Indian Institute of Technology Kanpur
CS771 Introduction to Machine Learning

Instructor: Purushottam Kar
Date: November 4, 2025
Total: 200 marks

MAJOR ASSIGNMENT 1

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

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

Major Assignment Package:
`https://www.cse.iitk.ac.in/users/purushot/courses/ml/2025-26-a/material/assignments/major1-2.zip`
Deadline for all submissions: Nov 27, 2025, 9:59PM IST
Code Validation Script: `https://colab.research.google.com/drive/15CM6izv8cO0sFTvahnvscmUx5OReFDSy?usp=sharing`
Code Submission: `https://forms.gle/TnmHwcJMjLzEqoyF7`
Report Submission: on Gradescope
There is no provision for "late submission" for this major 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. Unregistered auditors cannot make submissions to this major assignment.
3. 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.
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/TnmHwcJMjLzEqoyF7`

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 (Melbo is Hiring). Melbo decided to turn entrepreneurial and has set up a production company for ML videos. To produce each video, Melbo needs to figure out how many staff members to hire. This depends on three factors, the length of the video x, the fraction of the audience interested in the topic of the video z_1 and the difficulty level of the subject matter being discussed in that video z_2. x can take any non-negative value but $z_1, z_2 \in [0, 1]$. We use notation $\mathbf{z} \overset{\text{def}}{=} [z_1, z_2] \in \mathbb{R}^2$ to encapsulate the popularity and difficulty features in vector form. It is known that the number of staff members y needed to produce a video is proportional to the length of the video x but the dependence of y on $\mathbf{z}$ is non-linear i.e., it is known that

$$y = w(\mathbf{z}) \cdot x + b,$$

where $b \in \mathbb{R}$ is a scalar bias term and $w : \mathbb{R}^2 \to \mathbb{R}$ is a non-linear function describing how the popularity and difficulty influence the dependence of staff strength on video length. This is like linear regression but where each data point gets its own linear model based on some *auxiliary* variables (in this case $\mathbf{z}$).

To get around this problem, Melbii suggests using a novel technique called *semi-parametric regression* which assumes that $w(\mathbf{z}) = \mathbf{p}^\top \phi(\mathbf{z})$ where $\phi : \mathbb{R}^2 \to \mathcal{H}$ is a feature map to some high dimensional Hilbert space corresponding to some kernel K and $\mathbf{p} \in \mathcal{H}$ is the linear model in that Hilbert space. It is known that a polynomial kernel suffices for Melbo's semi-parametric problem, but Melbo does not know which polynomial kernel to use i.e. what values to use for `degree` and `coef0`. In other words, $K(\mathbf{m}, \mathbf{n}) = (\mathbf{m}^\top \mathbf{n} + c)^d$ but c, d need tuning as hyperparameters. This suggests a model of the following form:

$$y = \mathbf{p}^\top \phi(\mathbf{z}) \cdot x + b$$

The above form shows why the technique is called semi-parametric – it has elements of parametric model since it looks like a linear model, but it has non-parametric elements as well since the weights of the linear model are not constant across data points. Melbo does not want to write a new solver to learn this model and instead wants to use the `sklearn.kernel_ridge` module that can perform kernel ridge regression. However, a drawback with that module is that it does not allow an explicit bias term i.e. it only learns models of the form

$$y = \sum_{i=1}^{n} \alpha_i \cdot K(\mathbf{p}, \mathbf{p}_i)$$

where $\mathbf{p}_i$ are feature vectors of the training points and α_i are the dual variables learnt by the module. Meblii and Melbo want your help learning the semi-parametric model using `sklearn.kernel_ridge`. To help them, you need to convert this semi-parametric problem into a purely non-parametric problem i.e. a pure kernel regression problem. Design a new kernel $\tilde{K}$ with corresponding feature vector $\psi : \mathbb{R} \times \mathbb{R}^2 \to \tilde{\mathcal{H}}$ so that for any vector $\mathbf{p} \in \mathcal{H}$, there exists a vector $\tilde{\mathbf{p}} \in \tilde{\mathcal{H}}$ such that $\tilde{\mathbf{p}}^\top \psi(x, \mathbf{z}) = \mathbf{p}^\top \phi(\mathbf{z}) \cdot x + b$. This would solve Melbo's problem since all we need to do now is invoke `sklearn.kernel_ridge` using this new kernel $\tilde{K}$. You do not need to design or compute the feature map ψ as that could be high dimensional. All you need to do is specify the kernel $\tilde{K}$ corresponding to that feature map i.e. for any $x_1, \mathbf{z}_1, x_2, \mathbf{z}_2$, we always have $\tilde{K}((x_1, \mathbf{z}_1), (x_2, \mathbf{z}_2)) = \psi(x_1, \mathbf{z}_1)^\top \psi(x_2, \mathbf{z}_2)$.

Your Data. We have provided you with training data for 4000 points and test data for 1000 points. Each train/test data point corresponds to a video, and for each data point, you have been provided with a scalar value x describing the length of that video, a vector value $\mathbf{z}$ describing the popularity and difficulty of that video, and a scalar value y corresponding to the number of staff members needed for that video. If you wish, you may create (held out/k-fold) validation sets out of this data in any way you like. (100 marks)

Problem 1.2 (Delay Recovery by Inverting a XOR 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 $\mathbf{w} \in \mathbb{R}^{k+1}$ (bias hidden as last coordinate) 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 $w_k = \beta_{k-1}$.

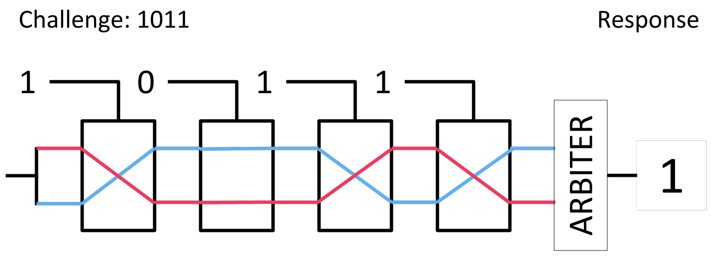

Figure 1: A simple arbiter PUF with 4 multiplexers

A XOR arbiter PUF is created by taking two arbiter PUFs, feeding them the same challenge, and taking the XOR of their output.

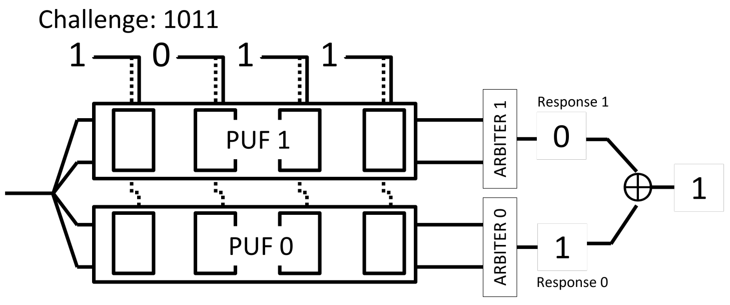

Figure 2: A XOR-Arbiter PUF with 4-bit challenges

As discussed in class, even XOR arbiter PUFs can be described using a linear model. Specifically, if $\mathbf{u} \in \mathbb{R}^{k+1}$ and $\mathbf{v} \in \mathbb{R}^{k+1}$ denote the linear models corresponding to the two arbiter PUFs, then the linear model $\mathbf{w}$ of a XOR arbiter PUF is described as $\mathbf{w} \overset{\text{def}}{=} \mathbf{u} \otimes \mathbf{v}$ where $\otimes$ is the Kronecker delta product. The Kronecker delta product is described using 3D and 2D vectors as an example. Suppose $\mathbf{p} = (p, q, r) \in \mathbb{R}^3$ and $\mathbf{s} = (s, t) \in \mathbb{R}^2$, then $\mathbf{p} \otimes \mathbf{s} = (ps, pt, qs, qt, rs, rt) \in \mathbb{R}^{3\times 2}$. Be careful that the Kronecker product is not symmetric since $\mathbf{s} \otimes \mathbf{p} = (ps, qs, rs, pt, qt, rt) \in \mathbb{R}^{3\times 2}$. You may refer to the API page for `numpy.kron` for clarification and details. Thus, if $\mathbf{u}, \mathbf{v} \in \mathbb{R}^{k+1}$ then $\mathbf{w} \in \mathbb{R}^{(k+1)^2}$.

Melbo has used leaked CRPs for a 32-bit XOR arbiter PUF to learn a linear model $\mathbf{w} \in \mathbb{R}^{1089}$ where $1089 = 33^2$. This XOR arbiter PUF has two arbiter PUFs. Let the delays of the first arbiter PUF be referred to as $a_i, b_i, c_i, d_i, 0 \leq i \leq 31$ and those of the second arbiter PUF be referred to as $p_i, q_i, r_i, s_i, 0 \leq i \leq 31$. Melbo is curious if the delays $a_i, b_i, c_i, d_i, p_i, q_i, r_i, s_i, 0 \leq i \leq 31$ can be back-calculated using only $\mathbf{w}$. 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 two vectors $\mathbf{u}, \mathbf{v}$ and non-zero scalar $c \neq 0$, we always have $\mathbf{u} \otimes \mathbf{v} = \left(\frac{1}{c} \cdot \mathbf{u}\right) \otimes (c \cdot \mathbf{v})$. Moreover, within any of the individual arbiter PUFs, say for the second one, 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 $p_i + \epsilon_i, q_i + \epsilon_i, r_i +$

$\eta_i, s_i + \eta_i, 0 \leq i \leq 63$.

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 for the XOR arbiter PUF would do. Given a linear model $\mathbf{w} \in \mathbb{R}^{1089}$, Melbo wishes to find out the $32 \times 4 \times 2 = 256$ delays that are non-negative real numbers $\hat{a}_i \geq 0, \hat{b}_i \geq 0, \hat{c}_i \geq 0, \hat{d}_i \geq 0, \hat{p}_i \geq 0, \hat{q}_i \geq 0, \hat{r}_i \geq 0, \hat{s}_i \geq 0, 0 \leq i \leq 31$ so that these delays generate the exact same linear model $\mathbf{w}$. 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 1089 real numbers that may be negative or positive or zero but will always lie between -1 and +1. Your job is to take each linear model and return $32 \times 4 \times 2 = 256$ non-negative real numbers that, if interpreted as delays for two arbiter PUFs in a XOR arbiter PUF, result in the same linear model. A distinct list of 256 delays is needed for each linear model. (100 marks)

Your Task. The following enumerates 5 parts to the question. Parts 1, 2, 4 need to be answered in the PDF file containing your report. Parts 3, 5 need 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 Hiring 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. Suppose the kernel that correctly describes the staff hiring problem is the polynomial kernel with degree d and coefficient c i.e. $y = \mathbf{p}^\top \phi(\mathbf{z}) \cdot x + b$ where $\phi(\mathbf{z}_1)^\top \phi(\mathbf{z}_2) = K(\mathbf{m}, \mathbf{n}) = (\mathbf{z}_1^\top \mathbf{z}_2 + c)^d$. What would be the corresponding kernel $\tilde{K}$ that solves the semi-parametric regression problem? In other words, if the feature map ψ converts the semi-parametric problem into a purely-non-parametric problem by assuring that $\tilde{\mathbf{p}}^\top \psi(x, \mathbf{z}) = \mathbf{p}^\top \phi(\mathbf{z}) \cdot x + b$, then what is the value of $\tilde{K}((x_1, \mathbf{z}_1), (x_2, \mathbf{z}_2)) = \psi(x_1, \mathbf{z}_1)^\top \psi(x_2, \mathbf{z}_2)$? Give detailed mathematical derivation. The kernel $\tilde{K}$ must take in two pairs of the form $(x_1, \mathbf{z}_1), (x_2, \mathbf{z}_2)$ and output a similarity value. (40 marks)

2. Which value of the degree d and coefficient c did you find to work optimally on the semi-parametric regression data provided to you? Show charts/graphs/tables showing various combinations of c, d that you tried as hyperparameters and the corresponding test results for those hyperparameter values. The test results must be reported in terms of R2 score (the `score` method available with `kernel_ridge` regressor objects can be used to calculate the R2 score on test data – see validation code). (20 marks)

3. Write code to compute your proposed kernel $\tilde{K}$. Your code should take two sets of data points $X1 \in \mathbb{R}^{n_1 \times 1}, Z1 \in \mathbb{R}^{n_1 \times 2}, X2 \in \mathbb{R}^{n_2 \times 1}, Z2 \in \mathbb{R}^{n_2 \times 2}$ and output a $n_1 \times n_2$ matrix of pairwise kernel values. These will be used to train a `kernel_ridge` regressor and evaluate on test data. Your code may use submodules of sklearn e.g. `sklearn.metrics.pairwise` to construct kernel Gram matrices. However, the use of non-linear models is not allowed. The use of other libraries such as scipy, keras, tensorflow is also not allowed. Submit code for your chosen method in `submit.py`. Note that your code must implement a method named `my_kernel()` to compute your proposed kernel. (40 marks)

4. Outline a method to take a 1089-dimensional linear model corresponding to a XOR arbiter PUF and produce 256 non-negative delays that generate the same linear model. This method should outline the equations that govern the model generation process of converting $32 \times 4 \times 2 = 256$ delays into a 1089-dimensional linear model. Then show how to invert these equations. This could be done, for example, by posing it as an (constrained) optimization problem or other ways – see hints below. (40 marks)

5. Write code to solve the XOR 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, such as decision trees, random forests, nearest neighbors, neural networks, kernel SVMs etc, is not allowed for this part (and not required either). Submit code for your chosen method in `submit.py`. Note that your code must implement a method named `my_decode()` that should take a single 1089-dimensional linear model as input and return eight 32-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). The output vectors must contain only non-negative entries (zeros are allowed). (60 marks)

Note that both methods my_kernel() and my_decode() must be provided in the same submit.py file and not in separate files. Parts 1, 2, 4 need to be answered in the (same, single) PDF file containing your report. Parts 3, 5 needs to be answered in the (same, single) Python file.

Hint for Part 1. Make sure that your new kernel corresponds to features that allow the new model $\tilde{\mathbf{p}}$ to (implicitly) absorb the bias term. ψ may depend on $\phi(\mathbf{z})$ and x.

Hint for Part 3. First try to de-Kronecker the model to get individual models for the two arbiter PUFs and then invert them to get the delays which is easier since an arbiter PUF model can be generated as a linear function of the delays. For the de-Kronecker step, you may find the equality $\mathbf{u} \otimes \mathbf{v} = \left(\frac{1}{c} \cdot \mathbf{u}\right) \otimes (c \cdot \mathbf{v})$ useful since it allows at least one of the coordinates to be fixed to a known quantity such as 0.5 or 1. 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 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, $A \in \mathbb{R}^{32\times 128}$ but for each row in A, very few 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 generated two staff hiring datasets – a public one and a secret one. Both have 4000 train points and 1000 test points, both have unidimensional x and bidimensional $\mathbf{z}$, both use the same kernel with same `degree` and `coef0` hyperparameters, i.e. the hyperparameters c, d you find working well on the public dataset should work well on the secret one as well and the kernel you propose that works well on the public dataset should work well on the secret one as well. However the two datasets use different values of α_i. We will use your `my_kernel()` method to train a `kernel_ridge` model on the secret train dataset and evaluate it on the secret test dataset. The R2 score (`https://en.wikipedia.org/wiki/Coefficient_of_determination`) will be used for evaluation. 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. How much time does your `my_kernel` method take to compute the train-train and test-train Gram matrices? (15 marks)
2. How much time does `kernel_ridge` take to train on your kernel – note that if your proposed kernel is a function of a polynomial kernel, then training time may increase with the degree d used by you. (15 marks)
3. What is the test R2 score offered by your kernel? (10 marks)

Lower is better for time evaluations whereas higher is better for R2 score. 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 XOR arbiter PUFs, each taking 32-bit challenges. The 20 XOR 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 1089-dimensional linear model. You have been given 10 of the models as a part of the assignment 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 1089-dimensional model and expect it to send us eight 32-dimensional arrays $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{s}$ representing the delays in the 32 multiplexers of the two arbiter PUFs. These delays will checked and any negative entries will be removed and then the cleaned-up delays will be used to reconstruct the 1089-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 1089-dimensional model. (30 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. (30 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/15CM6izv8cO0sFTvahnvscmUx5OReFDSy?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 assignment package itself (see the directory called `dummy`). However, note that these are just copies of the train, test datasets 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 with `my_kernel()` and `my_decode()` functions. 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 and submodules of sklearn e.g. metrics.pairwise to construct kernel Gram matrices. 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 major assignment, you should also not download any code available online or use code written by persons outside your assignment group. Direct copying of code from online sources or amongst assignment groups will be considered and act of plagiarism for this major assignment 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 major 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 4000 data points and 1000 test points for problem 1.1 in the files that have names such as `public_x_...`, `public_Z_...` and `public_y_...` containing the video length, (popularity, difficulty) and staff strength values respectively. You may use this as training data in any way to tune your hyperparameters (e.g. `degree`, `coef0`) 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 a kernel that can solve the semi-parametric problem.

3. We have provided you with ten 1089-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. degree, coef0 etc) inside that one Python file itself using variables/functions.

5. We generated two staff hiring datasets – a public one and a secret one. Both have 4000 train points and 1000 test points, both have unidimensional x and bidimensional $\mathbf{z}$, both use the same kernel with same `degree` and `coef0` hyperparameters, i.e. the hyperparameters c, d you find working well on the public dataset should work well on the secret one as well and the kernel you propose that works well on the public dataset should work well on the secret one as well. However the two datasets use different values of α_i. We will use your `my_kernel()` function to train a `kernel_ridge` regressor on our secret train set and use the learnt model to test on our secret test set.

6. We created 20 XOR arbiter PUFs, each taking 32-bit challenges and generated 1089-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 major assignment 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.