use your `my_map()` method to map it to $\tilde{D}$ dimensions and then apply the linear model as $\frac{1+\text{sign}(\mathbf{W}^\top \tilde{\phi}(\tilde{\mathbf{x}}_t)+b)}{2}$ to predict the ML-PUF response. Note that the secret train/test set may not have the same number of train/test points as the public train/test set that we have provided you. However, we assure you that the public and secret ML-PUFs are similar in that hyperparameter choices that seem good to you during validation (e.g. C, tol, solver etc), should work decently on our secret dataset as well. We will repeat the evaluation process described below 5 times and use the average performance so as to avoid any unluckiness due to random choices inside your code in a particular trial. We will award marks based on four performance parameters:

1. What dimensionality does your feature map use to predict the ML-PUF responses i.e. how large is $\tilde{D}$? (10 marks)
2. How much time does your `my_fit` method take to finish training (15 marks)

3. How much time does your `my_map` method take to generate test features (15 marks)

4. What is the test misclassification rate offered by your model for the response?(15 marks)

For all performance parameters, lower is better (e.g. smaller D will get higher marks). For more details, please check the evaluation script on Google Colab (linked below). Once we receive your code, we will execute the evaluation script to award marks to your submission.

Evaluation Measures and Marking Scheme for Problem 1.2. We created 20 arbiter PUFs, each taking 64-bit challenges. The 20 arbiter PUFs are independent and the model/delays for one PUF says nothing about the model/delays for another PUF. For each PUF, we used its delays to generate a 65-dimensional linear model (last dimension being the bias term). You have been given 10 of the models as a part of the mini-project package to allow you to figure out how well your algorithm works. The rest 10 are safe with us and will be used to evaluate your code.

We will use your `my_decode()` method and send it a single 65-dimensional model and expect it to send us four 64-dimensional arrays $\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{s}$ representing the delays in the 64 multiplexers of the arbiter PUF. These delays will checked and any negative entries will be removed and then the cleaned-up delays will be used to reconstruct the 65-dimensional model which will be evaluated against the original model in terms of Euclidean distance. Note that there is no *train-test* split in this part since this the task here is to take a single linear model and invert it to get non-negative delays. We will repeat the evaluation process described below 5 times and use the average performance so as to avoid any unluckiness due to random choices inside your code in a particular trial. We will award marks based on two performance parameters:

1. How much time does your `my_decode` method take to get the 256 delays given a 65-dimensional model (last dimension being bias term). (15 marks)

2. What is the Euclidean distance between the model generated by the delays output by your `my_decode` method and the linear model that was sent as input. (15 marks)

For all performance parameters, lower is better (e.g. smaller Euclidean distance will get higher marks). For more details, please check the evaluation script on Google Colab (linked below). Once we receive your code, we will execute the evaluation script to award marks to your submission.

Validation on Google Colab. Before making a submission, you must validate your submission on Google Colab using the script linked below.

Link: `https://colab.research.google.com/drive/1agYL6z-HNe5UwAlOrhWHCyrixcTkSNcq?usp=sharing`

Validation ensures that your submitted file `submit.py` does work with the automatic judge and does not give errors. Please use the IPYNB file at the above link on Google Colab and the dummy secret train, dummy secret test set and dummy model files (details below) to validate your submission.

Please make sure you do this validation on Google Colab itself. **Do not download the IPYNB file and execute it on your machine – instead, execute it on Google Colab itself.** This is because most errors we encounter are due to non-standard library versions on students personal machines. Thus, running the IPYNB file on your personal machine defeats the whole purpose of validation. You must ensure that your submission runs on Google Colab to detect any library conflict. **Please note that there will be penalties for submissions which were not validated on Google Colab and which subsequently give errors with our automated judge.**

Dummy Submission File and Dummy Secret Files. In order to help you understand how we will evaluate your submission using the evaluation script, we have included a dummy secret train, secret test set and secret model file in in the mini-project package itself (see the directory called `dummy`). However, note that these are just copies of the train, test dataset and 10 public models we provided you. The reason for providing the dummy files is to allow you to check whether the evaluation script is working properly on Google Colab or not. Be warned that the secret files on which we actually evaluate your submission will be different. We have also included a dummy submission file `dummy_submit.py` to show you how your code must be written to return a model with `my_fit()`, features with `my_map()` and delay retrieval with `my_decode()`. Note that the algorithms used in `dummy_submit.py` are very bad which will give poor results. However, this is okay since its purpose is only to show you the code format.

Using Internet Resources. You are allowed to refer to textbooks, internet sources, research papers to find out more about this problem and for specific derivations e.g. the arbiter-PUF problem. However, if you do use any such resource, cite it in your PDF file. There is no penalty for using external resources with attribution but claiming someone else's work (e.g. a book or a research paper) as one's own work without crediting the original author will attract penalties.

Restrictions on Code Usage. You are allowed to use the numpy module in its entirety and any sklearn submodule that learns linear models. To do so, you may include submodules of sklearn e.g. `import sklearn.svm` or `import sklearn.linearmodel` etc. However, **the use of any non-linear model is prohibited** e.g. decision trees, random forests, nearest neighbors, neural networks, kernel SVMs etc. The use of other machine learning libraries such as **libsvm, keras, tensorflow is also prohibited**. The only exception to this rule is the use of the `scipy.linalg.khatri_rao` routine from SciPy (all other SciPy routines are prohibited too). Use of prohibited modules and libraries for whatever reason will result in penalties. For this mini-project, you should also not download any code available online or use code written by persons outside your mini-project group. Direct copying of code from online sources or amongst mini-project groups will be considered and act of plagiarism for this mini-project and penalized according to pre-announced policies.

2 How to Prepare the PDF File

Use the following style file to prepare your report.
`https://media.neurips.cc/Conferences/NeurIPS2023/Styles/neurips_2023.sty`

For an example file and instructions, please refer to the following files
`https://media.neurips.cc/Conferences/NeurIPS2023/Styles/neurips_2023.tex`
`https://media.neurips.cc/Conferences/NeurIPS2023/Styles/neurips_2023.pdf`

You must use the following command in the preamble

```
\usepackage[preprint]{neurips_2023}
```

instead of `\usepackage{neurips_2023}` as the example file currently uses. Use proper LaTeX commands to neatly typeset your responses to the various parts of the problem. Use neat math expressions to typeset your derivations. Remember that all parts of the question need to be answered in the PDF file. All plots must be generated electronically - hand-drawn plots are unacceptable. All plots must have axes titles and a legend indicating what are the plotted quantities. Insert the plot into the PDF file using LaTeX `\includegraphics` commands.

3 How to Prepare the Python File

The mini-project package contains a skeleton file `submit.py` which you should fill in with the code of the method you think works best among the methods you tried. You must use this skeleton file to prepare your Python file submission (i.e. do not start writing code from scratch). This is because we will autograde your submitted code and so your code must have its input output behavior in a fixed format. Be careful not to change the way the skeleton file accepts input and returns output.

1. The skeleton code has comments placed to indicate non-editable regions. **Do not remove those comments**. We know they look ugly but we need them to remain in the code to keep demarcating non-editable regions.

2. We have provided you with data points in the file `public_trn.txt` in the mini-project package that has 6400 data points, having a 8-bit challenge and a 1-bit response. You may use this as training data in any way to tune your hyperparameters (e.g. `C`, `tol` etc) by splitting into validation sets in any fashion (e.g. held out, k-fold). You are also free to use any fraction of the training data for validation, etc. Your job is to do really well in terms of coming up with an algorithm that can learn a model to predict the responses in the test set accurately and speedily, produce a model that is not too large, and also perform learning as fast as possible.

3. We have provided you with ten 65-dimensional linear models (last dimension being bias term) in the file `public_mod.txt`. Use these to create a method to take a PUF linear model and recover non-negative delays.

4. The code file you submit should be self contained and should not rely on any other files (e.g. other .py files or else pickled files etc) to work properly. Remember, your ZIP archive should contain only one Python (.py) file. This means that you should store any hyperparameters that you learn (e.g. step length etc) inside that one Python file itself using variables/functions.

5. We created two ML-PUFs – a public one and a secret one. Using the public ML-PUF, we created CRPs that were split into the train and test set we have provided you with the mini-project package. We similarly created CRPs using the secret ML-PUF and created a secret train and secret test set with it. We will use your code to train on our secret train set and use the learnt model to test on our secret test set. Both the public and secret ML-PUFs have 8-bit challenges and use two arbiter PUFs each. However, the delays in the PUFs are not the same so a linear model that is able to predict the public ML-PUF responses will do poorly at predicting the secret ML-PUF delays and vice versa. Moreover, note that the secret train set is not guaranteed to contain 6400 points and the secret test set is not guaranteed to contain 1600 points (for example, our secret test set may have 10000 points or 20000 points etc). However, we assure you that the public and secret PUFs look similar otherwise so that hyperparameter choices that seem good to you during validation (e.g. step length, stopping criterion etc), should work decently on our secret dataset as well.

6. We created 20 arbiter PUFs, each taking 64-bit challenges and generated 65-dimension linear models for each using their delays (last dimension being bias). Each PUF is independent of all others in terms of its delays i.e. the linear model for one PUF is not expected to do well to predict responses on another PUF. The mini-project package contains 10 of these models and the rest 10 are secret and will be used to evaluate your submission.

7. Certain portions of the skeleton code have been marked as non-editable. Please do not change these lines of code. Insert your own code within the designated areas only. If you tamper with non-editable code (for example, to make your code seem better or faster than it really is), the auto-grader may refuse to run your code and give you a zero instead (we will inspect each code file manually).

8. You are allowed to freely define new functions, new variables, new classes in inside your submission Python file while not changing the non-editable code.

9. The use of any non-linear model is prohibited e.g. decision trees, random forests, nearest neighbors, neural networks, kernel SVMs etc. The use of other machine learning libraries such as scipy, libsvm, keras, tensorflow is also prohibited.

10. You are allowed to use basic Python libraries such as numpy, time and random which have already been included for you. You are also allowed to use linear model learning methods from the sklearn library (e.g. sklearn.svm or sklearn.linearmodel) and the Khatri-Rao product method from the scipy library. Please note that you are not allowed to use any other routine from the scipy library.

11. Do take care to use broadcasted and array operations as much as possible and not rely on loops to do simple things like take dot products, calculate norms etc otherwise your solution will be slow and you may get less marks.

12. Please do not perform file operations in your code or access the disk – you should not be using libraries such as sys, pickle in your code.

13. The use of any non-linear model is prohibited e.g. decision trees, random forests, nearest neighbors, neural networks, kernel SVMs etc. The use of other machine learning libraries such as libsvm, keras, tensorflow is also prohibited.

14. Before submitting your code, make sure you validate on Google Colab to confirm that there are no errors etc.

15. You do not have to submit the evaluation script to us – we already have it with us. We have given you access to the Google Colab evaluation script just to show you how we would be evaluating your code and to also allow you to validate your code.