From Foundational Circuits to Quantum Machine Learning: A Comprehensive Guide to the Variational Quantum Classifier

Introduction: From Fixed Algorithms to Trainable Quantum Models

The journey into Quantum Machine Learning (QML) represents a natural and seamless evolution for anyone who has mastered the fundamentals of quantum circuit construction and execution. The foundational workflow—mapping a problem to a QuantumCircuit, executing it with a primitive, and analyzing the results—forms the very bedrock upon which QML algorithms are built [User Query]. An algorithm such as Deutsch-Jozsa, for instance, utilizes a fixed, predefined quantum circuit to demonstrate a specific quantum principle. A Variational Quantum Classifier (VQC), the focus of this report, employs a strikingly similar workflow but introduces a pivotal new concept: parts of the quantum circuit are *trainable* [User Query]. This is not a replacement of existing skills but rather the addition of a sophisticated new layer to the circuits one already knows how to build and operate.

The VQC is a cornerstone of contemporary QML research and a principal example of a broader class of algorithms known as Variational Quantum Algorithms (VQAs). VQAs have emerged as the dominant and most promising computational paradigm for the current Noisy Intermediate-Scale Quantum (NISQ) era. Today's quantum processors are characterized by a limited number of qubits (typically 50-1000) and are inherently susceptible to noise from environmental interactions (decoherence) and imperfect gate operations. This hardware reality has profoundly shaped the direction of quantum algorithm design.

The development of VQAs reflects a crucial strategic pivot in the field of quantum computing. Early, theoretically powerful algorithms often presuppose the existence of large-scale, fault-tolerant quantum computers that are not yet available. The NISQ era's constraints necessitated a different approach. Instead of demanding perfect hardware for our algorithms, the field began designing algorithms that can tolerate, and in some cases even leverage, the characteristics of imperfect hardware. VQAs are the direct result of this pragmatic shift. They are hybrid quantum-classical algorithms where a classical computer optimizes the parameters of a quantum circuit. This

iterative loop allows the algorithm to "learn" and potentially compensate for systematic errors in the quantum device, making VQAs a practical and powerful framework for pursuing a quantum advantage on near-term hardware. Therefore, mastering the VQC is not merely about learning a new algorithm; it is about being inducted into the dominant, pragmatic philosophy that defines the current state of applied quantum computing.

Navigating the Modern Qiskit Application Ecosystem

A significant practical challenge for any learner entering the QML space today is navigating the substantial architectural evolution of the Qiskit software framework. Understanding this new landscape is an essential prerequisite for writing functional, modern QML code.

The Great Refactoring: Why Old Tutorials Might Fail

Until early 2021, Qiskit's high-level algorithms for applications in chemistry, finance, optimization, and machine learning were consolidated within a single, monolithic package named qiskit.aqua. However, qiskit.aqua has been officially deprecated and is no longer supported in modern versions of Qiskit (post-0.25.0). This change means that a vast number of tutorials, academic papers, and code repositories available online are now outdated. Attempting to run code from these older resources with a current Qiskit installation will invariably lead to ModuleNotFoundError errors, a common source of frustration for newcomers.

This transition was not a simple renaming of modules but a fundamental refactoring of the Qiskit ecosystem into a collection of domain-specific application packages. The goal was to create a more modular, maintainable, and extensible framework, reflecting the maturation of the field from academic exploration to the development of specialized scientific tools. This modular architecture is a hallmark of a professional software ecosystem, signaling a move towards stable and sustainable tools for serious research and industrial applications.

The New Modular Structure

The functionality once contained in qiskit.aqua has been reorganized into several key packages :

- **qiskit-machine-learning**: The central library for QML applications and the primary focus of this report. It contains implementations of Quantum Neural Networks (QNNs), Quantum Kernels, and high-level algorithms like the VQC and the Quantum Support Vector Classifier (QSVC).
- **qiskit-optimization**: For solving optimization problems.
- **qiskit-nature**: For applications in quantum chemistry and physics.
- **qiskit-finance**: For financial applications.

Furthermore, the core algorithmic components and classical optimizers that were once part of Aqua have been moved into qiskit-terra (the foundational Qiskit package) and a new library, qiskit_algorithms.

The Primitives: A New Paradigm for Execution

A cornerstone of the modern Qiskit architecture is the introduction of two abstract primitives for executing quantum circuits: the Sampler and the Estimator. These primitives provide a standardized, stable Application Programming Interface (API) for interacting with quantum backends, whether they are local simulators or real cloud-based hardware. They replace the now-deprecated QuantumInstance class, which previously handled the configuration and execution of jobs. This separation of concerns allows algorithm developers to write code that is agnostic to the specific backend being used, while backend providers can innovate on their hardware and simulators without breaking user code.

- **Sampler**: The Sampler primitive's job is to accept one or more quantum circuits, execute them for a given number of shots, and return the resulting quasi-probability distributions of the measurement outcomes (i.e., the counts for each bitstring). The VQC is built upon an underlying SamplerQNN, which uses the Sampler to generate outputs that are then interpreted as class probabilities, making it the relevant primitive for this report.

- **Estimator**: The Estimator primitive is designed to efficiently calculate the expectation values of quantum observables with respect to quantum circuits. This is particularly useful in algorithms like the Variational Quantum Eigensolver (VQE), where the goal is to find the energy (expectation value) of a Hamiltonian.

To aid in navigating this new ecosystem, the following table serves as a "Rosetta Stone," mapping the conceptual components of a VQC to their modern classes and module paths, and noting their deprecated equivalents.

Table 1: Key Classes in the Modern Qiskit Machine Learning Ecosystem

Component/Concept	Modern Qiskit Class	Module Path	Role/Notes (and Deprecated Equivalent)
Classifier Algorithm	VQC	qiskit_machine_learning.algorithms	High-level classifier. (Formerly in qiskit.aqua.algorithms)
Data Encoding Circuit	ZZFeatureMap, etc.	qiskit.circuit.library	Maps classical data to quantum states. (Location largely unchanged)
Trainable Circuit	EfficientSU2, etc.	qiskit.circuit.library	The parameterized part of the model. (Location largely unchanged)
Optimizer	COBYLA, SPSA, etc.	qiskit_algorithms.optimizers	Classical algorithm to update parameters. (Formerly in qiskit.aqua.components.optimizers)
Execution Backend	Sampler	qiskit.primitives	Executes circuits and returns measurement counts. (Replaces the QuantumInstance class)

Underlying QNN	SamplerQNN	qiskit_machine_learning.neural_networks	The neural network object that VQC wraps. (New, more modular concept)

The Anatomy of a Variational Quantum Classifier

The VQC operates as a hybrid quantum-classical algorithm, orchestrating a feedback loop between a quantum processing unit (QPU) and a classical computer to perform supervised learning. This process iteratively refines a quantum model to learn a mapping from input features to class labels by minimizing a cost function on a training dataset. The anatomy of this process can be broken down into its core components.

The entire procedure follows a distinct, cyclical workflow :

1. **Data Encoding**: A classical data point is encoded into the state of a quantum circuit.
2. **Variational Circuit Execution**: This state is evolved by a second, trainable quantum circuit.
3. **Measurement**: The final state is measured to extract classical information.
4. **Classical Post-Processing & Cost Calculation**: The measurement results are used to calculate the model's error.
5. **Parameter Update**: A classical optimizer suggests new parameters for the trainable circuit to reduce the error.
6. **Iteration**: The loop repeats until the model's performance converges.

Data Encoding: The Quantum Feature Map

The first step in any QML workflow is to "upload" classical data into the quantum computer. This is the role of the quantum feature map, a parameterized quantum circuit denoted as U(x), where the parameters are the features of a classical data point x. The purpose of the feature map is to encode this classical information into a quantum state $|\psi(x)\rangle = U(x)|0\rangle^{\otimes n}$, which lives in a high-dimensional Hilbert space. The geometry of the data within this quantum feature space is critical; a well-designed

feature map can transform a complex, non-linearly separable dataset into one that is more easily classified. It is within this vast computational space, potentially inaccessible to classical computers, that a quantum advantage for machine learning may be found.

A powerful and commonly used choice is the ZZFeatureMap. This is a type of PauliFeatureMap that is particularly adept at capturing second-order interactions between input features. Its structure typically consists of three parts :

1. A layer of Hadamard gates (H) applied to each qubit to create an initial uniform superposition.
2. A layer of single-qubit rotation gates (e.g., Pauli-Z rotations) whose angles are determined by a function of the individual data features, xi.
3. An entangling layer composed of two-qubit gates (like CNOTs) that are intertwined with two-qubit rotations (e.g., ZZ interactions). The angles of these rotations are determined by a non-linear function of pairs of data features, (xi,xj). In Qiskit's ZZFeatureMap, this function defaults to (π−xi)(π−xj).

The unitary operation for the entangling component of a general Pauli feature map can be expressed as:
$$ U_{\Phi(\vec{x})} = \exp\left(i \sum_{S \subseteq {1, \dots, n}, |S|=k} \phi_S(\vec{x}) \prod_{j \in S} P_j\right) $$
where Pj are Pauli matrices. For the ZZFeatureMap, the interaction order is k=2, and the Pauli products are of the form ZiZj. This structure allows the feature map to embed non-linear relationships from the original data into the quantum state, creating a rich feature space for classification.

The Model: The Parameterized Ansatz

If the feature map is where the data is loaded, the variational ansatz, W(θ), is the model itself. It is a quantum circuit with a fixed structure but containing a set of tunable classical parameters, θ, which are learned during the training process. The "expressivity" of the ansatz—its ability to generate a wide variety of quantum states—determines the complexity of the decision boundaries the VQC can learn. A more expressive ansatz can, in principle, approximate more complex functions.

A prevalent and practical choice for the ansatz is the EfficientSU2 circuit. This circuit is known as a "hardware-efficient" ansatz because its structure is designed to map well onto the physical constraints of real NISQ devices. It typically involves layers of

single-qubit rotations and entangling gates that only act on adjacent qubits (e.g., linear entanglement). This locality is crucial because it reduces the number of additional SWAP gates required during the transpilation process—the process of rewriting a circuit to match the specific qubit connectivity of a given hardware device. Fewer SWAP gates lead to shorter, less noisy circuits, which is paramount for successful execution on near-term quantum computers. The EfficientSU2 circuit consists of alternating layers of :

1. **Single-qubit Rotations**: These are typically general SU(2) gates, like RY(θi) and RZ(ϕi), applied to each qubit. These parameterized gates are the "knobs" that the classical optimizer turns during training.
2. **Entangling Gates**: These are typically two-qubit gates like CNOT (CX) that create entanglement between the qubits, allowing the circuit to explore correlations and generate complex, non-trivial quantum states necessary for solving the classification task.

The combination of the feature map and the ansatz creates the full parameterized quantum circuit for the VQC: $U_{VQC}(x,\theta)=W(\theta)U(x)$. The design of this complete circuit embodies a critical trade-off that is central to all VQAs. A natural instinct is to maximize the complexity of the circuit (e.g., by increasing the number of repetitions, or reps, of the feature map and ansatz layers) to increase its expressivity. However, this approach is fraught with peril. Deeper circuits are more susceptible to decoherence and gate errors on NISQ hardware, which can corrupt the computation. More fundamentally, they are prone to a trainability issue known as the **barren plateau phenomenon**. In a barren plateau, the landscape of the loss function becomes exponentially flat as the number of qubits or circuit depth increases. This means the gradient of the loss function with respect to the parameters becomes vanishingly small across most of the parameter space, providing no meaningful direction for the classical optimizer to follow and causing the training process to stall. Consequently, designing a VQC is not a matter of maximizing complexity, but of finding a delicate balance: the circuit must be expressive enough to solve the problem but simple enough to be trainable and executable on near-term hardware. This is precisely why hardware-efficient ansaetze like EfficientSU2 are so prevalent in research.

The Engine: Measurement and the Classical Optimization Loop

After the full quantum circuit $U_{VQC}(x,\theta)$ has been executed on the QPU, the final state must be measured to extract classical information for the optimization loop.

- **Measurement and Interpretation**: For a VQC built with a SamplerQNN, the quantum computer is run for a number of "shots," resulting in a dictionary of counts for each measured bitstring (e.g., {'00': 480, '11': 520}). An interpret function is then required to map these bitstrings to the problem's class labels. A common choice for binary classification is a parity function, which might map bitstrings with an even number of '1's to class 0 and those with an odd number of '1's to class 1. This process yields a probability distribution over the classes.
- **Loss Function**: The classical computer uses this output probability distribution to calculate a loss (or cost) function. This function measures the dissimilarity between the VQC's predicted output and the true label from the training data. A standard choice for classification tasks is the cross-entropy loss.
- **Classical Optimizer**: The value of the loss function is fed to a classical optimization algorithm. The optimizer's task is to propose a new set of circuit parameters, θ', that is expected to result in a lower loss value in the next iteration. There are many choices for optimizers, which can be broadly categorized as gradient-based (like Adam) or gradient-free (like COBYLA or SPSA). Gradient-free methods do not require calculating the gradient of the loss function, which can be advantageous as gradient calculation on quantum hardware can be resource-intensive and noisy. However, they may converge more slowly than their gradient-based counterparts. The choice of optimizer is another critical hyperparameter that can significantly affect the training performance of the VQC.

The Abstraction: The High-Level VQC Class

Instead of manually coding this entire hybrid optimization loop, a practitioner can leverage the high-level VQC class provided by the qiskit-machine-learning library. This class acts as a convenient wrapper that encapsulates the entire workflow. It is instantiated by providing the pre-configured quantum and classical components: the sampler for execution, the feature_map for data encoding, the ansatz as the trainable model, and the optimizer for training.

This abstraction provides an interface that is familiar to anyone with experience in classical machine learning libraries like scikit-learn. Once instantiated, the VQC object has .fit() and .predict() methods that hide the complexity of the underlying

quantum-classical interactions. The .fit(X_train, y_train) method initiates the full hybrid loop, iteratively calling the quantum backend via the sampler and using the classical optimizer to update the ansatz parameters until convergence. This powerful abstraction allows the user to focus on the high-level design of the QML model rather than the low-level implementation details of the training loop.

A Step-by-Step Implementation: Building Your First Quantum Classifier

This section provides a complete, end-to-end Python script for implementing a Variational Quantum Classifier. The code is written for the modern Qiskit 1.x framework and is presented in logical blocks with detailed explanations. The process mirrors the scientific methodology of applied QML research: beginning with a well-defined problem and dataset, designing a model architecture, running the experiment while monitoring its progress, and finally, analyzing the results with quantitative and qualitative metrics. We will tackle a classic binary classification problem: separating two concentric circles of data points, a task that is impossible for a simple linear classifier.

4.1. Setup and Data Preparation

The first phase of any machine learning experiment is to prepare the data. This involves importing the necessary libraries, generating a suitable dataset, scaling the features appropriately for our quantum model, splitting the data into training and testing sets, and visualizing it to establish a baseline understanding.

Python

```
# General-purpose imports
import numpy as np
import matplotlib.pyplot as plt
```

```
# Scikit-learn for data generation and processing
from sklearn.datasets import make_circles
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# Qiskit imports
from qiskit.primitives import Sampler
from qiskit.circuit.library import ZZFeatureMap, EfficientSU2
from qiskit_algorithms.optimizers import COBYLA
from qiskit_machine_learning.algorithms.classifiers import VQC
from qiskit_machine_learning.utils.algorithm_globals import algorithm_globals

# Set a random seed for reproducibility
algorithm_globals.random_seed = 42

# --- 1. Data Generation ---
# Generate 100 data points forming two concentric circles
X, y = make_circles(n_samples=100, noise=0.1, factor=0.5, random_state=1)

# --- 2. Data Scaling ---
# Scale features to the range [0, pi] for angle-based encoding in the feature map
scaler = MinMaxScaler(feature_range=(0, np.pi))
X_scaled = scaler.fit_transform(X)

# --- 3. Data Splitting ---
# Split the data into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=algorithm_globals.random_seed
)

# --- 4. Data Visualization ---
plt.figure(figsize=(8, 6))
plt.scatter(X_train[y_train == 0][:, 0], X_train[y_train == 0][:, 1], c='royalblue', marker='o',
label='Class 0 (Train)')
plt.scatter(X_train[y_train == 1][:, 0], X_train[y_train == 1][:, 1], c='tomato', marker='s',
label='Class 1 (Train)')
plt.scatter(X_test[y_test == 0][:, 0], X_test[y_test == 0][:, 1], c='darkblue', marker='o',
label='Class 0 (Test)')
plt.scatter(X_test[y_test == 1][:, 0], X_test[y_test == 1][:, 1], c='darkred', marker='s',
label='Class 1 (Test)')
```

```
plt.title("Synthetic 'Circles' Dataset")
plt.xlabel("Feature 1 (scaled)")
plt.ylabel("Feature 2 (scaled)")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```

Explanation:

- **Imports**: All necessary modules are imported from numpy, matplotlib, sklearn, and the modern Qiskit libraries. Note the specific imports from qiskit.primitives, qiskit_algorithms.optimizers, and qiskit_machine_learning.algorithms, which reflect the new modular ecosystem.
- **Data Generation**: We use make_circles from scikit-learn to create a dataset that is not linearly separable, providing a non-trivial challenge for our classifier.
- **Data Scaling**: This is a critical pre-processing step. The ZZFeatureMap uses angle encoding, meaning it maps classical feature values to the rotation angles of quantum gates. It is standard practice to scale the data to a suitable range, such as [0,π] or [0,2π], to ensure the full expressive range of the rotation gates is utilized. Here, MinMaxScaler transforms our features into the range [0,π].
- **Data Splitting**: The data is partitioned into a training set (which the model will learn from) and a testing set (which will be used to evaluate the model's performance on unseen data), a standard practice to prevent overfitting.
- **Visualization**: Plotting the data helps to understand its structure and provides a visual baseline for evaluating the classifier's performance later.

4.2. Building the Quantum Components

Next, we define the core quantum components of our VQC: the Sampler primitive for execution, the ZZFeatureMap for data encoding, and the EfficientSU2 circuit for the trainable ansatz. This is the model design phase of our experiment.

Python

```
# --- 1. Define the Sampler Primitive ---
```

```
# The Sampler is our execution backend. It runs circuits and returns counts.
sampler = Sampler()

# --- 2. Define Quantum Circuit Components ---
# The number of qubits is determined by the number of features in our data.
num_qubits = X_train.shape

# Feature Map: ZZFeatureMap
# 'reps=2' means the feature encoding structure is repeated twice for more expressivity.
feature_map = ZZFeatureMap(feature_dimension=num_qubits, reps=2)

# Ansatz: EfficientSU2
# This is our trainable circuit. 'reps=3' gives it more trainable parameters.
# 'entanglement="linear"' is a hardware-efficient choice.
ansatz = EfficientSU2(num_qubits=num_qubits, reps=3, entanglement='linear')

# We can inspect the components to understand their structure
print("--- Feature Map ---")
print(feature_map.decompose())
print("\n--- Ansatz ---")
print(ansatz.decompose())
```

Explanation:

- **Sampler**: We instantiate a basic Sampler from qiskit.primitives. By default, this will use a local, high-performance, statevector-based simulator to execute our circuits and return the measurement counts.
- **Feature Map**: We choose the ZZFeatureMap. The feature_dimension is set to the number of features in our data (2 in this case), which also determines the number of qubits required. The reps parameter controls how many times the basic encoding block is repeated. Higher reps can create a more complex mapping but also a deeper circuit, illustrating the expressivity-noise trade-off.
- **Ansatz**: We use the EfficientSU2 ansatz. The reps parameter here controls the number of repeating layers of rotation and entanglement gates, directly affecting the number of trainable parameters and the model's expressive power. We choose 'linear' entanglement, where qubit i is entangled with qubit i+1, a common hardware-efficient strategy that helps mitigate noise and connectivity issues on real devices.
- **Inspection**: Calling .decompose() on the circuit objects allows us to print and visualize their underlying gate structure, making the abstract components tangible.

4.3. Configuring and Training the VQC

Now we assemble the components into the high-level VQC class, configure the classical optimizer, and define a callback function to monitor the training progress in real-time. This is the experimentation phase.

Python

```
# --- 1. Define the Classical Optimizer ---
# COBYLA is a gradient-free optimizer. 'maxiter' is the max number of iterations.
optimizer = COBYLA(maxiter=100)

# --- 2. Define a Callback Function for Live Training Plot ---
objective_func_vals =
plt.rcParams["figure.figsize"] = (12, 6)

def callback_graph(weights, obj_func_eval):
    objective_func_vals.append(obj_func_eval)
    # Use IPython's clear_output to redraw the plot in the same cell
    from IPython.display import clear_output
    clear_output(wait=True)
    # Plot data
    plt.title("Objective Function Value vs. Iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Objective Function Value")
    plt.plot(range(len(objective_func_vals)), objective_func_vals)
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.show()

# --- 3. Instantiate the VQC ---
# We combine all our components into the VQC class.
vqc = VQC(
    sampler=sampler,
    feature_map=feature_map,
```

```
    ansatz=ansatz,
    optimizer=optimizer,
    callback=callback_graph,
)

# --- 4. Train the Model ---
# Clear the objective function values list before starting a new training run
objective_func_vals =
print("Training the VQC...")
vqc.fit(X_train, y_train)
print("Training complete.")
```

Explanation:

- **Optimizer**: We select COBYLA (Constrained Optimization By Linear Approximation). It is a gradient-free optimizer, making it a good choice for initial experiments as it avoids the complexities of quantum gradient calculation. We limit it to 100 iterations (maxiter=100).
- **Callback Function**: This function is a powerful tool for "seeing inside" the black box of training. The VQC.fit method will call this function after each iteration of the optimizer, passing it the current weights and the objective function (loss) value. Our function appends the value to a list and generates a plot, giving us a live view of how the loss is decreasing over time.
- **VQC Instantiation**: We create an instance of the VQC class, passing all our previously defined components: the sampler, feature_map, ansatz, optimizer, and our callback function.
- **Training**: The single command vqc.fit(X_train, y_train) starts the hybrid quantum-classical training loop. The VQC will now iteratively execute quantum circuits, measure the results, calculate the loss, and update the ansatz parameters to learn the patterns in our training data.

4.4. Model Evaluation and Visualization

Finally, after the model is trained, we enter the analysis and validation phase. We evaluate its performance quantitatively on both the training and unseen test data and visualize the decision boundary it has learned qualitatively.

Python

```
# --- 1. Evaluate Model Performance ---
train_score = vqc.score(X_train, y_train)
test_score = vqc.score(X_test, y_test)

print(f"VQC training accuracy: {train_score:.4f}")
print(f"VQC testing accuracy:  {test_score:.4f}")

# --- 2. Visualize the Decision Boundary ---
plt.rcParams["figure.figsize"] = (8, 6)

# Create a mesh grid of points to plot the decision boundary
h =.05  # step size in the mesh
x_min, x_max = X_scaled[:, 0].min() -.1, X_scaled[:, 0].max() +.1
y_min, y_max = X_scaled[:, 1].min() -.1, X_scaled[:, 1].max() +.1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Predict the class for each point in the mesh
# np.c_ concatenates the flattened xx and yy arrays into a 2D array of points
Z = vqc.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary and the data points
plt.contourf(xx, yy, Z, cmap=plt.cm.RdBu, alpha=0.8)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.RdBu, edgecolors='k',
label='Train')
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=plt.cm.RdBu, edgecolors='k',
marker='s', label='Test')
plt.title("VQC Decision Boundary")
plt.xlabel("Feature 1 (scaled)")
plt.ylabel("Feature 2 (scaled)")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```

Explanation:

- **Evaluation**: The vqc.score() method computes the mean accuracy of the classification on the given data. We compute this for both the training and testing sets. Comparing these two scores is essential for diagnosing overfitting—a situation where the model memorizes the training data but fails to generalize to new, unseen data.
- **Decision Boundary Visualization**: This is a powerful technique for understanding a classifier's behavior. We create a fine grid of points spanning the entire feature space. We then use our trained vqc to predict the class for every single point on this grid. By plotting these predictions as a colored background (plt.contourf), we can see the regions of the space that the VQC has learned to associate with each class. The line separating these colored regions is the decision boundary. Overlaying our original data points provides the ultimate visual confirmation that our quantum model has successfully learned to solve the non-linear classification problem by separating the two circles.

Contextualizing the VQC: A Tale of Two QML Philosophies

Learning the VQC in isolation can lead to a narrow understanding of the QML landscape. To provide a richer context, it is essential to contrast it with the other major paradigm in quantum classification: the Quantum Support Vector Machine (QSVM). This comparison reveals a fundamental strategic split in how researchers are attempting to leverage quantum computers for machine learning.

An Alternative Paradigm: Quantum Kernel Methods

To understand the QSVM, one must first understand its classical predecessor, the Support Vector Machine (SVM). An SVM is a powerful classification algorithm that finds an optimal hyperplane to separate data points of different classes. For data that is not linearly separable, the SVM employs the "kernel trick." A kernel is a function, K(xi,xj), that calculates the similarity (or inner product) between two data points in a higher-dimensional feature space, without ever having to explicitly compute the transformation into that space.

The QSVM, implemented in Qiskit as the QSVC (Quantum Support Vector Classifier), operates on a simple but powerful principle: it replaces the classical kernel computation with a quantum one. The process is as follows :

1. A quantum feature map, U(x), is chosen to encode classical data points xi and xj into quantum states $|\psi(xi)\rangle$ and $|\psi(xj)\rangle$.
2. The quantum computer is used to estimate the overlap, or fidelity, between these two states. This fidelity serves as the kernel value: $K(xi,xj)=|\langle\psi(xi)|\psi(xj)\rangle|2$.
3. This process is repeated for pairs of data points in the training set to construct a full quantum kernel matrix.
4. This quantum kernel matrix is then fed into a standard, classical SVM solver, which finds the optimal separating hyperplane in the quantum feature space.

Crucially, in this standard QSVM approach, only the kernel computation is quantum. The optimization part of the algorithm remains entirely classical.

VQC vs. QSVM: Two Philosophies of Quantum Classification

The distinction between the VQC and QSVM is not merely a choice between two algorithms; it reflects two different philosophies for achieving a quantum advantage.

- The **QSVM** represents a more conservative, **"quantum-enhanced"** philosophy. It identifies a specific, classically hard component of a well-understood and powerful classical algorithm (the kernel computation) and replaces only that part with a quantum subroutine. It aims to augment a proven classical framework with a potential quantum advantage in feature representation.
- The **VQC** embodies a more radical, **"end-to-end" quantum** philosophy. It discards the classical SVM framework entirely and instead builds a new type of classifier from the ground up, inspired by neural networks. Here, the entire model—from data encoding to the final transformation—is a quantum circuit, and the training process directly optimizes the parameters of this circuit to perform the classification task.

This philosophical difference leads to significant practical consequences. The QSVM inherits the robust optimization properties of classical SVMs; finding the optimal hyperplane given a kernel is a convex optimization problem, meaning it is guaranteed to find the global minimum and is not plagued by issues like spurious local minima. However, its power is entirely dependent on the choice of a fixed, non-trainable

feature map. The VQC, on the other hand, is far more flexible, as the variational ansatz can be trained to adapt to the specific problem. This gives it the potential to be more powerful, but this flexibility comes at the cost of a much more difficult training process. The optimization landscape for a VQC is generally non-convex and, as discussed, can suffer from barren plateaus, making training a significant challenge.

Recent research suggests that these differences lead to distinct performance characteristics. Several studies have found that QSVM can perform well, sometimes even outperforming classical SVMs, particularly in scenarios with limited data. In contrast, the VQC, while potentially struggling with small datasets, appears to have a higher ceiling for accuracy on larger and more complex datasets, assuming it can be trained successfully. The following table crisply summarizes this critical comparison, providing a quick, at-a-glance understanding of the crucial trade-offs and empowering a practitioner to think strategically about which tool to use for a given problem.

Table 2: VQC vs. QSVM - A Comparative Overview

Feature	Variational Quantum Classifier (VQC)	Quantum Support Vector Machine (QSVM/QSVC)
Core Mechanism	End-to-end trainable quantum circuit.	Classical SVM using a quantum-computed kernel.
What is Trained?	The parameters (θ) of the variational ansatz.	The support vector coefficients (alphas) in the classical SVM solver.
Role of Quantum Computer	Executes the entire classification circuit (feature map + ansatz) in each step of the optimization loop.	Computes the kernel matrix K(xi,xj) once (or a few times) before classical training.
Classical Analogy	Neural Network.	Support Vector Machine with a custom kernel.
Optimization Problem	Generally non-convex; gradient-based or gradient-free optimization of circuit parameters.	Convex; solved by classical quadratic programming.

Primary Challenge	Trainability issues (barren plateaus), sensitivity to noise.	Finding an effective feature map; the "curse of dimensionality" can make the kernel matrix trivial.
Best Suited For...	Potentially higher accuracy on larger, more complex datasets.	Good performance on smaller datasets; leverages robust classical optimization.

Advanced Topics, Practical Challenges, and Future Outlook

While the implementation of a VQC on a noise-free simulator can yield impressive results, transitioning to real-world problems and hardware exposes several significant challenges that are at the forefront of QML research. These challenges of trainability, noise, and architecture design are not independent problems; they are deeply interconnected facets of the same core issue—how to extract useful computational work from limited quantum resources.

The Trainability Problem: Barren Plateaus

As previously mentioned, the barren plateau phenomenon is one of the most significant obstacles to scaling VQAs. It refers to the observation that for many PQC architectures, particularly those that are deep or highly entangling, the variance of the gradient of the cost function with respect to the parameters vanishes exponentially as the number of qubits increases. An optimizer navigating a flat landscape has no information to guide its search, effectively halting the training process long before a solution is found.

Research into mitigating barren plateaus is an active and critical field. Key strategies include :

- **Careful Circuit Design**: Avoiding deep, unstructured ansaetze in favor of shallower, hardware-efficient circuits with local entanglement structures can reduce the likelihood of encountering a barren plateau.

- **Parameter Initialization**: Clever initialization strategies that correlate parameters or start the optimization in a region of non-zero gradients can help the training process get off the ground.
- **Local Cost Functions**: Using cost functions that depend only on measurements of a few qubits (local observables) can prevent the global nature of the problem from washing out gradient information, thus preserving a trainable landscape.
- **Geometric Quantum Machine Learning (GQML)**: This emerging approach leverages principles of symmetry in the problem to design ansaetze that are guaranteed to have non-zero gradients, effectively engineering the circuit to avoid barren plateaus from the start.

The Reality of Noise

NISQ devices are inherently noisy. Quantum states are fragile and quickly lose their quantum properties (decoherence) due to unwanted interactions with their environment. Furthermore, the quantum gates used to manipulate them are imperfect. This noise corrupts the final quantum state of the VQC, leading to incorrect measurement statistics and a distorted evaluation of the loss function, which can severely degrade or completely prevent successful training.

A design choice to increase expressivity, for example, by making the ansatz deeper, directly exacerbates the impact of hardware noise and simultaneously increases the risk of hitting a barren plateau. Addressing noise is therefore paramount. While full quantum error correction is not feasible on NISQ hardware, a suite of techniques known as **Quantum Error Mitigation (QEM)** can be used to reduce the impact of noise on computed results. These techniques often involve running additional calibration circuits and performing classical post-processing to extrapolate an estimate of the ideal, noise-free result. Another fascinating research direction involves making the model itself more robust to noise by, for example, learning quantum observables that are inherently less sensitive to the specific noise channels present in a device.

The Importance of Hyperparameter Tuning

The performance of a VQC is highly sensitive to a wide range of hyperparameters. As demonstrated in the implementation, choices regarding the feature map, the ansatz, their respective repetition counts (reps), the entanglement strategy ('linear', 'full', etc.), and the classical optimizer all have a profound impact on the final result. There is currently no universal theory that prescribes the optimal architecture for a given problem. QML remains a field of intense heuristic exploration, where researchers must experiment with different configurations to find a combination that balances expressivity, trainability, and noise resilience for their specific task. Navigating this complex, multi-dimensional problem space where every design choice has cascading effects is the central art of applied QML research.

Conclusion and Next Steps

This report has provided a comprehensive journey into the Variational Quantum Classifier, a cornerstone algorithm of near-term quantum machine learning. We have explored its theoretical underpinnings as a hybrid quantum-classical algorithm, navigated the modern Qiskit software ecosystem required for its implementation, and provided a complete, step-by-step guide to building and training a VQC from scratch. Furthermore, by contrasting the VQC with the QSVM, we have contextualized it within the broader strategic landscape of QML, revealing the different philosophies guiding the search for a quantum advantage.

The VQC, and VQAs in general, represent a promising pathway toward achieving practical quantum advantage on NISQ hardware. However, significant and interconnected challenges related to trainability, noise, and architecture design remain at the forefront of the field. For the ambitious learner, the path forward is clear. The provided code should serve not as a final destination, but as a sandbox for experimentation and deeper inquiry. The following steps are recommended for moving from a student of quantum machine learning to an active participant in its exciting development :

- **Explore different datasets**: Test the VQC on other classical machine learning datasets from libraries like scikit-learn to understand how its performance varies with data complexity and structure.
- **Vary the quantum components**: Swap the ZZFeatureMap for a PauliFeatureMap or the EfficientSU2 ansatz for RealAmplitudes. Systematically observe how changes in reps and the entanglement strategy affect accuracy, training time, and the shape of the loss curve.
- **Experiment with optimizers**: Replace the gradient-free COBYLA with a different

optimizer like SPSA (Simultaneous Perturbation Stochastic Approximation) or one of the gradient-based optimizers available in Qiskit to compare their convergence properties.

- **Advance to more complex models**: Investigate techniques like Quantum Kernel Training, which uses the QuantumKernelTrainer class to optimize the parameters of a feature map itself, bridging the gap between the VQC and QSVM paradigms. Explore the integration of VQCs with classical deep learning frameworks using the TorchConnector, which allows quantum layers to be embedded within larger PyTorch models.

By building upon the foundation laid in this report and actively engaging with these open challenges and advanced techniques, one can begin to contribute to the ongoing quest to unlock the power of quantum computation for machine learning.

Works cited

1. [2504.16131] Introduction to Quantum Machine Learning and Quantum Architecture Search - arXiv, accessed August 6, 2025, https://arxiv.org/abs/2504.16131
2. Introduction to Quantum Machine Learning and Quantum Architecture Search The views expressed in this article are those of the authors and do not represent the views of Wells Fargo. This article is for informational purposes only. Nothing contained in this article should be construed as investment advice. Wells Fargo makes no express or implied warranties and expressly disclaims all legal - arXiv, accessed August 6, 2025, https://arxiv.org/html/2504.16131v1
3. Empowering complex-valued data classification with the variational quantum classifier, accessed August 6, 2025, https://www.frontiersin.org/journals/quantum-science-and-technology/articles/10.3389/frqst.2024.1282730/full
4. comparative analysis of classical and quantum svm ... - DergiPark, accessed August 6, 2025, https://dergipark.org.tr/en/download/article-file/4943339
5. What is a variational quantum classifier? | by Krish Mittal - Medium, accessed August 6, 2025, https://medium.com/@typekrish/what-is-a-variational-quantum-classifier-888e40f83b24
6. Image classification with rotation-invariant variational quantum circuits | Phys. Rev. Research - Physical Review Link Manager, accessed August 6, 2025, https://link.aps.org/doi/10.1103/PhysRevResearch.7.013082
7. Impact of Amplitude and Phase Damping Noise on Quantum ..., accessed August 6, 2025, https://arxiv.org/abs/2503.24069
8. Learning Robust Observable to Address Noise in Quantum Machine ..., accessed August 6, 2025, https://arxiv.org/abs/2409.07632
9. Variational quantum classifiers through the lens of the Hessian ..., accessed

August 6, 2025, https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0262346
10. Variational Quantum Classifier - Dipòsit Digital UB, accessed August 6, 2025, https://diposit.ub.edu/dspace/bitstream/2445/140318/1/GIL%20FUSTER%20Elies%20Miquel.pdf
11. Learning to Program Quantum Measurements for Machine Learning - arXiv, accessed August 6, 2025, https://arxiv.org/pdf/2505.13525?
12. arxiv.org, accessed August 6, 2025, https://arxiv.org/html/2504.10073v2
13. On the Applicability of Quantum Machine Learning - PMC, accessed August 6, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC10377777/
14. Qiskit Variational Quantum Classifier on the Pulsar Classification Problem - arXiv, accessed August 6, 2025, https://arxiv.org/pdf/2505.15600
15. Quantum Machine Learning in 2025: The Power of QML, VQCs, and ..., accessed August 6, 2025, https://gganbumarketplace.com/quantum-machine-learning-in-2025-the-power-of-qml-vqcs-and-qsvms-with-qiskit/
16. qiskit-community/qiskit-aqua: Quantum Algorithms & Applications (**DEPRECATED** since April 2021 - see readme for more info) - GitHub, accessed August 6, 2025, https://github.com/qiskit-community/qiskit-aqua
17. Sklearn wrapper around QSVM and VQC · Issue #240 - GitHub, accessed August 6, 2025, https://github.com/Qiskit/qiskit-machine-learning/issues/240
18. Qiskit 0.33 release notes | IBM Quantum Documentation, accessed August 6, 2025, https://quantum.cloud.ibm.com/docs/api/qiskit/release-notes/0.33
19. Qiskit 0.46 and 1.0 (IBM) - LRZ Dokumentationsplattform, accessed August 6, 2025, https://doku.lrz.de/qiskit-0-46-and-1-0-ibm-1243353879.html
20. Taming the Qiskit Ecosystem: How to Build Quantum Algorithms After 2.0 - PyQML, accessed August 6, 2025, https://www.pyqml.com/qiskit-ecosystem
21. Unable to import Qiskit Aqua and terra Algorithm QSVM as per the documentation mentioned on Qiskit site - Stack Overflow, accessed August 6, 2025, https://stackoverflow.com/questions/76742125/unable-to-import-qiskit-aqua-and-terra-algorithm-qsvm-as-per-the-documentation-m
22. Building a Variational Quantum Classifier - Q-munity, accessed August 6, 2025, https://qmunity.thequantuminsider.com/2024/06/11/building-a-variational-quantum-classifier/
23. Your Qiskit Code Running in Quantum Computers on Azure with a Few Lines - Medium, accessed August 6, 2025, https://medium.com/@mancillamontero/your-qiskit-code-running-in-real-quantum-computers-on-azure-with-a-few-lines-1564a9fc0acc
24. Substitute for QuantumInstance in latest qiskit==1.0 - Quantum Computing Stack Exchange, accessed August 6, 2025, https://quantumcomputing.stackexchange.com/questions/39344/substitute-for-quantuminstance-in-latest-qiskit-1-0
25. Release Notes - Qiskit Aer 0.17.1, accessed August 6, 2025, https://qiskit.github.io/qiskit-aer/release_notes.html

26. Qiskit - IBM Quantum Computing, accessed August 6, 2025, https://www.ibm.com/quantum/qiskit
27. qiskit-community/qiskit-machine-learning: Quantum Machine Learning - GitHub, accessed August 6, 2025, https://github.com/qiskit-community/qiskit-machine-learning
28. Release Notes - Qiskit Optimization 0.6.1, accessed August 6, 2025, https://qiskit-community.github.io/qiskit-optimization/release_notes.html
29. qiskit-machine-learning/docs/tutorials ... - GitHub, accessed August 6, 2025, https://github.com/qiskit-community/qiskit-machine-learning/blob/main/docs/tutorials/02_neural_network_classifier_and_regressor.ipynb
30. Glossary | PennyLane, accessed August 6, 2025, https://pennylane.ai/qml/glossary
31. VQC - Qiskit Machine Learning 0.8.3 - GitHub Pages, accessed August 6, 2025, https://qiskit-community.github.io/qiskit-machine-learning/stubs/qiskit_machine_learning.algorithms.VQC.html
32. qiskit_machine_learning.algorithms.classifiers.vqc - Qiskit Machine Learning 0.8.3, accessed August 6, 2025, https://qiskit-community.github.io/qiskit-machine-learning/_modules/qiskit_machine_learning/algorithms/classifiers/vqc.html
33. ZZFeatureMap (latest version) | IBM Quantum Documentation, accessed August 6, 2025, https://quantum.cloud.ibm.com/docs/api/qiskit/qiskit.circuit.library.ZZFeatureMap
34. Qiskit Variational Quantum Classifier on the Pulsar Classification Problem - arXiv, accessed August 6, 2025, https://arxiv.org/html/2505.15600v1
35. EfficientSU2 (latest version) | IBM Quantum Documentation, accessed August 6, 2025, https://quantum.cloud.ibm.com/docs/api/qiskit/qiskit.circuit.library.EfficientSU2
36. EfficientSU2 (v0.19) | IBM Quantum Documentation, accessed August 6, 2025, https://docs.quantum.ibm.com/api/qiskit/0.19/qiskit.circuit.library.EfficientSU2
37. QISKIT's EfficientSU2 2-local circuit with 2 layers. The coloured gates... - ResearchGate, accessed August 6, 2025, https://www.researchgate.net/figure/QISKITs-EfficientSU2-2-local-circuit-with-2-layers-The-coloured-gates-are-the-gates_fig3_367975624
38. ZZFeatureMap entanglement setting in qiskit - EntangledQuery, accessed August 6, 2025, https://entangledquery.com/t/zzfeaturemap-entanglement-setting-in-qiskit/33/
39. qiskit-community-tutorials/machine_learning/vqc.ipynb at master ..., accessed August 6, 2025, https://github.com/qiskit-community/qiskit-community-tutorials/blob/master/machine_learning/vqc.ipynb
40. qiskit.algorithms.optimizers.COBYLA - IBM Quantum Platform, accessed August 6, 2025, https://quantum.cloud.ibm.com/docs/api/qiskit/0.26/qiskit.algorithms.optimizers.COBYLA
41. qiskit.optimization.algorithms.CobylaOptimizer - IBM Quantum Documentation, accessed August 6, 2025,

https://docs.quantum.ibm.com/api/qiskit/0.24/qiskit.optimization.algorithms.CobylaOptimizer
42. Saving, Loading Qiskit Machine Learning Models and Continuous Training - GitHub Pages, accessed August 6, 2025, https://qiskit-community.github.io/qiskit-machine-learning/tutorials/09_saving_and_loading_models.html
43. Quantum Support Vector Machine (QSVM) | by Thiago Veloso de Souza - Medium, accessed August 6, 2025, https://medium.com/mit-6-s089-intro-to-quantum-computing/quantum-support-vector-machine-qsvm-134eff6c9d3b
44. arxiv.org, accessed August 6, 2025, https://arxiv.org/html/2504.10073v2#:~:text=QSVM%20builds%20upon%20the%20classical,trainable%20quantum%20circuits%20optimized%20variationally.&text=A%20benchmark%20dataset%20from%20the,is%20used%20for%20model%20evaluation.
45. Quantum Support Vector Machines (QSVM) using Qiskit. | by Devmallya Karar - Medium, accessed August 6, 2025, https://medium.com/@devmallyakarar/quantum-support-vector-machines-qsvm-using-qiskit-eee347e81d83
46. Quantum_Kernel_Tutorial.ipynb - Colab, accessed August 6, 2025, https://colab.research.google.com/drive/1NTZDsNbk6KWO4b3T8ZYF2aX0NzXq7P7G?usp=sharing
47. Papers on how Quantum Support Vector Machines (QSVM) work - Reddit, accessed August 6, 2025, https://www.reddit.com/r/QuantumComputing/comments/1lqb035/papers_on_how_quantum_support_vector_machines/
48. Comparative Analysis of QSVM and VQC Models for B-cell Epitope Prediction in Vaccine Design Using Quantum Machine Learning - arXiv, accessed August 6, 2025, https://arxiv.org/html/2504.10073v1
49. [2504.10073] Comparative Analysis of Quantum Support Vector Machines and Variational Quantum Classifiers for B-cell Epitope Prediction in Vaccine Design - arXiv, accessed August 6, 2025, https://arxiv.org/abs/2504.10073
50. Is noise the only reason that makes the results of Quantum Support Vector Machine (QSVM) and Classical SVM differ, when we use a large dataset?, accessed August 6, 2025, https://quantumcomputing.stackexchange.com/questions/21170/is-noise-the-only-reason-that-makes-the-results-of-quantum-support-vector-machin
51. Comparative Performance Analysis of Quantum Machine Learning Architectures for Credit Card Fraud Detection - arXiv, accessed August 6, 2025, https://arxiv.org/html/2412.19441v1
52. Quantum Kernel Training for Machine Learning Applications - Qiskit ..., accessed August 6, 2025, https://qiskit-community.github.io/qiskit-machine-learning/tutorials/08_quantum_kernel_trainer.html