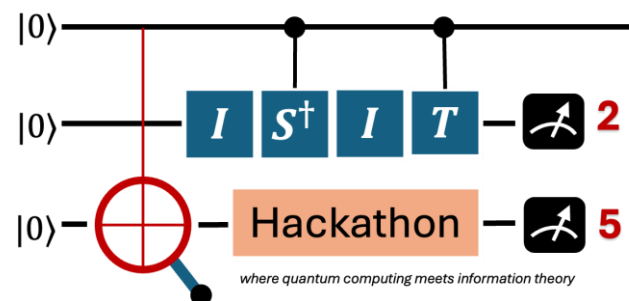


Quantum Kernels and Feature Maps

Concepts and Examples



Outline

- Objective
- Kernel Methods for Machine Learning
- Classification & Clustering
- Kernel Principal Component Analysis
- Conclusion

Machine Learning

- ML finds and studies patterns in data
- Better with higher dimensional feature space
- ML algorithms based on it are kernel methods
- Useful for classification and clustering

Machine Learning

■ Class A:

- (0, 0):

$$\phi(0, 0) = (0^2, 0^2, \sqrt{2} \cdot 0 \cdot 0) = (0, 0, 0)$$

- (1, 1):

$$\phi(1, 1) = (1^2, 1^2, \sqrt{2} \cdot 1 \cdot 1) = (1, 1, \sqrt{2})$$

■ Class B:

- (0, 1):

$$\phi(0, 1) = (0^2, 1^2, \sqrt{2} \cdot 0 \cdot 1) = (0, 1, 0)$$

- (1, 0):

$$\phi(1, 0) = (1^2, 0^2, \sqrt{2} \cdot 1 \cdot 0) = (1, 0, 0)$$

Machine Learning

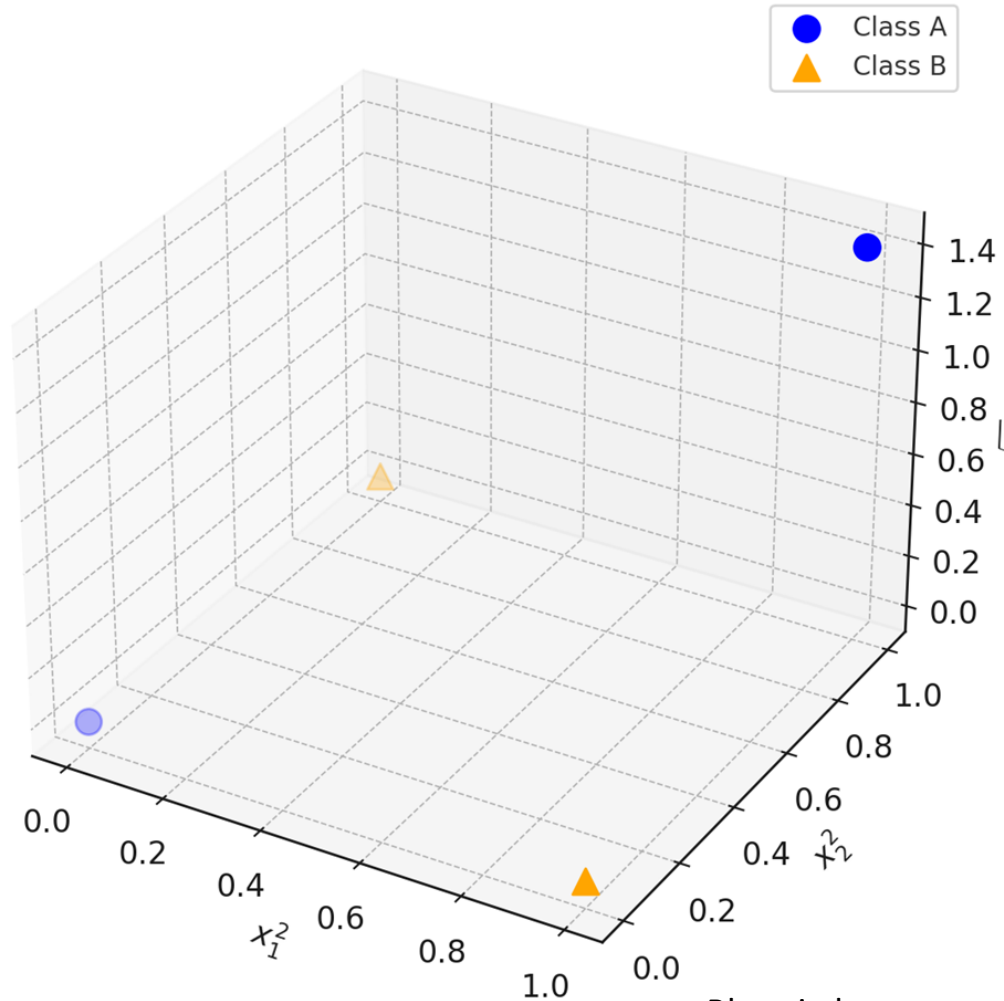
✦ Transformed Points in 3D Space:

Original Point	Class	Transformed Point $\phi(x_1, x_2)$
(0, 0)	A	(0, 0, 0)
(1, 1)	A	(1, 1, $\sqrt{2}$)
(0, 1)	B	(0, 1, 0)
(1, 0)	B	(1, 0, 0)

Now, in this 3D space, it becomes possible to **separate Class A from Class B with a plane**, which was not possible in the original 2D space.

Machine Learning

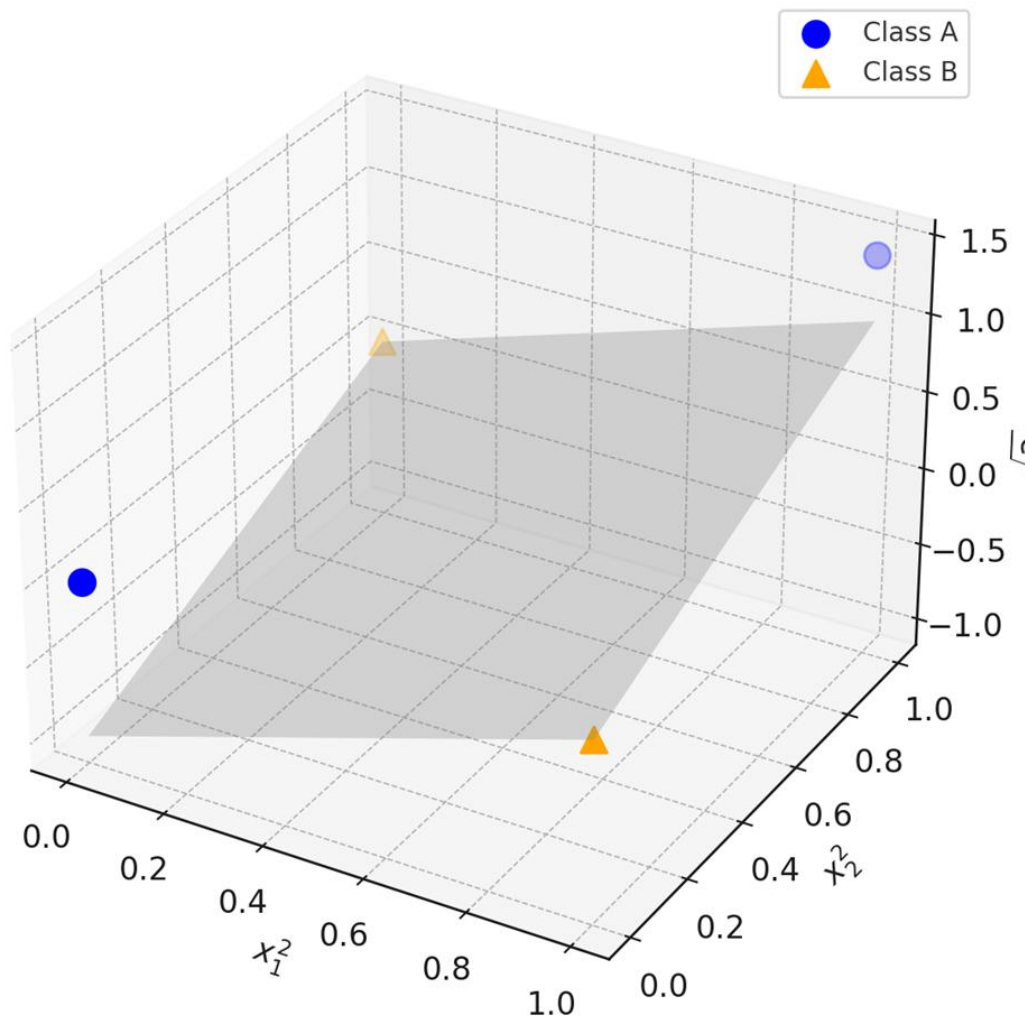
3D Feature Space for XOR Kernel Mapping



- Blue circles represent Class A: $(0, 0)$ and $(1, 1)$
- Orange triangles represent Class B: $(0, 1)$ and $(1, 0)$

Machine Learning

3D Feature Space with Separating Plane

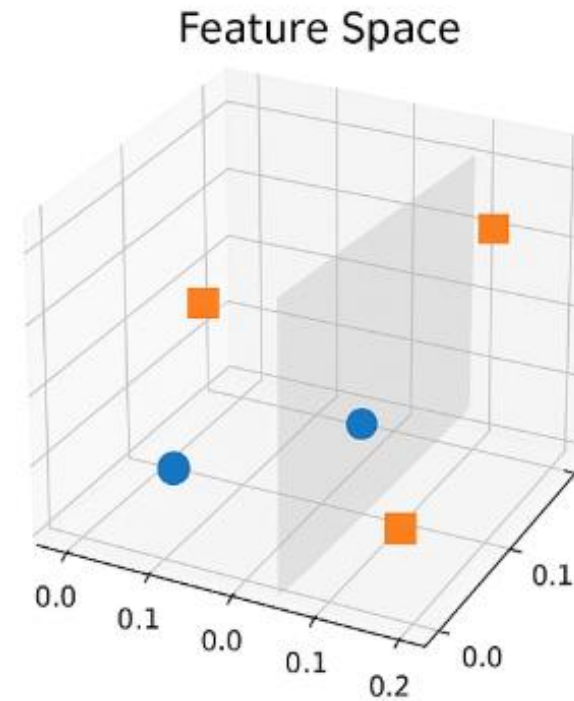
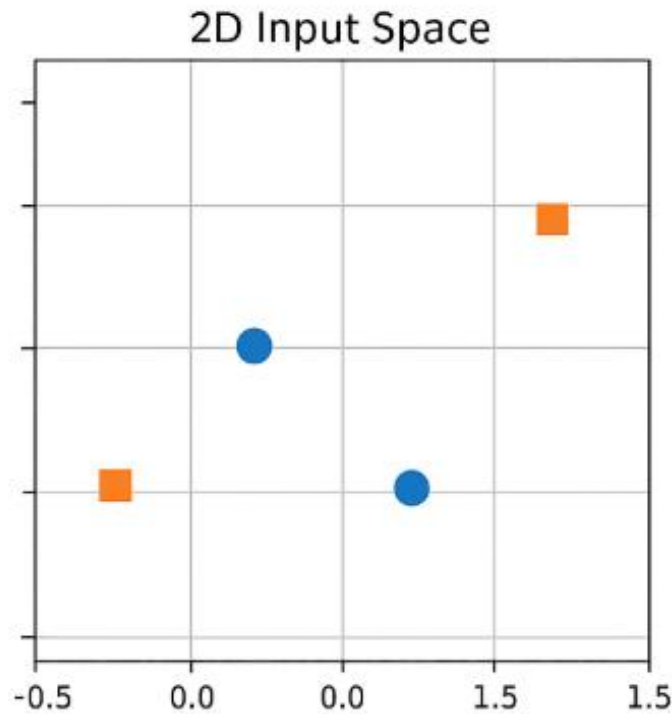


Kernel Methods

- Algorithms using kernel functions
- Best applied in Support Vector Machine (SVM)
- Supervised learning for classification tasks
- For non-linearly separable data spaces using kernels to find boundaries
- Kernel functions imply maps into high dimensional space
- ‘Kernel trick’ & ‘Spectral Clustering’

Kernel Methods

Kernel Methods



Kernel Functions

$$k(\vec{x}_i, \vec{x}_j) = \langle f(\vec{x}_i), f(\vec{x}_j) \rangle$$

- where:
- k is the kernel function
 - \vec{x}_i, \vec{x}_j are n dimensional inputs
 - f is a map from n -dimension to m -dimension space and
 - $\langle a, b \rangle$ denotes the inner product

Dimensional inputs

Space definition for mapping $n \times m$ dimensions

Inner product

Kernel Functions

Common Kernel Functions:

- **Linear kernel:** $K(x, x') = x^T x'$
(No transformation—useful for linearly separable data)
- **Polynomial kernel:** $K(x, x') = (x^T x' + c)^d$
(Allows modeling interactions up to degree d)
- **RBF (Gaussian) kernel:** $K(x, x') = \exp(-\gamma \|x - x'\|^2)$
(Implicitly maps to infinite dimensions, very powerful)

Kernel Functions

For finite data, a kernel function can be represented as a matrix

$$K_{ij} = k(\vec{x}_i, \vec{x}_j)$$

In this context, similarity refers to how aligned or close two data points are in the feature space created by the kernel.

Kernel Functions

✓ Example:

Take two 2D vectors:

- $x = [1, 2]$
- $x' = [3, 4]$

Step 1: Compute the dot product $x^T x'$

$$x^T x' = 1 \cdot 3 + 2 \cdot 4 = 3 + 8 = 11$$

Step 2: Plug into the polynomial kernel formula:

$$K(x, x') = (11 + 1)^2 = 12^2 = 144$$

Kernel Functions

What does 144 mean here?

The result **144** is a measure of similarity between x and x' in the higher-dimensional space induced by this kernel. A **larger number means higher similarity**.

If we had used a different pair of vectors, the number would be lower or higher depending on how aligned those vectors are in the transformed space.

What Would the Mapping Look Like (Optional)?

If we were to explicitly compute the polynomial transformation ϕ , it might look like:

$$\phi([x_1, x_2]) = [x_1^2, x_2^2, \sqrt{2}x_1x_2, \sqrt{2}x_1, \sqrt{2}x_2, 1]$$

This maps 2D vectors to a **6D space**, where the inner product of $\phi(x)$ and $\phi(x')$ would also yield 144. But the kernel function lets us get there without all that extra computation.

Application:

When training an SVM or performing kernel PCA, we would compute **pairwise kernel values** like this for all training examples and store them in a **kernel matrix**. This matrix acts like a similarity grid that the algorithm uses to find structure in the data.

Kernel Functions

In practical terms, when we work with a finite dataset, we represent the kernel function as a matrix called the kernel matrix or Gram matrix. This matrix stores the pairwise similarities between all data points using the kernel function.

When we have a dataset with multiple input vectors, we compute pairwise kernel values for every combination of data points. These values are arranged into a square matrix called the kernel matrix or Gram matrix.

Formally, for a dataset $X = \{x_1, x_2, \dots, x_n\}$, the kernel matrix K is an $n \times n$ matrix where:

$$K_{ij} = K(x_i, x_j)$$

This matrix:

- Is **symmetric** ($K_{ij} = K_{ji}$),
- Is **positive semi-definite**,
- Encodes the **similarities** of every data point with every other data point in the kernel-induced feature space.

This is what algorithms like SVMs and kernel PCA actually operate on.

Kernel Functions

✓ Numerical Example Using Polynomial Kernel

Let's consider a tiny dataset with 3 vectors in 2D:

$$x_1 = [1, 0], \quad x_2 = [0, 1], \quad x_3 = [1, 1]$$

We'll use the polynomial kernel:

$$K(x, x') = (x^T x' + 1)^2$$

12 Step-by-Step Computation:

Let's compute each pairwise kernel value:

- $K(x_1, x_1) = (1 \cdot 1 + 0 \cdot 0 + 1)^2 = (1 + 1)^2 = 4$
- $K(x_1, x_2) = (1 \cdot 0 + 0 \cdot 1 + 1)^2 = (0 + 1)^2 = 1$
- $K(x_1, x_3) = (1 \cdot 1 + 0 \cdot 1 + 1)^2 = (1 + 1)^2 = 4$
- $K(x_2, x_2) = (0 \cdot 0 + 1 \cdot 1 + 1)^2 = (1 + 1)^2 = 4$
- $K(x_2, x_3) = (0 \cdot 1 + 1 \cdot 1 + 1)^2 = (1 + 1)^2 = 4$
- $K(x_3, x_3) = (1 \cdot 1 + 1 \cdot 1 + 1)^2 = (2 + 1)^2 = 9$

Since the matrix is symmetric, we now have:

$$K = \begin{bmatrix} K(x_1, x_1) & K(x_1, x_2) & K(x_1, x_3) \\ K(x_2, x_1) & K(x_2, x_2) & K(x_2, x_3) \\ K(x_3, x_1) & K(x_3, x_2) & K(x_3, x_3) \end{bmatrix} = \begin{bmatrix} 4 & 1 & 4 \\ 1 & 4 & 4 \\ 4 & 4 & 9 \end{bmatrix}$$

Kernel Functions

Interpretation:

- The diagonal entries (4, 4, 9) are the **self-similarities**—they tell us how similar each point is to itself.
 - The off-diagonal entries represent how similar points are to each other.
 - For example, $K(x_1, x_2) = 1$ is the **lowest similarity**, which makes sense since the vectors are orthogonal.
 - $K(x_1, x_3) = 4$ means x_1 and x_3 are more similar—they share more directional alignment in the transformed space.
-

Why Is This Matrix Useful?

This **Gram matrix** becomes the actual input to:

- **SVM classifiers**, where it's used to compute margins,
- **Kernel PCA**, where eigenvectors of this matrix represent principal components,
- **Spectral clustering**, where it's used to define graph Laplacians.

You never need the original vectors—just their kernel similarities.

Dataset



Iris

Donated on 6/30/1988

A small classic dataset from Fisher, 1936. One of the earliest known datasets used for evaluating classification methods.

Dataset Characteristics

Tabular

Subject Area

Biology

Associated Tasks

Classification

Feature Type

Real

Instances

150

Features

4

iris setosa



petal

sepal

iris versicolor



petal

sepal

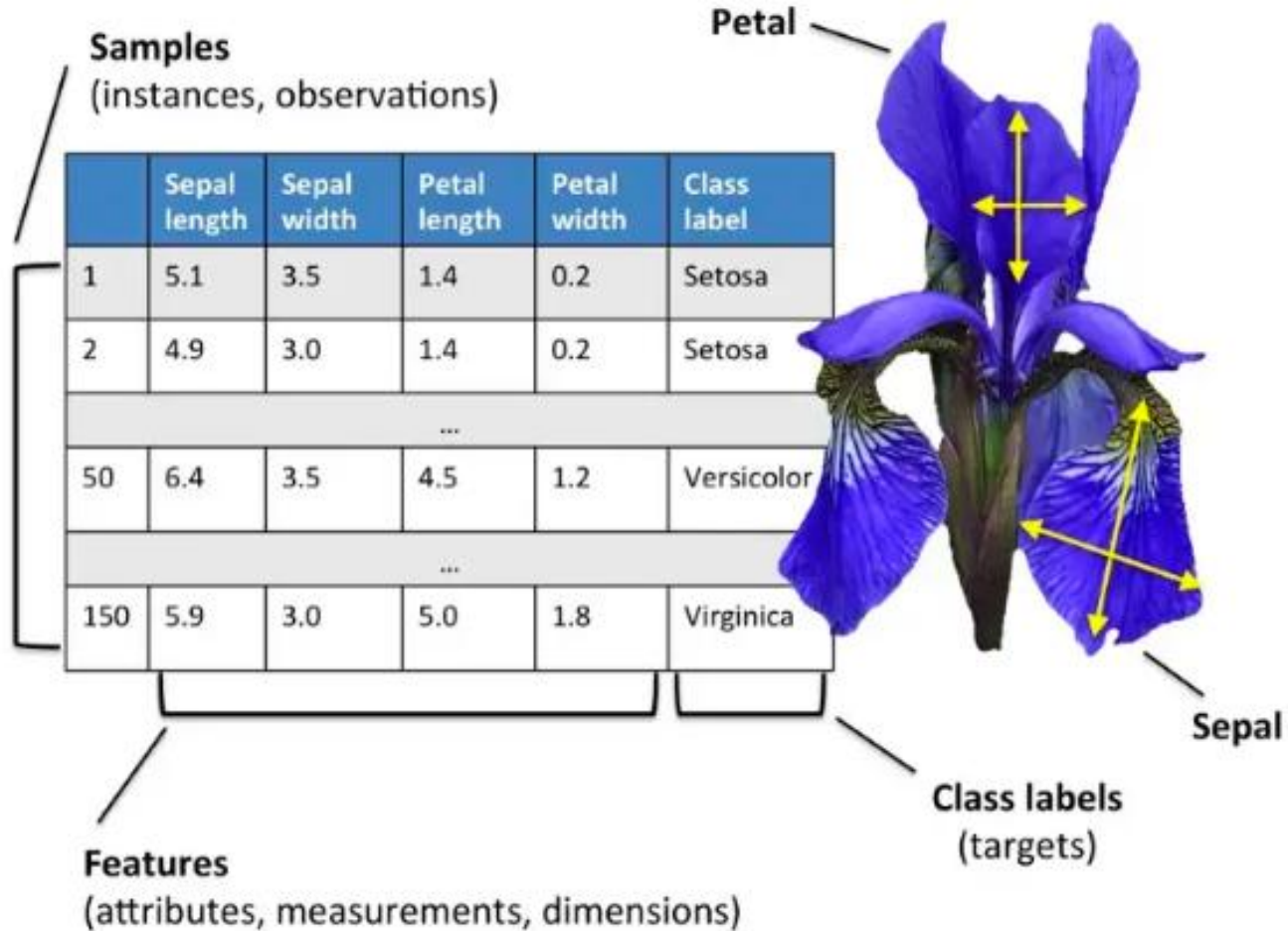
iris virginica



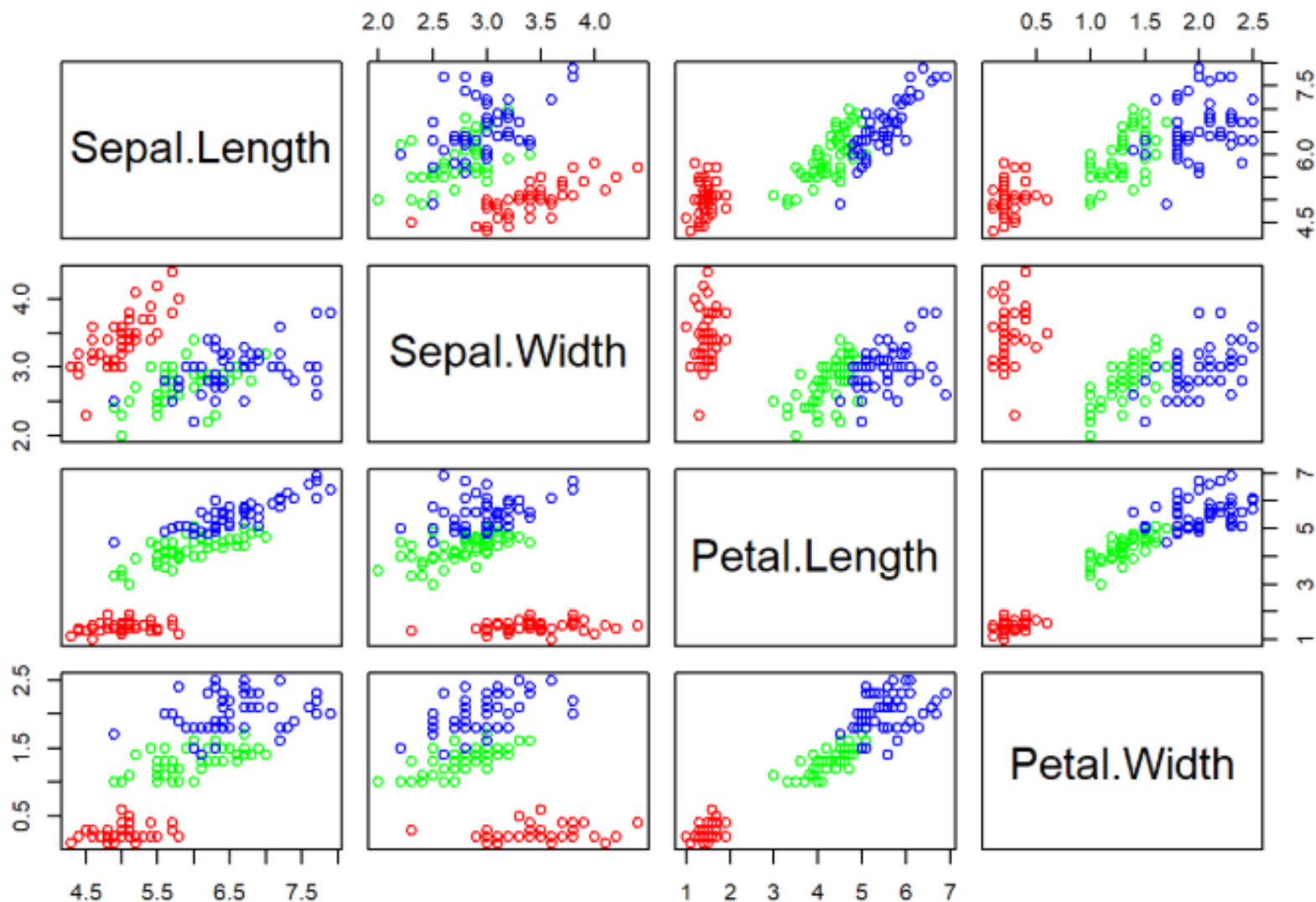
petal

sepal

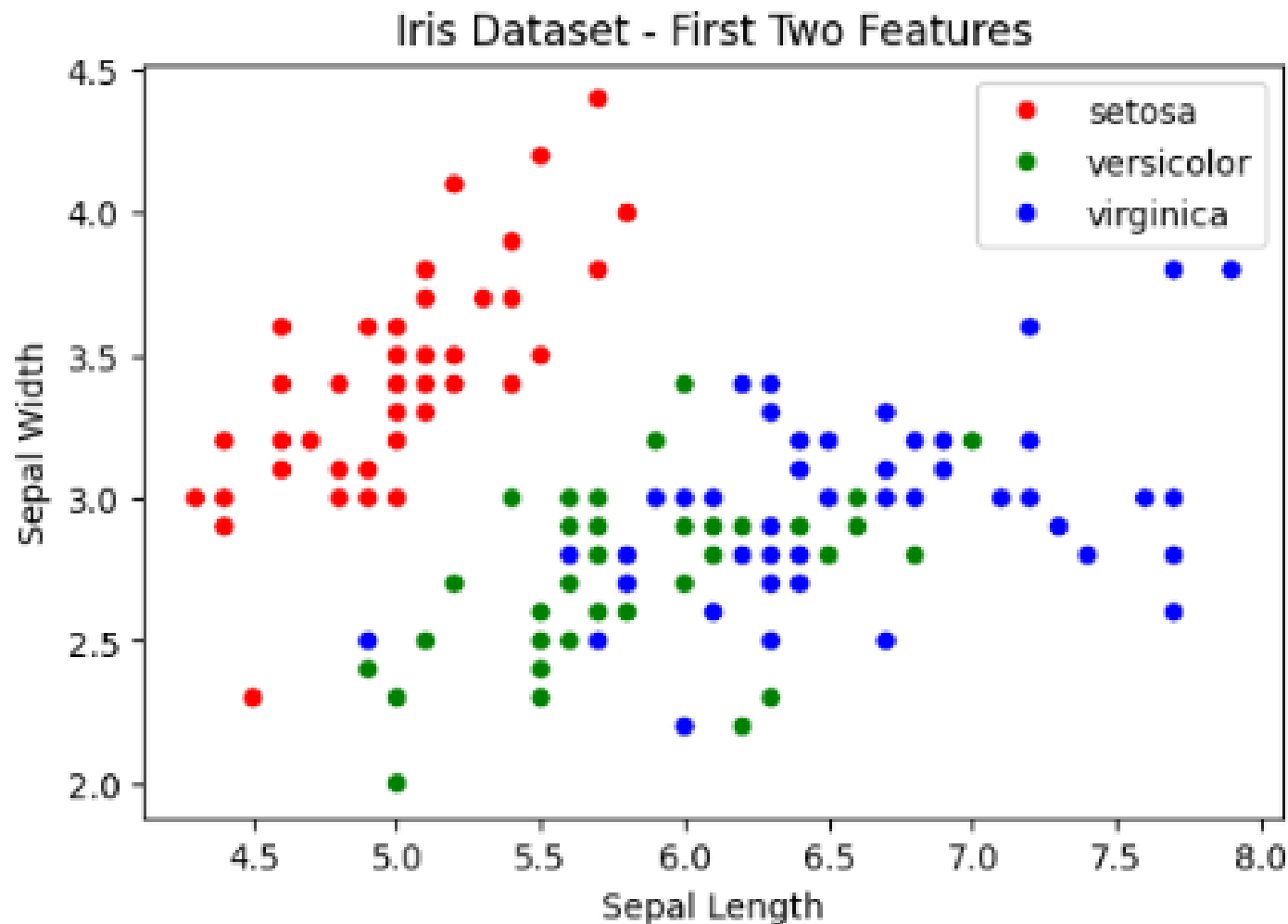
Iris dataset



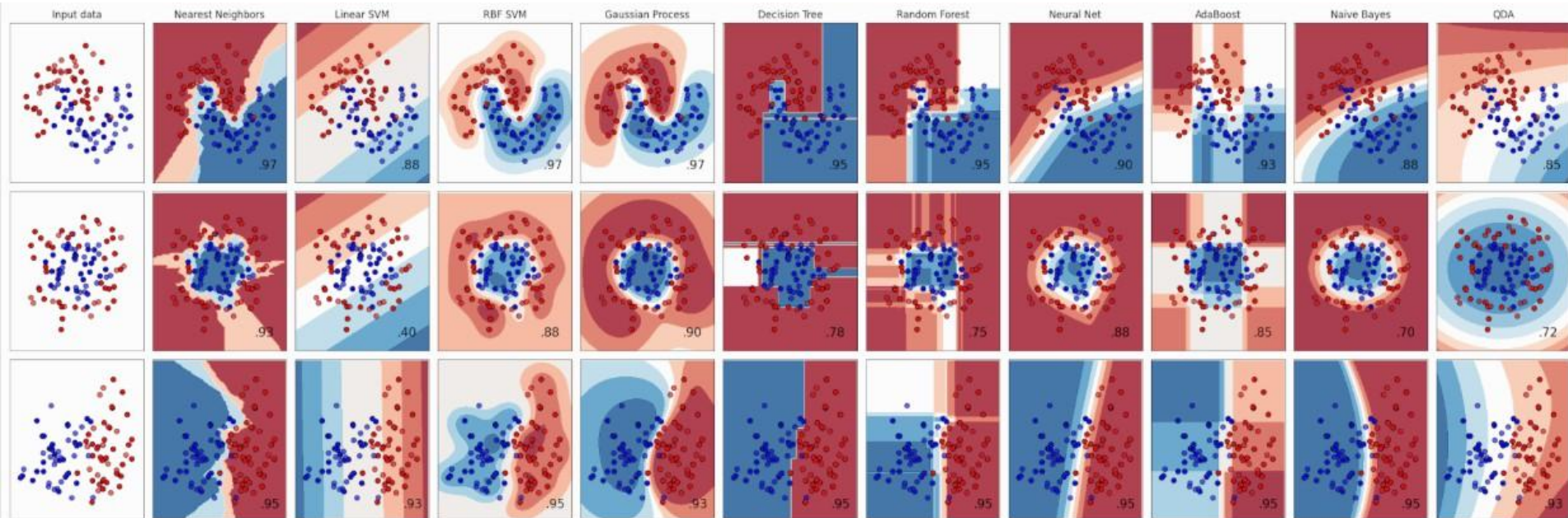
Classification



Classification



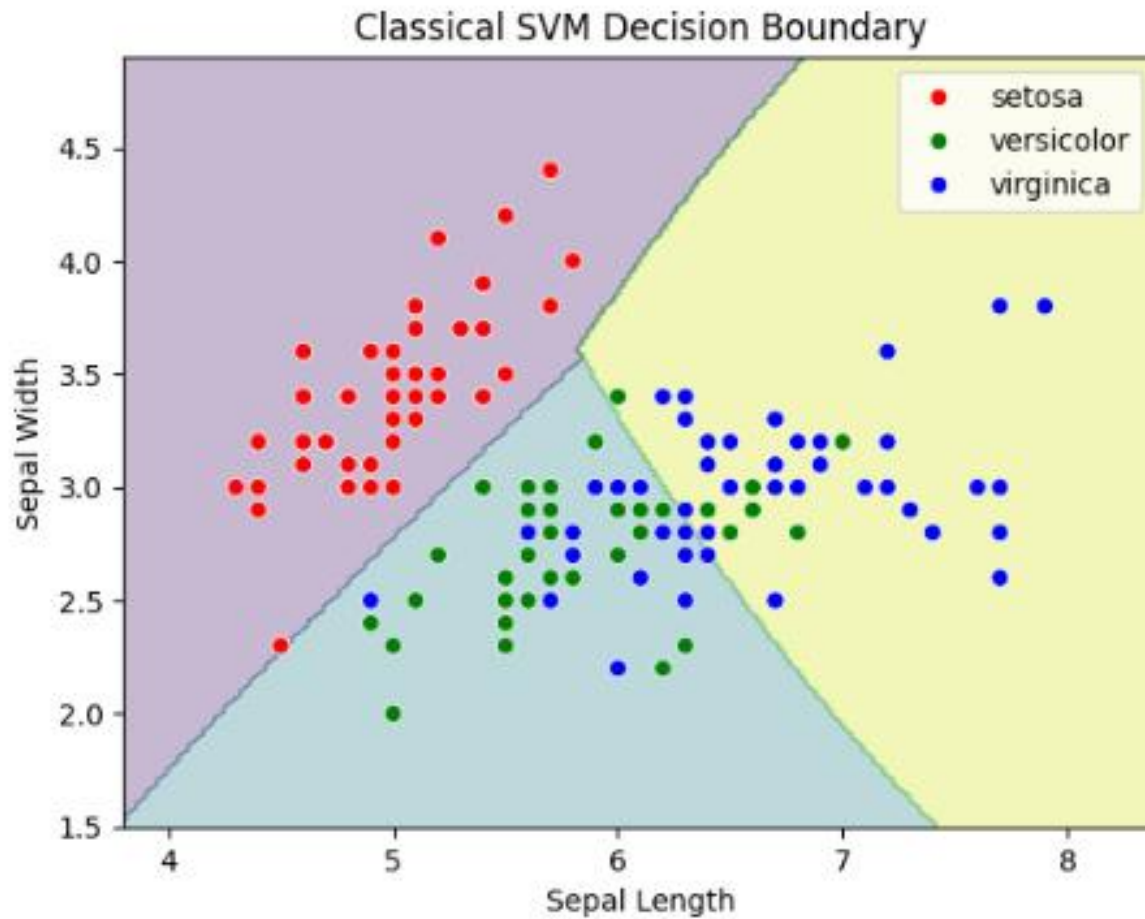
Classification



SVM Training

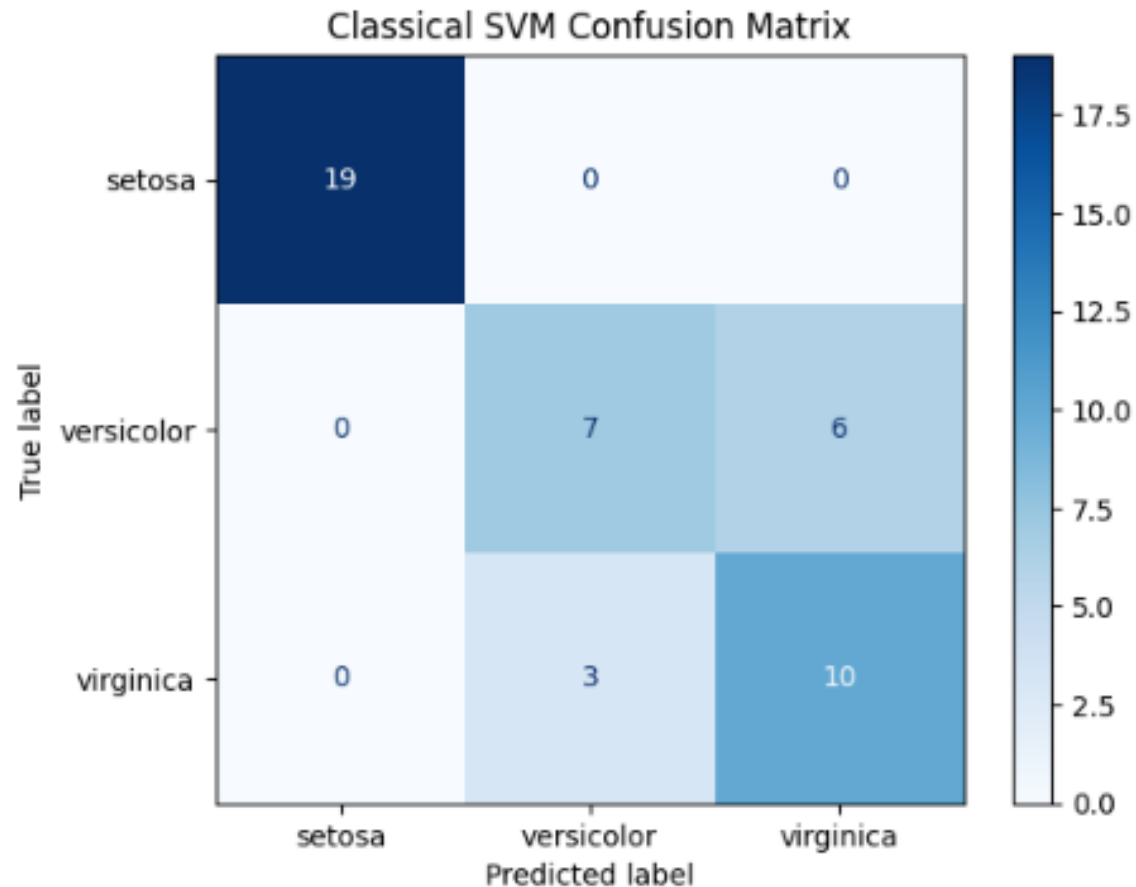
- Use kernel matrix with SVC
- Optimize 'C' with GridSearchCV

Classical SVM classification



Classical SVM Accuracy: 0.80

Classical SVM classification



Classical SVM Accuracy: 0.80

Classical SVM classification

Concept	Purpose	Notes
SVM	Classify data by separating with a hyperplane	Effective in high dimensions
RBF Kernel	Allows nonlinear separation	Common default kernel
Accuracy	Overall correctness	Simple and widely used
Confusion Matrix	Detailed error breakdown	Shows per-class performance
Decision Boundary	Visual intuition	Only possible in 2D or 3D

Qiskit Implementation

- Install Qiskit packages, prepare data
- Defining Qiskit feature map

Feature Map	What it Does	Best For
ZFeatureMap	Applies only Z-rotations	Simple data
ZZFeatureMap	Adds ZZ entanglement between qubits	Captures correlations
PauliFeatureMap	Uses X, Y, Z rotations	High flexibility

Quantum Kernels

- Quantum kernel machine learning
- Quantum states overlapping measure similarity
- Quantum feature mapping selection
- Enable SVMs with complex data boundaries

Quantum Kernels

⚙️ Why Use Quantum Kernels?

Quantum kernels may offer advantages in capturing:

- **Complex correlations** in the data (especially those that classical feature maps might miss),
- **Entanglement**, which naturally models dependencies between features,
- Exponentially **richer feature spaces** in some cases (though this is theoretical and still under research).

This means that **quantum SVMs** using quantum kernels could, in principle, classify data that classical SVMs cannot—if the data has **quantum-relevant structure** or the right circuit ansatz is used.

⚠️ But Be Cautious:

Quantum kernels **do not always outperform** classical kernels, especially:

- If the quantum circuit is too shallow or too noisy,
- If the dataset is small or not suitable for quantum advantages,
- If the quantum feature map doesn't encode useful structure.

We'll see some of these limitations later in the performance comparison.

Quantum Kernels

$$K_{ij} = |\langle \phi(\vec{x}_i) | \phi(\vec{x}_j) \rangle|^2$$

where:

- K_{ij} is the kernel matrix
- \vec{x}_i, \vec{x}_j are n dimensional inputs
- $\phi(\vec{x})$ is the quantum feature map
- $|\langle a | b \rangle|^2$ denotes the overlap of two quantum states a and b

Quantum Kernels

- Install Qiskit packages, prepare data
- Defining Qiskit feature map

Feature Map	What it Does	Best For
ZFeatureMap	Applies only Z-rotations	Simple data
ZZFeatureMap	Adds ZZ entanglement between qubits	Captures correlations
PauliFeatureMap	Uses X, Y, Z rotations	High flexibility

Quantum Kernels

What Does That Mean?

The idea is this:

- In classical kernels, we choose a transformation $\phi(x)$ into a high-dimensional space, and compute $\langle \phi(x), \phi(x') \rangle$.
- In quantum kernels, we use a **quantum feature map** $U_\phi(x)$ to encode data x into a **quantum state** $|\phi(x)\rangle$, and compute the **inner product between these states**:

$$K_Q(x, x') = |\langle \phi(x) | \phi(x') \rangle|^2$$

This is the **quantum kernel value**.

The crucial difference:

➡ Instead of computing dot products between vectors, we compute overlaps between quantum states prepared by quantum circuits.

Example: How It Works in Practice

1. You take a classical input x , say, a 4-feature vector from the Iris dataset.
2. You encode it into a quantum state using a **quantum feature map circuit**—this uses rotation and entanglement gates based on x .
3. You do the same for another point x' .
4. You run the quantum circuit to estimate how similar $|\phi(x)\rangle$ and $|\phi(x')\rangle$ are—this is your kernel value.

This is repeated for **every pair of inputs**, just like in classical kernel matrix construction—but now using a quantum device or simulator.

Feature Maps

- ZFeatureMap: Z-Rotations
- ZZFeatureMap: Entanglement
- PauliFeatureMap: X-Y-Z Rotations

Feature Maps

What Do We Mean by Rotation and Entanglement in Quantum Kernel Processing?

In quantum computing, we use **quantum circuits** to encode classical data into **quantum states**. These circuits are built from quantum gates—operations that manipulate **qubits**, which are the basic units of quantum information.

In the context of quantum kernels, two types of operations are central to quantum feature maps:

1. Rotation Gates – Encoding Classical Data

- A **rotation gate** changes the *state* of a qubit based on input data.
- For example, a Z-rotation gate $RZ(\theta)$ rotates a qubit around the Z-axis of the Bloch sphere by an angle θ . In a feature map, this angle θ is often a function of your classical data (like x_i).

In Quantum Kernel Processing:

- Each component of the classical input vector $x = [x_1, x_2, \dots, x_n]$ is used to apply rotations to corresponding qubits.
- These rotations “encode” the data into the quantum state.

Example:

For $x_1 = 0.5$, we might apply:

$$RZ(2\pi x_1) = RZ(\pi)$$

to qubit 1. This transforms the qubit’s state based on the value of x_1 .

- ✓ This is how **ZFeatureMap** works: it applies only data-driven rotations.

Feature Maps

2. Entanglement – Capturing Feature Interaction

- **Entanglement** is a uniquely quantum phenomenon where the state of one qubit becomes linked with the state of another.
- This means you can't describe one qubit independently of the other—there's a **non-local correlation**.

In Quantum Kernels:


- Entanglement is introduced by **multi-qubit gates** like the **CNOT** or **ZZ gate**.
- This allows the quantum circuit to capture **interactions between features**, not just the features individually.

Example:

A **ZZFeatureMap** includes terms like:

$$\exp(-i \cdot x_i x_j \cdot Z_i \otimes Z_j)$$

This applies a rotation conditioned on the **product of two features**, entangling the qubits and encoding correlations between features.

 Without entanglement, your circuit can only capture separable (independent) relationships. With entanglement, you model **feature interactions** directly.

Feature Maps

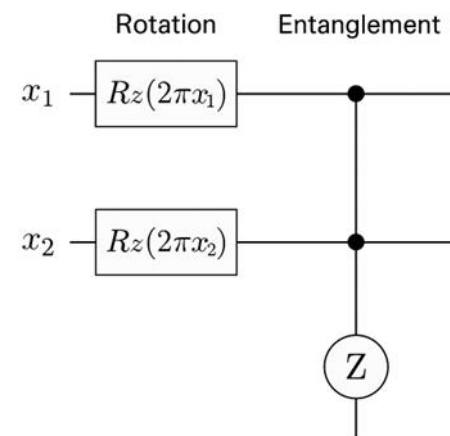
Summary:

Operation	Function	In Feature Map
Rotation	Encode classical features into qubit states	ZFeatureMap, PauliFeatureMap
Entanglement	Model correlations between features	ZZFeatureMap, PauliFeatureMap

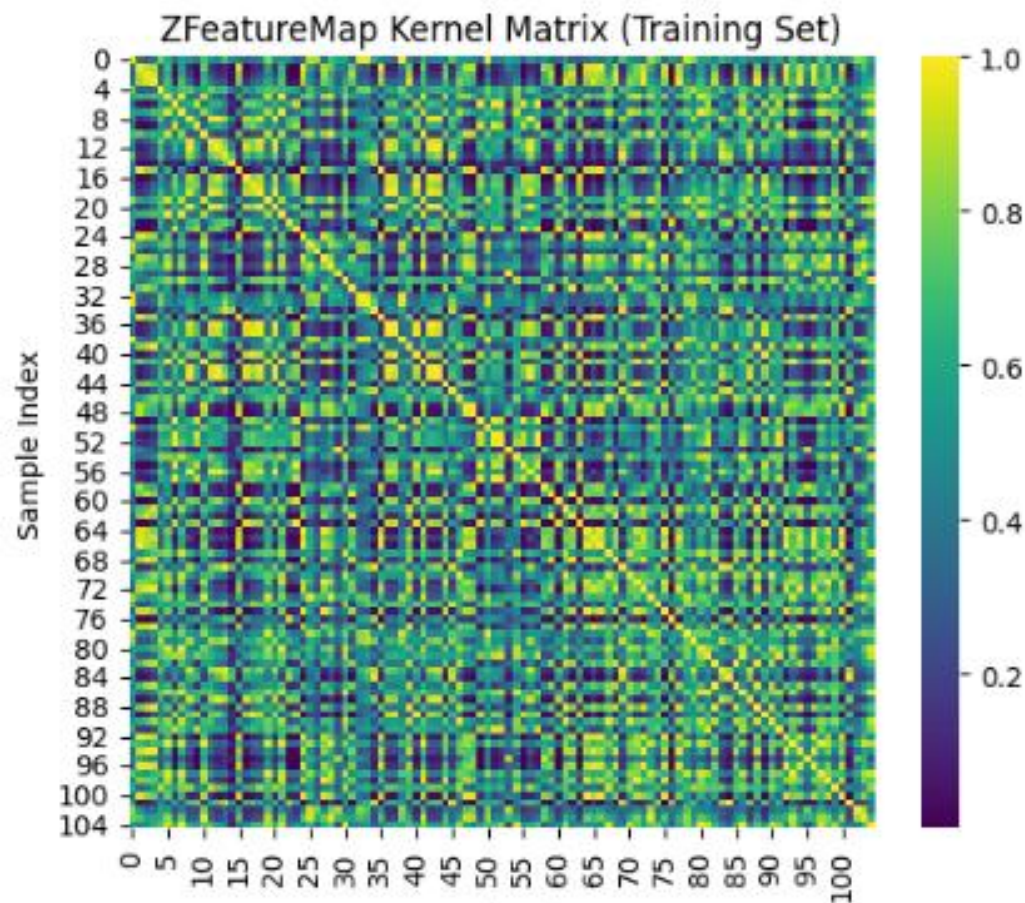
Why Does This Matter?

In classical ML, capturing interactions between features often requires adding cross-terms manually (e.g., $x_1 x_2$). Quantum kernels can **naturally and compactly** encode these relationships via **entangled circuits**.

This gives quantum feature maps their expressive power. The hope is that, in some problems, this leads to **quantum advantage** in classification performance or sample efficiency.

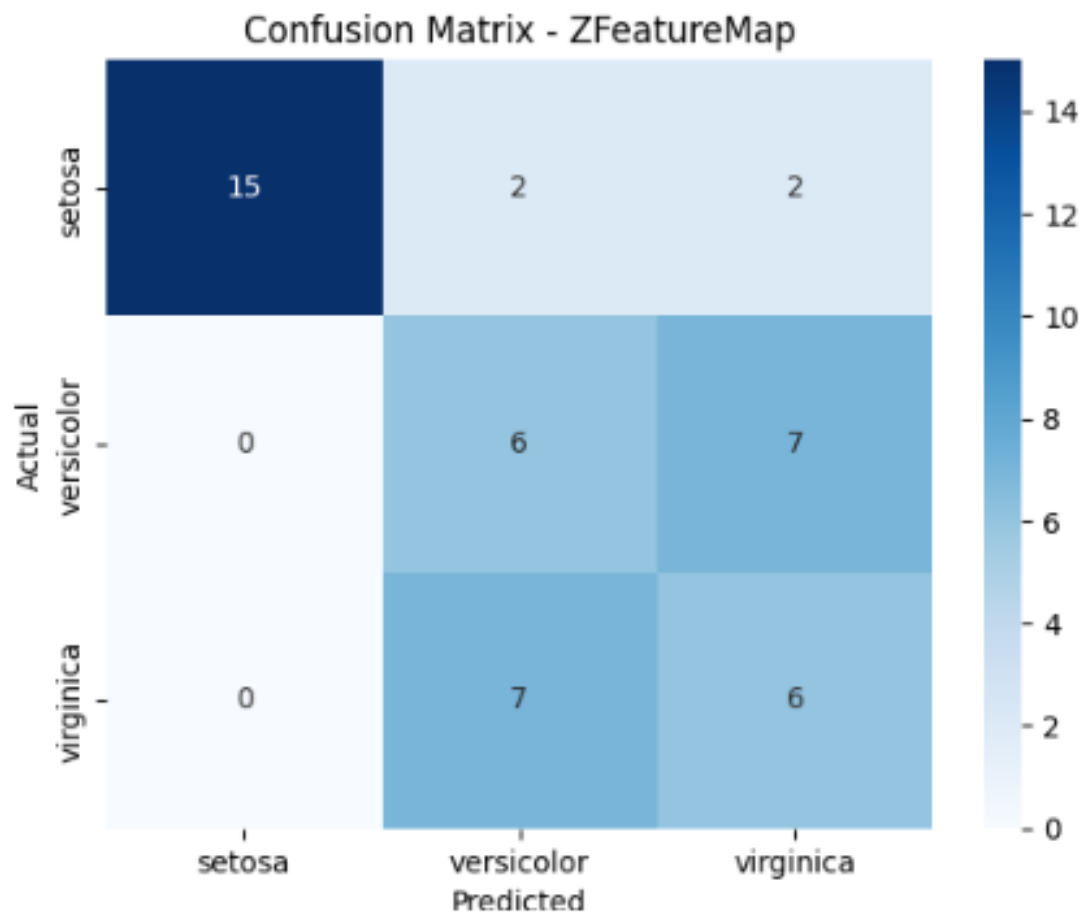


ZFeatureMap



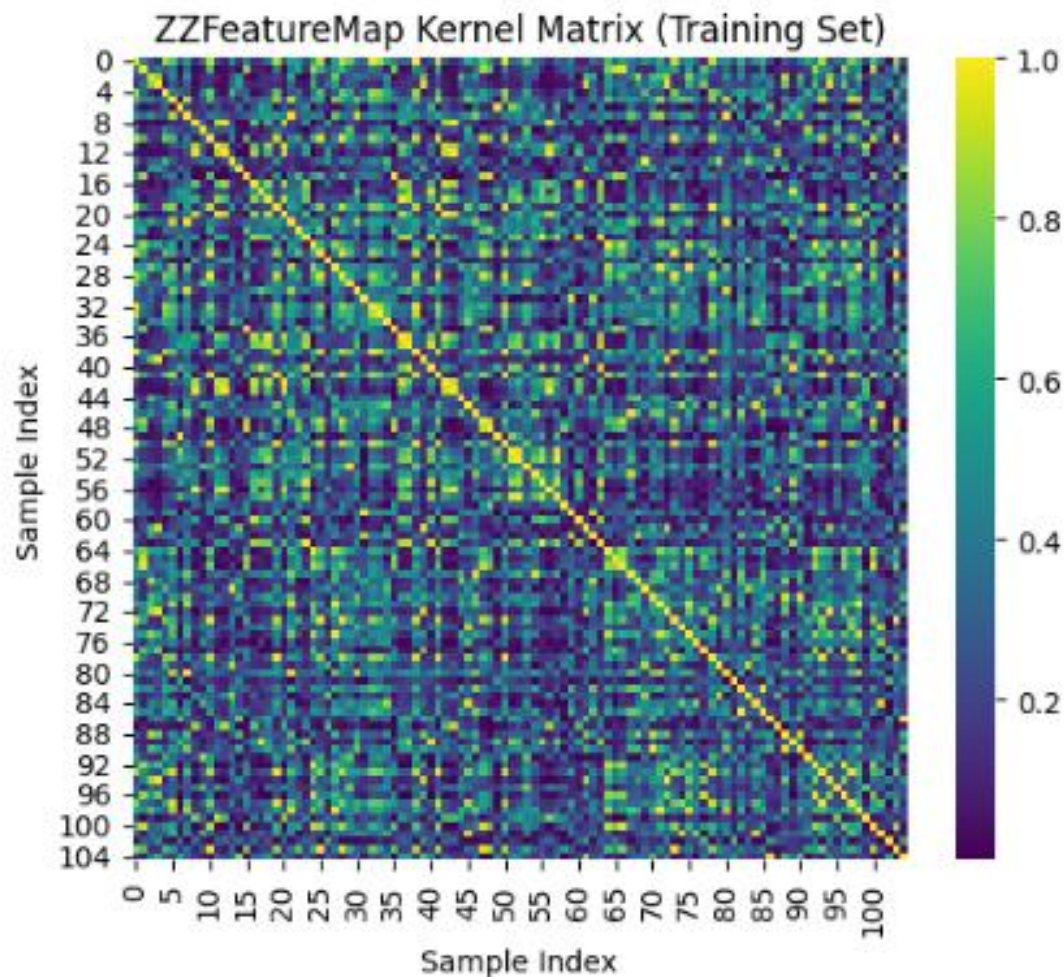
Quantum SVM Accuracy with ZFeatureMap: 0.60

ZFeatureMap



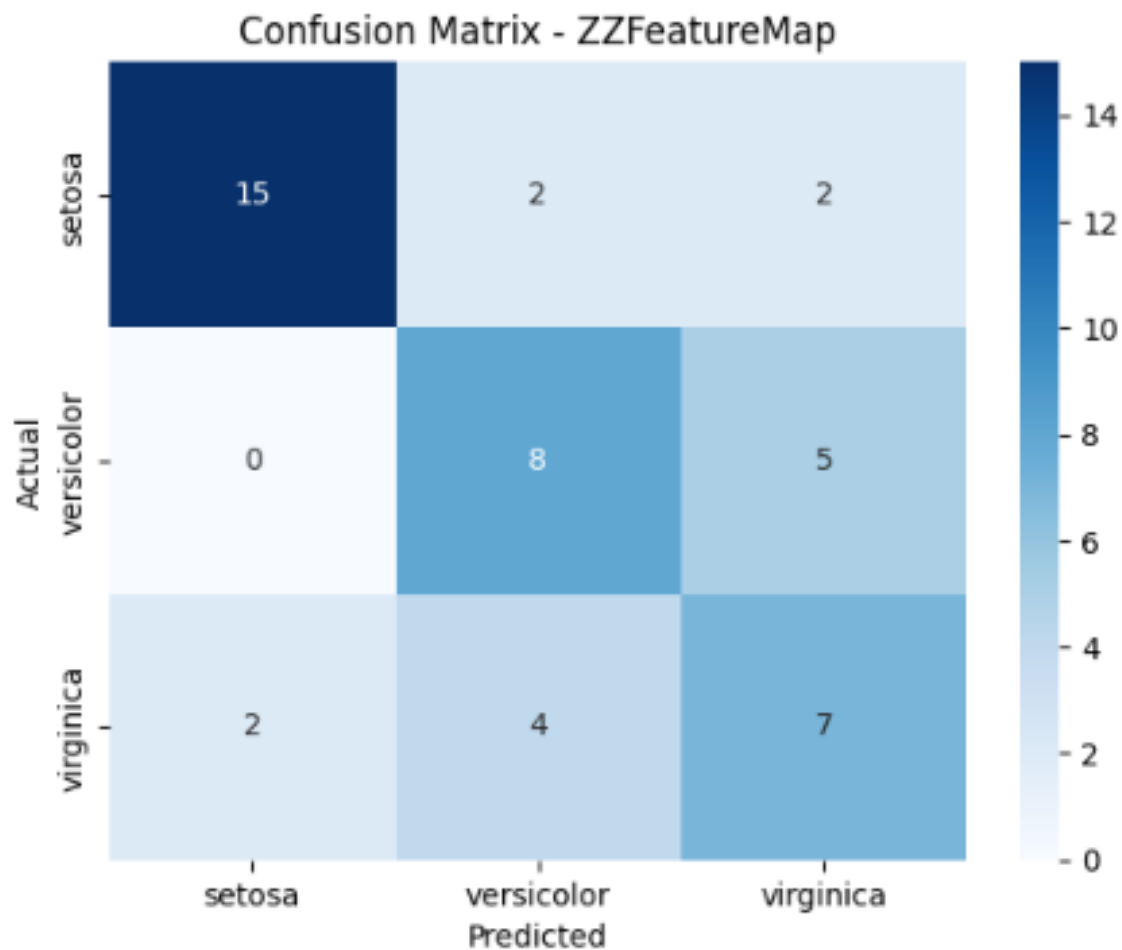
Quantum SVM Accuracy with ZFeatureMap: 0.60

ZZFeatureMap



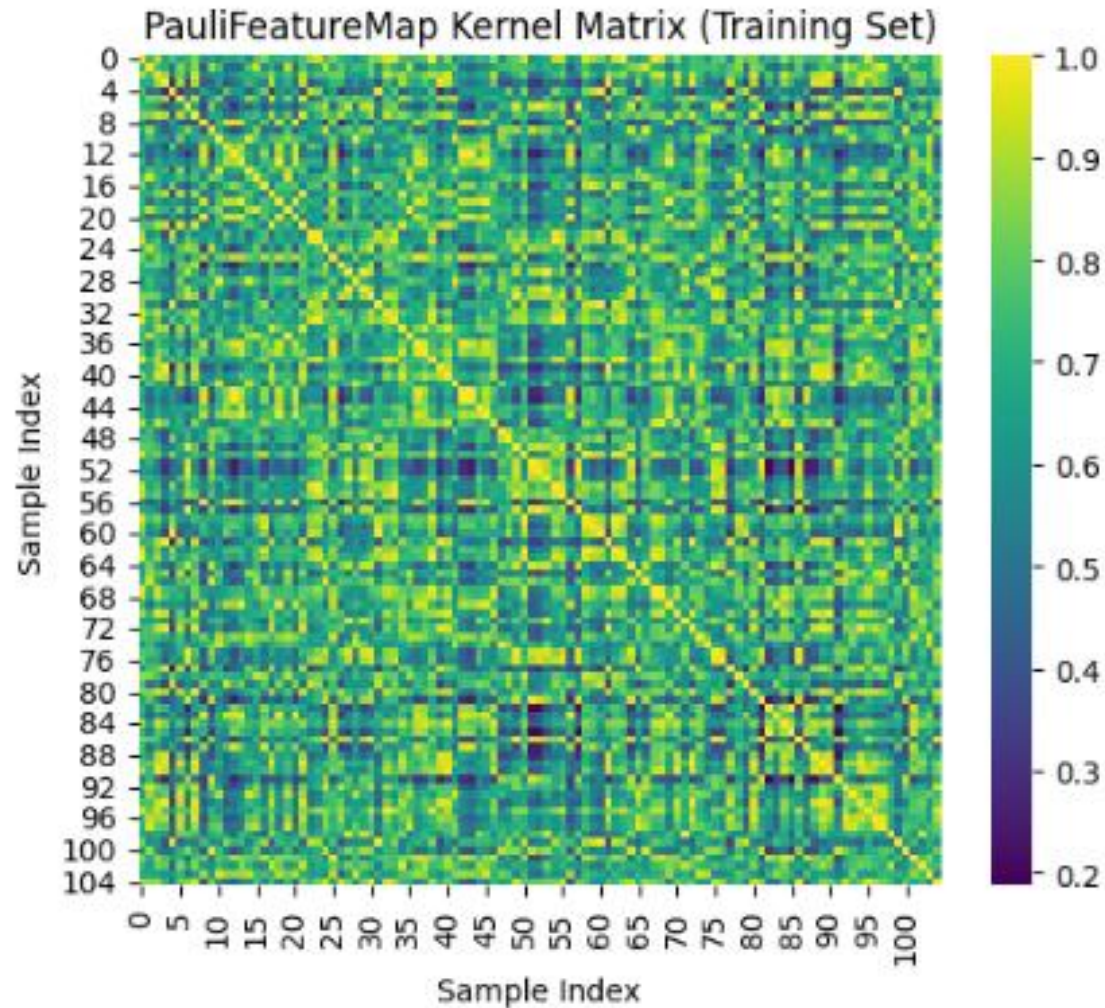
Quantum SVM Accuracy with ZZFeatureMap: 0.67

ZZFeatureMap



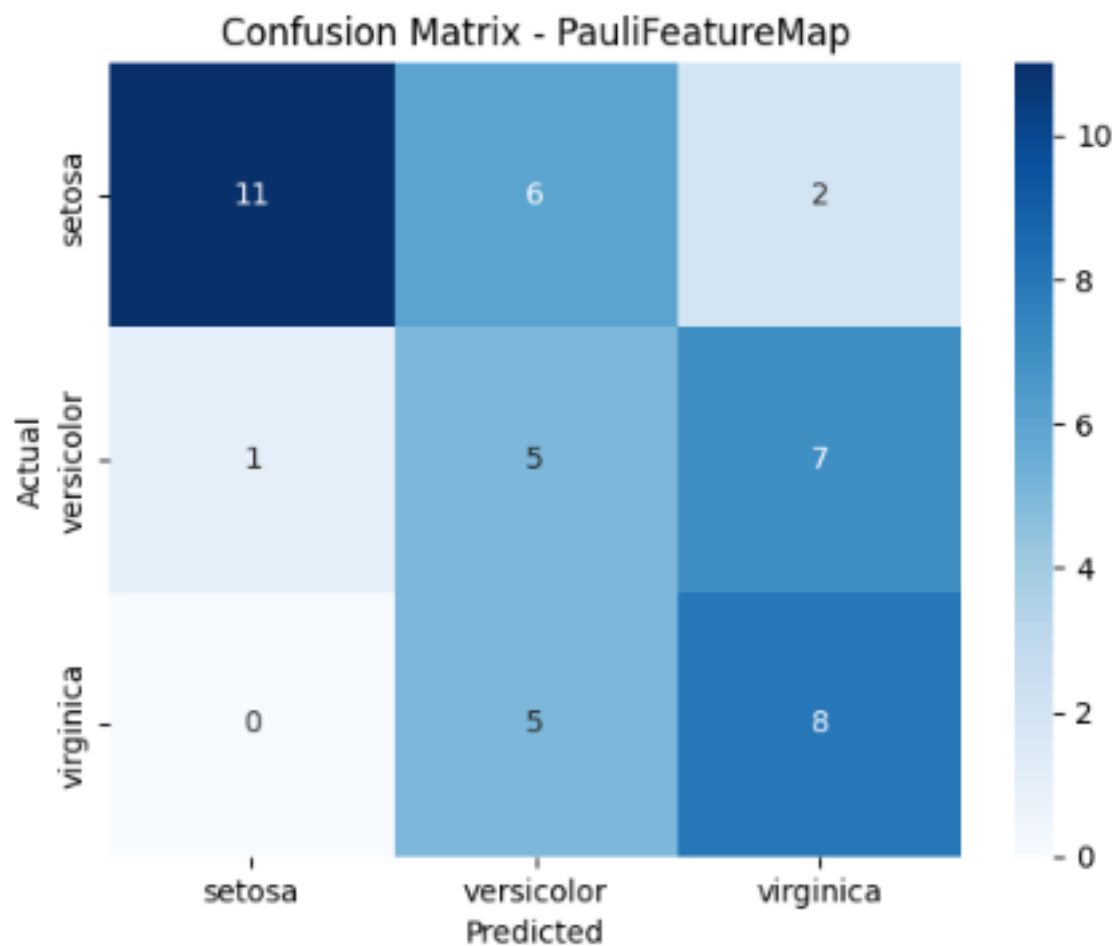
Quantum SVM Accuracy with ZZFeatureMap: 0.67

PauliFeatureMap



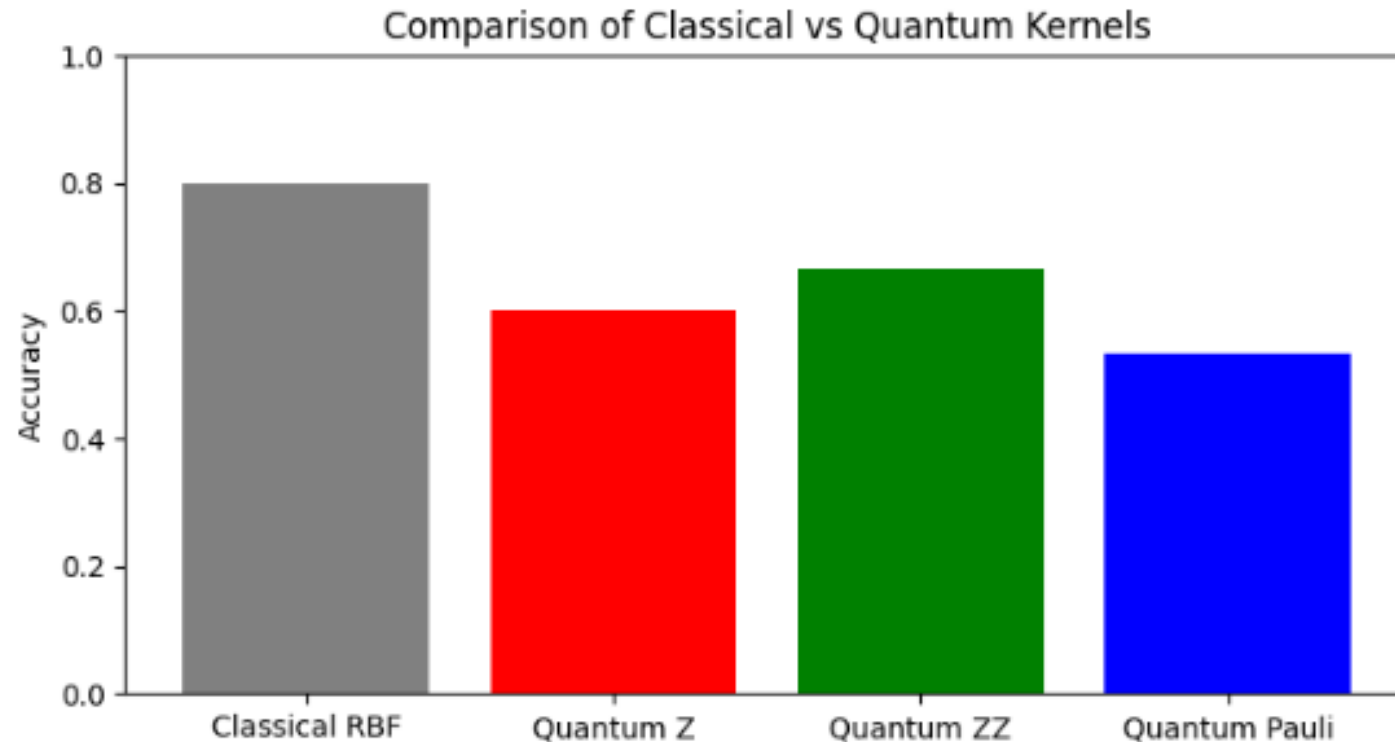
Quantum SVM Accuracy with PauliFeatureMap: 0.53

PauliFeatureMap



Quantum SVM Accuracy with PauliFeatureMap: 0.53

Results Comparison



Classical SVM Accuracy: 0.80

Quantum SVM Accuracy with ZFeatureMap: 0.60

Quantum SVM Accuracy with ZZFeatureMap: 0.67

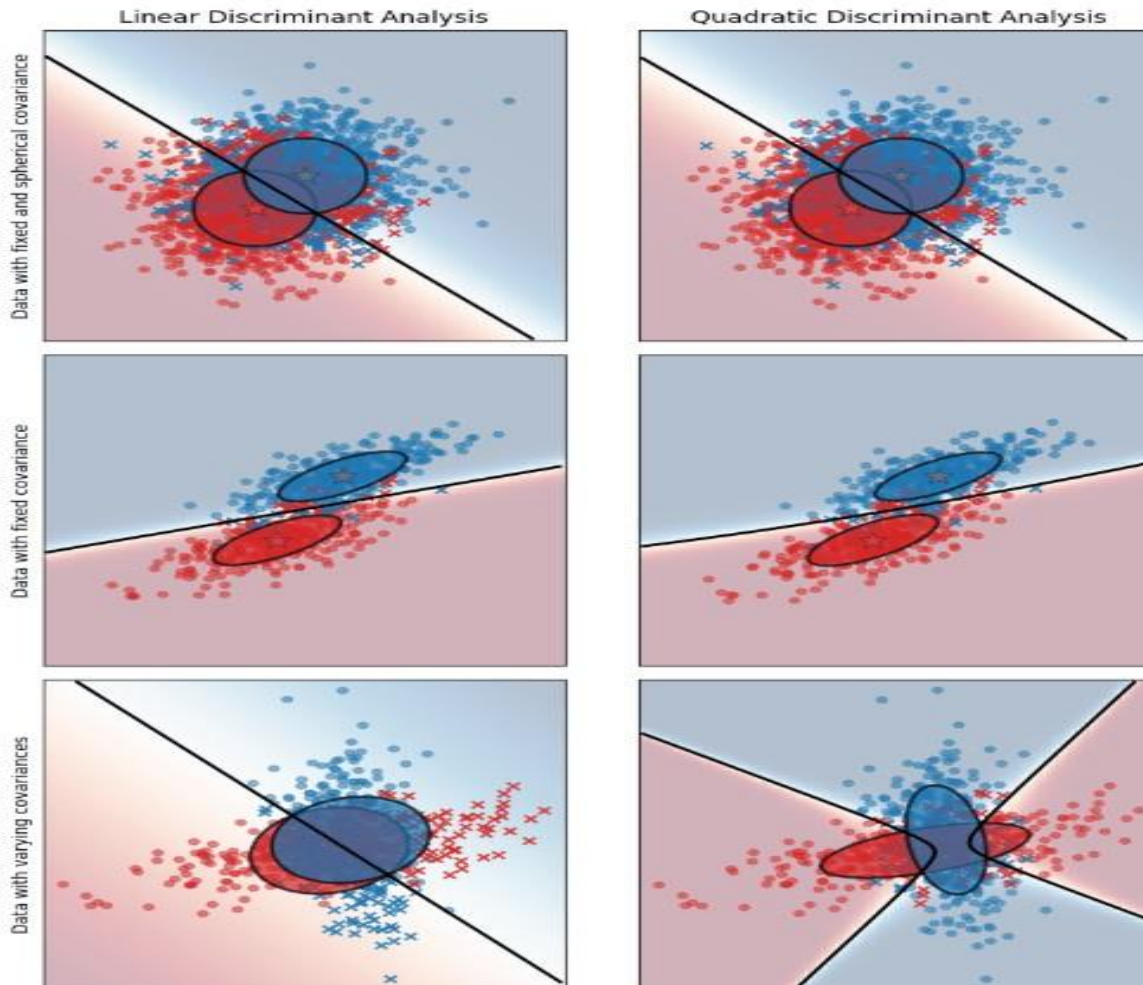
Quantum SVM Accuracy with PauliFeatureMap: 0.53

Underperformance Diagnosis

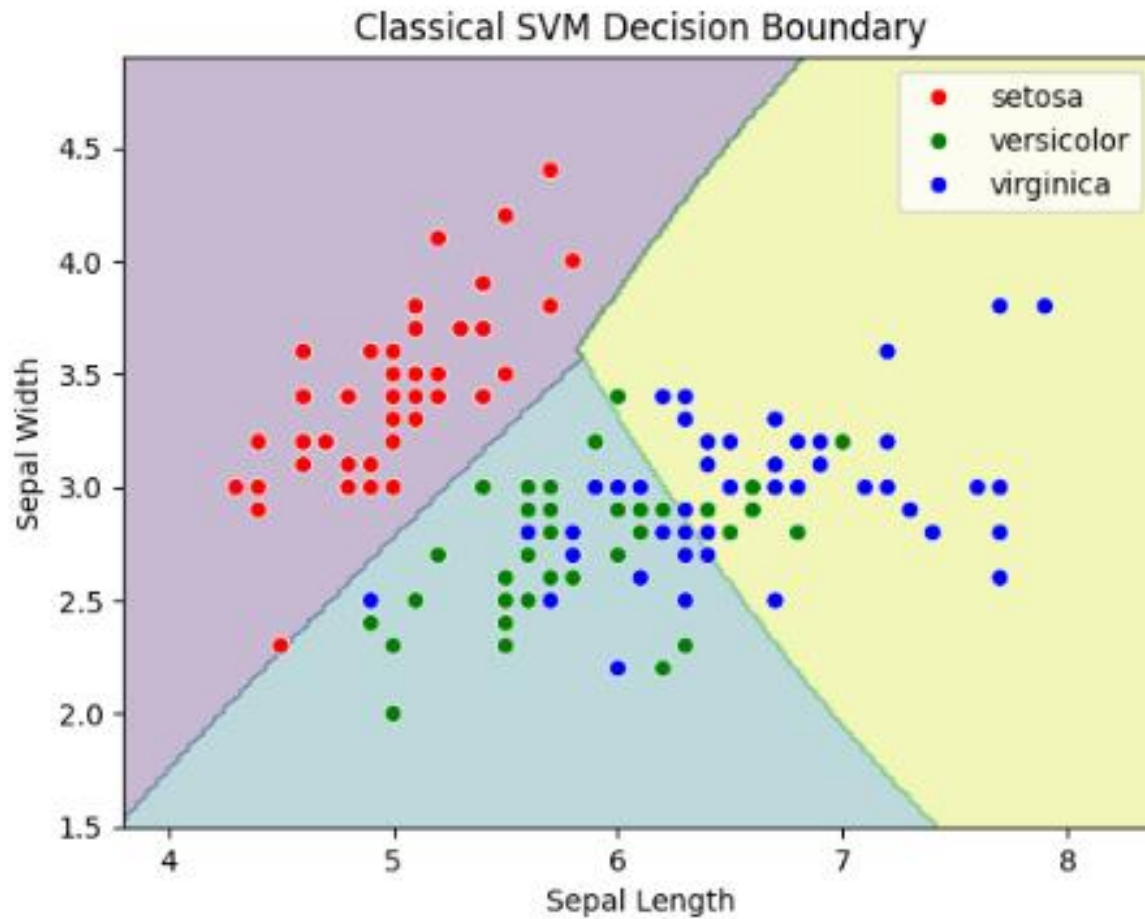
- Feature Dimensionality
- Number of Qubits and Feature Map Depth
- Quantum Kernel Evaluation Limitations
- Model Complexity and Class Structure

Analysis Comparison

Linear Discriminant Analysis vs Quadratic Discriminant Analysis

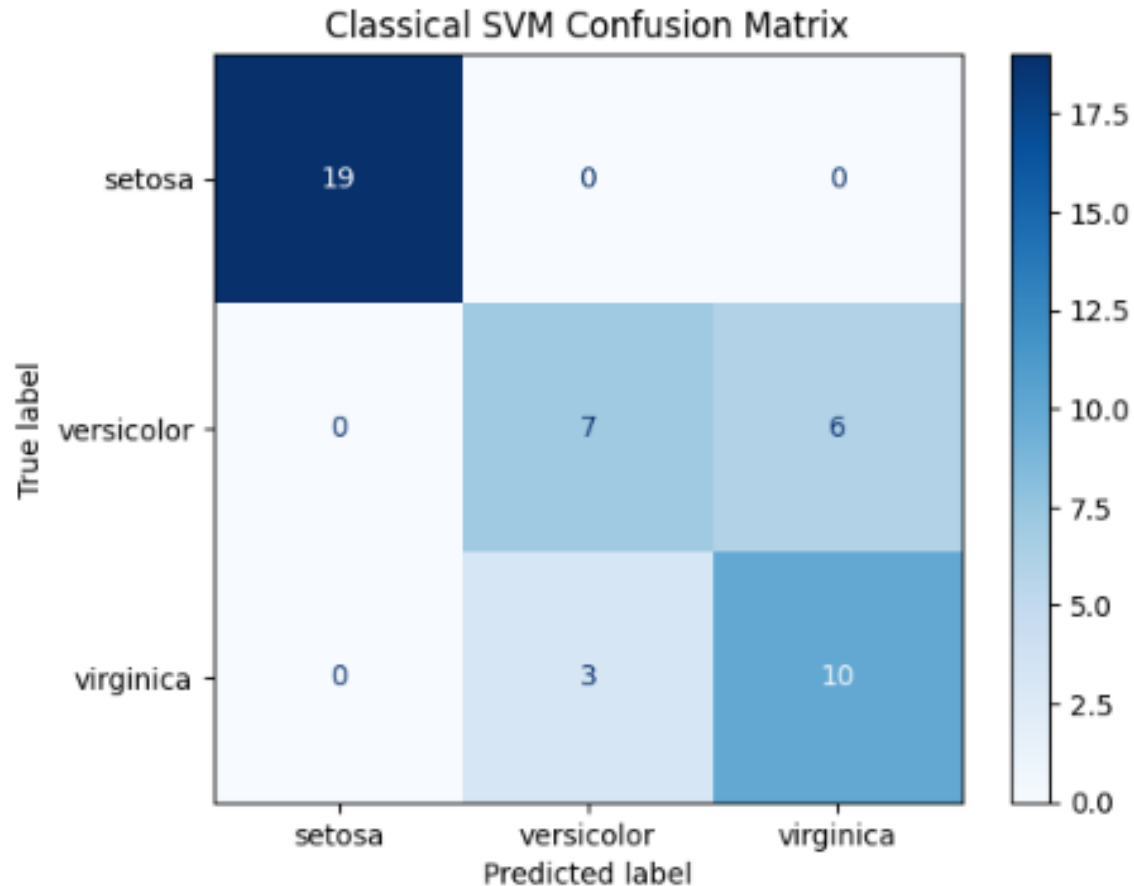


Classical SVM classification



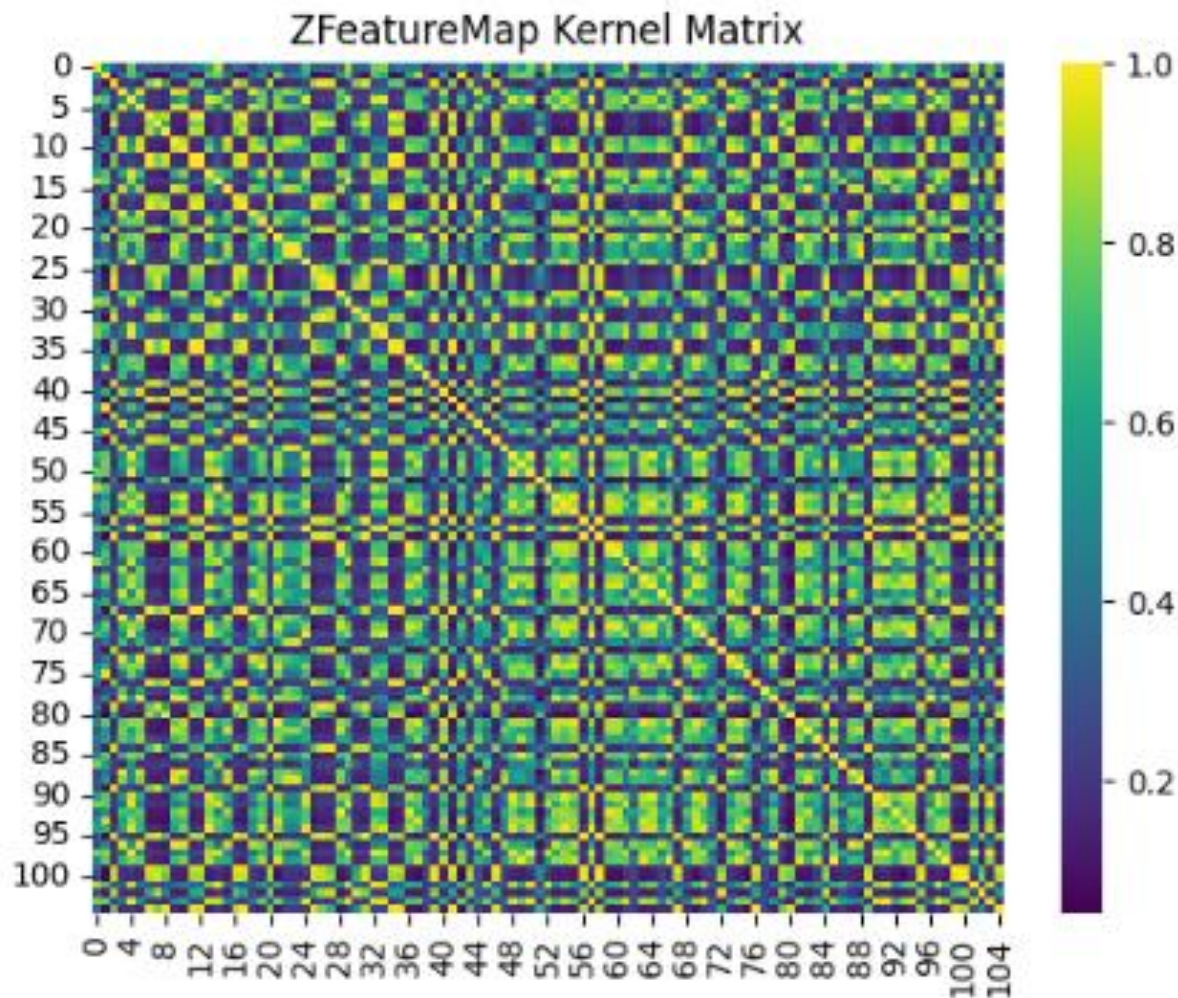
Classical SVM Accuracy: 0.80

Classical SVM classification



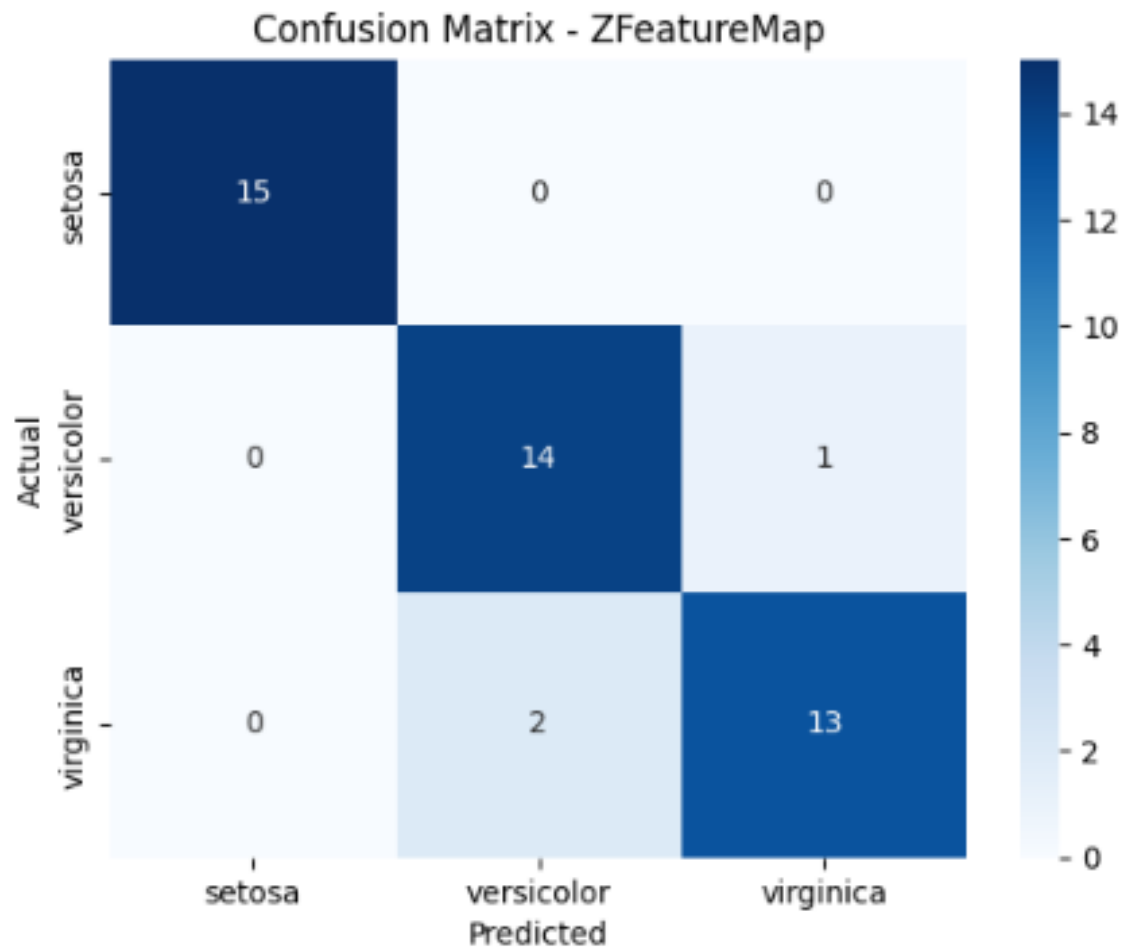
Classical SVM Accuracy: 0.80

ZFeatureMap



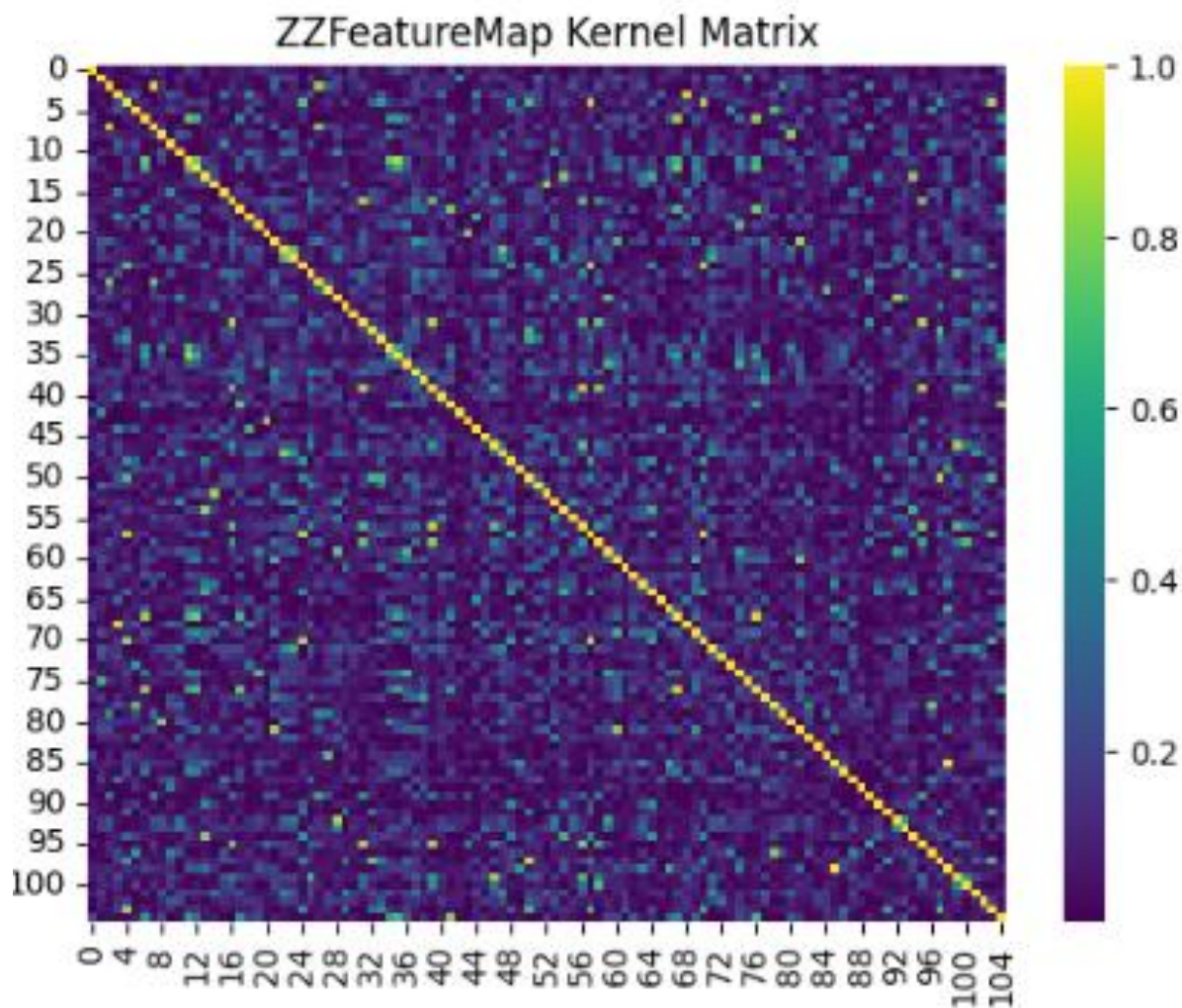
ZFeatureMap QSVN Accuracy: 0.93

ZFeatureMap



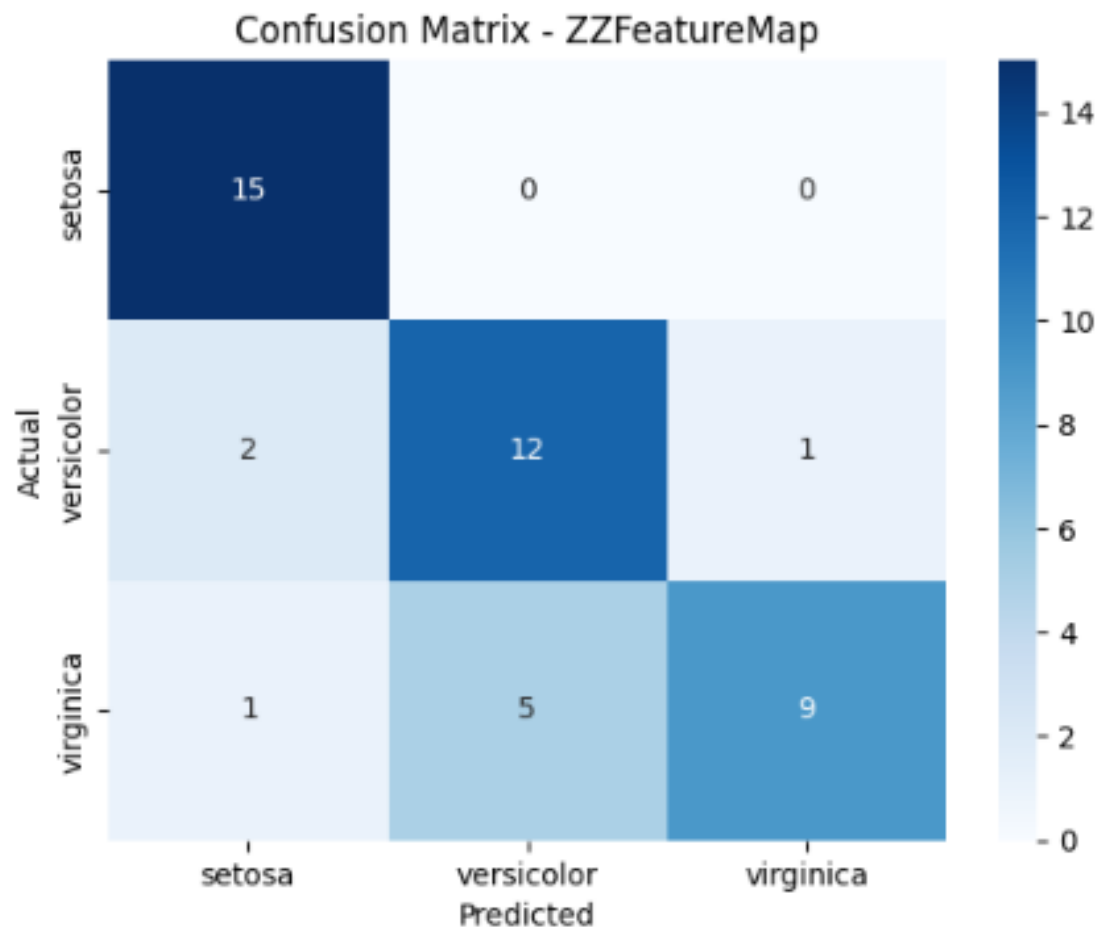
ZFeatureMap QSVN Accuracy: 0.93

ZZFeatureMap



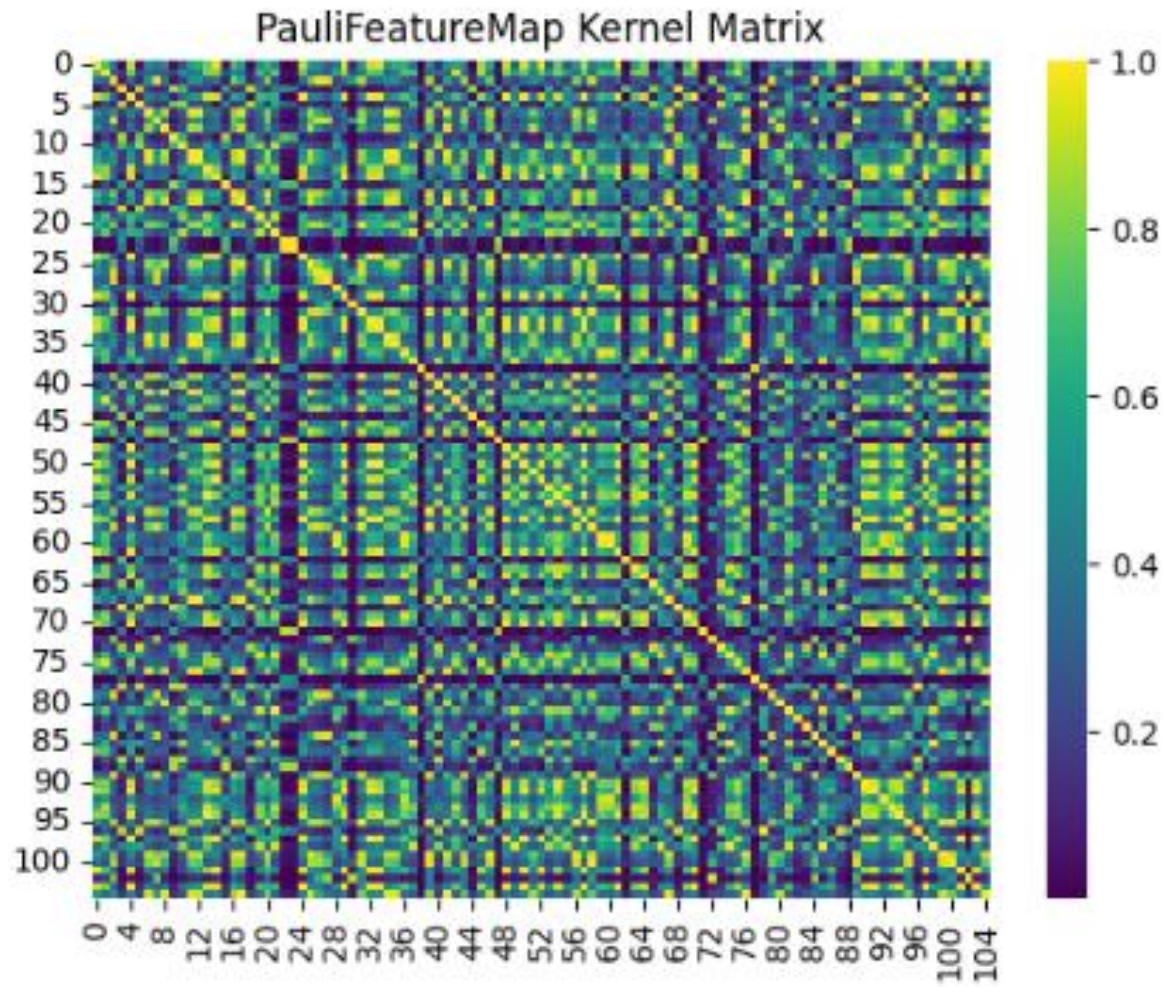
ZZFeatureMap QSVM Accuracy: 0.80

ZZFeatureMap



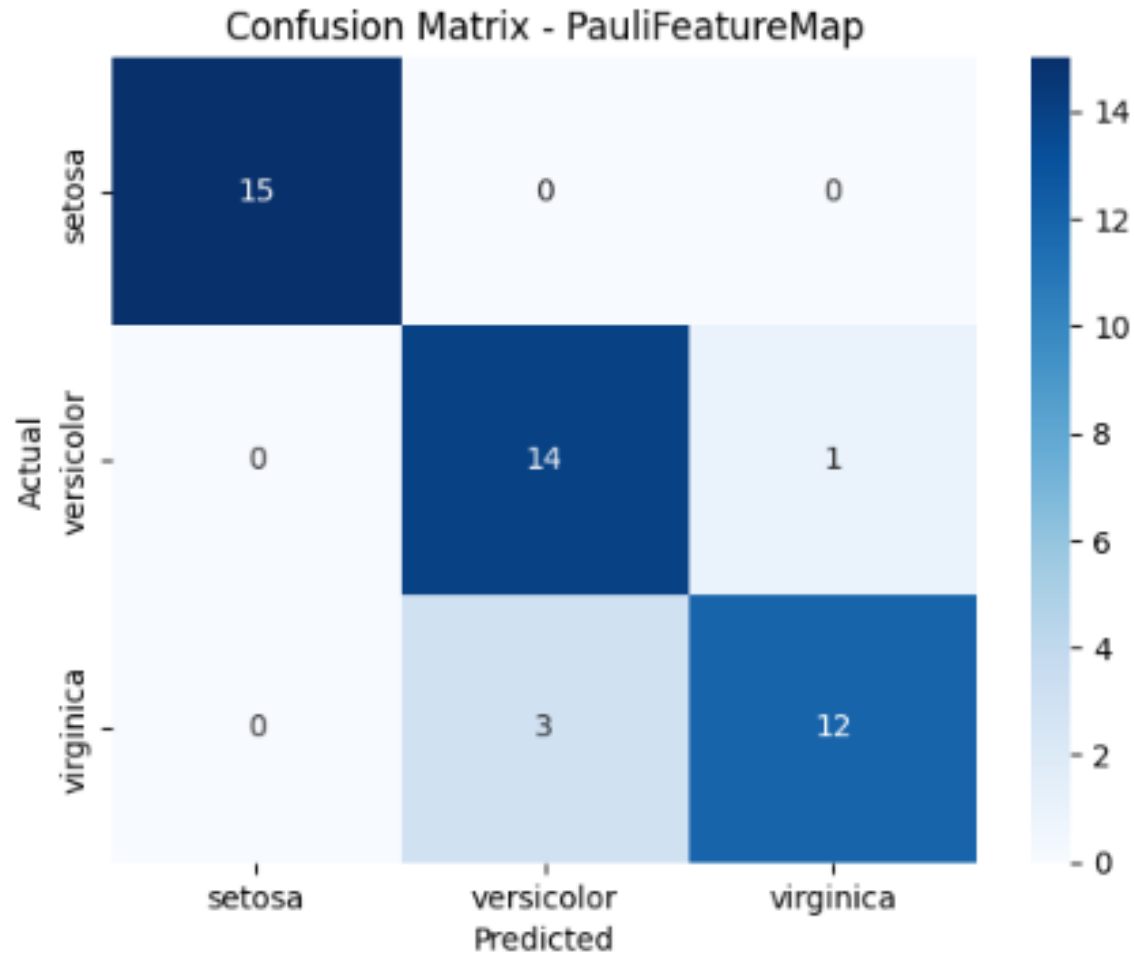
ZZFeatureMap QSVM Accuracy: 0.80

PauliFeatureMap



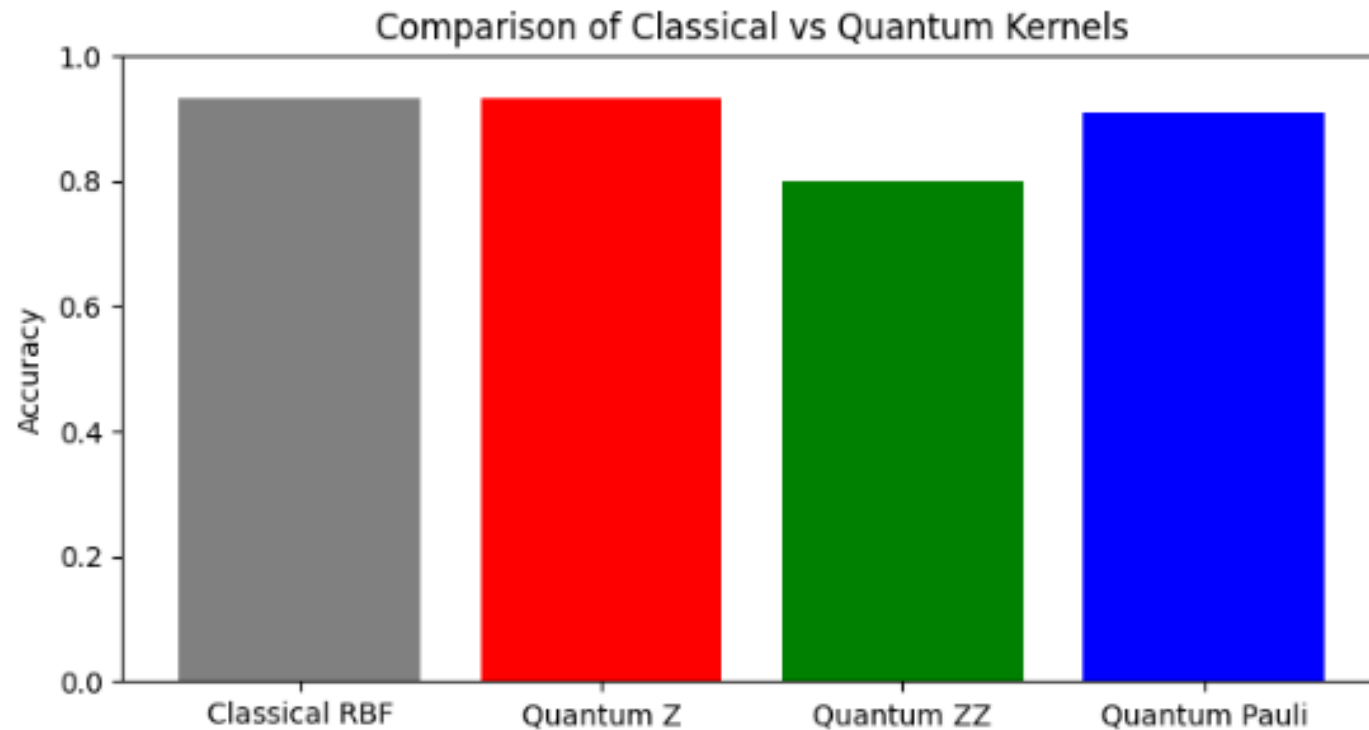
PauliFeatureMap QSVN Accuracy: 0.91

PauliFeatureMap



PauliFeatureMap QSVM Accuracy: 0.91

Results Comparison



ClassicalFeatureMap: 0.93

ZFeatureMap: 0.93

ZZFeatureMap: 0.80

PauliFeatureMap: 0.91

Summary

- Use All Four Features
- Increase Feature Map Depth
- Normalize Inputs
- Avoid Overly Complex Grid Search
- Adding Regularization to the Kernel