Quantum Machine Learning Data Encoding Research

Project Vision

Research Question: How do (1) statistical structures of classical data, (2) classical transformations of data, and (3) quantum encoding methods influence the performance of Quantum Machine Learning algorithms?

Key Distinction: This is NOT an analysis of model performance by fine-tuning hyperparameters. This is an analysis of the influence of **data representation** on QML algorithm performance.

Goal: Develop and implement efficient data encoding techniques for quantum devices, emphasizing the preprocessing pipeline that transforms classical data into quantum states suitable for NISQ-era QML.

Location: `qward/examples/papers/qml_data_encoding/`

Research Hypotheses

H1: Statistical Structure Hypothesis

The statistical properties of classical data (distribution shape, correlations, dimensionality, sparsity) significantly impact the expressiveness requirements of quantum encoding circuits.

H2: Classical Transformation Hypothesis

Classical preprocessing transformations (normalization, PCA, kernel methods) can reduce the quantum resources required for effective encoding.

H3: Encoding Method Hypothesis

Different quantum encoding methods (amplitude, angle, basis, IQP) have domain-specific advantages that depend on data characteristics rather than model architecture.

H4: Real-World Data Hypothesis

Standard preprocessing techniques developed for benchmark datasets (MNIST, Iris) are inadequate for real-world datasets with complex structures, requiring adaptive encoding strategies.

Scientific Background

The Challenge

Current QML research predominantly uses:

- **Benchmark datasets**: MNIST, Iris, Wine, Breast Cancer
- **Standard preprocessing**: MinMax scaling, PCA to reduce dimensions
- **Fixed encoding schemes**: Often amplitude or angle encoding without justification

This approach fails when:

- Real-world data has non-standard distributions (heavy tails, multimodal)
- Feature correlations don't align with PCA assumptions
- Data dimensionality exceeds available qubits significantly
- Class boundaries are not linearly separable in any encoding space

Our Approach

1. **Systematic encoding taxonomy**: Classify and implement major encoding schemes
2. **Data characterization framework**: Quantify statistical properties relevant to encoding
3. **Encoding-data compatibility analysis**: Map data characteristics to optimal encodings
4. **Real-world validation**: Test on non-benchmark datasets with complex structures
5. **Hybrid preprocessing pipeline**: Combine classical transformations with quantum encoding

Encoding Methods to Study

Encoding	Qubits Required	Circuit Depth	Best For
Basis Encoding	n (binary)	O(1)	Discrete/categorical data
Amplitude Encoding	$\log_2(n)$	O(n)	High-dimensional vectors
Angle Encoding	n features	O(1)	Continuous bounded data
IQP Encoding	n features	$O(n^2)$	Non-linear feature maps
Data Re-uploading	n features	O(L×n)	Expressive classification

Team & Responsibilities

Agent	Phase(s)	Primary Deliverables
`quantum-research-lead`	1, 6	Research direction, feasibility, synthesis
`quantum-computing-researcher`	2	Encoding theory, expressibility analysis, proofs
`test-engineer`	3	TDD test suite for all encoding methods
`python-architect`	4	Encoding library, QML models, pipelines
`quantum-data-scientist`	5	Statistical analysis, encoding comparison, visualizations
All	6	Review, iteration, conclusions

Phase 1: Ideation & Research Scoping (Lead + Researcher)

Objective

Define research scope, identify target datasets, and establish evaluation framework.

Tasks

Research Lead:

- [ ] Literature review: Data encoding in QML (2022-2025 papers)
- [ ] Survey existing encoding implementations in Qiskit, PennyLane, Cirq
- [ ] Identify gaps in current approaches for real-world data
- [ ] Select target datasets spanning different statistical profiles
- [ ] Define evaluation metrics (accuracy, resource usage, expressibility)
- [ ] Assess NISQ constraints for encoding circuits

Researcher (support):

- [ ] Theoretical foundations of quantum feature maps
- [ ] Expressibility bounds for different encoding schemes
- [ ] Relationship between encoding and kernel methods
- [ ] Barren plateau analysis for encoding circuits

Deliverables

```
qml-data-encoding/
  phase1_literature_review.md      # Comprehensive literature survey
  phase1_dataset_selection.md      # Target datasets with justification
  phase1_evaluation_framework.md   # Metrics and comparison methodology
  phase1_nisq_constraints.md       # Hardware limitations analysis
```

Target Datasets

Benchmark Datasets (Baseline Comparison)

Dataset	Samples	Features	Classes	Statistical Profile
Iris	150	4	3	Low-dim, Gaussian-like, separable
Wine	178	13	3	Mixed scales, moderate correlations
Breast Cancer	569	30	2	High-dim, correlated features
MNIST (subset)	1000	784→16	10	Image, requires heavy reduction

Real-World Datasets (Primary Focus)

Dataset	Samples	Features	Challenge
Credit Card Fraud	284,807	30	Extreme class imbalance (0.17%)
Network Intrusion (NSL-KDD)	148,517	41	Mixed categorical/continuous
Sensor Data (HAR)	10,299	561	High-dim time series
Medical (Heart Disease)	303	13	Missing values, mixed types
Financial (Stock Returns)	~5000	50+	Heavy tails, non-stationary

Dataset	Samples	Features	Challenge
Genomics (Gene Expression)	~1000	20,000+	Ultra-high dimensional, sparse

Evaluation Framework

Axis 1: Data Characteristics

- Distribution shape (Gaussian, multimodal, heavy-tailed)
- Feature correlations (Pearson, mutual information)
- Intrinsic dimensionality (PCA explained variance, intrinsic dim estimators)
- Class separability (Fisher's ratio, silhouette score)
- Sparsity (% zeros, information density)

Axis 2: Encoding Properties

- Qubit count required
- Circuit depth
- Expressibility (Haar-random distance)
- Entanglement capability
- Trainability (gradient variance)

Axis 3: QML Performance

- Classification accuracy
- Training convergence speed
- Generalization gap (train vs test)
- Resource efficiency (accuracy per qubit)
- Noise robustness

Success Criteria

Metric	Threshold
Encoding taxonomy completeness	≥ 5 encoding families implemented
Dataset coverage	≥ 3 benchmark + ≥ 3 real-world datasets
Statistical characterization	≥ 10 metrics per dataset
Performance comparison	Statistical significance ($p < 0.05$)
Actionable guidelines	Data→Encoding mapping recommendations

Handoff Checklist

- [ ] Literature review approved
- [ ] Datasets selected with statistical profiles
- [ ] Evaluation framework defined
- [ ] NISQ constraints documented

Phase 2: Theoretical Design (Researcher)

Objective

Formalize encoding methods, analyze expressibility, and design the experimental framework.

Tasks

2.1 Encoding Method Formalization

- [ ] Define mathematical framework for quantum feature maps
- [ ] Formalize each encoding scheme with explicit circuits
- [ ] Analyze qubit/depth requirements
- [ ] Prove encoding equivalences and distinctions

2.2 Expressibility Analysis

- [ ] Define expressibility metrics for encoding circuits
- [ ] Analyze encoding circuits as kernel methods
- [ ] Derive capacity bounds for different encodings
- [ ] Identify encoding-data compatibility conditions

2.3 Classical Preprocessing Theory

- [ ] Analyze how classical transformations affect encoded states
- [ ] Derive conditions for lossless dimensionality reduction
- [ ] Study normalization schemes compatible with quantum bounds
- [ ] Formulate hybrid classical-quantum preprocessing pipelines

2.4 QML Model Selection

- [ ] Select supervised models: Variational Quantum Classifier (VQC), QSVM
- [ ] Select unsupervised models: Quantum k-means, QAE (Quantum Autoencoder)
- [ ] Define model architectures that isolate encoding effects
- [ ] Establish baseline configurations (fixed across experiments)

Deliverables

```
qml-data-encoding/
  phase2_encoding_theory.md         # Mathematical formalization
  phase2_expressibility_analysis.md # Expressibility bounds and proofs
  phase2_preprocessing_theory.md    # Classical transformation analysis
  phase2_experimental_design.md     # Controlled experiment framework
```

Mathematical Specifications

Quantum Feature Map Definition

A quantum feature map is a parameterized unitary U(x) that encodes classical data $x \in \mathbb{R}^n$ into a quantum state:

```
|φ(x)⟩ = U(x)|0⟩^⊗m

where:
- x = (x₁, x₂, ..., xₙ) is the classical feature vector
- m = number of qubits
- U(x) is a unitary operation parameterized by x
```

Encoding Schemes

1. Basis Encoding

```
x = (b₁, b₂, ..., bₙ) ∈ {0,1}ⁿ → |x⟩ = |b₁b₂...bₙ⟩

Circuit: X gates on qubits where bᵢ = 1
Qubits: n
Depth: O(1)
```

2. Amplitude Encoding

```
x = (x₁, x₂, ..., xₙ) → |φ(x)⟩ = (1/||x||) Σᵢ xᵢ |i⟩

where |i⟩ is computational basis state
Qubits: ⌈log₂(n)⌉
Depth: O(n) or O(poly(log n)) with QRAM
```

3. Angle Encoding (Rotation Encoding)

```
x = (x₁, x₂, ..., xₙ) → |φ(x)⟩ = ⊗ᵢ Rʸ(xᵢ)|0⟩

where Rʸ(θ) = exp(−iθY/2)
Qubits: n
Depth: O(1)
Constraint: xᵢ ∈ [0, 2π] (requires normalization)
```

4. IQP (Instantaneous Quantum Polynomial) Encoding

```
U_IQP(x) = H^⊗n · D(x) · H^⊗n

D(x) = exp(i Σᵢ xᵢ Zᵢ + i Σᵢ<ⱼ xᵢxⱼ ZᵢZⱼ)
```

```
Qubits: n
Depth: O(n²) for full connectivity
Creates non-linear feature interactions
```

5. Data Re-uploading

$$U(x,\theta) = \prod_l [W(\theta_l) \cdot S(x)]$$

where:
- $S(x)$ = encoding layer (angle encoding)
- $W(\theta_l)$ = trainable variational layer

Qubits: n (or fewer with repeated encoding)
Depth: O(L×n) for L layers
Universal approximator property

Expressibility Metric

$$Expr(U) = D_{KL}(P_U \,||\, P_{Haar})$$

where:
- P_U = distribution of fidelities $F = |\langle\psi|\varphi\rangle|^2$ for random $U(x)$, $U(x')$
- P_{Haar} = Haar-random distribution

Lower Expr → closer to Haar-random → more expressive

Kernel Connection

Quantum encoding defines an implicit kernel:

$$K(x, x') = |\langle\varphi(x)|\varphi(x')\rangle|^2$$

For angle encoding:
$$K(x, x') = \prod_i \cos^2((x_i - x'_i)/2)$$

For IQP encoding:
$$K(x, x') = (1/2^n)|\Sigma_i \exp(i(f(x) - f(x'))_i)|^2$$

Classical Preprocessing Transformations

1. Normalization Schemes

```
MinMax: x' = (x − min)/(max − min) × 2π    # For angle encoding
Z-score: x' = (x − μ)/σ                     # Standardization
```

```
L2: x' = x/||x||                              # For amplitude encoding
```

2. Dimensionality Reduction

```
PCA: x' = W_k^T x                             # Linear, preserves variance
t-SNE: Non-linear, preserves local structure (not invertible)
Autoencoder: x' = Enc(x)                      # Non-linear, learned
```

3. Feature Engineering

```
Polynomial: φ(x) = (1, x₁, x₂, x₁², x₁x₂, x₂², ...)
RBF approximation: φ(x) = (cos(w₁·x), sin(w₁·x), ...)
```

Experimental Design

Controlled Variables (Fixed Across Experiments)

- **QML Model**: Variational Quantum Classifier (VQC)
- **Ansatz**: RealAmplitudes with 2 repetitions
- **Optimizer**: COBYLA with maxiter=100
- **Shots**: 1024
- **Train/Test Split**: 80/20 stratified

Independent Variables (Systematically Varied)

1. **Encoding method**: Basis, Amplitude, Angle, IQP, Re-uploading
2. **Classical preprocessing**: None, MinMax, Z-score, PCA
3. **Dataset**: As listed in Phase 1

Dependent Variables (Measured)

- Classification accuracy (test set)
- Training convergence (loss curve)
- Circuit metrics (depth, gates, via QWARD)
- Computational cost (circuit evaluations)

Handoff Checklist

- ☐ All encoding methods formalized
- ☐ Expressibility analysis complete
- ☐ Preprocessing theory documented
- ☐ Experimental design specified with controls

Phase 3: Test Design (Test Engineer)

Objective

Create comprehensive TDD test suite for all encoding methods and QML pipelines.

Tasks

- [ ] Create test fixtures for all datasets
- [ ] Write encoding correctness tests
- [ ] Write preprocessing transformation tests
- [ ] Write QML model tests
- [ ] Write end-to-end pipeline tests
- [ ] Write statistical validation tests

Deliverables

```
qml-data-encoding/
  tests/
    conftest.py                          # Shared fixtures
    test_data_characterization.py        # Statistical property tests
    test_preprocessing.py                # Classical transformation tests
    test_encoding_basis.py               # Basis encoding tests
    test_encoding_amplitude.py           # Amplitude encoding tests
    test_encoding_angle.py               # Angle encoding tests
    test_encoding_iqp.py                 # IQP encoding tests
    test_encoding_reuploading.py         # Data re-uploading tests
    test_expressibility.py               # Encoding expressibility tests
    test_qml_classifier.py               # VQC/QSVM tests
    test_qml_unsupervised.py             # Clustering tests
    test_pipeline_integration.py         # End-to-end pipeline tests
    test_classical_baseline.py           # Sklearn baseline (must pass)
```

Test Categories

1. Data Characterization Tests

```python
import pytest
import numpy as np
from scipy import stats
from sklearn.datasets import load_iris, load_wine

@pytest.fixture
def iris_dataset():
    """Load Iris dataset with statistical profile."""
    X, y = load_iris(return_X_y=True)
    return {'X': X, 'y': y, 'name': 'iris'}

@pytest.fixture
def real_world_dataset():
    """Load credit card fraud dataset (imbalanced)."""
```

```
    # Assumes data is downloaded
    df = pd.read_csv('data/creditcard.csv')
    X = df.drop('Class', axis=1).values
    y = df['Class'].values
    return {'X': X, 'y': y, 'name': 'credit_fraud'}

def test_data_distribution_characterization(iris_dataset):
    """Verify data distribution metrics are computed correctly."""
    X = iris_dataset['X']
    profile = compute_data_profile(X)

    assert 'mean' in profile
    assert 'std' in profile
    assert 'skewness' in profile
    assert 'kurtosis' in profile
    assert 'correlation_matrix' in profile
    assert profile['n_features'] == 4
    assert profile['n_samples'] == 150

def test_class_separability_metric(iris_dataset):
    """Verify Fisher's discriminant ratio computation."""
    X, y = iris_dataset['X'], iris_dataset['y']
    fisher_ratio = compute_fisher_ratio(X, y)

    # Iris should have good separability
    assert fisher_ratio > 1.0  # Classes are separable

def test_intrinsic_dimensionality(iris_dataset):
    """Verify intrinsic dimensionality estimation."""
    X = iris_dataset['X']
    intrinsic_dim = estimate_intrinsic_dimension(X)

    # Iris is 4D but intrinsic dim likely lower
    assert 1 <= intrinsic_dim <= 4
```

2. Preprocessing Transformation Tests

```
def test_minmax_normalization_bounds():
    """MinMax scales data to [0, 2π] for angle encoding."""
    X = np.random.randn(100, 4) * 10 + 5
    X_scaled = minmax_for_angle_encoding(X)

    assert X_scaled.min() >= 0
    assert X_scaled.max() <= 2 * np.pi

def test_amplitude_encoding_normalization():
    """L2 normalization for amplitude encoding."""
    X = np.random.randn(100, 4)
    X_norm = normalize_for_amplitude_encoding(X)

    norms = np.linalg.norm(X_norm, axis=1)
```

```
    assert np.allclose(norms, 1.0)

def test_pca_preserves_variance():
    """PCA reduction preserves specified variance."""
    X = np.random.randn(100, 20)
    X_reduced, explained_var = pca_reduce(X, n_components=5)

    assert X_reduced.shape[1] == 5
    assert explained_var >= 0.8  # At least 80% variance

def test_preprocessing_invertibility():
    """Invertible preprocessing can reconstruct original."""
    X = np.random.randn(100, 4)
    preprocessor = InvertiblePreprocessor()
    X_transformed = preprocessor.fit_transform(X)
    X_reconstructed = preprocessor.inverse_transform(X_transformed)

    assert np.allclose(X, X_reconstructed, atol=1e-10)
```

3. Encoding Correctness Tests

```
def test_basis_encoding_binary():
    """Basis encoding produces correct computational basis state."""
    x = np.array([1, 0, 1, 1])  # Binary input
    circuit = basis_encoding(x)

    backend = AerSimulator()
    result = backend.run(circuit, shots=1).result()
    measured = list(result.get_counts().keys())[0]

    assert measured == '1011'  # Reversed due to Qiskit convention

def test_angle_encoding_state():
    """Angle encoding produces correct rotation angles."""
    x = np.array([np.pi/2, np.pi])
    circuit = angle_encoding(x)

    # |ψ⟩ = Ry(π/2)|0⟩ ⊗ Ry(π)|0⟩ = |+⟩ ⊗ |1⟩
    backend = AerSimulator(method='statevector')
    result = backend.run(circuit).result()
    statevector = result.get_statevector()

    # Expected: (1/√2)(|00⟩ + |01⟩) for first qubit being |+⟩
    # With second qubit |1⟩: (1/√2)(|01⟩ + |11⟩)
    expected = np.array([0, 1/np.sqrt(2), 0, 1/np.sqrt(2)])
    assert np.allclose(np.abs(statevector), np.abs(expected), atol=0.01)

def test_amplitude_encoding_normalization():
    """Amplitude encoding requires normalized input."""
    x = np.array([3, 4])  # Not normalized
    x_norm = x / np.linalg.norm(x)  # [0.6, 0.8]
```

```
    circuit = amplitude_encoding(x_norm)
    backend = AerSimulator(method='statevector')
    result = backend.run(circuit).result()
    statevector = result.get_statevector()

    # |ψ⟩ = 0.6|0⟩ + 0.8|1⟩
    assert np.allclose(np.abs(statevector[:2]), [0.6, 0.8], atol=0.01)

def test_iqp_encoding_entanglement():
    """IQP encoding creates entanglement between qubits."""
    x = np.array([1.0, 1.0])
    circuit = iqp_encoding(x)

    # Measure entanglement via concurrence or Schmidt decomposition
    backend = AerSimulator(method='statevector')
    result = backend.run(circuit).result()
    statevector = result.get_statevector()

    entanglement = compute_entanglement(statevector, 2)
    assert entanglement > 0  # Non-zero entanglement

def test_reuploading_encoding_depth():
    """Data re-uploading has expected depth scaling."""
    x = np.array([1.0, 2.0, 3.0])
    n_layers = 3

    circuit = reuploading_encoding(x, n_layers=n_layers)

    # Depth should scale with layers
    assert circuit.depth() >= n_layers * 2  # At least encoding +
entangling per layer
```

4. Expressibility Tests

```
def test_expressibility_angle_vs_iqp():
    """IQP encoding should be more expressive than angle encoding."""
    n_qubits = 4
    n_samples = 1000

    expr_angle = compute_expressibility(angle_encoding, n_qubits,
n_samples)
    expr_iqp = compute_expressibility(iqp_encoding, n_qubits, n_samples)

    # Lower expressibility value = more expressive (closer to Haar)
    assert expr_iqp < expr_angle

def test_encoding_kernel_computation():
    """Encoding defines valid kernel (positive semi-definite)."""
    X = np.random.randn(50, 4)
    X_scaled = minmax_for_angle_encoding(X)
```

```
    K = compute_quantum_kernel(X_scaled, encoding='angle')

    # Kernel matrix must be PSD
    eigenvalues = np.linalg.eigvalsh(K)
    assert np.all(eigenvalues >= -1e-10)  # Allow numerical tolerance
```

5. QML Model Tests

```
def test_vqc_with_angle_encoding(iris_dataset):
    """VQC with angle encoding achieves reasonable accuracy on Iris."""
    X, y = iris_dataset['X'], iris_dataset['y']
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

    # Preprocess
    X_train_scaled = minmax_for_angle_encoding(X_train)
    X_test_scaled = minmax_for_angle_encoding(X_test)

    # Train VQC
    vqc = VariationalQuantumClassifier(
        encoding='angle',
        ansatz='RealAmplitudes',
        n_layers=2
    )
    vqc.fit(X_train_scaled, y_train)
    accuracy = vqc.score(X_test_scaled, y_test)

    # Should achieve at least 70% on Iris
    assert accuracy > 0.7

def test_vqc_encoding_comparison(iris_dataset):
    """Different encodings produce different accuracies (data matters)."""
    X, y = iris_dataset['X'], iris_dataset['y']
    X = X[:100]  # Subset for speed
    y = y[:100]

    accuracies = {}
    for encoding in ['angle', 'iqp', 'reuploading']:
        vqc = VariationalQuantumClassifier(encoding=encoding)
        scores = cross_val_score(vqc, X, y, cv=3)
        accuracies[encoding] = scores.mean()

    # Encodings should produce different results
    assert len(set(round(a, 2) for a in accuracies.values())) > 1

def test_qsvm_kernel_classification(iris_dataset):
    """QSVM with quantum kernel achieves reasonable accuracy."""
    X, y = iris_dataset['X'], iris_dataset['y']
    y_binary = (y == 0).astype(int)  # Binary classification
```

```
    X_train, X_test, y_train, y_test = train_test_split(X, y_binary,
test_size=0.2)

    qsvm = QuantumSVM(encoding='iqp')
    qsvm.fit(X_train, y_train)
    accuracy = qsvm.score(X_test, y_test)

    assert accuracy > 0.8  # Binary Iris should be easy
```

6. Pipeline Integration Tests

```
def test_full_pipeline_benchmark_dataset():
    """End-to-end pipeline works on benchmark dataset."""
    pipeline = QMLPipeline(
        preprocessor='minmax',
        encoding='angle',
        model='vqc',
        model_params={'n_layers': 2}
    )

    X, y = load_iris(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

    pipeline.fit(X_train, y_train)
    accuracy = pipeline.score(X_test, y_test)
    predictions = pipeline.predict(X_test)

    assert accuracy > 0.6
    assert len(predictions) == len(y_test)

def test_full_pipeline_real_world_dataset(real_world_dataset):
    """Pipeline handles real-world imbalanced data."""
    X, y = real_world_dataset['X'][:1000], real_world_dataset['y'][:1000]

    pipeline = QMLPipeline(
        preprocessor='pca',
        pca_components=8,
        encoding='iqp',
        model='vqc'
    )

    # Use stratified split for imbalanced data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, stratify=y
    )

    pipeline.fit(X_train, y_train)

    # For imbalanced data, check F1 score not just accuracy
    from sklearn.metrics import f1_score
```

```
    predictions = pipeline.predict(X_test)
    f1 = f1_score(y_test, predictions)

    assert f1 > 0.0  # At least some minority class predictions
```

7. Classical Baseline Tests (Must Pass First)

```
def test_sklearn_baseline_iris():
    """Classical SVM baseline for comparison."""
    from sklearn.svm import SVC
    from sklearn.preprocessing import StandardScaler

    X, y = load_iris(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    svm = SVC(kernel='rbf')
    svm.fit(X_train_scaled, y_train)
    accuracy = svm.score(X_test_scaled, y_test)

    # Classical baseline should achieve >90% on Iris
    assert accuracy > 0.9

def test_classical_preprocessing_correctness():
    """Verify preprocessing produces expected transformations."""
    X = np.array([[1, 2], [3, 4], [5, 6]])

    # MinMax to [0, 1]
    from sklearn.preprocessing import MinMaxScaler
    scaler = MinMaxScaler()
    X_scaled = scaler.fit_transform(X)

    assert X_scaled.min() == 0
    assert X_scaled.max() == 1
```

Handoff Checklist

- [ ] All test files created
- [ ] Classical baseline tests PASS
- [ ] Encoding tests FAIL (red phase - implementation needed)
- [ ] Test documentation complete
- [ ] Fixtures cover benchmark and real-world datasets

Phase 4: Implementation (Python Architect)

Objective

Build comprehensive QML encoding library passing all tests.

Tasks

4.1 Data Characterization Module

- [ ] Implement statistical profile computation
- [ ] Implement intrinsic dimensionality estimation
- [ ] Implement class separability metrics
- [ ] Create data profiling report generator

4.2 Preprocessing Module

- [ ] Implement encoding-aware normalizers
- [ ] Implement dimensionality reduction (PCA, feature selection)
- [ ] Implement feature transformations
- [ ] Create preprocessing pipeline builder

4.3 Encoding Library

- [ ] Implement basis encoding
- [ ] Implement amplitude encoding (with efficient state preparation)
- [ ] Implement angle encoding (with variants: Rx, Ry, Rz)
- [ ] Implement IQP encoding
- [ ] Implement data re-uploading
- [ ] Implement expressibility computation

4.4 QML Models

- [ ] Implement Variational Quantum Classifier (VQC)
- [ ] Implement Quantum SVM (kernel-based)
- [ ] Implement Quantum k-means clustering
- [ ] Wrap models in sklearn-compatible API

4.5 Pipeline Integration

- [ ] Create unified QMLPipeline class
- [ ] Implement experiment runner for systematic comparisons
- [ ] Integrate with QWARD for circuit analysis

Deliverables

```
qml-data-encoding/
  src/
    __init__.py
    data/
```

```
    __init__.py
    characterization.py         # Statistical profiling
    datasets.py                 # Dataset loaders
    profile.py                  # Data profile class
  preprocessing/
    __init__.py
    normalizers.py              # Encoding-aware normalization
    dimensionality.py           # PCA, feature selection
    transformers.py             # Feature engineering
    pipeline.py                 # Preprocessing pipeline
  encoding/
    __init__.py
    base.py                     # Abstract encoding class
    basis.py                    # Basis encoding
    amplitude.py                # Amplitude encoding
    angle.py                    # Angle encoding (Rx, Ry, Rz)
    iqp.py                      # IQP encoding
    reuploading.py              # Data re-uploading
    expressibility.py           # Expressibility metrics
    kernel.py                   # Quantum kernel computation
  models/
    __init__.py
    vqc.py                      # Variational Quantum Classifier
    qsvm.py                     # Quantum SVM
    qkmeans.py                  # Quantum k-means
    base.py                     # Abstract QML model
  pipeline/
    __init__.py
    qml_pipeline.py             # Unified pipeline
    experiment_runner.py        # Systematic experiments
  analysis/
    __init__.py
    comparison.py               # Encoding comparison tools
    visualization.py            # Results visualization
```

Class Architecture

```python
# encoding/base.py
from abc import ABC, abstractmethod
from typing import Optional, Union
import numpy as np
from qiskit import QuantumCircuit

class QuantumEncoding(ABC):
    """Abstract base class for quantum data encodings."""

    def __init__(self, n_qubits: Optional[int] = None):
        self.n_qubits = n_qubits
        self._circuit_template: Optional[QuantumCircuit] = None

    @abstractmethod
```

```
    def encode(self, x: np.ndarray) -> QuantumCircuit:
        """Encode classical data into quantum circuit.

        Args:
            x: Feature vector of shape (n_features,)

        Returns:
            QuantumCircuit with encoded data
        """
        pass

    @abstractmethod
    def required_qubits(self, n_features: int) -> int:
        """Calculate qubits needed for given feature dimension."""
        pass

    @property
    @abstractmethod
    def name(self) -> str:
        """Encoding method name."""
        pass

    def encode_batch(self, X: np.ndarray) -> list[QuantumCircuit]:
        """Encode batch of data points."""
        return [self.encode(x) for x in X]

    def compute_kernel(self, X1: np.ndarray, X2: np.ndarray) ->
np.ndarray:
        """Compute quantum kernel matrix K[i,j] = |⟨φ(x1_i)|φ(x2_j)⟩|²."""
        from .kernel import quantum_kernel_matrix
        return quantum_kernel_matrix(X1, X2, self)
```

```
# encoding/angle.py
import numpy as np
from qiskit import QuantumCircuit
from .base import QuantumEncoding

class AngleEncoding(QuantumEncoding):
    """Angle (rotation) encoding.

    Encodes each feature as a rotation angle on a separate qubit:
    |φ(x)⟩ = ⊗ᵢ R(xᵢ)|0⟩

    Args:
        rotation_axis: 'x', 'y', or 'z' for Rx, Ry, Rz gates
        repetitions: Number of encoding repetitions (for expressibility)
    """

    def __init__(
        self,
        rotation_axis: str = 'y',
```

```
        repetitions: int = 1,
        n_qubits: Optional[int] = None
    ):
        super().__init__(n_qubits)
        self.rotation_axis = rotation_axis.lower()
        self.repetitions = repetitions

        if self.rotation_axis not in ['x', 'y', 'z']:
            raise ValueError(f"rotation_axis must be 'x', 'y', or 'z', got
{rotation_axis}")

    @property
    def name(self) -> str:
        return f"Angle-R{self.rotation_axis.upper()}"

    def required_qubits(self, n_features: int) -> int:
        return n_features

    def encode(self, x: np.ndarray) -> QuantumCircuit:
        """Encode feature vector using rotation gates.

        Args:
            x: Feature vector, should be in [0, 2π] range

        Returns:
            Quantum circuit encoding the data
        """
        n_features = len(x)
        n_qubits = self.n_qubits or n_features
        qc = QuantumCircuit(n_qubits)

        for rep in range(self.repetitions):
            for i, xi in enumerate(x):
                if i >= n_qubits:
                    break
                if self.rotation_axis == 'x':
                    qc.rx(xi, i)
                elif self.rotation_axis == 'y':
                    qc.ry(xi, i)
                else:
                    qc.rz(xi, i)

        return qc
```

```
# encoding/iqp.py
import numpy as np
from qiskit import QuantumCircuit
from itertools import combinations
from .base import QuantumEncoding

class IQPEncoding(QuantumEncoding):
```

```
    """Instantaneous Quantum Polynomial (IQP) encoding.

    Creates non-linear feature interactions through ZZ gates:
    U_IQP(x) = H^⊗n · exp(i Σᵢⱼ xᵢxⱼ ZᵢZⱼ) · exp(i Σᵢ xᵢ Zᵢ) · H^⊗n

    This encoding is more expressive than angle encoding due to
    feature-feature interactions that create entanglement.
    """

    def __init__(
        self,
        n_qubits: Optional[int] = None,
        interaction_depth: int = 1
    ):
        super().__init__(n_qubits)
        self.interaction_depth = interaction_depth

    @property
    def name(self) -> str:
        return "IQP"

    def required_qubits(self, n_features: int) -> int:
        return n_features

    def encode(self, x: np.ndarray) -> QuantumCircuit:
        """Encode with IQP circuit.

        Args:
            x: Feature vector

        Returns:
            IQP-encoded quantum circuit
        """
        n_features = len(x)
        n_qubits = self.n_qubits or n_features
        qc = QuantumCircuit(n_qubits)

        # Initial Hadamard layer
        qc.h(range(min(n_features, n_qubits)))

        for _ in range(self.interaction_depth):
            # Single-qubit Z rotations
            for i, xi in enumerate(x):
                if i >= n_qubits:
                    break
                qc.rz(xi, i)

            # Two-qubit ZZ interactions
            for (i, j) in combinations(range(min(n_features, n_qubits)),
2):
                qc.rzz(x[i] * x[j], i, j)

        # Final Hadamard layer
        qc.h(range(min(n_features, n_qubits)))
```

```
        return qc
```

```
# models/vqc.py
from typing import Optional, Union, Callable
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin
from qiskit import QuantumCircuit
from qiskit.circuit.library import RealAmplitudes
from qiskit_algorithms.optimizers import COBYLA
from qward.algorithms import QuantumCircuitExecutor

from ..encoding.base import QuantumEncoding
from ..encoding.angle import AngleEncoding

class VariationalQuantumClassifier(BaseEstimator, ClassifierMixin):
    """Variational Quantum Classifier with configurable encoding.

    Sklearn-compatible quantum classifier that separates encoding
    from ansatz/training to isolate encoding effects.

    Args:
        encoding: QuantumEncoding instance or string name
        ansatz: Variational ansatz ('RealAmplitudes', 'EfficientSU2', or
QuantumCircuit)
        n_layers: Number of ansatz layers
        optimizer: Classical optimizer
        shots: Number of measurement shots
        noise_preset: QWARD noise preset name
    """

    def __init__(
        self,
        encoding: Union[str, QuantumEncoding] = 'angle',
        ansatz: str = 'RealAmplitudes',
        n_layers: int = 2,
        optimizer: str = 'COBYLA',
        max_iter: int = 100,
        shots: int = 1024,
        noise_preset: Optional[str] = None
    ):
        self.encoding = encoding
        self.ansatz = ansatz
        self.n_layers = n_layers
        self.optimizer = optimizer
        self.max_iter = max_iter
        self.shots = shots
        self.noise_preset = noise_preset

        self._encoder: Optional[QuantumEncoding] = None
        self._ansatz_circuit: Optional[QuantumCircuit] = None
```

```
        self._optimal_params: Optional[np.ndarray] = None
        self._classes: Optional[np.ndarray] = None
        self._n_qubits: Optional[int] = None
        self._loss_history: list = []

    def _build_encoder(self, n_features: int) -> QuantumEncoding:
        """Build encoding circuit."""
        if isinstance(self.encoding, QuantumEncoding):
            return self.encoding
        elif self.encoding == 'angle':
            return AngleEncoding()
        elif self.encoding == 'iqp':
            from ..encoding.iqp import IQPEncoding
            return IQPEncoding()
        elif self.encoding == 'reuploading':
            from ..encoding.reuploading import DataReuploading
            return DataReuploading(n_layers=self.n_layers)
        else:
            raise ValueError(f"Unknown encoding: {self.encoding}")

    def _build_ansatz(self, n_qubits: int) -> QuantumCircuit:
        """Build variational ansatz."""
        if self.ansatz == 'RealAmplitudes':
            return RealAmplitudes(n_qubits, reps=self.n_layers)
        elif self.ansatz == 'EfficientSU2':
            from qiskit.circuit.library import EfficientSU2
            return EfficientSU2(n_qubits, reps=self.n_layers)
        else:
            raise ValueError(f"Unknown ansatz: {self.ansatz}")

    def fit(self, X: np.ndarray, y: np.ndarray) ->
'VariationalQuantumClassifier':
        """Train the classifier.

        Args:
            X: Training features, shape (n_samples, n_features)
            y: Training labels, shape (n_samples,)

        Returns:
            self
        """
        self._classes = np.unique(y)
        n_features = X.shape[1]

        # Build encoder and ansatz
        self._encoder = self._build_encoder(n_features)
        self._n_qubits = self._encoder.required_qubits(n_features)
        self._ansatz_circuit = self._build_ansatz(self._n_qubits)

        n_params = self._ansatz_circuit.num_parameters

        # Optimization
        def loss_function(params):
            predictions = self._predict_proba_with_params(X, params)
```

```
            loss = self._cross_entropy_loss(predictions, y)
            self._loss_history.append(loss)
            return loss

        optimizer = self._get_optimizer()
        initial_params = np.random.randn(n_params) * 0.1

        result = optimizer.minimize(loss_function, initial_params)
        self._optimal_params = result.x

        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict class labels."""
        proba = self.predict_proba(X)
        return self._classes[np.argmax(proba, axis=1)]

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """Predict class probabilities."""
        return self._predict_proba_with_params(X, self._optimal_params)

    @property
    def loss_history(self) -> list:
        """Training loss history."""
        return self._loss_history
```

```
# pipeline/qml_pipeline.py
from typing import Optional, Union, Dict, Any
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.decomposition import PCA

from ..preprocessing.normalizers import AngleNormalizer,
AmplitudeNormalizer
from ..encoding.base import QuantumEncoding
from ..models.vqc import VariationalQuantumClassifier

class QMLPipeline(BaseEstimator, ClassifierMixin):
    """Unified Quantum Machine Learning pipeline.

    Combines classical preprocessing, quantum encoding, and QML model
    in a single sklearn-compatible interface.

    Example:
        >>> pipeline = QMLPipeline(
        ...     preprocessor='minmax',
        ...     encoding='iqp',
        ...     model='vqc',
        ...     model_params={'n_layers': 2}
```

```
        ... )
        >>> pipeline.fit(X_train, y_train)
        >>> accuracy = pipeline.score(X_test, y_test)
    """

    def __init__(
        self,
        preprocessor: str = 'minmax',
        pca_components: Optional[int] = None,
        encoding: str = 'angle',
        model: str = 'vqc',
        model_params: Optional[Dict[str, Any]] = None,
        noise_preset: Optional[str] = None
    ):
        self.preprocessor = preprocessor
        self.pca_components = pca_components
        self.encoding = encoding
        self.model = model
        self.model_params = model_params or {}
        self.noise_preset = noise_preset

        self._pipeline: Optional[Pipeline] = None

    def _build_preprocessor(self):
        """Build preprocessing steps."""
        steps = []

        # Dimensionality reduction
        if self.pca_components:
            steps.append(('pca', PCA(n_components=self.pca_components)))

        # Normalization for encoding
        if self.preprocessor == 'minmax':
            steps.append(('normalizer', AngleNormalizer()))
        elif self.preprocessor == 'amplitude':
            steps.append(('normalizer', AmplitudeNormalizer()))
        elif self.preprocessor == 'standard':
            steps.append(('normalizer', StandardScaler()))

        return steps

    def _build_model(self):
        """Build QML model."""
        params = {
            'encoding': self.encoding,
            'noise_preset': self.noise_preset,
            **self.model_params
        }

        if self.model == 'vqc':
            return VariationalQuantumClassifier(**params)
        elif self.model == 'qsvm':
            from ..models.qsvm import QuantumSVM
            return QuantumSVM(**params)
```

```
        else:
            raise ValueError(f"Unknown model: {self.model}")

    def fit(self, X: np.ndarray, y: np.ndarray) -> 'QMLPipeline':
        """Fit the pipeline."""
        steps = self._build_preprocessor()
        steps.append(('model', self._build_model()))
        self._pipeline = Pipeline(steps)
        self._pipeline.fit(X, y)
        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict class labels."""
        return self._pipeline.predict(X)

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        """Compute accuracy score."""
        return self._pipeline.score(X, y)

    def get_encoding_circuit(self, x: np.ndarray) -> 'QuantumCircuit':
        """Get encoding circuit for a single sample (for analysis)."""
        # Apply preprocessing
        x_transformed = x.reshape(1, -1)
        for name, transformer in self._pipeline.named_steps.items():
            if name == 'model':
                break
            x_transformed = transformer.transform(x_transformed)
        return
self._pipeline.named_steps['model']._encoder.encode(x_transformed[0])
```

```
# pipeline/experiment_runner.py
from typing import List, Dict, Any, Optional
import numpy as np
import pandas as pd
from itertools import product
from sklearn.model_selection import cross_val_score
from dataclasses import dataclass

from .qml_pipeline import QMLPipeline
from ..data.characterization import compute_data_profile

@dataclass
class ExperimentConfig:
    """Configuration for a single experiment."""
    dataset_name: str
    preprocessor: str
    encoding: str
    model: str
    model_params: Dict[str, Any]

@dataclass
```

```
class ExperimentResult:
    """Result of a single experiment."""
    config: ExperimentConfig
    accuracy_mean: float
    accuracy_std: float
    loss_history: List[float]
    circuit_depth: int
    circuit_gates: int
    data_profile: Dict[str, Any]

class EncodingExperimentRunner:
    """Systematic experiment runner for encoding comparison.

    Runs controlled experiments varying encoding methods while
    keeping model architecture fixed.
    """

    def __init__(
        self,
        datasets: Dict[str, tuple],  # name -> (X, y)
        encodings: List[str] = ['angle', 'iqp', 'reuploading'],
        preprocessors: List[str] = ['minmax', 'standard'],
        model: str = 'vqc',
        model_params: Dict[str, Any] = None,
        cv_folds: int = 5,
        noise_presets: Optional[List[str]] = None
    ):
        self.datasets = datasets
        self.encodings = encodings
        self.preprocessors = preprocessors
        self.model = model
        self.model_params = model_params or {'n_layers': 2, 'max_iter':
100}
        self.cv_folds = cv_folds
        self.noise_presets = noise_presets or [None]

    def run_all(self) -> pd.DataFrame:
        """Run all experiment combinations."""
        results = []

        for dataset_name, (X, y) in self.datasets.items():
            # Compute data profile once per dataset
            profile = compute_data_profile(X, y)

            for preprocessor, encoding, noise in product(
                self.preprocessors, self.encodings, self.noise_presets
            ):
                config = ExperimentConfig(
                    dataset_name=dataset_name,
                    preprocessor=preprocessor,
                    encoding=encoding,
                    model=self.model,
                    model_params=self.model_params
                )
```

```
                result = self._run_single(X, y, config, profile, noise)
                results.append(result)

        return self._results_to_dataframe(results)

    def _run_single(
        self,
        X: np.ndarray,
        y: np.ndarray,
        config: ExperimentConfig,
        profile: Dict,
        noise_preset: Optional[str]
    ) -> ExperimentResult:
        """Run single experiment configuration."""
        pipeline = QMLPipeline(
            preprocessor=config.preprocessor,
            encoding=config.encoding,
            model=config.model,
            model_params=config.model_params,
            noise_preset=noise_preset
        )

        # Cross-validation
        scores = cross_val_score(pipeline, X, y, cv=self.cv_folds)

        # Get circuit metrics from final fit
        pipeline.fit(X, y)
        circuit = pipeline.get_encoding_circuit(X[0])

        return ExperimentResult(
            config=config,
            accuracy_mean=scores.mean(),
            accuracy_std=scores.std(),

loss_history=pipeline._pipeline.named_steps['model'].loss_history,
            circuit_depth=circuit.depth(),
            circuit_gates=circuit.size(),
            data_profile=profile
        )
```

Integration with QWARD

```
from qward import Scanner
from qward.metrics import QiskitMetrics, ComplexityMetrics

# Analyze encoding circuits
def analyze_encoding_circuit(encoding, sample_data):
    """Use QWARD to analyze encoding circuit properties."""
    circuit = encoding.encode(sample_data)
```

```
    scanner = Scanner(circuit)
    metrics = scanner.scan()

    return {
        'depth': metrics.get_metric('depth'),
        'gate_count': metrics.get_metric('total_gates'),
        'two_qubit_gates': metrics.get_metric('cx_count'),
        'complexity': metrics.get_metric('complexity_score')
    }

# Visualization
scanner.visualize(save=True, show=False, output_path='img/')
```

Handoff Checklist

- [ ] All tests pass (green phase)
- [ ] Code follows .pylintrc standards
- [ ] Type hints complete
- [ ] Sklearn-compatible API
- [ ] QWARD integration verified
- [ ] All 5 encoding methods implemented

Phase 5: Execution & Analysis (Data Scientist)

Objective

Run systematic experiments comparing encoding methods across datasets and produce publication-ready analysis.

Tasks

5.1 Data Characterization Study

- [ ] Compute statistical profiles for all datasets
- [ ] Visualize distribution differences
- [ ] Identify data characteristics that may affect encoding choice
- [ ] Create data profile dashboard

5.2 Encoding Comparison Experiments

- [ ] Run all encoding × preprocessing × dataset combinations
- [ ] Compute statistical significance of differences
- [ ] Identify best encoding for each data profile
- [ ] Analyze failure cases

5.3 Expressibility vs Performance Analysis

- [ ] Correlate encoding expressibility with classification accuracy

- [ ] Analyze circuit depth vs accuracy trade-off
- [ ] Study data-encoding compatibility patterns

5.4 Real-World Dataset Evaluation

- [ ] Apply findings to real-world datasets
- [ ] Compare to standard preprocessing approaches
- [ ] Identify when QML provides advantages
- [ ] Document failure modes and limitations

5.5 Noise Impact Study

- [ ] Run experiments with IBM Heron and Rigetti noise models
- [ ] Characterize encoding robustness to noise
- [ ] Recommend noise-aware encoding strategies

Deliverables

```
qml-data-encoding/
  results/
    data_profiles/
      benchmark_profiles.csv           # Benchmark dataset statistics
      realworld_profiles.csv           # Real-world dataset statistics
    encoding_comparison/
      accuracy_matrix.csv              # Encoding × Dataset accuracy
      significance_tests.csv           # Statistical significance results
      expressibility_analysis.csv      # Expressibility metrics
    real_world/
      credit_fraud_results.csv
      har_results.csv
      medical_results.csv
    noise_analysis/
      noise_robustness.csv
      encoding_noise_interaction.csv
  img/
    data_profiles/
      distribution_comparison.png      # Data distribution visualizations
      correlation_heatmaps.png         # Feature correlation analysis
    encoding_comparison/
      accuracy_heatmap.png             # Encoding × Dataset accuracy
heatmap
      encoding_by_data_type.png        # Boxplots by data characteristics
      expressibility_vs_accuracy.png   # Scatter plot correlation
      circuit_complexity.png           # Depth/gates comparison
    convergence/
      loss_curves_by_encoding.png      # Training convergence
      convergence_by_dataset.png       # Dataset-specific convergence
    real_world/
      real_world_comparison.png        # Benchmark vs real-world
      failure_case_analysis.png        # Where encodings fail
    noise/
```

```
    noise_robustness.png              # Accuracy vs noise level
    encoding_noise_heatmap.png        # Encoding × Noise interaction
analysis/
  main_findings.md                    # Key results summary
  statistical_analysis.md             # Full statistical report
  encoding_recommendations.md         # Data→Encoding guidelines
  limitations.md                      # Failure modes and limitations
```

Analysis Plan

1. Data Profile Comparison

Dataset	n	d	Distribution	Correlations	Separability	Intrinsic Dim
Iris	150	4	Gaussian	Low	High (2.1)	2
Credit Fraud	284K	30	Heavy-tail	High	Low (0.3)	8
HAR	10K	561	Mixed	Moderate	Moderate	15
...	...	...	...	...	...	...

2. Encoding × Dataset Accuracy Matrix

	Angle	IQP	Re-upload	Amplitude	Basis
Iris	0.93	0.95	0.96	0.91	N/A
Credit Fraud	0.72	0.81	0.83	0.69	N/A
HAR	0.68	0.75	0.78	0.65	N/A

3. Statistical Hypothesis Testing

For each pair of encodings, test:

- H_0: μ_encoding1 = μ_encoding2 (equal accuracy)
- Use paired t-test or Wilcoxon signed-rank test
- Report p-values and effect sizes
- Correct for multiple comparisons (Bonferroni)

4. Data Characteristic → Encoding Mapping

Data Characteristic	Recommended Encoding	Reason
Gaussian, low-dim	Angle	Simple, sufficient expressibility
Heavy-tailed	IQP or Re-upload	Need non-linear feature interactions
High correlations	PCA + Angle	Reduce redundancy first
Class imbalance	Re-upload + oversampling	Need adaptive boundaries

Data Characteristic	Recommended Encoding	Reason
Mixed types	Hybrid	Basis for categorical, angle for continuous

Visualization Requirements

```python
import matplotlib.pyplot as plt
import seaborn as sns
from qward import Scanner

# 1. Data profile visualization
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
for ax, (name, profile) in zip(axes.flat, profiles.items()):
    ax.hist(profile['distribution'], bins=30)
    ax.set_title(f"{name}: Skew={profile['skewness']:.2f}")
fig.savefig('img/data_profiles/distribution_comparison.png')

# 2. Encoding comparison heatmap
accuracy_matrix = results.pivot(
    index='dataset',
    columns='encoding',
    values='accuracy_mean'
)
plt.figure(figsize=(10, 8))
sns.heatmap(accuracy_matrix, annot=True, cmap='YlGn', fmt='.2f')
plt.title('Classification Accuracy by Encoding and Dataset')
plt.savefig('img/encoding_comparison/accuracy_heatmap.png')

# 3. Expressibility vs accuracy scatter
plt.figure(figsize=(8, 6))
plt.scatter(
    results['expressibility'],
    results['accuracy_mean'],
    c=results['encoding'].map(color_map),
    s=100
)
plt.xlabel('Expressibility (lower = more expressive)')
plt.ylabel('Classification Accuracy')
plt.savefig('img/encoding_comparison/expressibility_vs_accuracy.png')

# 4. QWARD circuit analysis
for encoding_name, encoding in encodings.items():
    circuit = encoding.encode(sample_data)
    scanner = Scanner(circuit)
    scanner.scan().visualize(
        save=True,
        output_path=f'img/circuits/{encoding_name}/'
    )
```

Key Research Questions to Answer

1. **Does data structure predict optimal encoding?**
    - Correlate data profiles with best-performing encoding
    - Build predictive model: data features → encoding choice
2. **Is expressibility the key factor?**
    - Test if more expressive encodings always win
    - Identify when simpler encodings suffice
3. **How does preprocessing interact with encoding?**
    - Test preprocessing × encoding interactions
    - Identify optimal preprocessing for each encoding
4. **What fails on real-world data?**
    - Document specific failure modes
    - Propose mitigations
5. **How robust are encodings to noise?**
    - Rank encodings by noise robustness
    - Identify noise-optimal encodings

Handoff Checklist

- [ ] All experiments completed
- [ ] Statistical analysis with significance tests
- [ ] Visualizations saved to img/
- [ ] Data→Encoding recommendations documented
- [ ] Limitations and failure modes identified

Phase 6: Review & Synthesis (All Agents)

Objective

Evaluate research findings, iterate if needed, and produce final recommendations.

Decision Points

Finding	Threshold	Action if Not Met
Significant encoding differences	$p < 0.05$	Data Scientist: more samples, different metrics
Clear data→encoding pattern	$R^2 > 0.6$	Researcher: refine theoretical framework
Real-world improvement	> 5% over standard	Architect: tune preprocessing pipeline
Noise robustness	< 10% degradation	Architect: add error mitigation

Key Questions for Review

1. **Did we answer the research questions?**
   - H1: Statistical structure influence → Validated/Refuted
   - H2: Classical transformation influence → Validated/Refuted
   - H3: Encoding method influence → Validated/Refuted
   - H4: Real-world inadequacy → Validated/Refuted
2. **Are the findings generalizable?**
   - Do patterns hold across multiple datasets?
   - Are recommendations statistically robust?
3. **What are the limitations?**
   - NISQ constraints
   - Dataset biases
   - Model architecture confounds
4. **What are the practical recommendations?**
   - Actionable guidelines for practitioners
   - Decision tree for encoding selection

Iteration Routing

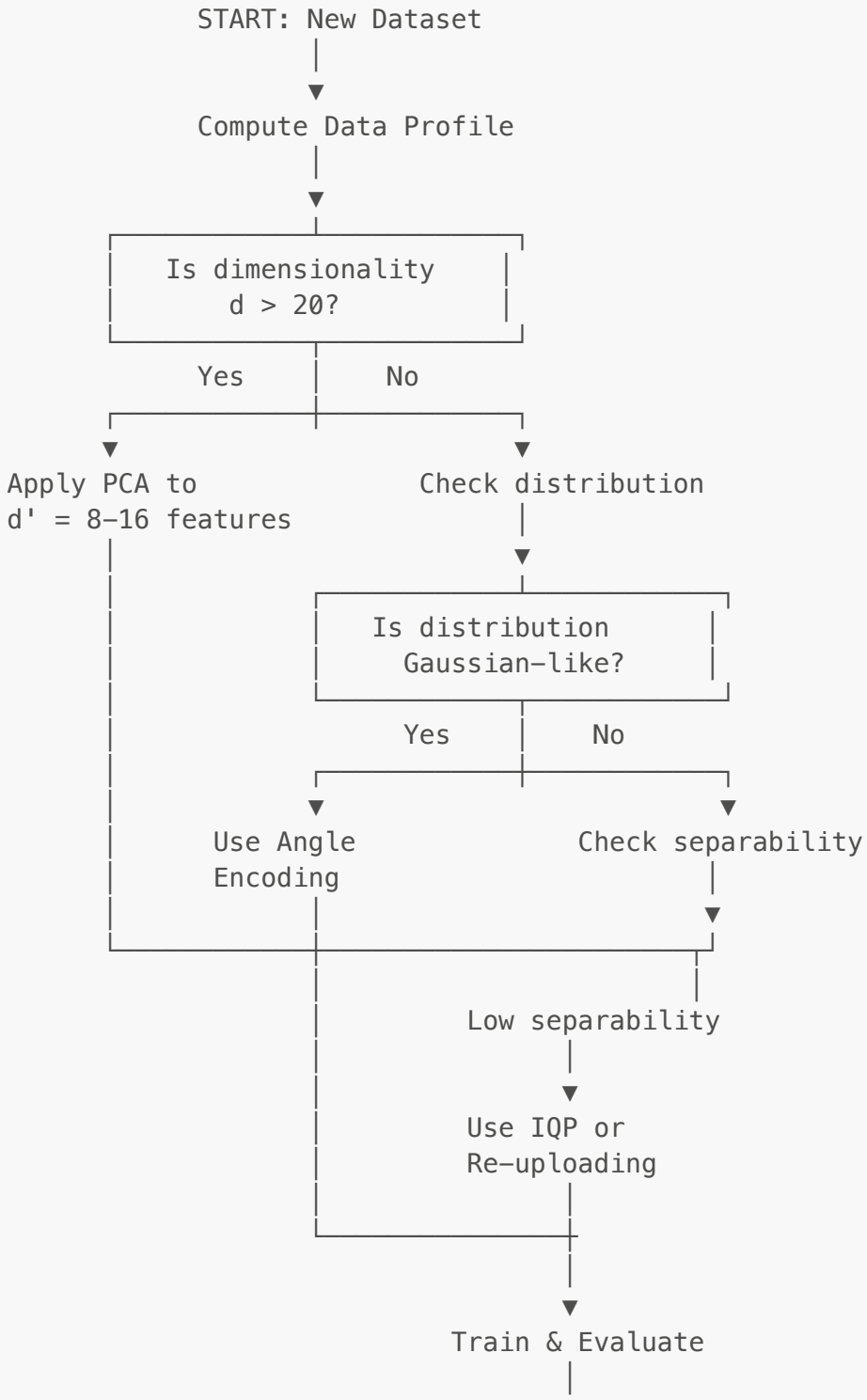
```
IF statistical_significance_poor:
    → Data Scientist: larger sample sizes, stratified analysis

IF no_clear_pattern:
    → Researcher: refine theoretical framework
    → Data Scientist: additional data characteristics

IF real_world_performance_poor:
    → Architect: custom preprocessing for specific data types
    → Researcher: hybrid encoding schemes

IF findings_contradict_theory:
    → Researcher: re-examine theoretical assumptions
    → Lead: literature comparison
```

Final Deliverables

```
qml-data-encoding/
  README.md                          # Project overview
  phase1_literature_review.md        # Background research
  phase1_dataset_selection.md        # Dataset justification
  phase1_evaluation_framework.md     # Metrics definition
  phase2_encoding_theory.md          # Mathematical foundations
```

```
phase2_experimental_design.md        # Controlled experiment design
src/                                 # Implementation
tests/                               # Test suite
results/                             # Raw results
img/                                 # Visualizations
analysis/                            # Analysis reports
final_report.md                      # Comprehensive findings
encoding_selection_guide.md          # Practical recommendations
decision_tree.png                    # Visual decision aid
```

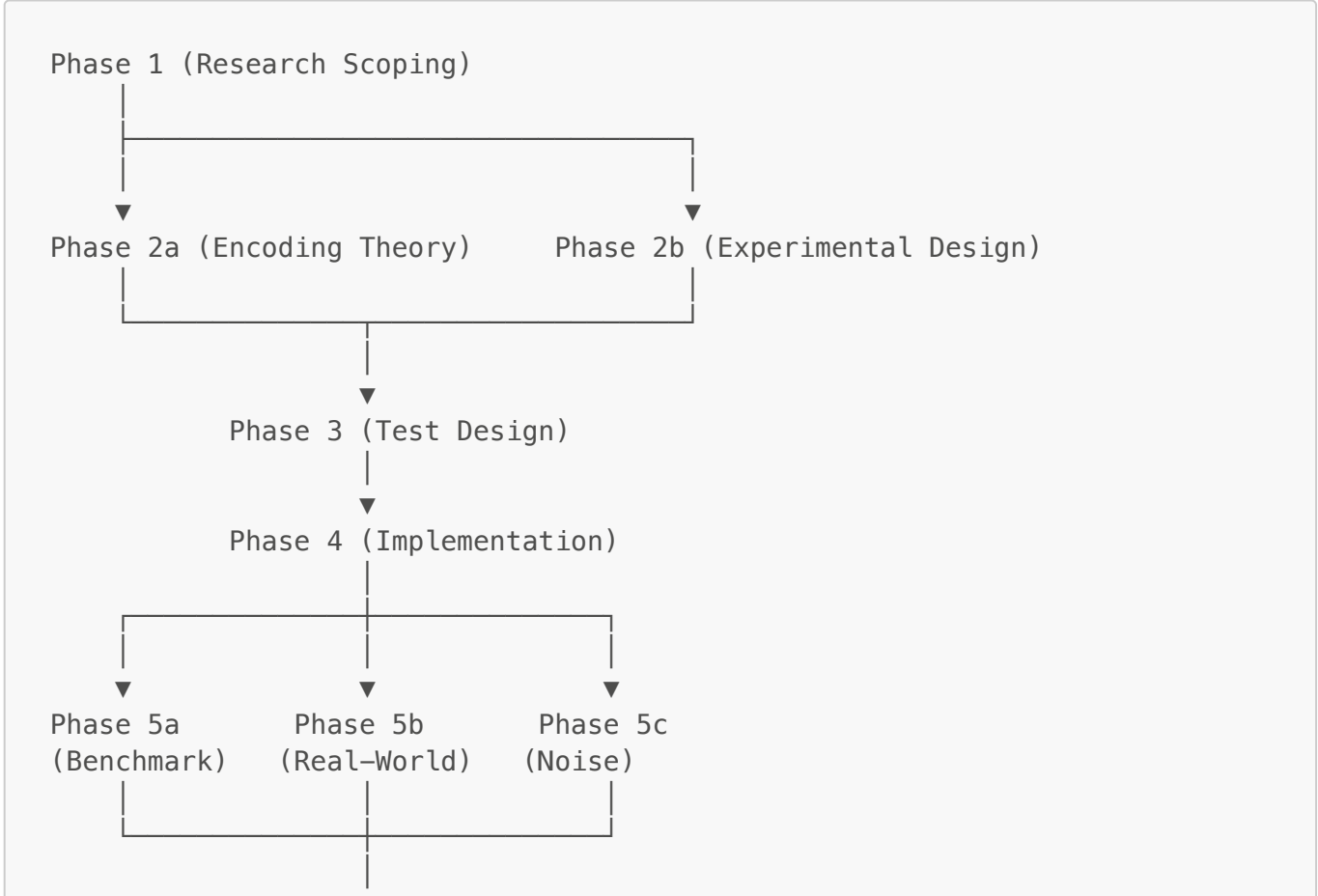
Encoding Selection Decision Tree (Final Output)

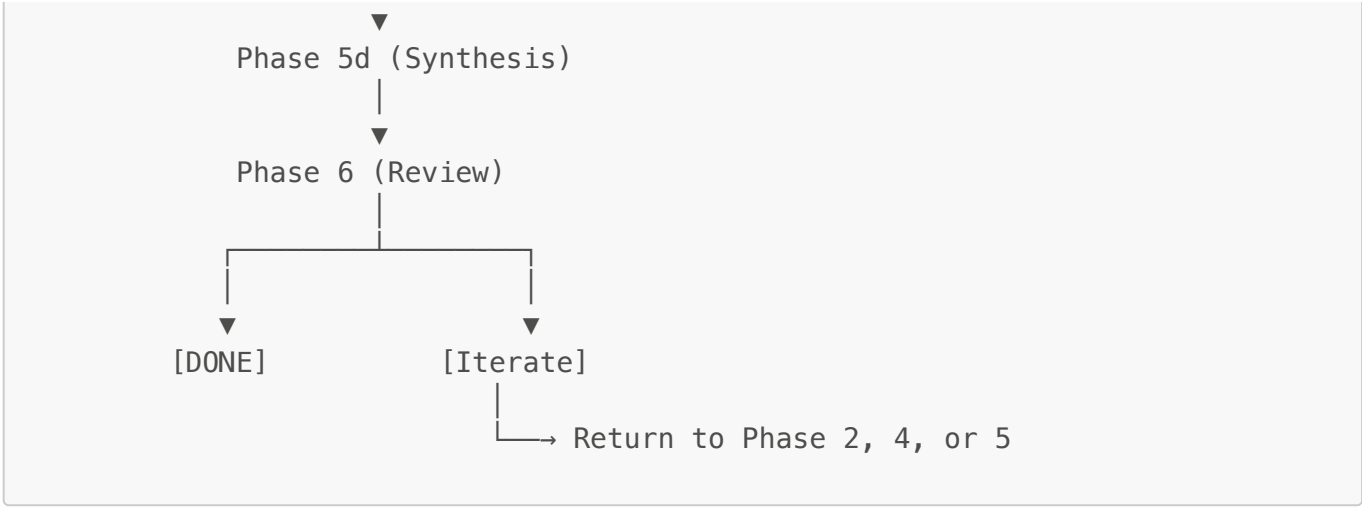


Success Metrics Summary

Metric	Target	Owner
Encoding methods implemented	≥ 5	Architect
Datasets analyzed	≥ 6 (3 benchmark + 3 real)	Data Scientist
Statistical significance	$p < 0.05$ for key comparisons	Data Scientist
Data→Encoding pattern	Identifiable correlations	Researcher + Data Scientist
Practical guidelines	Actionable decision tree	Lead
All tests pass	100%	Test Engineer
Code coverage	≥ 80%	Architect

Timeline & Dependencies





Dependencies & Requirements

Python Packages

```
# Core quantum (from requirements.qward.txt)
qiskit==2.1.2
qiskit-aer==0.17.1
qiskit-ibm-runtime==0.41.1

# Machine learning
scikit-learn>=1.2.0
torch==2.8.0

# Analysis
numpy>=1.24.0
pandas>=2.0.0
matplotlib>=3.7.0
seaborn>=0.12.0
scipy>=1.10.0
statsmodels

# Data
ucimlrepo>=0.0.3            # UCI ML Repository access
imbalanced-learn>=0.10.0    # For imbalanced datasets
```

External Datasets

```
# Download instructions
# 1. UCI ML Repository (automatic via ucimlrepo)
# 2. Kaggle datasets (manual download, API key required)
#    - Credit Card Fraud: kaggle.com/mlg-ulb/creditcardfraud
# 3. UCI HAR: archive.ics.uci.edu/dataset/240/human+activity+recognition
```

Quick Start Commands

```
# Install additional dependencies
pip install ucimlrepo imbalanced-learn seaborn

# Download datasets
python -m qward.examples.papers.qml_data_encoding.scripts.download_data

# Run classical baseline tests (must pass first)
pytest qward/examples/papers/qml-data-
encoding/tests/test_classical_baseline.py -v

# Run all encoding tests
pytest qward/examples/papers/qml-data-encoding/tests/ -v

# Run systematic experiments
python -m qward.examples.papers.qml_data_encoding.scripts.run_experiments

# Generate analysis report
python -m qward.examples.papers.qml_data_encoding.scripts.generate_report
```

Appendix: Reference Materials

Key Papers

1. **Data Encoding Survey**: Schuld & Petruccione, "Supervised Learning with Quantum Computers", Springer (2018)
2. **Expressibility**: Sim et al., "Expressibility and entangling capability of parameterized quantum circuits", Advanced Quantum Technologies (2019)
3. **Quantum Kernels**: Havlíček et al., "Supervised learning with quantum-enhanced feature spaces", Nature (2019)
4. **Data Re-uploading**: Pérez-Salinas et al., "Data re-uploading for a universal quantum classifier", Quantum (2020)
5. **IQP Circuits**: Shepherd & Bremner, "Temporally unstructured quantum computation", Proceedings of the Royal Society A (2009)
6. **QML Limitations**: Cerezo et al., "Challenges and opportunities in quantum machine learning", Nature Computational Science (2022)

Project Resources

- **QWARD Documentation**: See `skills/qward-development/` for API reference
- **Project Standards**: See `.pylintrc` and `requirements.qward.txt`
- **Workflow**: See `workflows/collaborative-development.md`