CS771 - Major Assignment 1&2 by team train_n_learn

Varad Shinde
251110081

Roopesh
251040633

Roshan
220918

Fahad
242040608

Hriddhit Datta
251110405

Abstract

The document presents our solutions to the questions asked in the assignment. The answers to questions 3 and 5 are given as code blocks in this assignment for reference.

1 Question 1

Lets start with the equation of the semi-parametric regression model as

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

where $x \in \mathbb{R}$ denotes the video length, $z \in \mathbb{R}^m$ denotes the video feature vector, $b \in \mathbb{R}$ is a scalar bias, and $w(z)$ is an unknown non-linear function. Lets assume that $w(z)$ lies in an RKHS with feature map $\phi(z)$, so that

$$w(z) = p^\top \phi(z),$$

and the inner product in this RKHS corresponds to the polynomial kernel

$$K_{\text{poly}}(z_1, z_2) = \langle \phi(z_1), \phi(z_2) \rangle = (z_1^\top z_2 + c)^d.$$

Step 1: Expanding the Model

Later substituting $w(z) = p^\top \phi(z)$ into the original equation we get:

$$y = (p^\top \phi(z)) \cdot x + b.$$

Since x is scalar, we can write

$$y = p^\top \left(\phi(z)\, x\right) + b.$$

Step 2: Incorporating the Bias Term

Kernel ridge regression typically does not include an explicit bias term. So we use the hint provided to rewrite as follows:

$$y = \underbrace{p^\top (\phi(z)x)}_{\text{variable part}} + \underbrace{b \cdot 1}_{\text{bias part}}.$$

Define the augmented parameter vector

$$\tilde{p} = \begin{bmatrix} p \\ b \end{bmatrix}, \qquad \Psi(x, z) = \begin{bmatrix} \phi(z)\, x \\ 1 \end{bmatrix}.$$

Further we will get,

$$y = \tilde{p}^\top \Psi(x, z),$$

this way semi-parametric model becomes fully linear in the transformed feature map Ψ.

Preprint. Under review.

Step 3: Computing the New Kernel

The kernel which is associated with Ψ is

$$\tilde{K}\big((x_1, z_1), (x_2, z_2)\big) = \langle \Psi(x_1, z_1), \Psi(x_2, z_2) \rangle.$$

When we use the definition of Ψ we get,

$$\tilde{K} = \left\langle \begin{bmatrix} \phi(z_1)\, x_1 \\ 1 \end{bmatrix}, \begin{bmatrix} \phi(z_2)\, x_2 \\ 1 \end{bmatrix} \right\rangle = (\phi(z_1)\, x_1)^\top (\phi(z_2)\, x_2) + 1.$$

Since we know that x_1, x_2 are scalars,

$$(\phi(z_1)\, x_1)^\top (\phi(z_2)\, x_2) = (x_1 x_2) \cdot \phi(z_1)^\top \phi(z_2).$$

Step 4: Substituting the Polynomial Kernel

Using $\phi(z_1)^\top \phi(z_2) = (z_1^\top z_2 + c)^d$, we will obtain

$$\tilde{K}\big((x_1, z_1), (x_2, z_2)\big) = (x_1 x_2)\, (z_1^\top z_2 + c)^d + 1.$$

Final Result

Finally we get transformed semi-parametric regression problem correlating with the kernel ridge regression according to the following kernel.

$$\boxed{\tilde{K}\big((x_1, z_1), (x_2, z_2)\big) = (x_1 x_2)\, K_{\text{poly}}(z_1, z_2) + 1}.$$

This kernel is computed by taking the elementwise product of the polynomial kernel matrix on Z with the outer product of video lengths X, and then adding 1 to all entries.

2 QUESTION 2

2.1 Hyperparameter Tuning Strategy

To determine the optimal hyperparameters for the polynomial kernel $K(m, n) = (m^\top n + c)^d$, we performed an exhaustive grid search over the degree $d \in \{1, 2, 3, 4, 5\}$ and the coefficient $c \in \{0, 0.1, 0.5, 1, 2, 5, 10, 15\}$. We split the training dataset into 80% training set and a 20% validation set to evaluate the R^2 score for each combination. The R2 scores achieved with all the pairs of d and e are documented in the table 1

2.2 Initial Observations and Graph Analysis

Our experiments revealed that the model performance is highly sensitive to the coefficient c (coef_0) when $c = 0$, but stabilizes quickly for any $c > 0$.

- **Effect of Degree (d):** As shown in Figure 2, lower degrees as ($d = 1, 2$) shows robust performance across a wide range of coefficients. Higher degrees ($d = 4, 5$) have shown some instability at $c = 0$ (yielding very low R^2 scores), though they have recovered as we icnreased $c > 0$.
- **Effect of Coefficient (c):** Setting $c = 0$ we saw that it resulted in poor performance for non-linear kernels. But when we added even a small bias (e.g., $c = 0.1$) it improved the score a lot.

When we approached with fully accuracy based results (based on the graphs alone) as seen in 1, the highest validation R^2 score was achieved at:

Degree (d) = 1 and **Coefficient (c) = 0.003**

Though we had a higher R^2 score in validation script (on G_test) with degree = 2 and coefficient = 0.1 as **0.9699458448220089** and R^2 score in the grid search as **0.97032**.

Still we chose degree = 1 and coefficient = 0.003 with R^2 score **0.9698386992189694** (on validation script, that is (on G_test)), because of lesser degree which provides higher stability .

2.3 Reasoning for chosen solution in Trade-off and Final Decision

NOTE : We obesrved that when we connect with colab and run the code, it gives minimal time within certain ranges but during the same login when we restrat the session, the time shoots up. So the results we will be mentioning are the first session login and the very first run of the validation script

While Degree 2 appeared optimal in terms of raw accuracy in the validation script (on G_test), later we faced a tradeoff between computational cost and predictive performance.

The evaluation criteria for this assignment emphasized more on getting lesser execution time of the kernel computation. During validation benchmarking, we observed the following:

- **Degree 1 ($c = 0.003$):** Achieved an R^2 of 0.9698386992189694 (on the validation script (on G_test)) with a kernel computation time of $\approx \mathbf{0.2714522823999971s}$.
- **Degree 2 ($c = 0.1$):** Achieved an R^2 of ≈ 0.9699 (on the validation script (on G_test)) with a kernel computation time of $\approx \mathbf{0.5079177448002156s}$.

We claim that degree 2 has contributed very little in accuracy ($+0.0001$), while it increases the kernel computation time by approximately 50%. If it had been a very huge dataset having degree 2 would have increased the time even more, so we prioritized scalability and efficiency.

Final Decision: We selected **Degree (d) = 1** and **Coefficient (c) = 0.003** for the final model which gives the R^2 score as **0.9698386992189694**, t_kernel as **0.2714522823999971s** and t_train as **1.2986371638000036s** on the google colab validation script. This linear kernel provides identical predictive power to the quadratic kernel while significantly reducing the computational overhead, ensuring the most optimal score according to the evaluation metrics.

2.4 Performance Visualization

The following plots illustrate the R^2 scores for various combinations of c and d observed during the grid search (training time).

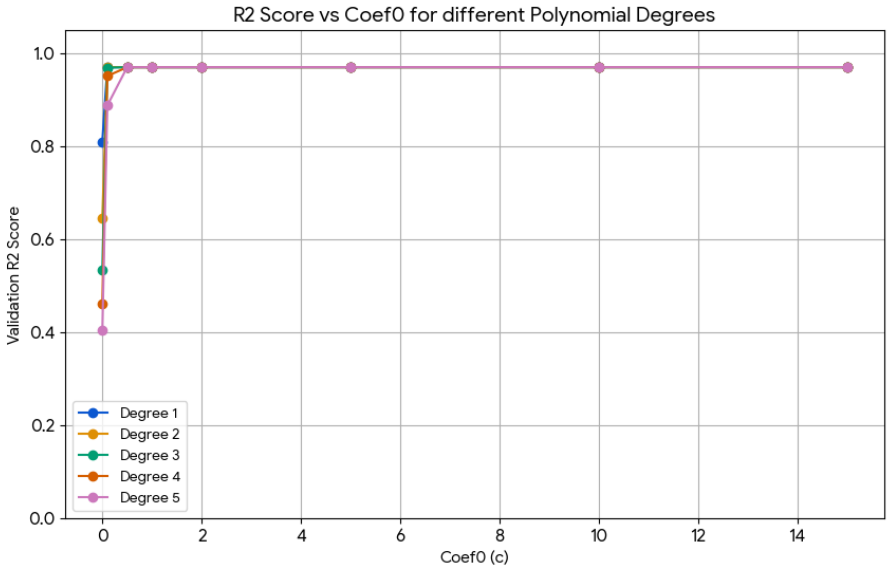
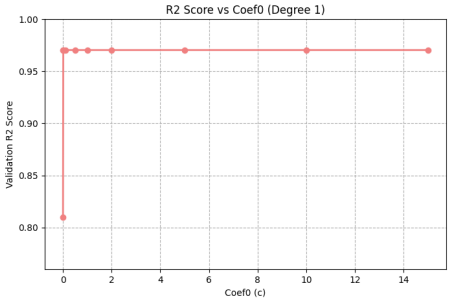
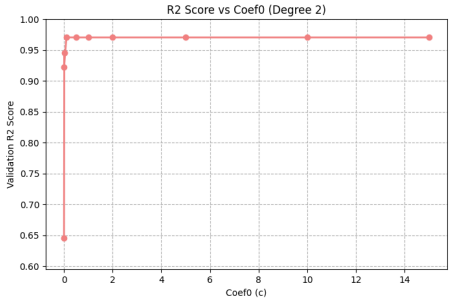


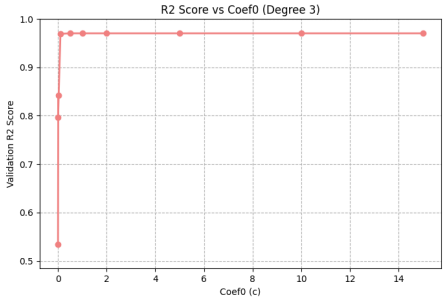
Figure 1: Combined comparison of R^2 scores across all degrees ($d = 1$ to 5). Note the sharp drop at $c = 0$ for higher degrees.



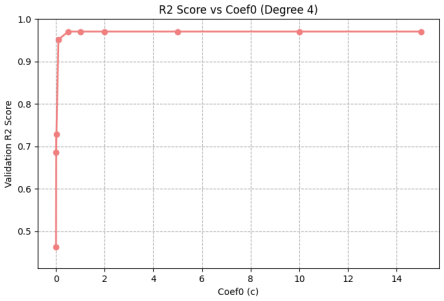
(a) Degree $d = 1$(better, optimized solution)



(b) Degree $d = 2$



(c) Degree $d = 3$



(d) Degree $d = 4$

Figure 2: Individual performance charts for degrees 1 through 4. These charts are from the training period.

Degree (d)	$c = 0$	$c = 0.003$	$c = 0.1$	$c = 0.5$	$c = 1$	$c = 2$	$c = 5$	$c = 10$	$c = 15$
1	0.8101	**0.97038**	0.9702	0.9702	0.9702	0.9702	0.9702	0.9702	0.9702
2	0.6451	0.92196	*0.97032*	0.9702	0.9702	0.9702	0.9702	0.9702	0.9702
3	0.5346	0.79688	0.9691	0.9702	0.9702	0.9702	0.9702	0.9702	0.9702
4	0.4624	0.68498	0.9512	0.9702	0.9702	0.9702	0.9702	0.9702	0.9702
5	0.4036	0.60353	0.8878	0.9702	0.9702	0.9702	0.9702	0.9702	0.9702

Table 1: This table reports the $R^2 score$ achieved by various combinations during the grid search we did for finding the best hyperparameter. Here we can see while training the combination d=1 and c=0.003 has given the highest R^2 score, but during testing (validation on google colab) we observed d=2 and c=0.1 giving the best R^2 scores. As per our reasoning and explaination given above, we conclude that most optimized combination is **degree = 1 and c = 0.003, as it gives 50 % less computational overhead along with on par R2 score.**

3 QUESTION 3

3.1 Implementation of Proposed Kernel

Mathematical derivation was done in Part 1 and the best hyperparameters were decided in Part 2, the final kernel $\tilde{K}$ was implemented using efficient vectorized operations in Python. The code utilizes `numpy` for the linear component and `sklearn.metrics.pairwise` for the polynomial component.

```
import numpy as np
from sklearn.metrics.pairwise import polynomial_kernel

def my_kernel( X1, Z1, X2, Z2 ):

    # Since d=1, K_poly(Z) = gamma * (Z1 @ Z2.T) + coef0
```

```
    # We use gamma=1.0, coef0=0.1
    K_Z = Z1 @ Z2.T + 0.003

    # Compute Linear Kernel for X (Outer Product)
    # Matrix multiplication is the fastest way to compute the outer
    product
    K_X = X1 @ X2.T

    # Combine
    # Element-wise multiplication (*) followed by scalar addition
    K_final = (K_X * K_Z) + 1

    return K_final
```

Listing 1: Python implementation of the semi-parametric kernel function

The implementation relies on `sklearn.metrics.pairwise.polynomial_kernel` for the non-linear component. The linear component is computed using standard NumPy matrix multiplication (`@`), wiht the aim so that $O(N^2)$ we get lesser time.

4 QUESTION 4

4.1 XOR Arbiter PUR model

A single Arbiter PUF with $k = 32$ stages is defined by the delays (p_i, q_i, r_i, s_i) for $i \in \{0, \ldots, 31\}$. The following variables are then defined:

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

The linear weights $u \in \mathbb{R}^{33}$ for a single PUF:

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

Therefore, the combined linear model $w \in \mathbb{R}^{1089}$ for an XOR Arbiter PUF which consists of two independent PUFs with weight vectors u and v is the Kronecker product:

$$w = u \otimes v.$$

4.2 Motivation

The main challenges to de-kroneckerization of the XOR Arbiter PUF model weight matrix are the following:

- **Scaling Ambiguity:** $u \otimes v = (\frac{1}{c}u) \otimes (cv)$. Even after scaling and descaling u and v the product remains the same.
- **Delay Redundancy:** $p - q = (p + k) - (q + k)$. Equally translating the delays would result in the same model weights.

For this assignment we just have to produce one set of non-negative delays that produce the same XOR Arbiter PUF weight matrix.

A direct non-linear optimization approach would prove to be expensive as well as require complicated projection operations or barrier functions. It would also require hyperparameter tuning and could end up reaching a suboptimal local minima due to poor initialization. On the other hand an approach that avoids optimization would not just be faster computationally faster but also more stable, guaranteeing valid non-negative delays.

Time complexity for optimization based approaches:

$$O(N_{iter} \times K)$$

where N_{iter} is the number of iterations for optimization and K is the number of parameters.

$$\min_{u,v} ||w - u \otimes v||^2$$
$$s.t.\ \ p, q, r, s \geq 0$$

The Kronecker product of two column vectors u and v can be reshaped to be an outer product of the two.

$$w_{(1089\times 1)} = u_{(33\times 1)} \otimes v_{(33\times 1)} = \begin{bmatrix} u_0v_0 \\ u_0v_1 \\ ... \\ u_0v_{32} \\ u_1v_0 \\ ... \\ u_{32}v_{32} \end{bmatrix}$$

$$vu^\top = \begin{bmatrix} u_0v_0 & u_0v_1 & ... & u_0v_{32} \\ u_1v_0 & u_1v_0 & ... & u_1v_{32} \\ & & ... & \\ u_{32}v_0 & u_{32}v_1 & ... & u_{32}v_{32} \end{bmatrix}$$

This outer product is a rank-1 matrix due to each ith row containing the same scalar factor u_i multiplied to the vector v. According to the Eckart-Young-Mirsky theorem SVD provides the optimal rank-1 approximation of a matrix in terms of the Frobenius norm. This allows us to recover the vectors u and v directly.

$$W = U\Sigma V^\top$$

Due to W being a rank-1 matrix, all but one singular value for the decomposition would be non-zero (rest would be approximately zero, but generally not zero due to noise). The first column of U becomes the best approximation for u, and the first row of $V^\top$ becomes the best approximation for v. The singular values σ_1 become the scaling factor.

Finally we have an under-determined system of linear equations (u - 33 knowns, α, β - 64 intermediate unknowns and 128 unknown delays). Again, instead of attempting a complicated optimization based approach we can simplify the problem by setting some unknowns to be 0 in a way to make the systems determined and obtaining one valid set of delays (and intermediate values). The process is detailed in section 4.3 step 2.

This approach is mainly bottle-necked by SVD on the XOR arbiter PUF weight matrix. It approximately is:

$$O(M^3)$$

where $M \times M$ is the dimension of the reshaped weight matrix.

4.3 Inversion

We perform the recovery of the 256 non-negative delays of the 2 32-bit Arbiter PUFs from the 33×33 XOR Arbiter PUF matrix (created from the 1089 given weights) in 3 steps:

Step 1: Singular Value Decomposition

As shown, $w = u \otimes v$ implies that reshaping the vector w into a matrix $W \in \mathbb{R}^{33\times 33}$ results in W being a matrix of rank 1 (or very close to it, considering numerical noise). Specifically, $W = uv^\top$.

To recover u and v, we perform a Singular Value Decomposition (SVD) on W:

$$W = U\Sigma V^\top \approx \sigma_1 \mathbf{x}\mathbf{y}^\top.$$

We approximate u and v using the principal singular vectors:

$$u \approx \sqrt{\sigma_1} \cdot \mathbf{x}, \quad v \approx \sqrt{\sigma_1} \cdot \mathbf{y}.$$

Step 2: Non-Negative Delay Recovery

Before we get valid non-negative delay values, we have to construct values for α and β from the values for u and v that we got from SVD.

We currently have:

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

This system is obviously under-determined (infinitely many possible values for α_i and β_{i-1} for $(1 \leq i \leq 31)$. To simplify the search, we can arbitrarily set $\beta_0, \beta_1, ..., \beta_{30} = 0$, making the system determined and getting valid values for α and β.

We now have the following two equations:

$$\begin{cases} p - q = \alpha + \beta \\ r - s = \alpha - \beta \end{cases}$$

Let $X = \alpha + \beta$ and $Y = \alpha - \beta$. We need to find $p, q \geq 0$ such that $p - q = X$.

- If $X > 0$, we set $p = X$ and $q = 0$.
- If $X \leq 0$, we set $p = 0$ and $q = |X|$.

This ensures that $p, q \geq 0$ and $p - q = X$. Identical logic is used for r and s using Y.

5 QUESTION 5

5.1 Code Implementation for Delay Recovery

The following code implements a solver to recover one set of correct possible delays for the problem described in Question 4 in python. The function "my_decode" takes the linear a 1089 dimensional vector and returns two sets of 4 delays for each arbiter PUF.

```
import numpy as np

def my_decode(w):
    # Reshaping 1089x1 vector to 33x33 matrix
    W_mat = w.reshape((33, 33))

    # Performing SVD on rank-1 matrix
    U, S, Vh = np.linalg.svd(W_mat)

    # Getting best approximation for u and v
    scale = np.sqrt(S[0])
    u = U[:, 0] * scale
    v = Vh[0, :] * scale

    def solve_single_apuf(model_weights):
        # Recovering values for alpha and beta from model weights
    after setting most of beta to be 0
        alpha = model_weights[:32]
        beta = np.zeros(32)
        beta[31] = model_weights[32]

        # Getting differences
        diff_pq = alpha + beta
        diff_rs = alpha - beta

        # Ensuring non-negativity of delay values
        p_vec = np.maximum(diff_pq, 0)
        q_vec = np.maximum(-diff_pq, 0)
        r_vec = np.maximum(diff_rs, 0)
```

```
        s_vec = np.maximum(-diff_rs, 0)

        return p_vec, q_vec, r_vec, s_vec

    # Solve for both APUFs
    a, b, c, d = solve_single_apuf(u)
    p, q, r, s = solve_single_apuf(v)

    return a, b, c, d, p, q, r, s
```

Listing 2: Python implementation of the XOR PUF inversion algorithm

This solution took around 0.25 milliseconds on the public dataset and the mean euclidean distance of the predicted and the ideal model weights was close to 0 (4.4×10^{-16})

5.2 Intermediate Steps Explanation

The solution is purely feedforward:

- **Matrix Reshaping:** The weight vectors are reshaped as matrices (dimensions changed from 1089×1 to 33×33).
- **SVD Decomposition:** `np.linalg.svd` decomposes the matrix. Taking only the first component ($U[:, 0]$ and $Vh[0, :]$) corresponding to the largest singular value separates the XOR PUF into two Arbiter PUFs.
- **Variable Recovery:** `solve_single_apuf` calculates `diff_pq` and `diff_rs`, representing the required net delay difference for each stage.
- **Rectification:** `np.maximum(x, 0)` function assigns the magnitude of the difference to either p or q depending on the sign, ensuring strict non-negativity for all delays.

6 Supplementary Material