CS771 Major Assignment Report

Group: 6 Nearest Neighbours (MLR-40)

Narayan Aniruddh Somaiya (231019), Shrey Jigneshbhai Solanki (231017),
Ayush Yadav (230272), Harshita Awasthi (230463),
Shelly Singhal (230963), Jaini (230494)

Indian Institute of Technology Kanpur

Part 1: Derivation of the Kernel $\widetilde{K}$ for the Semi–Parametric Model

We are given the semi–parametric model

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

where $x \geq 0$ is the video length, $\mathbf{z} = (z_1, z_2) \in \mathbb{R}^2$ contains popularity/difficulty values, and $\mathbf{w}(\mathbf{z}) = \mathbf{p}^\top \phi(\mathbf{z})$ varies with $\mathbf{z}$. The feature map ϕ corresponds to the polynomial kernel:

$$K(\mathbf{z}_1, \mathbf{z}_2) = (\mathbf{z}_1^\top \mathbf{z}_2 + c)^d.$$

Our goal is to express this semi–parametric model as a purely non–parametric kernel regression model compatible with `sklearn.kernel_ridge`, which does not include a bias term internally. To do this, we must construct a new feature map ψ and a corresponding kernel $\widetilde{K}$ such that

$$\widetilde{\mathbf{p}}^\top \psi(x, \mathbf{z}) = \mathbf{p}^\top \phi(\mathbf{z})\, x + b.$$

Motivation for the Augmented Feature Map

Since the slope term $\mathbf{p}^\top \phi(\mathbf{z})$ must scale with x, the feature map must contain $x\phi(\mathbf{z})$. Also, because kernel ridge regression has *no explicit bias*, the constant b has to be absorbed into the feature map via a constant coordinate.

This naturally leads us to define

$$\psi(x, \mathbf{z}) = \begin{bmatrix} x\,\phi(\mathbf{z}) \\ 1 \end{bmatrix}, \qquad \widetilde{\mathbf{p}} = \begin{bmatrix} \mathbf{p} \\ b \end{bmatrix}.$$

Then

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

which matches the original model.

Kernel Computation

For two inputs $A = (x_1, \mathbf{z}_1)$ and $B = (x_2, \mathbf{z}_2)$,

$$\widetilde{K}(A, B) = \psi(x_1, \mathbf{z}_1)^\top \psi(x_2, \mathbf{z}_2) = x_1 x_2\, \phi(\mathbf{z}_1)^\top \phi(\mathbf{z}_2) + 1.$$

Using the polynomial kernel identity,

$$\phi(\mathbf{z}_1)^\top \phi(\mathbf{z}_2) = (\mathbf{z}_1^\top \mathbf{z}_2 + c)^d,$$

we finally obtain:

$$\boxed{\widetilde{K}\big((x_1, \mathbf{z}_1), (x_2, \mathbf{z}_2)\big) = x_1 x_2\, (\mathbf{z}_1^\top \mathbf{z}_2 + c)^d + 1.}$$

This reduces the semi–parametric regression problem to a standard kernel regression problem.

Part 2: Hyperparameter Selection for the Semi–Parametric Kernel Model

In this part, we tune the hyperparameters of the polynomial kernel used inside our semi–parametric kernel:

$$d \in \{1, 2, 3, 4, 5, 6\}, \qquad c \in \{0, 0.5, 1, 1.5, 2, 5\}.$$

Predictive performance is measured using the R^2 score on the test set:

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}.$$

Step 1: Hyperparameter Grid Search

For each pair (d, c) in the grid, the following steps were carried out:

1. Construct the **training Gram matrix** G_{train} using the semi-parametric kernel:

$$K\big((x_i, z_i), (x_j, z_j)\big) = x_i x_j \, (z_i^\top z_j + c)^d + 1.$$

2. Train a `KernelRidge` regressor with the precomputed kernel G_{train}.
3. Compute the R^2 score on the test set using the corresponding **test Gram matrix** G_{test} and the `score` method of the regressor.
4. Repeat the above steps n_{trials} times to obtain stable estimates, then average the R^2 score and computation times (kernel + training).

We evaluated all 36 combinations of (d, c).
Table 1 lists the obtained test scores:

Table 1: Test R^2 scores and average computation time for different (d, c) values.

Degree d	**Coefficient c**	**Test R^2 Score**	**Avg Time (s)**
1	0	0.78984	0.55
2	0	0.57140	0.56
3	0	0.44039	0.56
4	0	0.36277	0.54
5	0	0.30730	0.55
6	0	0.26561	0.52
1	0.5	0.96969	0.54
2	0.5	0.96992	0.55
3	0.5	0.96994	0.57
4	0.5	0.96990	0.55
5	0.5	0.96979	0.55
6	0.5	0.96971	0.52
1	1	0.96969	0.55
2	1	0.96992	0.52
3	1	0.96993	0.55
4	1	0.96986	0.59
5	1	0.96970	0.60
6	1	0.96958	0.59
1	1.5	0.96969	0.61
2	1.5	0.96991	0.60
3	1.5	0.96993	0.58
4	1.5	0.96984	0.60
5	1.5	0.96969	0.60
6	1.5	0.96957	0.60
1	2	0.96969	0.60
2	2	0.96991	0.61
3	2	0.96993	0.59
4	2	0.96983	0.59
5	2	0.96968	0.58
6	2	0.96956	0.56
1	5	0.96968	0.58
2	5	0.96991	0.60
3	5	0.96993	0.59
4	5	0.96982	0.57
5	5	0.96967	0.59
6	5	0.96954	0.58

A heatmap visualization of test R^2 scores is shown in Fig. 1. The numbers in parentheses indicate the *average computation time (kernel + training) in seconds* for each (d, c) pair.

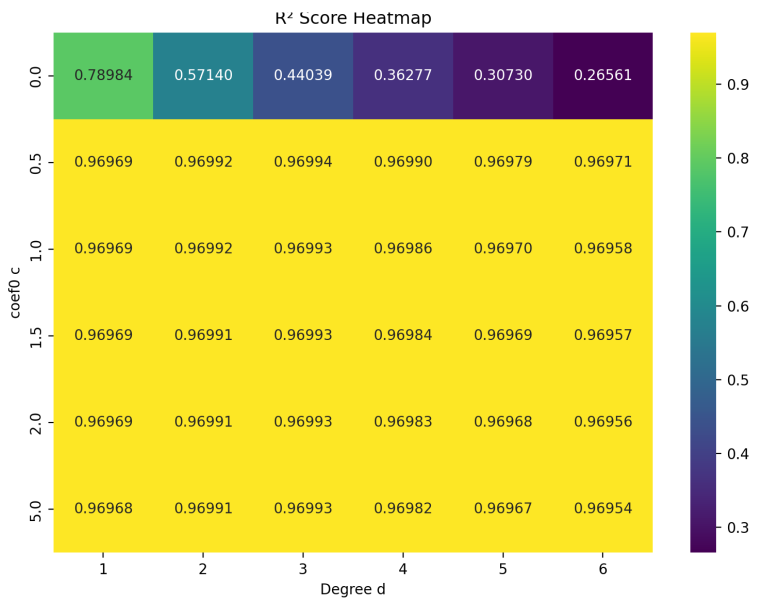

Figure 1: Heatmap of test R^2 scores for various (d, c) values. Average time (s) is shown in parentheses below each R^2.

Step 2: Model Selection

From the grid search, the highest R^2 score is achieved at $(d = 3, c = 0.5)$. However, this score is only slightly better than the score for $(d = 2, c = 1)$:

$$R^2_{\text{gain}} < 2 \times 10^{-5},$$

which is practically negligible.

At the same time, using $d = 3$ makes the kernel more complex, which increases the computation time for both the kernel and model training. Since $(d = 2, c = 1)$ gives almost the same accuracy but requires less computation, it is the better choice in practice.

Therefore, we select the **computationally efficient** configuration:

$$\boxed{d^* = 2, \qquad c^* = 1}$$

with the corresponding test performance:

$$\boxed{R^2_{\text{test}} = 0.96992}.$$

In short, we are prioritizing a combination that achieves nearly the best accuracy while keeping the model fast and simple.

Discussion

- Very small c leads to underfitting (poor performance when $c = 0$).
- Kernels with $d > 3$ show no significant improvement but higher cost.
- $(d = 2, c = 1)$ achieves strong accuracy while minimizing model complexity.

Thus, this choice offers the best trade-off between predictive performance and efficiency, making it well suited for later parts of the assignment.

Part 4: Method for Inverting the XOR Arbiter PUF

We are given a linear model

$$\mathbf{w} \in \mathbb{R}^{1089},$$

corresponding to a 2-XOR Arbiter PUF built from two independent 32-bit Arbiter PUFs. Since $(32+1)^2 = 1089$, the model can naturally be reshaped into a 33×33 matrix

$$W = \mathrm{reshape}(\mathbf{w}, 33, 33).$$

The goal is not to recover the original physical delays exactly (this is impossible due to scale and shift symmetries), but rather to construct *some* valid set of non-negative delays that generates a linear model identical to $\mathbf{w}$, as required in the assignment.

To describe our inversion method clearly, we first recall how the forward mapping from delays to the model vector is defined.

Forward Model of a Single Arbiter PUF

A 32-bit Arbiter PUF is described by four delay parameters per stage:

$$(p_i, q_i, r_i, s_i), \qquad 0 \le i \le 31.$$

Following the derivation from class, we define the delay-difference parameters

$$\alpha_i \;=\; \frac{p_i - q_i + r_i - s_i}{2}, \qquad \beta_i \;=\; \frac{p_i - q_i - r_i + s_i}{2}.$$

These quantities determine the PUF's $(32+1)$-dimensional linear model $u = (u_0, \ldots, u_{32})^\top$ via the recurrence

$$u_0 = \alpha_0, \qquad u_i = \alpha_i + \beta_{i-1} \quad (1 \le i \le 31), \qquad u_{32} = \beta_{31}.$$

Thus a single Arbiter PUF induces a linear function

$$f(C) = \mathrm{sign}(u^\top \Phi(C)),$$

where $\Phi(C)$ is the standard parity-preprocessed challenge vector.

For this assignment, we focus only on the linear model u and not on $f(C)$ itself.

Forward Model of a 2-XOR Arbiter PUF

Let $u, v \in \mathbb{R}^{33}$ denote the linear models of the two individual Arbiter PUFs. The XOR of their responses produces a combined model

$$w \;=\; u \otimes v,$$

where $\otimes$ denotes the Kronecker product. Concretely, if $u = (u_0, \ldots, u_{32})^\top$ and $v = (v_0, \ldots, v_{32})^\top$, then

$$w_{(i,j)} = u_i \, v_j, \qquad 0 \le i, j \le 32.$$

Equivalently, the 1089-dimensional vector $\mathbf{w}$ can be written as

$$\mathbf{w} = \mathrm{vec}(uv^\top),$$

so that reshaping gives

$$W = uv^\top.$$

In practice, numerical noise and non-idealities mean that W is only approximately rank–1, but we still have

$$\mathrm{rank}(W) \approx 1,$$

and this near rank–1 structure is what we exploit to invert the model.

Why the First Row and First Column Encode u and v

If $W = uv^\top$ held exactly, then

$$W_{0,j} = u_0\, v_j, \qquad W_{i,0} = u_i\, v_0.$$

In other words, the first row is proportional to $v^\top$ and the first column to u. Even when W is only approximately rank–1, these directions remain good summaries of the underlying models.

Motivated by this, we extract

$$r = W_{0,:}, \qquad c = \frac{W_{:,0}}{r_0},$$

which (up to an overall scale) serve as surrogates for $v^\top$ and u respectively. The normalization by r_0 compensates for the scalar ambiguity

$$u \otimes v = \frac{1}{\gamma}(\gamma u) \otimes v,$$

and ensures that r and c are on a consistent scale.

We can now treat r and c as approximate Arbiter-PUF model vectors and invert them stage-wise.

Inverting a Single Arbiter-PUF Model

Consider one of the extracted vectors, say

$$v = (v_0, \ldots, v_{31}, v_{32})^\top.$$

For an ideal Arbiter PUF, the recurrence described earlier can in principle be inverted:

$$\alpha_0 = v_0, \qquad \alpha_i = v_i - \beta_{i-1}, \qquad \beta_{31} = v_{32}.$$

However, the pairs (α_i, β_i) do not uniquely determine (p_i, q_i, r_i, s_i), because many different non-negative delay assignments lead to the same α_i and β_i. The assignment explicitly notes that uniqueness is *not* required: any consistent set of non-negative delays is acceptable.

To construct such delays efficiently, we interpret the first 32 coordinates of v as a baseline pattern of stagewise delay differences,

$$x = (v_0, \ldots, v_{31}),$$

and the last coordinate v_{32} as a terminal offset that mainly influences the final stage:

$$y = (0, \ldots, 0, v_{32})^\top.$$

We then form two simple candidates:

$$s = x + y, \qquad d = x - y.$$

These play the role of plausible stagewise differences,

$$X_i = p_i - q_i, \qquad Y_i = r_i - s_i.$$

We apply this construction independently to both r and c, giving us four candidate difference vectors in total.

Recovering Non-Negative Delays

Each difference vector z is converted into valid non-negative delays using

$$p_i = \max(z_i, 0), \qquad q_i = \max(-z_i, 0),$$

which ensures

$$z_i = p_i - q_i$$

with $p_i, q_i \geq 0$. Applying the same construction to another difference vector yields (r_i, s_i).

Thus each z gives us two delay vectors, and using both s and d for each of the two model directions (r, c) produces the required 8 non-negative delay vectors.

Summary

Our complete inversion procedure can be summarized as follows:

1. Use the Kronecker-product structure to reshape the XOR-PUF model into $W = \mathrm{reshape}(\mathbf{w}, 33, 33)$.
2. Extract model directions from the first row and normalized first column, which approximate $v^\top$ and u.
3. For each extracted model, form two stagewise difference candidates s and d, capturing different ways the terminal offset can influence the stages.
4. Convert each difference vector into valid non-negative delays via

$$p_i = \max(z_i, 0), \qquad q_i = \max(-z_i, 0).$$

References

- Course slides and lecture notes by Purushottam Kar, CS771, IIT Kanpur.