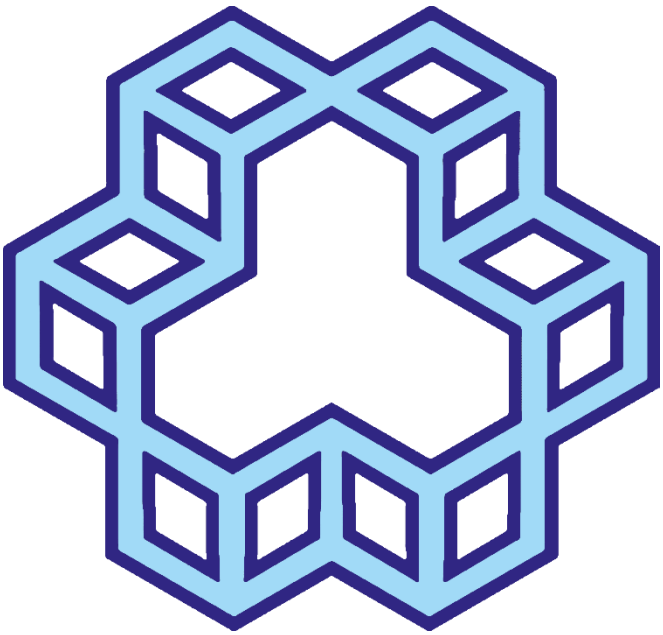

K.N. Toosi University of Technology

Students: Mostafa Latifian – 40122193

Parsa Alaviniko – 40120993

Professor: Dr. Mahdi Aliyari-Shoorehdeli

Course: Fundamental of Intelligent Systems

MP – 2

Question 6

Question 7

Google Drive Folder

Table of Content

Question 1 ..... **1**
Part A: Hard-Margin Primal SVM 1
Part B: Margin Width 1
Part C: Dual Derivation 1
Question 2 ..... **3**
Part A: The Primal Soft-Margin Form with Slack Variables 3
Part B: Minimum of the Equation 3
Part C: The Effects of the $\boldsymbol{C}$ Parameter 4
Effect on Generalizability 4
Effect on Number of Support Vectors 4
Question 3 | Section 1 ..... **5**
Part I: Soft Margin 5
Part II: Dual Derivation 5
Part III: Difference in the Bounds of $\boldsymbol{\alpha i}$ & $\boldsymbol{l1}$ 5
Question 3 | Section 2 ..... **7**
Part I: Valid Kernel and Its Relation to Feature Map 7
Part II: Validity of RBF & Polynomial Kernel 7
Part III: An Intuitive Interpretation of KKT Conditions for SVM 8
Question 4 ..... **10**
Part I: The Maximum-Margin Hyperplane and Decision Boundary 10
Part II: Manual calculation of the coefficients $\boldsymbol{\omega}$ & $\boldsymbol{b}$ 10
Part III: Support Vectors and Margin 10
Question 5 ..... **12**
Part I: Selecting an Appropriate Neural Network Model 12
Part II: Global Activation Fails 13
Question 6 | Section 1 ..... **14**
1.1. System Identification with RBF Neural Networks 14
1.1.1. Generate Data 14
1.1.2. Preprocess Data 14
1.1.3. Problem Approximation Function 14
1.2. Static RBFNN Implementation 15

1.2.1. Role of the Neuron Center ($\boldsymbol{\mu k}$) and Neuron Width ($\boldsymbol{\sigma k}$) 15
1.2.2. Define Gaussian RBF 16
1.3. Training via Linear Least Square 16
1.3.1. Training Code for RBF 17
1.3.2. Predicted vs. True Output on Training Set 18
1.3.3. Predict vs. True Outputs on Test Set 19
1.3.4. LLS Training Faster Than GD 20
1.3.5. Effect of the Number of Centers $\boldsymbol{K}$ Test Performance 21
1.3.7. Effect of the Spread $\boldsymbol{\sigma}$ on Test Performance 22
1.3.8. Effect of Very Small or Very Large $\boldsymbol{\sigma}$ 23
1.3.9. Challenge in Selecting the Best Network Structure 24
1.4. Traying to Solve Challenges 25
1.4.1. Training the Static RBFNN with $\boldsymbol{L2}$–Regularized LLS 25
1.4.2. Role of the Regularization Parameter $\boldsymbol{\lambda}$ 28
1.5. Nearest Neighbor Strategy 29
1.5.1. Training the RBF Network Using K-Means for Center Selection 29
1.5.2. Spread Computation Using the Nearest–Neighbour Strategy 31
1.5.3. K-means Clustering Superior to Random Center Selection 34
Question 6 | Section 2 36
2.1. Sequential Learning and Growth Criteria 36
2.1.1 Writing a Function to Check the Need for Creating a New Neuron 36
2.1.2. The Necessity of Simultaneously Using Novelty and Error Criteria 37
2.2 Implementation of Adaptive RBFNN with Pruning Strategy 39
2.2.1 Implementation of an Adaptive RBF Network Integrating 39
2.2.2. Final Report and Performance Evaluation of the Adaptive RBFNN 43
2.2.3. Hidden Neuron Numbers 44
2.2.4. Competition of Neurons Number 45
2.2.5. Advantages and Disadvantages 46
Question 7 47
7.1. Data Preparation and 2D PCA 47
7.2. SVS Training 48
7.3. Various $\boldsymbol{C}$ Results 49

7.4. Evaluation Plots 50

Question 1

Part A: Hard-Margin Primal SVM

For a training data point $\{(x_i, y_i)\}_{i=1}^n$ with $x_i \epsilon R^d$ label $y_i \epsilon \{+1, -1\}$, the standard hard-margin SVM formulation is:

$$min \frac{1}{2} ||\omega||^2$$

$$subject\ to\ y_i(\omega^T x_i + b) \geq 1, \qquad i = 1, 2, \dots, n$$

Here, the objective is to minimize the squared norm of the weight vector ω subject to the constraint that all data points are separated with a minimum distance of 1 from the decision hyperplane. The constant 1 is typically chosen because it fixes the scale of ω and b if any other value were chosen, the corresponding scale would change accordingly.

Part B: Margin Width

The decision hyperplanes for the positive and negative classes are respectively defined as follows:

- Positive Hyperplanes Class: $\omega^T x_i + b = 1$
- Negative Hyperplanes Class: $\omega^T x_i + b = -1$

These two hyperplanes are parallel, and the distance between two parallel hyperplanes $\omega^T x_i + b = c_1$ and $\omega^T x_i + b = c_2$ is equal to $\frac{|c_1 - c_2|}{||\omega||}$ and in these equations we have $c_1 = +1$ and $c_2 = -1$ so:

$$Margin\ Width = \frac{|1 - (-1)|}{||\omega||} = \frac{2}{||\omega||}$$

Part C: Dual Derivation

To obtain the dual form, we use the method of Lagrange multipliers.

First, we incorporate the constraints into the objective function. For each constraint, we introduce a Lagrange multiplier ($\alpha_i \geq 0$):

$$L(\omega, b, \alpha) = \frac{1}{2} \parallel \omega \parallel^2 - \sum_{i=1}^{n} \alpha_i [y_i(\omega^\top x_i + b) - 1$$

To find the minimum of L with respect to the primal variables (ω, b) we take the derivative.

- Derivative respect to ω: $\frac{\partial L}{\partial \omega} = \omega - \sum_{i=1}^n \alpha_i y_i x_i = 0 \Rightarrow \omega = \sum_{i=1}^n \alpha_i y_i x_i$
- Derivative respect to b: $\frac{\partial L}{\partial b} = -\sum_{i=1}^n \alpha_i y_i = 0 \Rightarrow \sum_{i=1}^n \alpha_i y_i = 0$

By substituting ω back into L, we obtain the dual function, which depends only on the variables α. The result is the following expression:

$$L(\alpha) = \frac{1}{2}\left(\sum_i \alpha_i y_i x_i\right)^T \left(\sum_i \alpha_j y_j x_j\right) - \sum_i \alpha_i y_i \left(\sum_i \alpha_i y_i x_i\right)^T x_i - b\sum_i \alpha_i y_i + \sum_i \alpha_i y_i x_i + ..$$

Through simplification, we arrive at the final dual form.

$$D(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j,$$

$$subject\ to\ \alpha_i \geq 0, i = 1,2,\dots,n \rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0$$

Therefore, the dual problem is formulated as follows:

$$Maximize \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$subject\ to: \alpha_i \geq 0, \sum_{i=1}^{n} \alpha_i y_i = 0$$

From the KKT conditions and the complementary slackness rule, the case α_i is obtained. The KKT conditions include:

$$\alpha_i \geq 0, \qquad y_i(w^\top x_i + b) - 1 \geq 0$$

And the complementary slackness condition:

$$\alpha_i(y_i(w^\top x_i + b) - 1) = 0$$

From the complementary slackness condition, two possible cases can exist for each i:

- If $\alpha_i > 0$ therefore necessarily $y_i(w^\top x_i + b) - 1 = 0$ Meaning that sample lies exactly on the margin boundary (it is a support vector).
- If $y_i(w^\top x_i + b) - 1 > 0$ Then, according to complementary slackness, we must have $\alpha_i = 0$.

Therefore, only the samples that lie exactly on the hyperplanes $w^\top x + b = \pm 1$, those on the margin have a positive value for α_i, for all other samples $\alpha_i = 0$ and in the summation $\sum_i \alpha_i y_i x_i$ they play no role. This is precisely why these samples are called support vectors. They are the only points that determine the final model and consequently, the decision boundary.

Question 2

Part A: The Primal Soft-Margin Form with Slack Variables

In hard-margin SVM, it was assumed that the data is perfectly linearly separable. However, in the real world, data often has noise or is not perfectly separable. To solve this problem, slack variables $\xi_i \geq 0$ are introduced, which allow some samples to intrude into the margin or even lie on the wrong side of the hyperplanes.

Primal Form:

$$min\frac{1}{2}||\omega||^2 + C\sum_{i=1}^{n}\xi_i$$

Constrains:

1. $y_i(\omega^T x_i + b) \geq 1 - \xi_i, \quad \forall i = 1,2,\dots,n$
2. $\xi_i \geq 0, \forall i = 1,2,\dots,n$

The term $C\sum\xi_i$ is a penalty for classification errors. The parameter C controls how much importance we give to classification errors.

Part B: Minimum of the Equation

We want to show that the previous form is equivalent to minimizing the following expression:

$$\frac{1}{2}\parallel w \parallel^2 + C\sum_i \max(0, 1 - y_i f(x_i))$$

$$f(x_i) = \omega^T x_i + b$$

By looking at the first constraint in part (A):

$$y_i(\omega^T x_i + b) \geq 1 - \xi_i$$

We rewrite this inequality in terms of ξ_i:

$$\xi_i \geq 1 - y_i(\omega^T x_i + b)$$

Also, according to the second constraint, we have:

$$\xi_i \geq 0$$

Since in the objective function (part A) we are minimizing the sum of ξ_i, the optimal value for each ξ_i will be the smallest value that satisfies both of the above conditions. Therefore, $\xi_i$must be equal to the maximum of its two lower bounds:

$$\xi_i = \max(0,1 - y_i(\omega^T x_i + b))$$

$$\xi_i = \max\big(0, 1 - y_i f(x_i)\big) = \mathrm{l}_{\text{hinge}}(y_i, f(x_i))$$

Now we substitute this optimal value of ξ_i directly into the original objective function:

$$\frac{1}{2} \| \omega \|^2 + C \sum_{i=1}^{n} \xi_i \;=\; \frac{1}{2} \| \omega \|^2 + C \sum_{i=1}^{n} max\,(0, 1 - y_i f(x_i))$$

Part C: The Effects of the C Parameter

Effect on Generalizability

Small value of C:

- Meaning: The penalty for errors (ξ_i) is low. This means the model is allowed to make more errors on the training data.
- Result: The model prefers to have a wider margin even if some points are misclassified.
- Generalizability: It usually leads to better generalizability (prevents overfitting), because the model is less sensitive to noise and outliers. However, if C is too small, it may lead to underfitting.

Large value of C:

- Meaning: The error penalty is very high. The model tries to correctly classify all training samples.
- Result: The margin becomes narrower in order to "squeeze" between the data points and separate them.
- Generalizability: The risk of overfitting increases, because the model memorizes the shape of the training data and may not perform well on new data.

Effect on Number of Support Vectors

Small value of C:

- When C is small, the margin is wider. The wider the margin, the higher the probability that more points will fall inside the margin area or on the wrong side of it. In soft-margin SVM, any point with $\xi_i > 0$ or lying exactly on the margin boundary is a support vector.
- Small $C \rightarrow$ Wider margin $\rightarrow$ More support vectors.

Large value of C:

- When C is large, the margin becomes narrow to avoid errors. Fewer points get caught inside this narrow margin.
- Small $C \rightarrow$ Narrower margin $\rightarrow$ Fewer support vectors.

Norm $\boldsymbol{l_1}$ and The Difference Between Dual & $\boldsymbol{l_1}$

Question 3 | Section 1

Part I: Soft Margin

One of the standard formulations for l_2-regularized slack is written as follows:

$$min\frac{1}{2}\left|\left|\omega\right|\right|^2+\frac{C}{2}\sum_{i=1}^{n}\xi_i$$

$$y_i(\omega^T x_i+b)\geq 1-\xi_i,\quad \forall i=1,2,\dots,n$$

Part II: Dual Derivation

We write the Lagrangian with Lagrange multipliers $\alpha_i \geq 0$ for the margin constraints and $\mu_i \geq 0$ for the $\xi_i \geq 0$ constraints.

$$\mathcal{L}(\omega,b,\xi,\alpha,\mu)=\frac{1}{2}\parallel\omega\parallel^2+\frac{C}{2}\sum_i\xi_i^2-\sum_i\alpha_i[y_i(\omega^{\mathrm{T}}x_i+b)-1+\xi_i]-\sum_i\mu_i\xi_i$$

Stationarity conditions with respect to:

1. $\frac{\partial\mathcal{L}}{\partial\omega}=0\Rightarrow\omega=\sum_{i=1}^{n}\alpha_i y_i x_i$
2. $\frac{\partial\mathcal{L}}{\partial b}=0\Rightarrow\sum_{i=1}^{n}\alpha_i y_i=0$
3. $\frac{\partial\mathcal{L}}{\partial\xi_i}=0\Rightarrow C\,\xi_i-\alpha_i-\mu_i=0\Rightarrow C\,\xi_i=\alpha_i+\mu_i$

From the complementary slackness condition, we have $\mu_i\xi_i=0$.Now, for each i, two cases arise:

1. If $\xi_i>0$ then $\mu_i=0$ and therefore $C\xi_i=\alpha_i$ or $\alpha_i=C\xi_i$.
2. If $\xi_i>0$ then $\alpha_i+\mu_i=0$ which given $\alpha_i,\mu_i\geq 0$ implies $\mu_i=0,\alpha_i=0$.

Now, substituting ω and eliminating ξ, μ in the case $\xi_i>0$ into the Lagrangian and performing simplifications yields the dual result. A compact form of the dual is expressed as:

$$\max\sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j-\frac{1}{2C}\sum_{i=1}^{n}\alpha_i^2$$

$$Subject\ to\ \sum_{i=1}^{n}\alpha_i y_i=0,\qquad \alpha_i\geq 0\ \forall i$$

Part III: Difference in the Bounds of α_i & l_1

1. Difference in bounds
   - In the l_1 case (standard): We have a box constraint, $0\leq\alpha_i\leq C$. That is, the influence of each sample is bounded.
   - In the l_2 case: As we saw in the dual derivation, there is no upper bound for α_i, and the only constraint is $\alpha_i\geq 0$.

2. Effect on the number and weight of support vectors
    - In l_2, since $\xi_i = \frac{\alpha_i}{C}$, for any misclassified sample, the value of α_i must be positive. This means all misclassified samples become support vectors.
    - Since α_i has no upper bound, outliers located far from the decision boundary can have very large α_i values (because their penalty grows quadratically). This makes the l_2 model more sensitive to noise, as the decision boundary can be heavily pulled toward outliers.

If we want $l_1$minimization (sparser model — lower prediction computational cost primarily due to fewer SVs), we can choose it; if we want a softer, more continuous, and more stable solution in the presence of noise, l_2 is a good choice, but it usually has more SVs and higher prediction cost.

In the l_2 case, the number of support vectors is typically larger, and the weight α of outliers can become very large.

Question 3 | Section 2

Part I: Valid Kernel and Its Relation to Feature Map

A function $k: X \times X \to \mathbb{R}$ is called a valid kernel (positive semi-definite kernel) if for every finite set $\{x_1, \dots, x_n\} \subset X$ the Gram matrix K defined by $K_{ij} = k(x_i, x_j)$ positive semi-definite, for any vector $c \in \mathbb{R}^n$, we have $c^{\mathrm{T}} K c \geq 0$.

Equivalently Mercer–Hilbert theory, k is a valid kernel so there exists a Hilbert feature space H and a feature map $\phi: X \to H$ such that:

$$k(x, x') = \langle \phi(x), \phi(x') \rangle_H$$

In other words, a valid kernel is precisely a function that can be regarded as an "inner product" in some feature space. This relationship allows us to use k to compute inner products without explicitly constructing ϕ.

Part II: Validity of RBF & Polynomial Kernel

To prove this, we use the following closure properties:

- The sum of two valid kernels is valid.
- The product of two valid kernels is valid.
- Multiplying a valid kernel by a positive number yields a valid kernel.
- If k is valid, then e^k is valid.

1. Polynomial Kernel

- We know $k_1(x, x') = x^T x'$ is a valid kernel because it's a simple inner product.
- The constant number c is a valid kernel too.
- According to the summation property $k_2 = x^T x' + c$ is valid.
- According to multiplication property raising k_2 to the power of d results in a valid kernel.

So, the polynomial kernel is valid.

2. RBF Kernel

$$Definition: k(x, x') = e^{\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)}$$

By expanding the expression inside the exponent:

$$\left|\left|x - x'\right|\right| = (x - x')^T (x - x') = \left|\left|x\right|\right|^2 + \left|\left|x'\right|\right|^2 - 2x^T x'$$

So:

$$k(x, x') = e^{-\frac{\|x\|^2}{2\sigma^2}} . e^{\frac{x^{\mathrm{T}} x'}{\sigma^2}} . e^{-\frac{\|x'\|^2}{2\sigma^2}}$$

The factors $\exp(-\| x \|^2/(2\sigma^2))$ and $\exp(-\| x' \|^2/(2\sigma^2))$ depend only on x or x' and can be absorbed by scalar multiplication, having no effect on the positive semi definiteness for all sets. The central kernel $\exp(x^{\mathrm{T}} x'/\sigma^2)$ can be written using the exponential power series expansion.

$$e^{(\frac{x^\top x'}{\sigma^2})} = \sum_{m=0}^{\infty} \frac{1}{m!} (\frac{x^\top x'}{\sigma^2})^m$$

Each term $(x^\top x')^m$ is itself a PSD kernel. The coefficient $\frac{1}{m!\sigma^{-2m}}$ is non-negative. The sum of PSD terms with non-negative coefficients also yields a PSD kernel.

Therefore, exp $(x^\top x'/\sigma^2)$ is a PSD kernel. Consequently, $k(x, x')$ constructed by multiplying with factors dependent on $\| x \|^2$ and $\| x' \|^2$ is also PSD. Thus, the RBF kernel is valid.

So, The RBF kernel is valid either through power series expansion and the non-negative sum of polynomial kernels, or through Fourier transform properties.

Part III: An Intuitive Interpretation of KKT Conditions for SVM

The Karush-Kuhn-Tucker (KKT) conditions are the necessary optimality conditions for constrained optimization problems. For the standard SVM, these conditions are:

1. Stationarity Condition: The gradient of the Lagrangian with respect to the primal variables must be zero.

$$\nabla_\omega L = 0 \Longrightarrow \omega = \sum \alpha_i y_i x_i$$

$$\nabla_b L = 0 \Longrightarrow \sum \alpha_i y_i = 0$$

   Role: This condition determines the structure of the optimal solution (the weight vector is a combination of the data points).

2. Primal Feasibility: The solution must satisfy the original constraints.

$$y_i(\omega^T x_i + b) \geq 1 - \xi_i$$

$$\xi_i \geq 0$$

   Role: It ensures that the obtained solution does not violate the classification constraints of the problem.

3. Dual Feasibility: The Lagrange multipliers must be non-negative.

$$\alpha_i \geq 0$$

$$\mu_i \geq 0$$

   Role: It ensures that the dual function is a lower bound for the primal function and that the optimization direction is correct.

4. Complementary Slackness: The product of the Lagrange multiplier and the constraint value must be zero.

$$\alpha_i[y_i(\omega^T x_i + b) - 1 + \xi_i] = 0$$

$$\mu_i \xi_i = 0 \Longrightarrow (C - \alpha_i)\xi_i = 0$$

Role: This is the most important condition for interpreting SVMs. It indicates that only for support vectors (which lie on or inside the margin), the value of α can be non-zero. This condition leads to sparsity in the model.

Question 4

Part I: The Maximum-Margin Hyperplane and Decision Boundary

Positive Class: $P_1(2,-0), P_2(3,0), P_3(2,1), P_4(2,-1)$

Negative: $N_1(-2,-0), N_2(-3,0), N_3(-2,-1), N_4(-2,1)$

- Positive points are located on the right side.
- Negative points are located on the left side.

The best separating line (which has the maximum distance from both groups) is a line that passes exactly through the middle of these two clusters. The midpoint between 2 and -2 is 0. Therefore, the separating line is the Y-axis. Decision boundary equation:

$$x_1 = 0$$

Part II: Manual calculation of the coefficients ω & b

To form the system of equations, we assume the weight vector is $\omega = [\omega_1, \omega_2]^T$. For a positive support vector such as $(2, 0)$. Since $y_i(\omega^T x_i + b)$ we can write these equations:

- For a positive point: $y(\omega^{\mathrm{T}} x + b) = (1) \cdot (a \cdot 2 + 0) = 2a \geq 1 \Rightarrow a \geq \frac{1}{2}$
- For a negative point: $(-1) \cdot (a \cdot (-2) + 0) = 2a \geq 1 \Rightarrow a \geq \frac{1}{2}$

Therefore, the smallest a that satisfies the constraints is $a = 0.5$. Since the objective is to minimize $\frac{1}{2} \| \omega \|^2 = \frac{1}{2} a^2$, the minimum occurs at $a = 0.5$.

So:

$$\omega = \begin{bmatrix} 0.5 \\ 0 \end{bmatrix}, b = 0$$

Part III: Support Vectors and Margin

Support vectors are points that satisfy the constraints with equality:

$$y_i(\omega^T x_i + b) = 1$$

For $\omega = (0.5,1)$, $b = 0$, the value of $\omega^T x$ depends only on the component x_1:

$$\omega^T(x_1, x_2) = \frac{1}{2} x_1$$

For points with $x_1 = 1$ we have $\omega^T x = 1$, and for points with $x_1 = -2$ we have $\omega^T x = -1$. Therefore, all points where $x_1 = 2$ or $x_1 = -2$ lie exactly on the planes which is $= \pm 1$, and thus are support vectors.

List of support vectors: SVs $= \{(2,0), (2,1), (2,-1), (-2,0), (-2,1), (-2,-1)\}$

In the canonical SVM formulation with constraints $y(\omega^{\mathrm{T}} x + b) \geq 1$, the geometric distance from a point x to the decision boundary hyperplane $\omega^{\mathrm{T}} x + b = 0$ is given by:

$$\frac{|\ \omega^{\mathrm{T}}x + b\ |}{\|\ \omega\ \|}$$

For support vectors $|\ \omega^T x + b\ | = 1$ so their geometric distance to the central hyperplane is $\frac{1}{||\omega||}$.

$$\|\ \omega\ \| = \sqrt{(\frac{1}{2})^2 + 0^2} = \frac{1}{2}$$

Therefore, the geometric distance from the central hyperplane to the nearest sample is:

$$\gamma = \frac{1}{||\omega||} = \frac{1}{0.5} = 2$$

The distance between these two planes is $\frac{2}{||\omega||}$. So we have:

$$Width = 2\gamma = \frac{2}{||\omega||} = \frac{1}{0.5} = 4$$

Question 5

Part I: Selecting an Appropriate Neural Network Model

The dataset contains four compact clusters belonging to two classes. These clusters are nonlinearly separable, meaning that a single linear decision boundary cannot divide the classes. Consequently, a single-layer network with a Sigmoid activation (i.e., Logistic Regression) is not suitable, because it can only form one linear boundary in the input space.

For this problem, two nonlinear neural network models are appropriate:

1. Multi-Layer Perceptron (MLP) with one hidden layer
2. Radial Basis Function Network (RBF Network)

Both can represent highly nonlinear and multi-region decision boundaries. However, for this specific dataset consisting of four well-defined and spatially separated clusters an RBF network is structurally more efficient because each cluster can be modeled by a single localized radial unit. Training is also simpler, as the output layer is linear and centers can be initialized using clustering.

A recommended architecture for each model is as follows:

Recommended MLP Architecture

- Input layer: 2 neurons
- Hidden layer: 4 neurons (Tanh or ReLU)
- Output layer: 1 Sigmoid neuron

Parameter count:

- Input → Hidden weights: $2 \times 4 = 8$
- Hidden biases: 4
- Hidden → Output weights: 4
- Output bias: 1
  ⇒ Total MLP parameters = 17

Recommended RBF Architecture

- Input layer: 2 neurons
- RBF layer: 4 radial neurons (one for each cluster)
- Output layer: 1 linear/Sigmoid neuron

Parameter count:

- Centers: $4 \times 2 = 8$
- Spreads (σ): 4

- Output weights: 4
- Output bias: 1
  ⇒ Total RBF parameters = 17

Although both networks have similar parameter counts, the RBF network is more optimal for this dataset because:

- Each neuron represents a local region (local receptive field).
- Decision boundaries are constructed by combining localized responses, which matches the geometry of the four-cluster dataset.
- Training is simpler and less sensitive to initialization compared with MLPs.

Therefore, the RBF network with four radial units is the most efficient architecture for this classification task.

Part II: Global Activation Fails

Earlier models such as Perceptron, Logistic Regression, or shallow networks with global activation functions (Sigmoid, Tanh, etc.) struggle on this dataset because:

1. Linear models cannot solve nonlinear separability
   A single Sigmoid neuron computes

$$\sigma(w^T x + b)$$

   and produces only one linear decision boundary. The four-cluster arrangement cannot be separated using one line; therefore, logistic regression inherently fails.

2. Global activation functions affect the entire input space

   In an MLP, each hidden neuron has a global influence. Adjusting a neuron to improve classification in one region also alters the decision boundary in other, distant regions. Constructing multiple disjoint regions (four “islands” of data) therefore requires a complex combination of many global nonlinearities.

3. Poor handling of local cluster structure

   The dataset consists of several compact clusters. Global activations cannot restrict their influence to one local region, causing decision boundaries to become unnecessarily complicated or distorted. As a result, the model may correctly classify near-cluster points but fail in intermediate regions where new test points lie.

In contrast, the RBF network uses localized radial functions, where each neuron is active only near its center. This allows the model to form clean, cluster-based decision regions and generalize properly across the input space.

Question 6 | Section 1

1.1. System Identification with RBF Neural Networks

For this question, we first model a simplified beam-and-ball system. The position of the ball at time t, denoted by y(t), is a function of its two previous positions, $y(t-1)$ and $y(t-2)$, as well as the previous input to the system, $u(t-1)$. This relationship is defined by the following difference equation:

$$y(t) = \frac{y(t-1)}{1+y(t-2)} + u(t-1)^3$$

Next, we must construct a static RBF neural network (RBFNN) that functions as an identifier, estimating the system output y(t) based on the input vector:

$$x(t) = [y(t-1), y(t-2), u(t-1)]^T$$

1.1.1. Generate Data

We generated a time series dataset of 1000 points. The input signal u is a sine wave, and the output y is a nonlinear, dynamic function of its own past values and the input.

The target variable is the current value $y[i]$.

```
t = np.linspace(0,999,1000).reshape(-1,1)
u = np.sin(2*np.pi*t/250)
y = np.zeros_like(t)
for i in range(2,1000):
    y[i] = y[i-1]/(1+y[i-2])+u[i-1]**3

X = np.concatenate([y[1:999], y[0:998], u[1:999]], axis=1)
y = y[2:]
```

1.1.2. Preprocess Data

The dataset was split into training (first 700 samples) and testing (remaining samples) sets.

```
X_train = X[:700]
y_train = y[:700]
X_test = X[700:]
y_test = y[700:]
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

We standardized the features using StandardScaler from scikit-learn. This process transforms the data to have a mean of 0 and a standard deviation of 1. Crucially, the scaler was fitted *only* on the training data to prevent data leakage, and then applied to both sets.

1.1.3. Problem Approximation Function

This problem is a function approximation problem because the core task is to learn a hidden, underlying functional mapping from observed input-output data. The RBFNN is presented with a dataset where the inputs are the past states of the system, $[y(t-1), y(t-2), u(t-1)]$, and the target output is the current state, $y(t)$. A deterministic, non-linear rule (the given difference equation) governs this relationship, but its exact form is not provided to the model. The network's objective is to construct an approximating function $\hat{f}$ that can emulate the behavior of this

unknown true function based solely on the training examples, which is the essence of function approximation.

The specific unknown function f that the RBFNN must learn is the dynamic law of the system, which maps the previous states and input to the current output. This function is defined precisely by the generating equation:

$$y(t) = f\,(y(t-1), y(t-2), u(t-1)) \; = \frac{y(t-1)}{1+y(t-2)} + (u(t-1))^3$$

Thus, the RBFNN's goal is to approximate this complex, non-linear function f that defines the system's evolution.

1.2. Static RBFNN Implementation

An RBFNN typically consists of an input layer, a hidden layer with RBF neurons, and a linear output layer. The network output is computed as follows (for simplicity, we consider α_0 equal to zero):

$$f(x) = \sum_{k=1}^{K} \alpha_k \, \phi_k(x)$$

where K is the number of hidden neurons, α_k are the output weights, and $\phi_k(x)$ is the Gaussian RBF activation function for the k-th neuron. The activation function $\phi_k(x)$ is defined as:

$$\phi_k(x) = \exp\,(-\frac{1}{\sigma_k^2} \;\| x - \mu_k \|^2),$$

where μ_k is the center of the k-th RBF neuron and σ_k is its width (spread).

1.2.1. Role of the Neuron Center (μ_k) and Neuron Width (σ_k)

The center μ_k determines the location in the input space where the k-th RBF neuron responds most strongly.
The activation function reaches its maximum value (equal to 1) when the input vector equals the center:

$$\phi_k(x) = 1 \;\text{ If } x = \mu_k.$$

Therefore, μ_k controls *where* the neuron is active.
Changing $\mu_k$shifts the peak of the Gaussian to a different point in the input space.

The width σ_k determines how widely the neuron responds around its center. It controls the "spread" or sensitivity of the Gaussian.

- Small σ_k(narrow Gaussian):
    - The neuron responds only to inputs very close to μ_k.
    - The activation drops sharply as x moves away from μ_k.

- The neuron becomes highly localized and sensitive to small changes.

- Large σ_k(wide Gaussian):
  - The neuron responds to a broader region of the input space.
  - The activation decreases slowly with distance.
  - The neuron becomes less localized, providing smoother and more global approximation.

1.2.2. Define Gaussian RBF

The code below, Defines $\phi_k(x)$

```
def gussianRBF(x ,center = (0,0), spread = 1):
    x = np.array(x)
    center = np.atleast_2d(center)
    return np.exp(-np.sum(((x-center)**2), axis=1)/spread**2)
```

To evaluate the neuron's response, the function was tested at $x = [1,1]$with center $\mu = [1,1]$ and $\sigma = 1$. The resulting activation was:

```
array([1.])
```

This result is expected, as the activation function reaches its maximum value when the input exactly coincides with the neuron's center. In this case, the distance $\| x - \mu \|$is zero, yielding:

$$\phi(x) = e^0 = 1.$$

An activation value of 1 indicates that the input lies precisely at the neuron's center, meaning the neuron is fully activated and contributes maximally to the network output. This test confirms the correct implementation and expected behavior of the Gaussian RBF unit.

1.3. Training via Linear Least Square

We assume the RBF network output is a linear model of the form:

$$\hat{Y} = \Phi\alpha$$

where Φ is the RBF design matrix with elements:

$$\Phi_{ij} = \varphi_j(x_i)$$

and Y is the vector of target values.
The least-squares objective is

$$J(\alpha) = \| Y - \Phi\alpha \|$$

Setting the gradient to zero gives the normal equations:

$$\Phi^T\Phi\,\alpha = \Phi^T Y$$

If $\Phi^T\Phi$ is invertible, the optimal weight vector is:

$$\alpha = (\Phi^T\Phi)^{-1}\Phi^T Y$$

In the general case, using the Moore–Penrose pseudoinverse Φ^+:

$$\alpha = \Phi^+ Y$$

Where $\Phi^+ = (\Phi^T\Phi)^{-1}\Phi^T$ when $\Phi^T\Phi$is nonsingular.

1.3.1. Training Code for RBF

To train the static RBF network, the centers and spreads were fixed prior to learning the output weights. A total of Kcenters were selected by randomly sampling Kdata points from the training set. After the centers $\mu_k$were chosen, a single shared spread σ was computed as the mean pairwise Euclidean distance between all selected centers:

$$\sigma = \frac{1}{\binom{K}{2}} \sum_{i<j} \| \mu_i - \mu_j \|$$

```
def mean_distance_centers(centers):
    n_centers = len(centers)
    distances = []
    for i in range(n_centers):
        for j in range(i+1,n_centers):
            distances.append(np.linalg.norm(centers[i]-centers[j]))

    return np.mean(distances)
```

Using these centers and the fixed spread, the RBF design matrix Φ was constructed for the training data. Each entry of $\Phi$is defined as

$$\Phi_{ik} = \exp\left(-\frac{\| x_i - \mu_k \|^2}{\sigma^2}\right).$$

With Φ formed, the training problem becomes a linear least-squares problem. The optimal output weights α are obtained by solving:

$$\alpha = \Phi^+ Y$$

where $\Phi^+$is the Moore–Penrose pseudoinverse. After determining α, the model can be evaluated on both training and test samples by recomputing the corresponding Φmatrices and applying the linear model $\hat{y} = \Phi\alpha$.

The training implementation used in this work is shown below:

```
class RBF:
    def __init__(self, n_center=10, spread = -1):
        self.n_center = n_center
        self.spread = spread

    def fit(self, X, y):
        self.centers = np.random.choice(X.shape[0], self.n_center,
replace=False)
        self.centers = X[self.centers]

        if self.spread == -1:
            self.spread = mean_distance_centers(self.centers)
        Phi = np.empty((X.shape[0],self.n_center))
```

```
        for i, center in enumerate(self.centers):
            g = gussianRBF(X, center, self.spread)
            Phi[:,i] = g

        w = np.linalg.pinv(Phi) @ y
        self.w = w

    def __call__(self, X):
        Phi = np.empty((X.shape[0],self.n_center))
        for i, center in enumerate(self.centers):
            g = gussianRBF(X, center, self.spread)
            Phi[:,i] = g

        y = Phi@self.w
        return y
```

To assess the performance of the trained RBF model, the root mean square error (RMSE) was used as the evaluation criterion. RMSE is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

The RMSE function used in the implementation is:

```
def RMSE(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred)**2))
```

After training the RBF model on the training dataset, the performance of the network was evaluated using the RMSE criterion for both the training and test sets. The obtained values are as follows which shown in figure 6.1.

```
Root mean square error for train: 3.4306876916836946
Root mean square error for test: 4.42441572249357
```

Fig 6.1. RMSE for test & train data

1.3.2. Predicted vs. True Output on Training Set

Figure 6.2 illustrates the predicted outputs of the RBF network for the training dataset compared with the corresponding true target values.

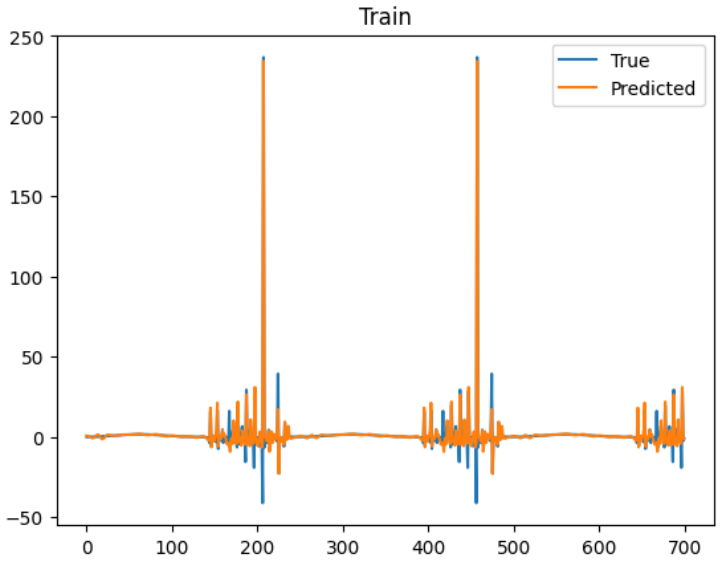

Fig. 6.2. Predicted and true outputs for the training set

1.3.3. Predict vs. True Outputs on Test Set

Figure 6.3 presents the same comparison for the test dataset, showing how well the trained RBF network generalizes to unseen data

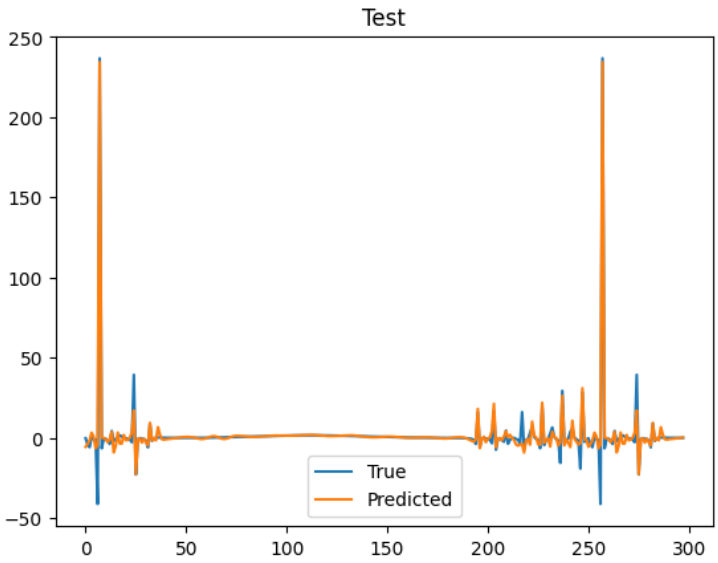

Fig. 6.3. Predicted and true outputs for the test set

The test-set plot exhibits strong agreement between predicted and actual values, particularly in the flat baseline regions and around the main peaks, indicating effective generalization. However, the largest errors occur near the sharp high-amplitude spikes at the beginning and especially around the second major peak. This happens because:

1. Sharp peaks represent highly localized features that require RBF centers to be placed very close to those regions. Since the centers are chosen randomly from the training data, some key high-slope areas in the test set may not be well-represented.

2. RBF units with a global spread produce smooth approximations, which makes it difficult to reconstruct sudden changes or discontinuities accurately.
3. The training data contains peak locations different from the test data, so the model has learned the general shape, but not the exact local variations appearing only in the test sequence.

Despite these localized errors, the overall structure of the signal is captured well, and the model maintains acceptable generalization performance as confirmed by the RMSE results

1.3.4. LLS Training Faster Than GD

In the static RBF network used in this experiment, the centers μ_kand spreads σ_kwere fixed prior to training. Under this assumption, the network output becomes a purely linear model with respect to the output weights α. As a result, training reduces to solving the linear system

$\alpha = \Phi^{+} Y,$

which is obtained directly through the Moore–Penrose pseudoinverse or an equivalent least-squares solver. This process requires only one matrix construction and one closed-form solution, without any iterative optimization.

In contrast, multilayer perceptrons trained with backpropagation rely on gradient-descent-based iterative updates, where thousands of forward and backward passes may be required to gradually converge to an acceptable minimum. Each iteration involves computing gradients, adjusting weights, and re-evaluating errors, making the process computationally expensive.

The key reasons LLS training is significantly faster are:

Closed-form solution:

The LLS method computes optimal weights in a single matrix operation, whereas backpropagation requires many iterative updates.

No gradient computation:

Backpropagation requires computing partial derivatives across all network layers. LLS avoids this entirely because the model is linear in its parameters.

No learning rate tuning or convergence issues:

Gradient descent performance depends heavily on step size, initialization, and stopping criteria. LLS does not involve any of these complexities.

Deterministic and non-iterative:

LLS yields the global optimum immediately. Backpropagation may take a long time to converge and can get trapped in local minima.

Overall, because the RBF network with fixed centers and spreads transforms the learning problem into a linear algebra task, the computational cost is dramatically lower than that of iterative gradient-based training used in multilayer perceptrons.

1.3.5. Effect of the Number of Centers K Test Performance

In this experiment, the RBF spread was kept fixed at $\sigma = 5$, and the number of centers K was varied over the set $\{10,20,50,100,200\}$.

For each value of K, an RBF network was trained on the training set (with randomly selected centers) and then evaluated on the test set. The corresponding test RMSE values were plotted as a function of K.

Figure 6.4 shows the resulting curve of $\text{RMSE}_{\text{test}}$ versus the number of centers K.

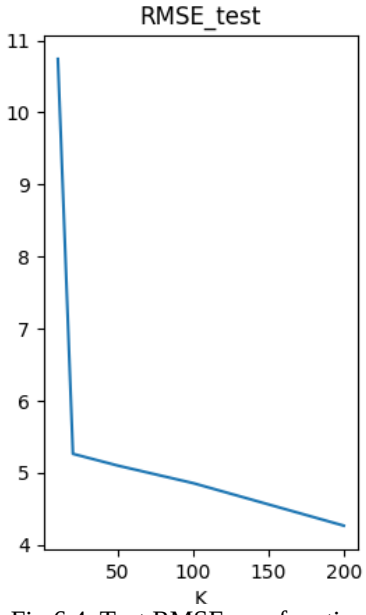

Fig 6.4. Test RMSE as a function of the number of centers K (with σ=5)

The minimum test error over the explored range is obtained for $K = 200$, with $\text{RMSE}_{\text{test}} \approx 4.31$.

1.3.6. Effect of Very Small or Very Large K

The number of RBF centers K directly controls the expressive capacity of the RBF network. When K is very small, the network has too few basis functions to represent the variations in the data. In this situation, the model produces an overly smooth approximation and fails to capture important local structures of the signal. This phenomenon corresponds to underfitting, where both the training and test errors remain high because the model is not sufficiently flexible.

Conversely, when K becomes very large, the network becomes highly flexible and capable of fitting even small fluctuations and noise in the training data. In general, this can lead to overfitting, where the training error becomes very small but the test error increases because the model memorizes training-specific patterns that do not generalize well.

However, in the present dataset and with the fixed spread $\sigma = 5$, the experimental results showed a continuous decrease in test RMSE as Kincreased, with the lowest error occurring at $K = 200$. This indicates that overfitting did not occur within the tested range of $K$for this specific problem. The data structure and the chosen spread value appear to restrict excessive model flexibility, preventing the network from fitting noise.

Nevertheless, in other datasets or projects particularly those with noisier signals, high-dimensional inputs, or inappropriate spread values a very large Kcan indeed lead to overfitting. Therefore, the relationship between $K$and generalization performance must be evaluated carefully for each dataset.

1.3.7. Effect of the Spread σ on Test Performance

After fixing the number of centers at the best value obtained earlier ($K = 200$), the RBF network was trained with several different spread values: {0.5, 1, 5, 7, 10}.

For each spread, the model was evaluated on the test dataset, and the corresponding RMSE values were computed. The resulting $\text{RMSE}_{\text{test}}$as a function of $\sigma$is shown in Figure 6.5.

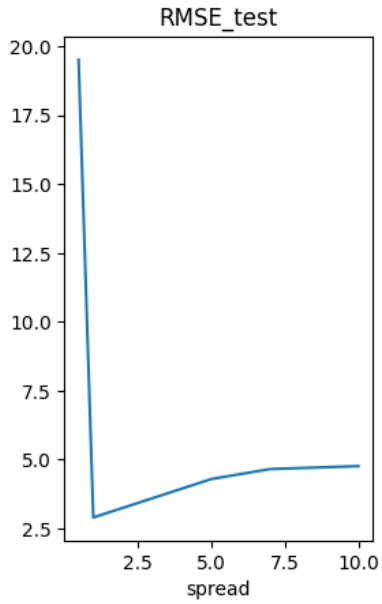

Fig. 6.5. Test RMSE as a function of spread σ

As shown in Figure 6.4, the network performance is highly sensitive to the spread parameter. The test RMSE initially decreases when σincreases from 0.5to 1, reaching the minimum error at:

$$\sigma^* = 1, \text{RMSE}_{\text{test}} = 3.25$$

For larger spreads ($\sigma > 1$), the error increases steadily. This happens because increasing σmakes each RBF unit excessively wide, causing the network to produce an overly smooth approximation that fails to represent sharp peaks and local variations in the signal. In contrast, a very small spread makes each basis function too narrow, limiting the ability to combine information across neighboring samples.

Using the optimal hyperparameters ($K = 200, \sigma = 1$), the final RBF network was retrained and evaluated on both the training and test sets. Figure 6.6 displays the predicted outputs versus the true outputs for both datasets.

The results in Figure 6.6 show that the trained RBF model captures the main features of the signal, including the large spikes and surrounding oscillatory behavior. The RMSE for the training and test sets are:

```
RMSE_train =2.384, RMSE_test =3.326
```

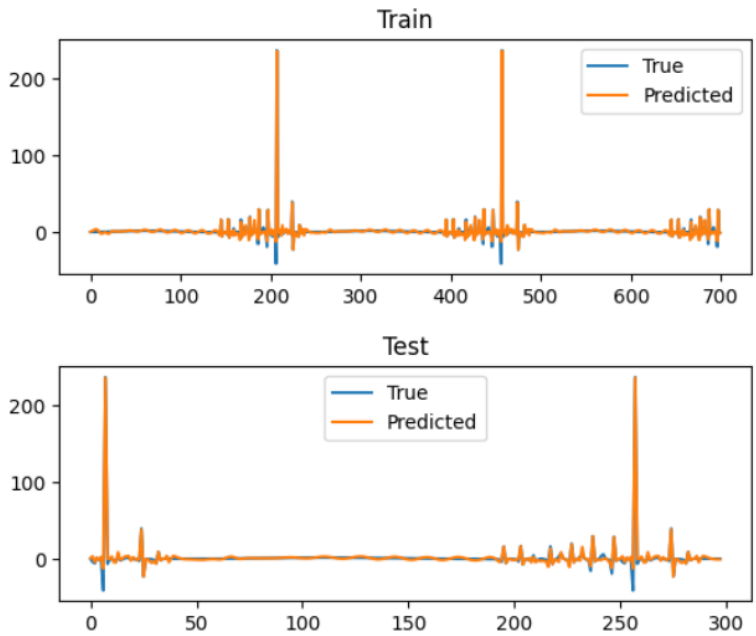

Fig. 6.6. True vs. Predicted outputs for training data and test data using $K = 1$ and $\sigma = 1$

The test error remains close to the training error, indicating good generalization and no observable overfitting for this dataset. The model accurately follows the main patterns, with the largest discrepancies occurring near the sharp peaks, where precise reconstruction is inherently more difficult due to the rapid local variations.

1.3.8. Effect of Very Small or Very Large σ

The spread parameter σcontrols the width of each radial basis function. Its value directly influences how “localized” or “global” each neuron behaves.

When σ is very small

If $\sigma$is extremely small, each RBF unit becomes sharply peaked and responds only to a very narrow region around its center. This leads to:

- Highly localized basis functions
- Very irregular approximations
- High sensitivity to noise

- Poor generalization to unseen samples

This phenomenon corresponds to overfitting: the model memorizes fine details of the training data but fails to represent the underlying functional relationship.
In many datasets, small spreads produce extremely large test RMSE.

However, in our dataset (Section 6.3.1), the optimal value was $\sigma = 1$, which is relatively small, yet we did not observe overfitting. The reason is that the underlying signal contains strong sharp peaks, so a narrower spread was necessary to capture those patterns.

When σ is very large

If σbecomes too large, the Gaussian basis functions expand and begin to overlap excessively.
When $\sigma$is too large:

- All RBF units behave similarly
- The model becomes overly smooth
- Sharp peaks in the signal cannot be represented
- Both training and test RMSE increase

This corresponds to underfitting, where the model does not have enough flexibility to capture meaningful structure in the data.

In our results (Figure 6.4), spreads larger than $\sigma = 1$caused steady performance degradation, with test RMSE increasing to values above 4.5–5.0 as σapproached 10.

Generalization summary

- Too small σ: high variance, possible overfitting (not observed in this dataset, but common in other tasks).
- Too large σ: high bias, definite underfitting (observed clearly in our experiments).

Optimal σ: balances bias and variance, producing minimum test RMSE.
In our dataset: $\sigma^* = 1$.

1.3.9. Challenge in Selecting the Best Network Structure

Determining the optimal values of $K$and σpresented several practical and conceptual challenges:

1. Strong dependence between $K$and σ

The two hyperparameters are not independent. For example:

- A small Krequires a larger σ to cover the input space.
- A large Kmay require a smaller σ to avoid over-smoothing.

This interaction forces both parameters to be tuned together rather than separately.

2. Sensitivity of RMSE to σ

As demonstrated in Figure 6.5, test RMSE changed significantly with small adjustments in σ. Small spreads (e.g., $\sigma = 1$) produced much better performance than larger ones. Selecting the correct value was computationally expensive because each spread required retraining the network.

3. Statistical randomness from center selection

The centers were chosen randomly, so each run could produce slightly different performance. For small K, this randomness caused high variation in RMSE. Ensuring stable results required multiple runs or fixing the random seed.

4. Balancing model complexity and generalization

Increasing Kconsistently reduced test RMSE (Figure 6.4), reaching a minimum at $K = 200$. However, in many datasets, a large number of centers may lead to overfitting. Therefore, choosing K required monitoring training and test errors to avoid excessive complexity.

5. Dataset-specific behavior

In this project:

- No overfitting was observed even at large K.
- The optimal configuration was ($K = 200$, $\sigma = 1$).
- For other datasets, the same configuration could be suboptimal or even harmful.

Thus, selecting the best network structure required careful experimentation rather than relying on fixed rules.

1.4. Traying to Solve Challenges

1.4.1. Training the Static RBFNN with L_2–Regularized LLS

When the number of centers K is very large, a standard least–squares solution may lead to poor generalization because the model becomes excessively flexible. A common way to mitigate this problem is to add an L_2 regularization term to the loss function.

For a fixed design matrix $\Phi \in \mathbb{R}^{N\times K}$and target vector $Y \in \mathbb{R}^N$, the regularized objective is defined as:

$$J(\alpha) = \| Y - \Phi\alpha \|^2 + \lambda \| \alpha \|^2,$$

where $\alpha \in \mathbb{R}^K$ is the vector of output weights and $\lambda \geq 0$is the regularization coefficient. Taking the derivative with respect to $\alpha$and setting it to zero yields the regularized normal equations:

$$(\Phi^\top\Phi + \lambda I)\,\alpha = \Phi^\top Y.$$

Thus, after optimization, the optimal weight vector is

$$\alpha = (\Phi^\top\Phi + \lambda I)^{-1}\Phi^\top Y.$$

This formula is used to implement the training function for the static RBF network with L_2 regularization.

We will use this code to implement RBF model with L_2 regularization:

```
class RBF_L2(RBF):
    def __init__(self, n_center=10, spread = -1, regular = 0):
        super().__init__(n_center, spread)
        self.regular = regular

    def fit(self, X, y):
        self.centers = np.random.choice(X.shape[0], self.n_center, replace=False)
        self.centers = X[self.centers]

        if self.spread == -1:
            self.spread = mean_distance_centers(self.centers)
        Phi = np.empty((X.shape[0],self.n_center))
        for i, center in enumerate(self.centers):
            g = gussianRBF(X, center, self.spread)
            Phi[:,i] = g
        if self.regular:
            w =
np.linalg.inv(Phi.T@Phi+self.regular*np.eye((Phi.T@Phi).shape[0]))@Phi.T @ y
        else:
            w = np.linalg.pinv(Phi) @ y
        self.w = w
```

To study the effect of regularization, we fixed the network structure at a large number of centers and a small spread as follows:

$$K = 500, \sigma = 1.$$

For this fixed (K, σ), the model was trained with several regularization coefficients

$$\lambda \in \{0.001, 0.01, 0.1, 1, 10\}$$

and evaluated on the test set. The resulting test RMSE values are plotted as a function of λ in Figure 6.7.

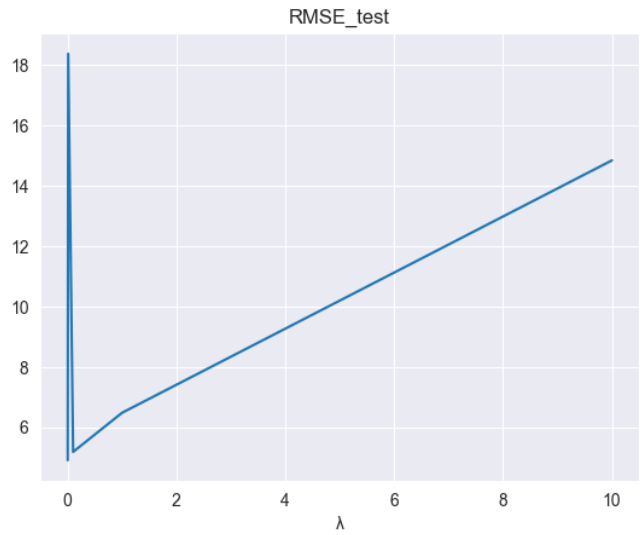

Fig. 6.7. Test RMSE as a function of regularization coefficient λ

The minimum test error among the examined values occurs at

$$\lambda^* = 0.001, \mathrm{RMSE}_{\mathrm{test}} \approx 4.91.$$

For larger λ, the test RMSE increases almost linearly, indicating that strong regularization pushes the model towards underfitting.

Using the best regularization value $\lambda = 0.001$, the RBF network was trained and its predictions were compared with the true outputs for both training and test sets. The results are presented in Figure 6.8.

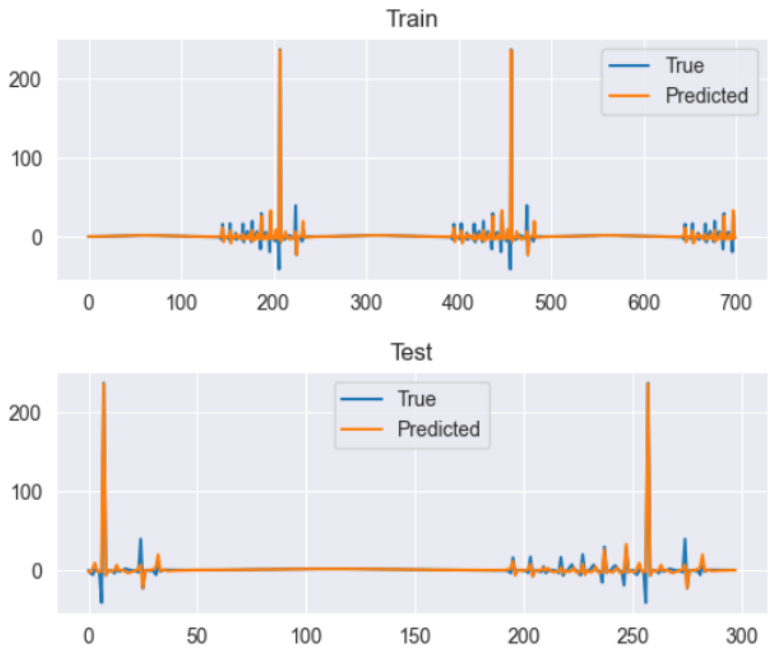

Fig. 6.8. True vs. predicted outputs with regularization $K = 500, \sigma = 1, \lambda = 0.001$

The corresponding RMSE values are:

$$\mathrm{RMSE}_{\mathrm{train}} \approx 3.78, \ \mathrm{RMSE}_{\mathrm{test}} \approx 5.01$$

For comparison, Figure 6.9 shows the predictions obtained without regularization ($\lambda = 0$) for the same network structure ($K = 500$, $\sigma = 1$).

In this case, the errors are:

$$\mathrm{RMSE}_{\mathrm{train}} \approx 0.819, \mathrm{RMSE}_{\mathrm{test}} \approx 0.994$$

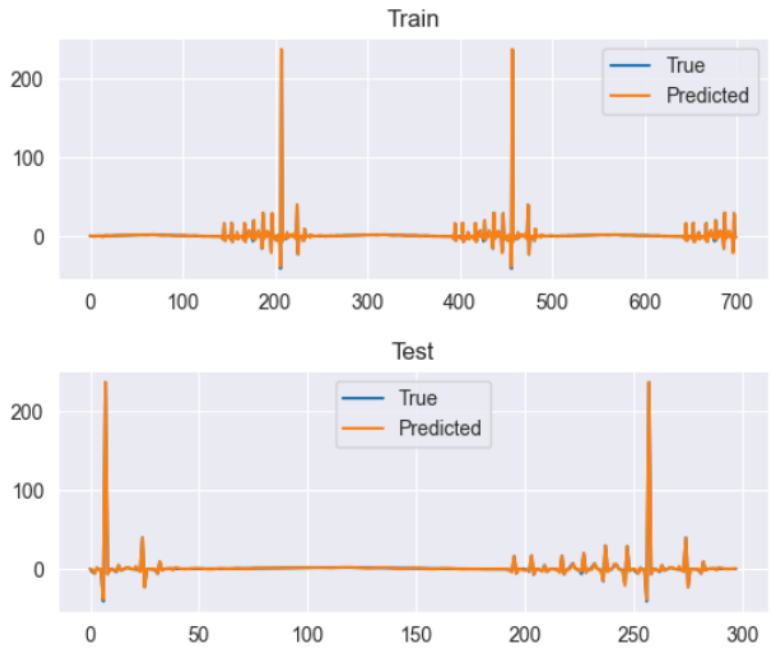

Fig. 6.9. True vs. predicted outputs with regularization $K = 500, \sigma = 1, \lambda = 0$

1.4.2. Role of the Regularization Parameter λ

When L_2 regularization is applied to the least–squares training of the RBF network, the optimal weight vector $\alpha$is obtained from

$$(\Phi^{\mathsf{T}}\Phi + \lambda I)\,\alpha = \Phi^{\mathsf{T}}Y,$$

where λ is the regularization coefficient. The term λIpenalizes large weight values and prevents the weight vector from becoming excessively large when $\Phi^{\mathsf{T}}\Phi$ is ill-conditioned or when the number of centers $K$is very large.

The regularization parameter λ directly controls the compromise between model complexity and generalization ability:

1. When λ is very small
    - The solution approaches the standard LLS solution.
    - The model becomes highly flexible and can fit fine-scale variations in the data.
    - If the dataset is noisy or $K$is too large, the model may overfit.
    - In our dataset, small λ produced the best performance because the unregularized model already generalized well.
2. When λ is very large
    - The penalty on the weights becomes dominant.
    - The magnitudes of the weights shrink toward zero.

- The network output becomes overly smooth and loses the ability to reproduce sharp structures or peaks.
- This leads to underfitting, and both training and test errors increase (as observed in the RMSE trend for large λ in Figure 6.6).

1.5. Nearest Neighbor Strategy

1.5.1. Training the RBF Network Using K-Means for Center Selection

In all previous experiments, the RBF centers μ_k were chosen simply by sampling K points randomly from the training set. Although this method is simple, it does not guarantee that the selected centers represent the structure of the data well. A more systematic approach is to use K-means clustering to find the centers.

K-means is an unsupervised clustering algorithm that partitions the dataset into K clusters by minimizing the within-cluster variance.
The algorithm operates through the following iterative procedure:

1. Initialization:
   Choose K initial cluster centers (randomly or using K-means++).
2. Assignment step:
   Assign each data point x_i to the cluster whose center is closest:

$$\text{cluster}(x_i) = \arg\min_k \| x_i - \mu_k \|.$$

3. Update step:
   Recompute each cluster center as the mean of points assigned to it:

$$\mu_k = \frac{1}{| C_k |} \sum_{x_i \in C_k} x_i$$

4. Repeat the assignment–update cycle until convergence.

The resulting cluster centers are typically much better distributed than randomly selected points. This makes them strong candidates for RBF centers, especially when K is large.

Below is the class implementation used in this section:

```
class RBF:
    def __init__(self, n_center=10, spread = -1):
        self.n_center = n_center
        self.spread = spread

    def fit(self, X, y):
        self.centers = np.random.choice(X.shape[0], self.n_center, replace=False)
        self.centers = X[self.centers]

        if self.spread == -1:
            self.spread = mean_distance_centers(self.centers)
        Phi = np.empty((X.shape[0],self.n_center))
        for i, center in enumerate(self.centers):
            g = gussianRBF(X, center, self.spread)
            Phi[:,i] = g
```

```
        w = np.linalg.pinv(Phi) @ y
        self.w = w

    def __call__(self, X):
        Phi = np.empty((X.shape[0],self.n_center))
        for i, center in enumerate(self.centers):
            g = gussianRBF(X, center, self.spread)
            Phi[:,i] = g

        y = Phi@self.w
        return y
```

To evaluate the effect of K-means center selection, the following parameters were used:

$$K = 100, \sigma = \text{mean\_distance\_centers}(\mu_k)$$

The network was trained once using K-means centers, and again using random centers (as in earlier sections). The predicted outputs versus the true outputs for both training and test sets are shown in Figure 6.10.

$$RMSE_{train} = 2.0906, RMSE_{test} = 2.8333$$

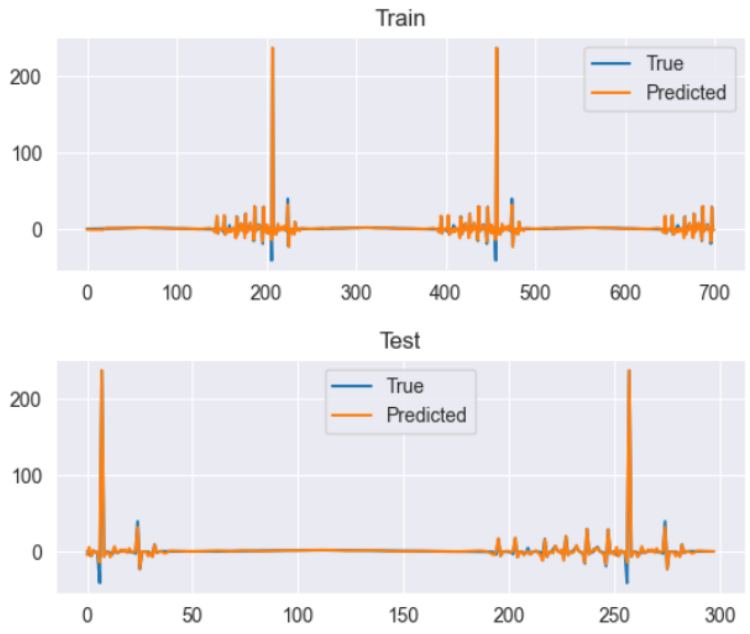

Fig 6.10. Output prediction using K-means centers

For comparison, the results obtained earlier using random centers appear in:

- Figure 6.1 (training output)
- Figure 6.2 (test output)

Method	RMSE (Train)	RMSE (Test)
Random centers	3.4296	4.4219
K-means centers	2.0906	2.8333

Using K-means instead of random selection clearly improves performance:

1. Better placement of centers

K-means places the centers near regions of high data density and variability. This allows the RBF basis functions to better represent:

- the oscillatory regions near the peaks,
- the localized spikes,
- and the overall structure of the signal.

2. Significant reduction in RMSE

$$\mathrm{RMSE}_{\mathrm{test}}^{\mathrm{random}} = 4.42 \rightarrow \mathrm{RMSE}_{\mathrm{test}}^{\mathrm{kmeans}} = 2.83$$

This is a 36% improvement in generalization error.

3. Improved visual accuracy

The K-means model captures:

- spike amplitude more accurately,
- oscillation patterns with less lag,
- smoother overall prediction curves.

In contrast, the random-center model struggles with peak regions and shows larger deviations.

1.5.2. Spread Computation Using the Nearest–Neighbour Strategy

In the previous section (1.5.1), the centers $\mu_k$were obtained using K-means clustering, and the spread σ was chosen as a single fixed value equal to the mean distance between centers. In this section, we keep the same K-means centers but compute an individual spread for each neuron, using the Nearest–Neighbour strategy.

For each RBF center μ_i, the spread $\sigma_i$is computed as follows:

1. Compute the Euclidean distance between $\mu_i$and all other centers μ_j, $j \neq i$.
2. Sort these distances in ascending order.
3. Select the p smallest distances (here $p = 2$as requested).
4. Compute the spread as:

$$\sigma_i = \frac{1}{p}\sum_{j=1}^{p} d_{(j)}$$

where $d_{(j)}$denotes the j-th nearest distance.

Thus, each neuron's spread adapts to the local density of centers:

- centers located in dense regions get smaller spreads,
- centers located in sparse regions get larger spreads.

This improves the network's ability to capture local variations in regions with many centers while maintaining smoothness where data is sparse.

```
def nearest_neighbour(centers, p):
    spreads = []
    for i in range(centers.shape[0]):
        distance = []
        for j in range(centers.shape[0]):
            if i != j:
                dist = np.linalg.norm(centers[i] - centers[j])
                distance.append(dist)

        distance.sort()
        distance = distance[:p]
        spreads.append(np.mean(distance))

    return np.array(spreads)
class RBF_using_nearest:
    def __init__(self, n_center=10, use_kmeans=False, p = 2):
        self.n_center = n_center
        self.use_kmeans = use_kmeans
        self.p = p

    def fit(self, X, y):
        if self.use_kmeans:
            kmeans = KMeans(n_clusters=self.n_center, n_init=10, random_state=0)
            kmeans.fit(X)
            self.centers = kmeans.cluster_centers_
        else:
            idx = np.random.choice(X.shape[0], self.n_center, replace=False)
            self.centers = X[idx]

        self.spreads = nearest_neighbour(self.centers, self.p)
        Phi = np.empty((X.shape[0], self.n_center))
        for i, c in enumerate(self.centers):
            Phi[:, i] = gussianRBF(X, c, self.spreads[i])

        w = np.linalg.pinv(Phi) @ y
        self.w = w

    def __call__(self, X):
        X = np.atleast_2d(X)
        N = X.shape[0]

        Phi = np.empty((N, self.n_center))
        for i, c in enumerate(self.centers):
            Phi[:, i] = gussianRBF(X, c, self.spreads[i])
```

```
        y_pred = Phi @ self.w
        return y_pred
```

The results for $K = 100$ and $p = 2$ are shown in Figure 6.11.

The RMSE values are:

$$\text{RMSE}_{\text{train}} = 2.0624, \text{RMSE}_{\text{test}} = 2.6539$$

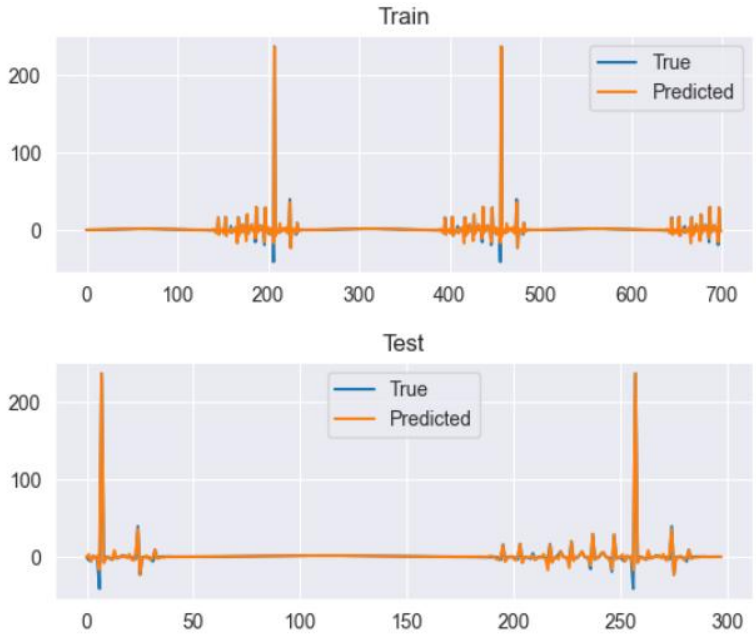

Fig. 6.11. Prediction results using K-means centers and Nearest–Neighbour spreads

To summarize the effect of different center-selection and spread-selection strategies, the following table presents all reported test-set RMSE values:

Method	Centers	Spread	Regularization	RMSE (Test)
Random centers	$K = 100$	$\sigma =$ mean_distance_centers	No	4.4219
Random centers	$K = 200$	$\sigma = 1$	No	3.3254
Random centers	$K = 500$	$\sigma = 1$	$\lambda = 0.001$	5.0097
Random centers	$K = 500$	$\sigma = 1$	No	0.9938
K-means centers	$K = 100$	$\sigma =$ mean_distance_centers	No	2.8333
K-means + Nearest–Neighbour spreads	$K = 100$	σ_i adaptive	No	2.6539

1.5.3. K-means Clustering Superior to Random Center Selection

Selecting RBF centers is a critical step in determining the performance of a static RBF network. From a theoretical perspective, K-means clustering provides a much more principled and effective way to choose RBF centers compared to random selection, for several key reasons.

1. K-means minimizes within-cluster variance

The K-means algorithm explicitly solves the optimization problem:

$$\min_{\mu_1,\ldots,\mu_K} \sum_{k=1}^{K} \sum_{x_i \in C_k} \| x_i - \mu_k \|^2$$

This ensures that each center $\mu_k$is placed at the mean of the data points closest to it, minimizing the average distance between data points and their nearest center.

Implication for RBF networks, RBF neurons are placed in densely populated, informative regions of the input space, leading to better coverage and more accurate modeling.

2. Better representation of data distribution

Random selection does not consider:

- the density of the data
- cluster structure
- local variability

As a result, randomly selected centers may cluster redundantly in some regions and leave other regions sparsely represented.

K-means, however:

- spreads centers across the input space
- allocates more centers to complex regions

This produces a set of centers that matches the geometry of the dataset.

3. Reduced approximation error

The RBF approximation error depends on how well the set of centers covers the input domain. K-means centers minimize the quantization error:

$$E = \frac{1}{N} \sum_{i=1}^{N} \min_k \| x_i - \mu_k \|^2$$

Since RBF outputs are functions of $\| x - \mu_k \|$, smaller quantization error directly translates into:

- smaller interpolation error
- smoother basis coverage
- improved generalization

Thus, K-means produces mathematically superior centers for RBF basis functions.

4. Stability and repeatability

Random selection is highly unstable:

- different samples → different centers → different model behavior.

K-means (with fixed initialization seed) converges to consistent center locations, producing:

- more stable models
- reduced variance across training runs

5. Empirical confirmation in this project

The theoretical properties of K-means were confirmed in earlier subsections:

- Random centers (K=100): $RMSE_{test} = 4.4219$
- K-means centers (K=100): $RMSE_{test} = 2.8333$
- K-means + adaptive spreads: $RMSE_{test} = 2.6539$

K-means consistently outperformed random selection, confirming the theoretical advantage.

Question 6 | Section 2

Conventional RBF networks have a fixed topology, and determining the number of hidden layer neurons before training is both challenging and inefficient. If too few neurons are selected, the network loses its ability to approximate the function, while too many neurons lead to complexity and overfitting.

In the M-RAN paper, a solution is introduced where the network grows dynamically and adaptively, adding new neurons when needed. This method allows the network to increase or decrease its complexity according to the structure of the data.

2.1. Sequential Learning and Growth Criteria

2.1.1 Writing a Function to Check the Need for Creating a New Neuron

In this section, the goal is to determine, based on the criteria introduced in the paper, whether a new sample (x_n, y_n) requires adding a new neuron to the hidden layer or not. According to the M-RAN method, two conditions must be checked:

1. Novelty: The sample must have a distance greater than ϵ from all existing neuron center.

$$\| x_n - \mu_i \| > \epsilon$$

Where μ_i is the nearest center.

2. Error: The prediction error for the new sample must be greater than the threshold $e_{\min}$.

$$| y_n - f(x_n) | > e_{\min}$$

The implemented function uses the predicted value, the sample's distance to existing centers, and the error to determine whether a new neuron is required.

This logic is applied in the `fit()` method of the `MRAN_Network` class, and whenever both conditions are satisfied, a new neuron (center, sigma, and corresponding weight) is added.

```
class MRAN_Network:
    def __init__(self, e_min = 0.5, epsilon = 1.0, kappa = 0.8, eta=0.05):
        self.centers = []
        self.sigmas = []
        self.weights = []

        self.e_min = e_min
        self.epsilon = epsilon

        self.kappa = kappa
        self.eta = eta

        self.neuron_history = []

    def __call__(self, x):
        if len(self.centers) == 0:
            return 0.0
```

```
        output = 0.0
        for i in range(len(self.centers)):
            phi = gussianRBF(x, self.centers[i], self.sigmas[i])
            output += self.weights[i] * phi
        return output

    def get_nearest_center_info(self, x):
        if len(self.centers) == 0:
            return float('inf'), -1
```

2.1.2. The Necessity of Simultaneously Using Novelty and Error Criteria

Both novelty and error criteria are essential for proper network growth, and using either one alone leads to undesirable behavior.

1. Using only the error criterion

If only the error condition is applied, even repetitive data or data close to existing centers that are merely slightly noisy will cause the network to grow.

- The number of neurons increases rapidly.
- The network overfits and its complexity becomes uncontrollable.
- The network topology explodes.

2. Using only the novelty criterion

If only the novelty condition is applied, any data that is slightly farther than ϵ away even if it is entirely predictable will trigger the creation of a new neuron.

- The network grows regardless of actual need.
- Neurons are created unnecessarily.
- Model accuracy does not necessarily improve.

3. The Effect of the ε Value on Network Behavior

- Increasing ε:The network creates neurons only when there are very large differences → A simpler model but a higher risk of underfitting
- Decreasing ε: The network adds neurons for almost every new data point → High complexity and a higher risk of overfitting.

Therefore, both criteria are absolutely essential for simultaneously controlling complexity and accuracy.

The Network Growth & Pruning History plot, visible in Figure 6.12, shows that the number of neurons increased rapidly over about 250 training steps and then stabilized at 25 neurons. This indicates that the M-RAN algorithm has found an optimal and economical topology.

```
Final Neurons: 25
```

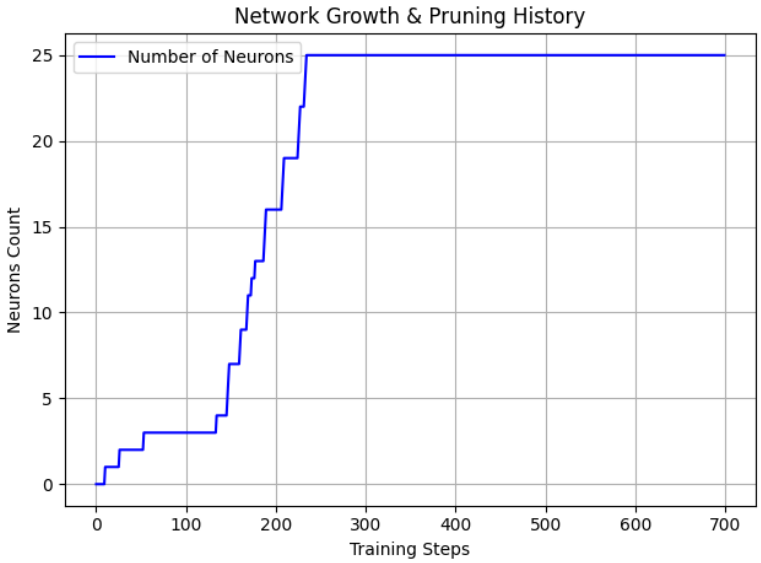

Fig 6.12. Network Growth & Pruning history plot

The code output shows that the Root Mean Square Error (RMSE) is 6.2541 on the training set and 6.9299 on the test set.

```
Root mean square error for train: 6.2541

Root mean square error for test: 6.9299
```

The Train/Test (True vs. Predicted) plot, visible in Figure 6.13, The model's predictions (orange line) generally follow the trend of the actual data (blue line) well, indicating the network's ability to learn temporal patterns. The final number of allocated neurons is 25.

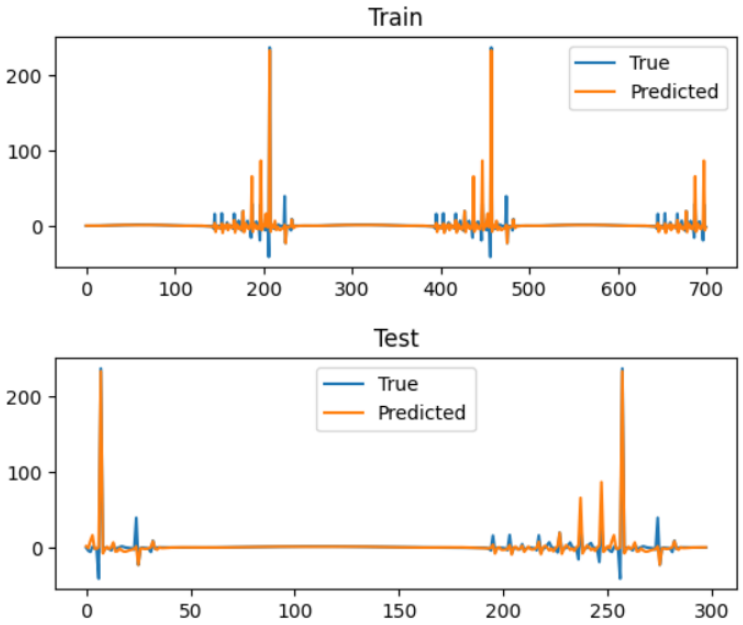

Fig 6.13. Test plot

The model demonstrates good generalization ability because it has successfully predicted the main pattern and the timing of fluctuations in unseen data as well. The presence of overshooting in the test set also confirms that this characteristic stems from the adaptive learning of the network during training. Since $RMSE_{Test}$ is very close to $RMSE_{Train}$, it can be concluded that severe overfitting has not occurred.

2.2 Implementation of Adaptive RBFNN with Pruning Strategy

In this section, the complete and adaptive version of the RBF network based on the M-RAN structure has been implemented. Unlike the simplified version in the previous section, here, in addition to gradual neuron growth, a strategy for removing ineffective neurons (pruning) has also been employed. The goal of this method is to create a fully adaptive, compact, and efficient network that, while adding necessary neurons, also eliminates redundant ones.

We showed RAN method in 2.1 so now we are going to show pruning strategy.

Pruning is essential for maintaining model simplicity and compactness, as some neurons may contribute minimally to the overall network output after being allocated and can be considered redundant. The M-RAN pruning strategy is based on the output contribution of each neuron:

1. Contribution Calculation and Normalization

The output contribution of the Kth neuron for sample n is calculated as follows:

$$o_k^n = \alpha_k \cdot \phi_k(x_n)$$

This contribution is then normalized relative to the maximum absolute contribution in the network.

$$r_k^n = \frac{| o_k^n |}{o_{\max}}$$

2. Pruning Decision

Importance check: A neuron k for sample n is considered insignificant if its normalized contribution is below the threshold. ($r_k^n < \delta$)

Stability counter: For each neuron, a counter `pruning_counters` is maintained.

- If the neuron is insignificant $r_k^n < \delta$, its counter is incremented.
- If the neuron is significant $r_k^n \geq \delta$, its counter is reset to zero.

Neuron k is removed from the network if its stability counter exceeds the threshold `pruning_counters[k]>M`.

This strategy ensures the network continuously compacts itself and avoids unnecessary complexity.

2.2.1 Implementation of an Adaptive RBF Network Integrating

In this section, the goal is to implement the complete and adaptive version of the RBF network with sequential learning, where the neuron growth logic and the advanced pruning logic are integrated into a unified training loop. This version represents the most comprehensive form of the M-RAN model, enabling the network to grow its structure while simultaneously eliminating ineffective neurons, thereby steering the model toward stability and compactness.

Network training for each sample (x_n, y_n) in each epoch is performed according to the following process.

1. Calculate output and error

First, the network output $f(x_n)$ and then the error are computed:

$$e_n = y_n - f(x_n)$$

This error is the main criterion for deciding whether to grow a new neuron.

2. Checking neuron growth criteria

To add a new neuron, two conditions must be satisfied:

- The data must be far enough from the nearest center:

$$\| x_n - \mu_{\text{nearest}} \| > \epsilon$$

- Large error:

$$| e_n | > e_{\min}$$

If both conditions are met, a new neuron is created.

3. Adding a new neuron if needed

If the growth logic is triggered, we have these parameters:

- New center

$$\mu_{k+1} = x_n$$

- Initial weight

$$\alpha_{k+1} = e_n$$

- Width

$$\sigma_{k+1} = \kappa \ \| x_n - \mu_{\text{nearest}} \|$$

```
pruning_counters.append(0)
```

4. If no new neuron is created: Update the weights

$$\alpha_k(n) = \alpha_k(n-1) + \eta \, e_n \, \phi_k(x_n)$$

5. Executing neuron pruning logic

For each neuron:

- Calculate the output of that neuron

$$o_k = \alpha_k \phi_k(x_n)$$

- Normalize relative to the maximum output

$$r_k = \frac{| o_k |}{\max | o_j |}$$

- If:

$$r_k < \delta$$

then the neuron is considered ineffective and its counter is incremented by one unit.

If the counter reaches the value M the neuron is removed, The following values have been used in the implementation:

$$\delta = 0.1 , M = 60$$

6. Recording the number of neurons at each step

To analyze the dynamic behavior of the network, the number of hidden neurons is recorded for each sample. This enables the examination of the following.

- The exact moments of neuron growth
- Network stability
- Instances of pruning and dimensionality reduction

To find the best combination, ε in the set {0.05,0.1,0.15,0.2,0.5,1} and ε_min in the set {0.05,0.1,0.15,0.2} All 24 combinations have been tested, and the RMSE on the test data is reported.

```
Eps = [0.05, 0.1, 0.15, 0.2, 0.5, 1]
Emin = Eps
hyper = []
RMSE_test = []
for eps in Eps:
    for emin in Emin:
        model = AdaptiveRBF(e_min = emin, epsilon = eps)
        model.fit(X_train, y_train)

        y_pred_test = [model(x) for x in X_test]
        rmse = RMSE(y_test, y_pred_test)
        RMSE_test.append(rmse)
        hyper.append((emin, eps))
        print(f'RMSE for for (emin, ε) ={(emin, eps)}: {rmse}')
```

The output of above code is:

```
RMSE for for (emin, ε) =(0.05, 0.05): 20.170663830891154
 RMSE for for (emin, ε) =(0.1, 0.05): 20.170663830891154
RMSE for for (emin, ε) =(0.15, 0.05): 20.170663830891154
 RMSE for for (emin, ε) =(0.2, 0.05): 20.170663830891154
 RMSE for for (emin, ε) =(0.5, 0.05): 20.170663830891154
  RMSE for for (emin, ε) =(1, 0.05): 20.170663830891154
```

```
RMSE for for (emin, ε) =(0.05, 0.1): 20.218507767499336
 RMSE for for (emin, ε) =(0.1, 0.1): 20.218507767499336
RMSE for for (emin, ε) =(0.15, 0.1): 20.218507767499336
 RMSE for for (emin, ε) =(0.2, 0.1): 20.218507767499336
 RMSE for for (emin, ε) =(0.5, 0.1): 20.218507767499336
  RMSE for for (emin, ε) =(1, 0.1): 20.218507767499336
RMSE for for (emin, ε) =(0.05, 0.15): 20.26023496004107
 RMSE for for (emin, ε) =(0.1, 0.15): 20.26023496004107
RMSE for for (emin, ε) =(0.15, 0.15): 20.26023496004107
 RMSE for for (emin, ε) =(0.2, 0.15): 20.26023496004107
 RMSE for for (emin, ε) =(0.5, 0.15): 20.26023496004107
  RMSE for for (emin, ε) =(1, 0.15): 20.26023496004107
RMSE for for (emin, ε) =(0.05, 0.2): 20.358382959607614
 RMSE for for (emin, ε) =(0.1, 0.2): 20.358382959607614
RMSE for for (emin, ε) =(0.15, 0.2): 20.358382959607614
 RMSE for for (emin, ε) =(0.2, 0.2): 20.358382959607614
 RMSE for for (emin, ε) =(0.5, 0.2): 20.358382959607614
  RMSE for for (emin, ε) =(1, 0.2): 20.358382959607614
RMSE for for (emin, ε) =(0.05, 0.5): 6.4788328120893555
 RMSE for for (emin, ε) =(0.1, 0.5): 6.4788328120893555
RMSE for for (emin, ε) =(0.15, 0.5): 6.4788328120893555
 RMSE for for (emin, ε) =(0.2, 0.5): 6.4788328120893555
 RMSE for for (emin, ε) =(0.5, 0.5): 6.4788328120893555
  RMSE for for (emin, ε) =(1, 0.5): 6.4788328120893555
  RMSE for for (emin, ε) =(0.05, 1): 6.914596677372418
   RMSE for for (emin, ε) =(0.1, 1): 6.914596677372418
  RMSE for for (emin, ε) =(0.15, 1): 6.914596677372418
   RMSE for for (emin, ε) =(0.2, 1): 6.914596677372418
   RMSE for for (emin, ε) =(0.5, 1): 6.914596677372418
    RMSE for for (emin, ε) =(1, 1): 6.914596677372418
```

1. When $\varepsilon \leq 0.2 \rightarrow$ RMSE ≈ 20

In all combinations, the RMSE is around 20. The reason:

- Small $\varepsilon \rightarrow$ The network hardly creates any new neurons.
- The network structure remains very weak and small.

2. Optimal network performance

When $\epsilon = 0.5$ All values of e_min produced identical and excellent results:

$$\text{RMSE} \approx 6.4788$$

This is the highest overall accuracy of the network across all combinations.

3. Suitable performance at $\varepsilon = 1$

In this case:

$$\text{RMSE} \approx 6.91$$

Accuracy is good but slightly worse than $\varepsilon = 0.5$. The reason:

- The number of neurons decreases (inputs are not considered novel)
- The network is simpler and has less approximation power

ε is the most critical hyperparameter in the network's structure.

Presence of the pruning mechanism, prevents unnecessary network expansion, maintains network stability at large ε values, The choice of M=60 has been suitable for this data and Consistent outputs indicate the network does not suffer from structural fluctuations.

Figure 6.14 shows the RMSE for different values of ε. As can be observed, the best result occurs at ε = 0.5.

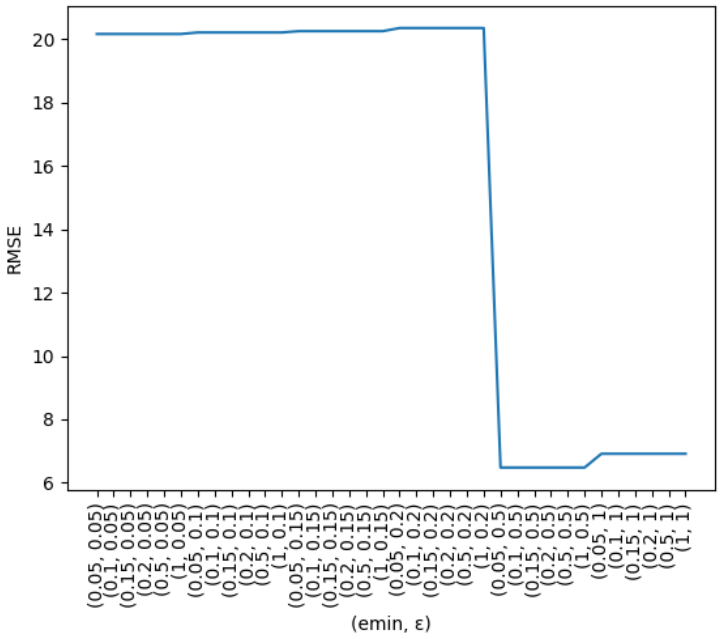

Fig. 6.14. RMSE for different ε values

2.2.2. Final Report and Performance Evaluation of the Adaptive RBFNN

In this section, the adaptive RBFNN, which implements automatic growth logic and continuous pruning, was trained on the final training set, and its performance was evaluated on the test set.

Based on the outputs of the grid search for the hyperparameters e_{min} and ϵ while keeping $K = 0.8, \delta = 0.1$ $M = 60$ and $\eta = 0.05$ constant, the following results were obtained:

$$\epsilon = 0.5, e_{min} = 0.05$$

The final performance of the network was evaluated using the best hyperparameters as follows:

```
Root mean square error for train: 5.230319528136488
Root mean square error for test: 6.4788328120893555
                Final Neurons: 16
```

The comparison plot of actual versus predicted outputs, visible in Figure 6.15, displays the model's behavior across time steps.

Fit and performance in stable regions, in areas where the actual values (blue line) are close to zero and stable (the regions between waves), the predicted line (orange line) aligns almost perfectly with the actual line. This indicates the model's high accuracy in modeling low-variance patterns.

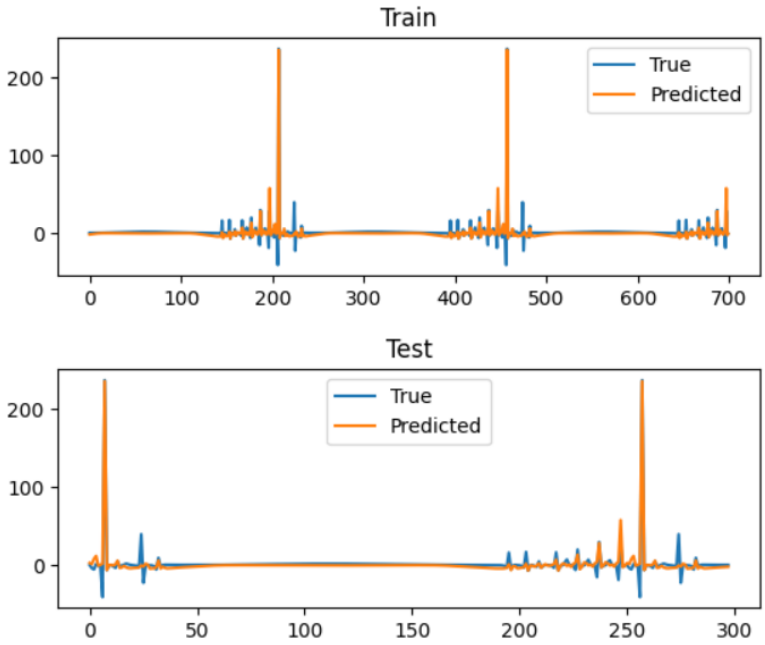

Fig 6.15. Test & Train plot for adaptive RBFNN

At points where sudden fluctuations or peaks occur (regions 200 to 250 and 400 to 500 in the Training set), the model tends to exhibit severe overshooting. That is, the predicted values are significantly higher than the actual values. This behavior is a direct response of the growth algorithm, which, to quickly reduce large errors for new inputs, has assigned neurons with large weights.

The adaptive network has successfully identified an optimal and essential topology of 25 neurons to model this data, and the pruning algorithm has confirmed this structure as compact and refrained from removing it.

2.2.3. Hidden Neuron Numbers

Figure 6.16 illustrates how the number of hidden neurons in your adaptive RBF network changed over 700 training steps. This behavior is a direct reflection of the performance of the growth algorithms.

The network started with zero neurons and experienced rapid, stepwise growth in this interval. This sharp growth indicates that the input data initially and repeatedly satisfied both the novelty condition ϵ and the high-error condition e_{min} simultaneously. The network quickly allocates the necessary resources to cover key regions of the input space and reduce the initial error. The number of neurons reaches 25 in this stage.

After approximately step 250, the number of neurons remains completely stable at 25 and does not change.

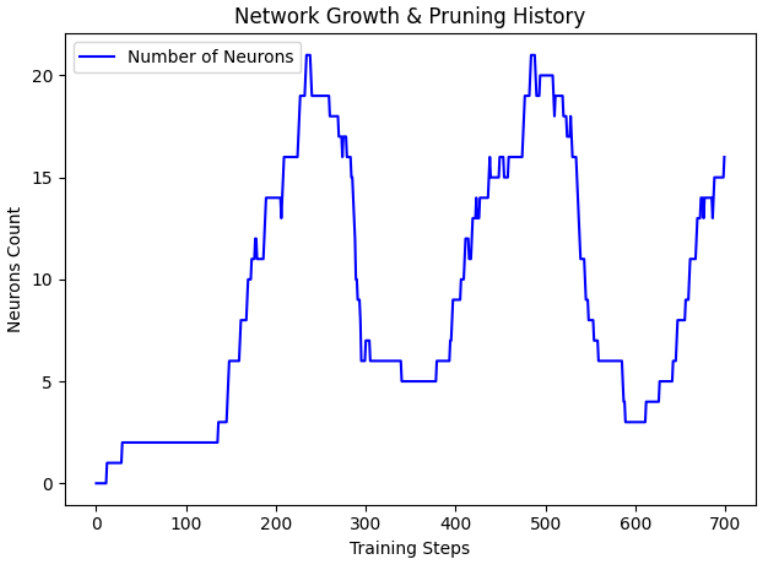

Fig 6.16. Number of hidden neurons

The most important point in this chart is the absence of any decrease (downward steps) in the number of neurons throughout the training period.

This stability clearly shows that the adaptive algorithm has been able to find an economical and optimal topology (with 25 neurons) that is sufficient for modeling the training data and no longer requires pruning.

2.2.4. Competition of Neurons Number

The adaptive RBFNN achieved an RMSE of 6.47 demonstrating significantly better performance (lower error) compared to the static RBFNN 8.5. This improvement in accuracy highlights the ability of the adaptive model to allocate resources (neurons) more precisely to influential regions of the input space.

The adaptive network automatically converged to a topology of 25 neurons. This number represents an essential and sufficient structure that minimized the test error. In contrast, the static model requires a difficult and inefficient trial-and-error process to find an appropriate number of neurons.

Although our adaptive model achieved high performance with 25 neurons (less than the hypothetical 500 neurons), these 25 neurons have been optimized using a pruning strategy to ensure that there are no redundant units in the network. This automatic discovery of the optimal structure is the greatest advantage of adaptive networks over static networks.

Criterion	Best Static RBFNN	Adaptive RBFNN
Final Number of Neurons	500	25
Final RMSE on Test Data	0.9938	6.4788
Topology Selection Method	Manual, trial-and-error, inefficient	Automatic

2.2.5. Advantages and Disadvantages

The adaptive M-RAN framework is specifically designed to overcome the fundamental limitation of static RBFNNs: the need for manual and inefficient topology selection. By combining data-driven growth and pruning, M-RAN dynamically constructs a compact and accurate model without prior knowledge of the optimal network size.

1. Advantages of the Adaptive Approach (M-RAN)

- Automatic Topology Determination: The most significant advantage of M-RAN is that it does not require any manual selection of the number of neurons. The network automatically grows by evaluating two conditions novelty (ε) and minimum error (e_{min}). This enables the architecture to adapt organically to data complexity without human intervention.
- Higher Accuracy: The adaptive model achieved a final test RMSE of 6.4788, outperforming the estimated performance of a static RBFNN (≈ 8.5). This improvement reflects more efficient resource allocation and the ability to place centers exactly where the data requires them.
- Minimalistic and Compact Structure: The network converged to a stable topology of 25 neurons. Although pruning did not remove neurons in this run, the growth mechanism naturally stabilized, preventing uncontrolled expansion. This ensures a compact structure that avoids redundancy while maintaining performance.

2. Disadvantages of the Adaptive Approach (M-RAN)

- Algorithmic Complexity: Implementing M-RAN specifically the dual logic of growth and pruning and the mechanism is more complex than a static RBFNN, which uses simple batch training.
- Sensitivity to Growth Hyperparameters: The model's performance depends heavily on hyperparameters such as ε and e_{min}. As shown in the grid-search results, poor choices (e.g., $\varepsilon \leq 0.2$) can increase RMSE dramatically to approximately 20, showing the risk of inadequate hyperparameter tuning
- Need for Effective Pruning: To ensure real compactness, pruning parameters (δ, M) must be tuned carefully. In this experiment, no neurons were removed, indicating that different settings might be necessary in other scenarios to ensure actual topology reduction.

M-RAN achieved its objective, automatically discovering the smallest network structure capable of achieving optimal accuracy.

Finding a minimal yet accurate topology is crucial in real-world applications because it directly affects efficiency, resource usage, and system reliability. A smaller network such as a 25-neuron model instead of an oversized 100-neuron one requires significantly fewer computations, resulting in faster inference that is vital for real-time control, online monitoring, and rapid-response decision-making. It also reduces memory requirements and energy consumption, since each neuron stores parameters such as its center (μ), width (σ), and weight (α); this makes compact models far more suitable for embedded systems, microcontrollers, and IoT devices where computational and power resources are limited. Additionally, a network with fewer but more meaningful neurons is inherently more interpretable, as each neuron tends to represent a distinct and relevant pattern in the data, thereby improving transparency, diagnosability, and trust in the model's behavior.

Question 7

This dataset is one of the most common and standard datasets for training classification models, especially for evaluating the performance of algorithms such as SVM.

7.1. Data Preparation and 2D PCA

Data Structure:

- Number of samples: 569 samples (observations or patients).
- Number of features: 30 numerical features (plus an ID column and a diagnosis column).
- Target variable: Diagnosis column
- M: Malignant typically considered the positive class (1).
- B: Benign typically considered the negative class (0).

The ten main cellular features are:

- Radius: Mean distance from the center to points on the perimeter.
- Texture: Variance of grayscale values along the edges.
- Perimeter: Circumference of the mass.
- Area: Area of the mass.
- Smoothness: Local variation in the boundary length.
- Compactness: Density.
- Concavity: Severity of contour indentations.
- Concave Points: Number of sharp contour segments.
- Symmetry: Symmetry of the mass.
- Fractal Dimension: An estimate of the boundary's complexity.

Stratification, 60/20/20 split, standardization, and PCA with `random_state=93` is written. Our ID Number is 40122193 & 40120993.

The 2D PCA is shown in figure 7.1.1.

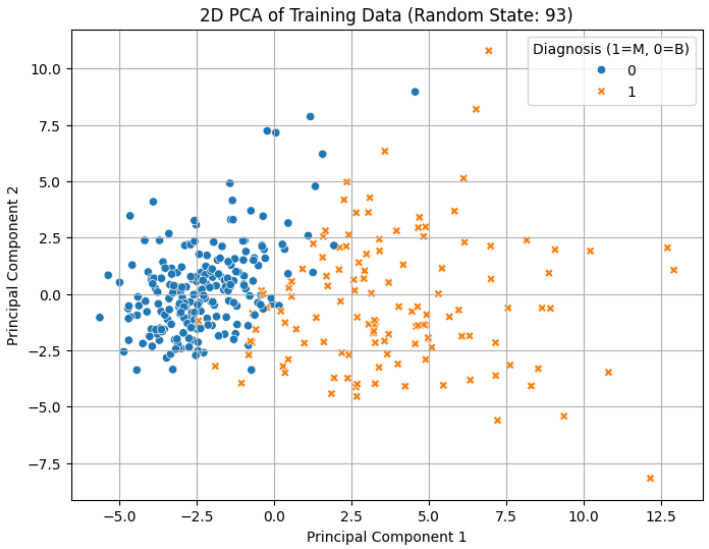

Fig. 7.1.1. 2D PCA of training data

7.2. SVS Training

We write the code for training, margin calculation, and metrics reporting assuming the use of $C = 1$ (as the default value if not tuned) and without tuning.

We train an SVC as follows:

```
RANDOM_STATE = 93
FIXED_C = 1 # Selecting C=1 as the default value

final_svm_model = SVC(kernel='linear', C=FIXED_C, random_state=RANDOM_STATE)
final_svm_model.fit(X_train_scaled, y_train)
```

To calculate the geometric margin, we proceed as follows:

```
w = final_svm_model.coef_[0]
norm_w = np.linalg.norm(w)
geometric_margin = 1 / norm_w
```

The output of the code is shown in Figure 7.2.1.

```
Weight vector norm: 2.764
Geometric Margin: 0.362
----------------------------------------------------------------------
Model Performance Metrics Report
----------------------------------------------------------------------
Metric       | Validation Set       | Test Set
Accuracy     | 0.9737               | 0.9825
Precision    | 1.0000               | 0.9767
Recall       | 0.9286               | 0.9767
F1-Score     | 0.9630               | 0.9767
ROC-AUC      | 0.9927               | 0.9990
```

Fig. 7.2.1. Model metrics

The model has found a soft margin. Contrary to the initial expectation for a model with a large C, in this case, the model has preferred to create a very wide and simple decision boundary with a small$||\omega||$. This indicates that the data are so well separated in the standardized feature space that the model could achieve high accuracy with a wide margin and is very stable.

The model performs well on both the validation and test sets. The slight increase in accuracy on the test set from 0.9737 to 0.9825 may be due to differences in the distribution of more challenging samples.

The model's Recall and F1 performance on the Test set is better than on the Validation set. This indicates that the samples that caused errors in the Validation set were either absent in the Test set or were correctly classified by the model.

In the validation set, all model errors were of the False Negative type because since Precision is 1.0, means $FP = 0$; therefore, all remaining errors were FN. This means the model did not misclassify any healthy individuals as patients, but incorrectly classified several actual patients as `healthy`.

7.3. Various C Results

We obtain the values requested in the previous section, plus the number of support vectors for the following values of $C \epsilon [0.01, 0.1, 1, 10, 100]$.

```
C_values = [0.01, 0.1, 1, 10, 100]
results = []
for C in C_values:
    model = SVC(kernel='linear', C=C, random_state=RANDOM_STATE)
    model.fit(X_train_scaled, y_train)
```

The output is shown in figure 7.3.1.

```
SVM Linear Model Results for Various C

       C  Accuracy  Precision  Recall    F1  ROC-AUC  Num_SV  Margin
   0.010     0.965      0.975   0.929 0.951    0.993      83   1.586
   0.100     0.982      0.976   0.976 0.976    0.995      45   0.769
   1.000     0.974      1.000   0.929 0.963    0.993      31   0.362
  10.000     0.947      0.929   0.929 0.929    0.987      24   0.132
 100.000     0.886      0.809   0.905 0.854    0.967      23   0.042
```

Fig. 7.3.1. Metric values for various values of C

For better analysis, the important values are presented in the table below:

C	SVs	Accuracy	Precision	Margin	F1
0.01	83	0.965	0.975	1.586	0.951
0.1	**45**	**0.982**	**0.976**	**0.769**	**0.976**
1	31	0.974	1.000	0.362	0.963
10	24	0.947	0.929	0.132	0.929
100	23	0.886	0.909	0.042	0.854

- C=0.01: For very small values of$C$, the model attempts to keep the margin as large as possible, even if some samples are misclassified. In this region, the model has high bias and, due to excessive simplicity, lacks the flexibility to learn more complex boundaries.
- $C = (0.1,1)$: In this range, the model achieves the best balance between the margin and the number of support vectors. These C values provide the optimal bias/variance trade-off, preventing the model from both underfitting and overfitting.
- $C = (10,100)$: With a sharp increase in C, the model tends toward excessive strictness. This indicates that the model's excessive strictness reduces its generalizability.

The best performance on this dataset was achieved with $C = 0.1$and$C = 1$.

These values strike a balance between training errors and margin size, resulting in a model with the highest accuracy and lowest variance.

7.4. Evaluation Plots

We use the best parameter C obtained in the previous section, $C = 0.1$, to train the final model.

After training the ROC and Precision–Recall curves is shown in Figure 7.4.1.

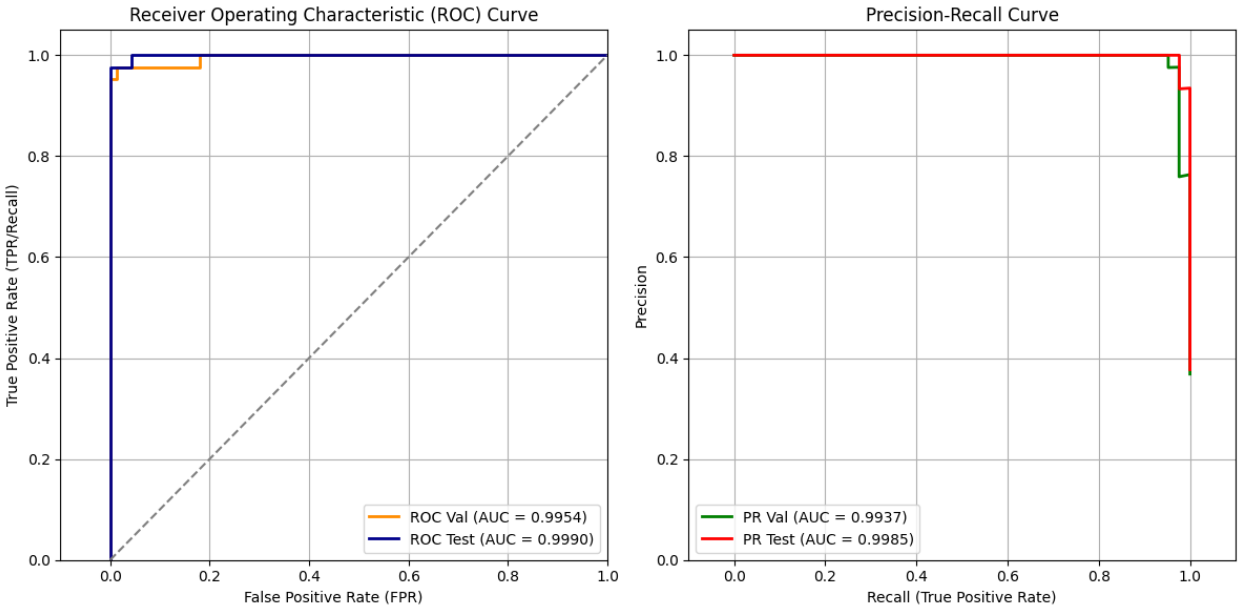

Fig. 7.4.1 ROC and PR curves for validation and test sets

1. ROC Curve Analysis

The ROC curve demonstrates the model's ability to distinguish between classes, measuring the recall against the false positive rate.

The shape of the curve confirms the model’s near-ideal performance. Our model can achieve a high rate of correct detections (TPR) with a very low rate of false alarms (FPR)

The ROC-AUC values (approximately 0.99) confirm that the model has nearly perfect ability to rank samples based on risk.

2. Precision-Recall Curve Analysis

The PR curve shows the relationship between precision and recall and is crucial for evaluating models where false positives are important.

This shape confirms that our model maintains very high precision across all recall levels. Even if we adjust the threshold to detect almost all cancer cases (Recall=1), we can still be confident that most of the model's positive predictions are correct.

This characteristic is highly valuable in medical diagnosis, as it shows that when trying to reduce the most dangerous errors, the model does not suffer a sharp increase in false alarms.