Indian Institute of Technology Kanpur
CS973 Machine Learning for Cyber Security

Instructor: Purushottam Kar
Date: October 09, 2023
Total: 60 marks

ASSIGNMENT 1

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

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

Deadline for all submissions: 03 November, 2024, 9:59PM IST
Code Validation Script: `https://colab.research.google.com/drive/1dRtJ_wK0jxsdac_2ES1pg57mpfQX-vCn?usp=sharing`
Code Submission: `https://forms.gle/45mWHemtLYdFkiLu7`
Report Submission: on Gradescope
There is no provision for "late submission" for this assignment

1.1 How to submit the PDF report file

1. The PDF file must be submitted using Gradescope in the *group submission mode*.
2. Make only one submission per assignment group on Gradescope, not one submission per student. Gradescope allows you to submit in groups - please use this feature to make a group submission.
3. Link all group members in your group submission. If you miss out on a group member while submitting, that member may end up getting a zero since Gradescope will think that person never submitted anything.
4. You may overwrite your group's submission (submitting again on Gradescope simply overwrites the old submission) as many times as you want before the deadline.
5. 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. Ensure that you validate your code 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.
2. 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 file.
3. 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.

4. Do not submit Jupyter notebooks or files in other languages such as C/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 may simply get a zero score).

5. 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.

6. Remember, your file is not under attack from hackers with access to supercomputers. This is just an added security measure so that your submission does not get rejected for containing scripts. 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.

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

8. Upload the password protected ZIP file using the following Google form which allows you to tell us other details as well such as the password `https://forms.gle/45mWHemtLYdFkiLu7`

9. 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.

10. Make sure you fill in the Google form with your file upload before the deadline. We will close the form at the deadline.

11. We will entertain no submissions over email, edX 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 submitted using the Google form at or before the deadline.

Problem 1.1 (Uncovering the Mask of Xorro)**.** Melbo was unhappy that the arbiter and XOR PUF devices were broken so easily using a simple linear model. In an effort to make an unbreakable PUF, Melbo came up with another idea using devices called ring oscillators. Below, we first describe a traditional ring oscillator, then we describe a more interesting ring oscillator using XOR gates (called a XOR Ring Oscillator which we will lovingly call XORRO) and finally we describe how to create a powerful PUF using multiple XORROs.

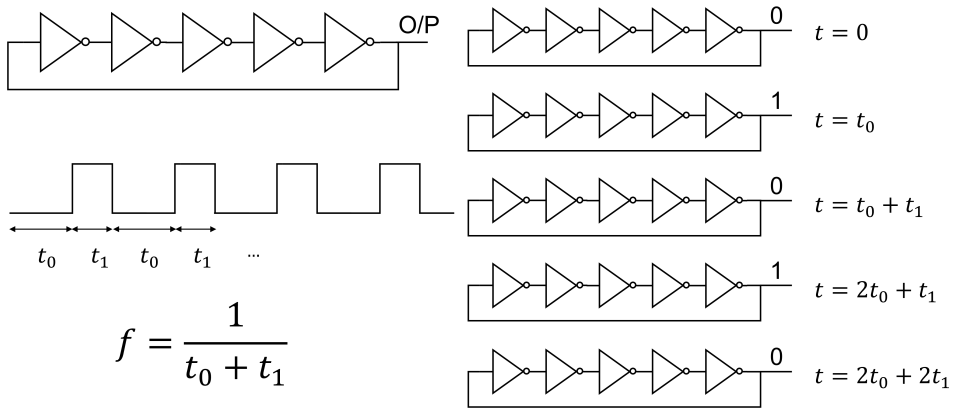

Figure 1: A simple ring oscillator using 5 inverters.

Ring Oscillator (RO). The figure above shows a simple ring oscillator created using 5 NOT gates. The NOT gates are inverters since they invert their input – a 0 input yields a 1 output and a 1 input gives a 0 output. The name of this device is very apt since the NOT gates are connected in a ring and the output of the last NOT gate (marked as O/P) oscillates between 0 and 1. To see why there is an oscillation, notice that if a bit is inverted an odd number of times, the final output will be inverted i.e. a 0 inverted an odd number of times times will yield a 1 and a 1 inverted an odd number of times will give us a 0.

Since the O/P is connected back as input to the first NOT gate, after 5 inversions, O/P will flip. Then that output is fed back again and thus, after 5 inversions, O/P will flip again. However, NOT gates do not give their output instantly – there is a delay. Suppose the 5 NOT gates are such that if 0 is input to them, they give their output (which will be 1) after δ_i^0 seconds $i = 0, 1, \ldots, 4$. Similarly, if the NOT gates are input 1, then they give the output 0 after δ_i^1 seconds $i = 0, 1, \ldots, 4$. We assume that delays caused due to the wires connecting the NOT gates is absorbed in the δ_j^i values themselves.

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. Note that it may be the case that $t_0 \neq t_1$. However, it can be safely assumed that t_0 and t_1 remain constant over time and do not change. It is also very difficult to manufacture an RO with precisely the same values of t_0, t_1 which is why the frequency of the RO f (with formula given below) can be treated as a unique fingerprint of the RO.

$$f \stackrel{\text{def}}{=} \frac{1}{t_0 + t_1}$$

XOR Ring Oscillator (XORRO). The simple RO does not allow multiple challenge-response pairs (CRP) to be created. To remedy this, we notice that the XOR gate can also act as a *configurable* inverter. The XOR gate takes two inputs and the output is 1 only if exactly one of the inputs is 1 and the other is 0 (we do not care which input is 1 and which input is 0). If both or neither of the inputs is 1 then the output is 0. Note that if the second input to

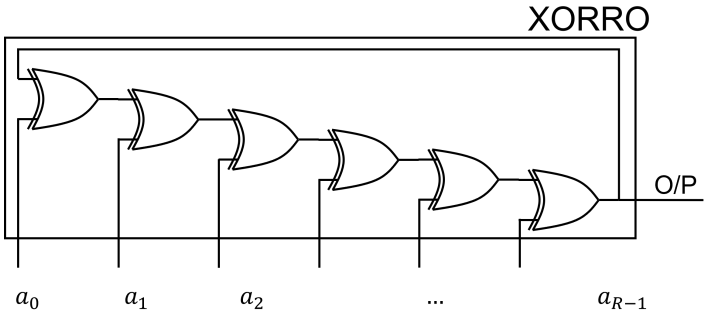

Figure 2: A configurable ring oscillator using 5 XOR gates.

a XOR gate is fixed to 1 then the XOR gate starts acting as an inverter gate with respect to the first input since XOR(0,1) = 1 and XOR(1,1) = 0. Similarly, if the second input to a XOR gate is fixed to 0 then the XOR gate starts acting as an identity gate with respect to the first input since XOR(0,0) = 0 and XOR(1,0) = 1. Using this, the XOR Ring Oscillator is created as shown in the figure above.

The XORRO has R XOR gates and for each gate, the second input is linked to a config bit that tells us whether that XOR gate will act as an inverter gate or an identity gate. The first inputs of all the XOR gates are connected in a ring to create a ring oscillator. Note that in order for the XORRO to indeed oscillate, an odd number of XOR gates must act as inverters. This means an odd number of config bits $a_0, \ldots a_{R-1}$ must be set to 1 otherwise the XORRO will not oscillate.

The XORRO offers us flexibility to create multiple challenge responses pairs since there are more delay parameters now. Let $\delta^i_{00}, \delta^i_{01}, \delta^i_{10}, \delta^i_{11}$ be the time that the i^{th} XOR gate takes before giving its output when the input to that gate is, respectively, 00, 01, 10 and 11. Note that the XORRO does not have a unique oscillation frequency as its oscillation frequency depends on the *config* bits $\mathbf{a} \stackrel{\text{def}}{=} [a_0, a_1, \ldots, a_{R-1}]$ send into the XORRO.

This gives us potentially 2^R different oscillation frequencies and we can set a particular frequency simply by setting the corresponding config bits. Note that it is very difficult to manufacture a XOR gate with exactly these 2^R frequencies but we assume that the frequencies remain stable over time.[1]

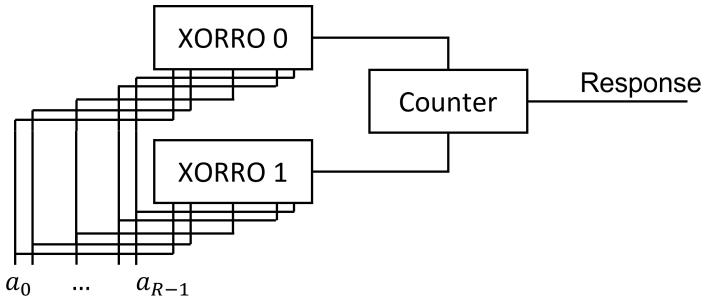

Figure 3: A Simple XORRO PUF using 2 XORROs.

Simple XORRO PUF. The above realization allows us to create a PUF as shown above. We take two XORROs. Our challenge is an R bit string that is fed into both XORROs as their config bits. Each of the XORROs will now start oscillating. The (oscillating) outputs of the

[1] Actually the number of potentially different oscillation frequencies is around 2^{R-1} since we must set an odd number of config bits to 1 and there are $\sum_{r=0}^{R/2-1} \binom{R}{2r+1} = 2^{R-1}$ ways of setting the config bits in this manner.

two XORROs is fed into a counter which can find out which XORRO has a higher frequency. If the upper XORRO (in this case XORRO 0) has a higher frequency, the counter outputs 1 else if the lower XORRO (in this case XORRO 1) has a higher frequency, the counter outputs 0. The output of the counter is our response to the challenge. Assume that it will never be the case that both XORROs have the same frequency i.e. the counter will never be confused.

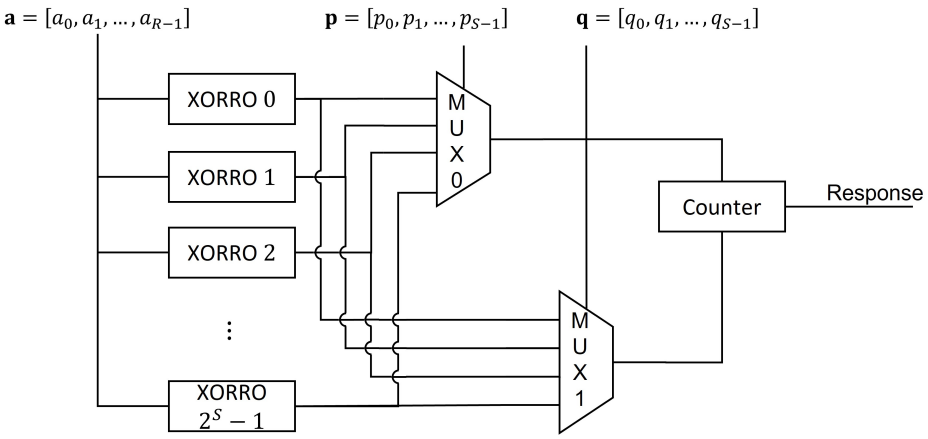

Figure 4: An Advanced XORRO PUF using 2^S XORROs.

Advanced XORRO PUF. Worried by the previous failures, Melbo does not want to take any chances this time and decides to make the PUF even stronger. Instead of having just 2 XORROs, Melbo now takes 2^S XORROs and extends the challenge vector to have $2S$ more bits which we call *select* bits. The first S of these select bits is interpreted as a number between 0 and $2^S - 1$ and used to select the corresponding XORRO as the first XORRO. This selection is done by using a multiplexer. The next S select bits are used to select another XORRO as the second XORRO. For example, if the $S = 3$ (i.e. there are $2^3 = 8$ XORROs) and the $2S$ bits are 110010 then the XORRO number 6 will get selected as the first XORRO (as $110 \equiv 6$) and XORRO number 2 will get selected as the second XORRO (as $010 \equiv 2$).

Note that for this selection scheme to work, the XORROs are numbered from 0 to 7, not 1 to 8. Also, notice that the counter will definitely get confused if the same XORRO is chosen twice e.g. if the select bits are 110110 or 010010. Thus, we are assured that the first S select bits are not the same as the second S select bits. The frequencies of the two selected XORROs are compared using a counter. If the XORRO selected by the upper MUX (in this case MUX 0) has a higher frequency, the counter outputs 1 else if the XORRO selected by the lower MUX (in this case MUX 1) has a higher frequency, the counter outputs 0. The output of the counter is our response to the challenge. Note that the challenge now has $R + 2S$ bits.

Melbo thinks that with 2^R possible frequencies in each XORRO and 2^S XORROS, there is no way any machine learning model, let alone a linear one, 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 even in this case, there does exist a model that can perfectly predict the responses of a XOR-PUF and this model can be learnt if given enough challenge-response pairs (CRPs).

Your Data. We have provided you with data from an Advanced XORRO PUF with $R = 64, S = 4$ i.e. it has $64 + 4 + 4 = 72$ bit challenges. Each row in the train/set file has 73 bits. The first 64 bits are the R bits to be sent as challenges to both the selected XORROs. The next $S = 4$ bits select the first XORRO. The next $S = 4$ bits select the second XORRO. The last bit is the response. The training set consists of 60000 CRPs and the test set consists of

40000 CRPs. If you wish, you may create (held out/k-fold) validation sets out of this data in any way you like. Your job is to learn a model that can accurately predict the responses on the test set. However, a twist in the tale is that unlike in the Arbiter/XOR PUF case where a single linear model was sufficient to break the PUF, here you will need an *ensemble* model that contains $M > 1$ linear models (more details below).

Your Task. The following enumerates 4 parts to the question. Parts 1,2,4 need to be answered in the PDF file containing your report. Part 3 needs to be answered in the Python file.

1. By giving a detailed mathematical derivation (as given in the lecture slides), show how a simple XORRO PUF can be broken by a single linear model. Recall that the simple XORRO PUF has just two XORROs and has no select bits and no multiplexers (see above figure and discussion on Simple XORRO PUF). Thus, the challenge to a simple XORRO PUF has just R bits. More specifically, give derivations for a map $\phi : \{0,1\}^R \to \mathbb{R}^D$ mapping R-bit 0/1-valued challenge vectors to D-dimensional feature vectors (for some $D > 0$) and show that for any simple XORRO PUF, there exists a linear model i.e. $\mathbf{w} \in \mathbb{R}^D, b \in \mathbb{R}$ such that for all challenges $\mathbf{c} \in \{0,1\}^R$, the following expression
$$\frac{1 + \text{sign}(\mathbf{w}^\top \phi(\mathbf{c}) + b)}{2}$$
gives the correct response. (10 marks)

2. Show how to extend the above linear model to crack an Advanced XORRO PUF. Do this by treating an advanced XORRO PUF as a collection of multiple simple XORRO PUFs. For example, you may use $M = 2^{S-1}(2^S - 1)$ linear models, one for each pair of XORROs, to crack the advanced XORRO PUF. (10 marks)

3. Write code to solve this problem by learning the M linear models $\mathbf{w}^1, \ldots, \mathbf{w}^M$. You may use any linear classifier formulation e.g. LinearSVC, LogisticRegression, RidgeClassifier etc., to learn the linear model. However, the use of non-linear models is not allowed. You are allowed to use scikit-learn routines to learn the linear model you have chosen (i.e. you do not need to solve the SVM/LR/RC optimization problems yourself). Submit code for your chosen method in `submit.py`. Note that your code will need to implement at least 2 methods namely
   (a) `my_fit()` that should take CRPs for the advanced XORRO PUF as training data and learn the M linear models.
   (b) `my_predict()` that should take test challenges and predict responses on them.

   We will evaluate your method on a different dataset than the one we have given you and check how good is the method you submitted (see below for details). **Note that your learnt model must be a collection of one or more linear models**. The use of non-linear models such as decision trees, random forests, nearest neighbors, neural networks, kernel SVMs etc is not allowed. (35 marks)

4. Report outcomes of experiments with both the sklearn.svm.LinearSVC and sklearn.linear_model.LogisticRegression methods when used to learn the ensemble linear model. In particular, report how various hyperparameters affected training time and test accuracy using tables, 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 the affect of training time and test accuracy of at least 2 of the following:

(a) changing the `loss` hyperparameter in LinearSVC (hinge vs squared hinge)

(b) setting `C` hyperparameter in LinearSVC and LogisticRegression to high/low/medium values

(c) changing the `tol` hyperparameter 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 required. 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 M models using LinearSVC) with hyperparameter settings that you found to work the best. (5 marks)

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

Evaluation Measures and Marking Scheme. We created two Advanced XORRO PUFs – a public one and a secret one. Both had $R = 64, S = 4$ i.e. 72-bit challenges but the PUFs all have different delays in their XOR gates so a model that is able to predict the public XORRO PUF delays is expected to do very poorly at predicting the secret XORRO PUF delays and vice versa. The train/test sets we have provided you with the assignment package were created using CRPs from the public XORRO PUF. We similarly created a secret train and secret test set using CRPs from the secret XORRO PUF. We will use your code to train on our secret train set and use the learnt model to make predictions on our secret test set. 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 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. We will repeat the evaluation process described below 5 times and use the average performance parameters so as to avoid any unluckiness due to random choices inside your code in one particular trial. We will award marks based on four performance parameters

1. How fast is your `my_fit` method able to finish training (7 marks)
2. What is the on-disk size of your learnt model (after a pickle dump) (8 marks)
3. How fast is your `my_predict` method able to make predictions (10 marks)
4. How high is the test prediction accuracy offered by your model? (10 marks)

Thus, the total marks for the code evaluation is 35 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/1dRtJ_wK0jxsdac_2ES1pg57mpfQX-vCn?usp=sharing`

Validation ensures that your submit.py does work with the automatic judge and does not give errors due to file formats etc. Please use IPYNB file at the above link on Google Colab and the dummy secret train and dummy secret test set (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 etc 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 and secret test set in in the assignment package itself (see the directory called `dummy`). However, note that these are just copies of the train and test dataset we provided you. The reason for providing the dummy secret dataset is to allow you to check whether the evaluation script is working properly on Google Colab or not. Be warned that the secret dataset on which we actually evaluate your submission will be a different one. 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()` and predictions with `my_predict()`. Note that the model used in `dummy_submit.py` is a very bad model which will give poor accuracies. However, this is okay since its purpose is 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 XORRO PUF problem. However, if you do use any such resource, cite it in your PDF file. There is no penalty for using external resources so long as they are acknowledged 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 **scipy, libsvm, keras, tensorflow is also prohibited**. Use of prohibited modules and libraries for whatever reason will result in penalties. For this assignment, you should also not download any code available online or use code written by persons outside your assignment group (we will relax this restriction for future assignments). Direct copying of code from online sources or amongst assignment groups will be considered and act of plagiarism for this assignment and penalized according to pre-announced policies.

(60 marks)

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 assignment 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 `train.dat` in the assignment package that has 60000 data points, having a 72-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. 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 inside that one Python file itself using variables/functions.

4. We created two XORRO PUFs – a public one and a secret one. Using the public XORRO PUF, we created CRPs that were split into the train and test set we have provided you with the assignment package. We similarly created CRPs using the secret XORRO 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 XOR-PUFs have $R = 64, S = 4$ i.e. 72-bit challenges. However, the delays in the public an secret PUFs are not the same so a linear model that is able to predict the public XORRO PUF responses will do poorly at predicting the secret XORRO PUF responses and vice versa. Moreover, note that the secret train set is not guaranteed to

contain 60000 points and the secret test set is not guaranteed to contain 40000 points (for example, our secret test set may have 59000 points or 41000 points etc). However, the public and secret PUFs look similar otherwise so that hyperparameter choices that seem good to you during validation (e.g. `C`, `tol` etc), should work decently on the secret dataset as well.

5. 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, perform system operations), we may simply refuse to run your code and give you a zero instead (we will inspect each code file manually).

6. 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.

7. 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.

8. 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.

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

10. 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.