Quantum Decision Tree

COVER PAGE

Title : Quantum Decision Tree(QDT) Algorithm

Submitted by: ENUKONDA VISHNU VARDHAN REDDY
Intern / Researcher
NIT–SRINAGAR , KASHMIR

Guided by:
HARINATH REDDY DUGGIREDDY
CEO
ANANT WAVE

Date:
April 2025

Project Type:
Quantum Computing Function Documentation (Part of Quantum Machine Learning Module / Research Internship)

Submitted to:
[ANANT WAVE]
[KADAPA, Country (INDIA)]

- **Table of Contents**

1. Introduction

1.1 Overview of the Problem
1.2 What The Function Does
1.3 Principles
1.4 Applications
1.5 Theoretical and Real world Use cases

2. Implementations And Testing's

2.1 Steps to implement the function (use PennyLane or any other suitable framework)
2.2 Code snippets
2.3 Testing & Evaluation
2.4 What metrics or benchmarks can be used
2.5 Timeline & Complexity

3.Requirements

3.1 Requirements
3.2 Hardware and Software Needs
3.3 Tools And Libraries
3.4 Prerequisites in terms of knowledge
3.5 Team Requirements

4. References & Resources

4.1 Foundational Reading
4.2 Machine Learning & Classical Counterparts
4.3 Quantum Machine Learning
4.4 Video Lectures & MOOCs
4.5 Tools & Environments

5. Link to papers, articles, videos, or GitHub repositories used for research

5.1 Research Papers & Articles
5.2 Documentation & Articles
5.3 Videos & Tutorials
5.4 GitHub Repositories

INTRODUCTION

FUNCTION OVERVIEW

➢ $\boldsymbol{QuantumTreeNode.\_init\_()}$

- Sets up a beginning node in the quantum decision tree.
- Sets placeholders for:
  - **weights:** The quantum circuit at the node has trainable parameters.
  - **left, right:** Child node pointers that enable branching.
  - **predicted_class:** Predict the label of the node as its class when it is a leaf.

➢ $\boldsymbol{QuantumDecisionTree.\_init\_(max\_depth = 4, n\_qubits = 2)}$

- This initializes the model used for quantum decision trees.
- **Parameters:**
  - max_depth: The level of tree depth (controlled by the parameter) determines the complexity of the model.
  - n_qubits: How many qubits are used in the quantum circuitry.
- Creates a PennyLane quantum device.

➢ $\boldsymbol{QuantumDecisionTree.circuit(weights, x)}$

- Defines the quantum circuit architecture.
- Inputs:
  - weights: Trainable parameters (rotation angles).
  - x: Trying out the classical input features to represent the data.
- Operation:
  - Uses data re-uploads that work based on RY rotations.
  - Entangles qubits with CNOT gates.
  - Gives the expectation value of **PauliZ** acting on qubit 0.

➢ $\boldsymbol{QuantumDecisionTree.create\_qnode()}$

- Uses $\boldsymbol{qml.QNode}$ to package the circuit method.
- It makes it possible to run quantum circuits and obtain their gradients on the device.

➢ $\boldsymbol{QuantumDecisionTree.cost(weights, X, y, qnode)}$

- Decides on the method to calculate the cost for training.

- Inputs:
  - weights: Trainable parameters.
  - X, y: Input features and labels.
  - $qnode$: Quantum circuit node.

- Computes:
  - Difference squared between the values predicted by the model and the actual labels (in the range of 0 and 1).
  - Adds L2 regularization to prevent overfitting.

➢ $\underline{\boldsymbol{QuantumDecisionTree.train\ node(X, y)}}$

- Trains a quantum circuit to do the best job of dividing the dataset when it reaches a certain step in the calculation.

- Uses gradient descent with a learning method called AdamOptimizer to help reduce the error in the net.

- Returns:
  - Optimal weights and associated $qnode$.

➢ $\boldsymbol{QuantumDecisionTree.build\_tree(X, y, depth = 0)}$

- Recursively builds the entire decision tree by using the function one more time for each branch.

- At each node:

- Trains a quantum circuit to break apart data into smaller parts.

- Applies the circuit to every data point to figure out if it should go into the left or right branch.

- Stops splitting if:

  - Maximum depth is reached.

  - All labels on the node say the same thing.

- All data points get put in one group (degenerate split).

- Returns:

- A fully constructed $\boldsymbol{QuantumTreeNode}$.

➢ $\boldsymbol{QuantumDecisionTree.fit(X, y)}$

- Triggers build each step of the tree starting from the root node.
- Trains the quantum decision tree by using the data set you give it.

➢ $\boldsymbol{QuantumDecisionTree.predict\_single(x, node)}$

- Predicts the class the sample will be put into or pick out what group a single sample belongs to.
- Recursively goes through the tree by going down the path that matches the outputs from each part of the quantum circuit until it hits a leaf node.

➢ $\boldsymbol{QuantumDecisionTree.predict(X)}$

- Predicts which classes the input samples will belong to.
- Applies $predict\_single()$ to each row in the given dataset.

WHAT THE FUNCTION DOES

➢ $\boldsymbol{QuantumTreeNode.\_\_init\_\_()}$

Creates a root node with holders for quantum circuit parameters, choices of the next node, and prediction for the class.

➢ $\boldsymbol{QuantumDecisionTree.\_\_init\_\_(max\_depth, n\_qubits)}$

Configures the main parts of the quantum decision tree, setting the depth, qubits, and loading the quantum simulator.

- $\boldsymbol{circuit(weights, x)}$

Explains how classical data is turned into a quantum circuit and the role trainable parameters play in the result. Outputs a single prediction, representing the expectation.

- $\boldsymbol{Create\_qnode()}$

Converts the quantum circuit into a $QNode$ that can be run on the simulator and help with gradient-based optimization.

- $\boldsymbol{Cost(weights, X, y, qnode)}$

Shows how successfully the quantum circuit with its current weights is accomplishing the task. It finds the average of how much each sample differs from the guess, and it also penalizes large weights.

- $\boldsymbol{Train\_node(X, y)}$

The quantum circuit is trained to separate the data at a tree node. It fine-tunes the circuit's weights to make the outputs differentiate between the classes.

- $\boldsymbol{Build\_tree(X, y, depth)}$

Builds the tree recursively:

- Use a quantum machine to learn and train its data at each node.
- Lowers the data signal using the circuit and sends it to the left or right side.
- The algorithm stops when all nodes have a single pure label or the max depth is met.
- It builds and delivers a full decision tree.

- $\boldsymbol{Fit(X, y)}$

Builds the quantum decision tree from the ground up, using the information in the given dataset. The code internally runs build_tree() to do its job.

- $\boldsymbol{Predict\_single(x, node)}$

Forecasts a value for one input by running it along the branches of the tree. Every decision depends on processing the waves on each node separately.

- $\boldsymbol{Predict(X)}$

$Predict\_single()$ is used to find a label for each input sample in the dataset and the resulting labels are returned.

UNDERLYING PRINCIPLES

1. Quantum Circuits for Decision Making

- Feature threshold points are used to build decision boundaries in classical decision trees.
- Each internal node on a Quantum Decision Tree calculates the decision boundary with the help of a quantum circuit.
- The circuit shall provide an outcome (interpreted as an expectation value) which determines whether the data goes to the left or right subtree.

2. Qubits and Quantum Gates

- A qubit is the quantum version of a digital bit, starting at a default (usually 0) state |0⟩|0⟩.
- To encode classical data using quantum mechanics, we use rotation gates (RY).
    - **qml.RY(x[i] * π, wires=i)** maps the classical configuration settings to rotations in quantum computing.
- Using CNOT gates, circuit designers can entangle qubits and allow the circuit to describe tough decision functions.

3. Data Encoding (Angle Encoding / Re-uploading)

- Angle encoding is used by the model to handle classical kinds of data.
- π is multiplied by each feature and each feature is then encoded via an RY gate.
- Re-uploading: Once the first encoding is finished, another set of trainable RY rotations is applied.
- It allows the model to be more expressive by increasing the mix of data-influenced and parameter-influenced layers.

4. Quantum Measurement

- At the final step, the measurement checks what is expected for PauliZ on the first qubit.

```
return qml.expval(qml.PauliZ(0))
```

- It is found in [−1,1] and is used to make decisions about a case.
    - Even when the number of branches is zero in one direction, it can still be one in a different direction.

5. Quantum Circuit Training

- The circuit uses learnable weights (RY rotation angles) to achieve the task.
- We use gradient descent (with the Adam optimizer which is an optimization algorithm) to help the model learn and reduce the loss function.
- Sum of the squared differences between expected outputs of ±1 and the actual outputs of the circuit.
- Using automatic differentiation, PennyLane can find gradients in quantum circuits using rules known as parameter-shift.

6. Hybrid Quantum-Classical Model

- The model is hybrid:
    - The main features of classical computation are data flow, tree creation and working with recursion.
    - At decision points in the process, quantum computation is invoked to determine the results and split data.

7. Quantum Regularization

- To avoid overfitting, the model gets L2 regularization to penalize model parameters that are too large.

```
regularization = 0.001 * np.sum(weights**2)
```

8. Summary of Concepts

Quantum Concept	Role in Model
Feature/variable	Each feature has its own quantum information carried by a qubit
RY gates	Turn classical data (features) into angles
CNOT gates	Create a connection between two qubits
PauliZ expectation	Variable used to choose how to divvy the data at any node
Quantum circuit training	Explore how each node in a quantum circuit makes a decision by looking at its boundaries
Re-Uploading	Using several layers lets you increase how you express your design

APPLICATIONS

1. Binary Classification Tasks

- Since QDTs are structured with branches, they are usually built for binary classification problems.
- Example: Deciding if something is spam or regular email and if a tumor has turned cancerous or not.

2. Quantum Machine Learning Prototypes

- Well suited for exploring what could be done in the field of quantum machine learning.
- Allows researchers to understand how quantum circuits can either serve instead of or add to the classical decision boundaries.

3. Quantum-Enhanced Tree Models

- Represents a quantum-based version of decision trees used in:
- Random Forests
- Gradient Boosted Trees

- It is possible to form ensemble quantum models derived from applying several QDTs at once.

4. Low-Dimensional Feature Spaces

- Works in areas that can be best handled by compressing data into 2–4 features (given the qubit constraint).
- Example: Biological data sets have been changed to fewer dimensions through PCA or other methods.

5. Quantum-Classical Hybrid Systems

- The framework is designed to be useful when data is pre-processed prior to a computation by a quantum algorithm.
- Example: After learning with a classical neural network, we rely on a quantum decision tree to decide.

6. Educational Tools

- Great for teaching:
  - Quantum encoding techniques (angle encoding).
  - Variational quantum circuits.
  - Hybrid quantum-classical training.

7. Exploring Quantum Advantage

- To measure the performance of quantum circuits when dealing with small data compared to classical circuits.

- A step toward answering: Are quantum circuits more capable of generalizing than classical tree networks?

8. Quantum Hardware Experiments

- It is possible to carry out this model on real quantum computers if the number of qubits is not too high.
- Enables check-ups of hardware to make sure it behaves as it should.

REAL-WORLD OR THEORETICAL USE CASES

1. Hybrid Quantum-Classical Machine Learning

- **Use Case:**
  - Quantum decision trees fit into the collection of hybrid models that make use of both classical and quantum methods.
- **How it's used:**
  - Decision-making takes place at every single node in a quantum circuit.
  - Classical computation deals with building tree structures, controlling how data moves and managing it.
  - Because this approach is compatible with NISQ devices, it can easily be tested in practice.

2. Small-Scale, High-Value Decision Problems

- **Use Case:**
  - For example, there are finance, logistics or healthcare industries that deal with little data, yet the data is complicated.
- **How it's used:**
  - Use PCA or pick the most important features to convert your input data to around 2–4 values.
  - Educate the QDT to use the enhanced model from quantum technology to decide.
  - Example: Detecting fraud, sorting patients or organizing how they are seen in the hospital.

3. Feature-Reduced Bioinformatics / Genomics

- **Use Case:**
  - There are often many noisy features in medical data collections. After picking out the best features, quantum trees can be used for pattern learning.
- **How it's used:**
  - Try out standard forms of dimensionality reduction, for example, PCA or Lasso.
  - Classify the levels or presence of genes using QDT analysis.

4. Benchmarking Quantum vs Classical Models

- **Use Case:**
  - Using quantum models and classical models to see if there is an advantage in using quantum approaches.
- **How it's used:**
  - The evaluation and training happen for QDTs similarly as they do for regular decision trees or SVMs.
  - Researchers check how accurately the model works, how it performs on new data and how much time it takes to train.
  - Usually, problems are solved on simulators or small quantum hardware that uses a few qubits.

5. Quantum AutoML (Automated Machine Learning)

- **Use Case:**
  - QDTs could be an important part of a quantum AutoML system, where the circuit and tree structures are optimized without manual input.
- **How it's used:**
  - Each decision node is trainable.
  - These types of search algorithms use tree structures to guide their searches.
  - Allows for the development of small and efficient models in quantum computing.

6. Quantum-Inspired Algorithms

- **Use Case:**

Even without the help of quantum computers, QDTs can still be a source of inspiration for quantum-inspired classical algorithms.

- **How it's used:**

  - Applying entangled feature encoding or reuploading data to classical models can lead to improvement.
  - These training data sets are run in simulation and then transformed into better classical ML pipelines.

7. Quantum Edge Computing (Theoretical)

- **Use Case:**
  - In upcoming distributed systems, edge smart sensors could host quantum computers that are running quantum distributed tasks.
- **How it's used:**

- Lightweight quantum models such as QDTs are implemented on specialized quantum processors that are embedded.
- They handle quick yes or no decisions without depending on the central nervous system.

8. Quantum Control Systems

- **Use Case:**
- Design adaptive types of measurements for quantum systems.
- **How it's used:**
- Adjust the settings of the experiment by using QDTs when you spot any differences from your predictions.

IMPLEMENTATIONS AND TESTING'S

Implementation

```
import pennylane as qml
from pennylane import numpy as np
import numpy.random as npr

# Node of Quantum Decision Tree
class QuantumTreeNode:
    def _init_(self):
        self.weights = None  # Quantum circuit weights
        self.left = None
        self.right = None
        self.predicted_class = None  # Class label at leaf

# Main Quantum Decision Tree Class
class QuantumDecisionTree:
    def _init_(self, max_depth=4, n_qubits=2):
        self.max_depth = max_depth
        self.root = None
        self.n_qubits = n_qubits
        self.dev = qml.device("default.qubit", wires=self.n_qubi

    # Define a deeper quantum circuit
    def circuit(self, weights, x):
        for i in range(self.n_qubits):
            qml.RY(x[i] * np.pi, wires=i)
        for i in range(self.n_qubits - 1):
            qml.CNOT(wires=[i, i+1])

        # More trainable layers
        for i in range(self.n_qubits):
            qml.RY(weights[i], wires=i)
        qml.CNOT(wires=[0,1])
        for i in range(self.n_qubits):
```

```
            qml.RY(weights[i+self.n_qubits], wires=i)

        return qml.expval(qml.PauliZ(0))

    # Wrap quantum circuit
    def create_qnode(self):
        return qml.QNode(self.circuit, self.dev)

    # Define cost function with L2 regularization
    def cost(self, weights, X, y, qnode):
        loss = 0
        for xi, yi in zip(X, y):
            pred = qnode(weights, xi)
            loss += (pred - (1 if yi == 1 else -1))**2
        loss = loss / len(X)

        # L2 Regularization
        regularization = 0.001 * np.sum(weights**2)
        return loss + regularization

    # Train a single quantum split at a node
    def train_node(self, X, y):
        weights = np.array(npr.randn(self.n_qubits * 2), requires_grad=True)  # More weights
        qnode = self.create_qnode()
        opt = qml.AdamOptimizer(stepsize=0.05)  # Better optimizer with smaller stepsize

        for it in range(300):  # More training iterations
            weights = opt.step(lambda w: self.cost(w, X, y, qnode), weights)
        return weights, qnode

    # Recursive tree building
    def build_tree(self, X, y, depth=0):
        node = QuantumTreeNode()
```

```
        # Stopping condition
        if depth >= self.max_depth or len(np.unique(y)) == 1:
            node.predicted_class = int(np.bincount(y).argmax())
            return node

        # Train quantum circuit for split
        weights, qnode = self.train_node(X, y)
        node.weights = weights

        # Predict splits
        preds = np.array([qnode(weights, xi) for xi in X])
        preds_binary = (preds > 0).astype(int)

        # Split left and right
        left_idx = preds_binary == 0
        right_idx = preds_binary == 1

        if np.sum(left_idx) == 0 or np.sum(right_idx) == 0:
            # Edge case: All data went one side → make leaf
            node.predicted_class = int(np.bincount(y).argmax())
            return node

        node.left = self.build_tree(X[left_idx], y[left_idx], depth + 1)
        node.right = self.build_tree(X[right_idx], y[right_idx], depth + 1)

        return node

    # Fit the full tree
    def fit(self, X, y):
        self.root = self.build_tree(X, y)

    # Recursive prediction for one sample
    def predict_single(self, x, node):
```

```
        if node.predicted_class is not None:
            return node.predicted_class

        qnode = self.create_qnode()
        pred = qnode(node.weights, x)

        if pred > 0:
            return self.predict_single(x, node.right)
        else:
            return self.predict_single(x, node.left)

    # Predict for batch of samples
    def predict(self, X):
        return np.array([self.predict_single(xi, self.root) for xi in X])
      from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA

# Load Iris dataset
data = load_iris()
X, y = data.data, data.target
X = X[y != 2]
y = y[y != 2]

# Scale and PCA
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=42)

# 2. Train Quantum Decision Tree
qdt = QuantumDecisionTree(max_depth=3)
qdt.fit(X_train, y_train)

# 3. Predict
y_pred_train = qdt.predict(X_train)
y_pred_test = qdt.predict(X_test)

# 4. Accuracy
train_acc = np.mean(y_pred_train == y_train)
test_acc = np.mean(y_pred_test == y_test)

print(f"Quantum Decision Tree Train Accuracy: {train_acc*100:.2f}%")
print(f"Quantum Decision Tree Test Accuracy: {test_acc*100:.2f}%")
```

STEPS TO IMPLEMENT THE FUNCTION

❖ STEP 1 : INSTALL REQUIRED LIBRARIES

Install PennyLane, scikit-learn, and numpy (if not already installed):

```
pip install pennylane scikit-learn
```

❖ STEP 2 : DEFINE THE QUANTUM TREE NODE

Create a class to represent each node in the decision tree:

```
class QuantumTreeNode:
    def __init__(self):
        self.weights = None
        self.left = None
        self.right = None
        self.predicted_class = None
```

❖ STEP 3 : BUILD THE QUANTUM DECISION TREE CLASS

The course content should contain:

1. **Initialization:**

- Set tree depth to a maximum.
- Build a quantum device.

2. **Quantum Circuit:**

- Ray gates are used to encode the features.
- The CNOT gates are used to achieve entanglement.
- Bases the weights on received training examples.

3. **Cost Function:**

Also incorporates ADL and MSE loss

4. **Training the Nodes One by One.**

Use the Adam optimizer whenever you want to update the weights.

5. **Recursive Tree Building**

- A process of dividing a set into smaller parts.
- End at the pass of a maximum depth or a pure label leaf.
- Subdivide the tree according to the predictions made by quantum physics.

6. **Prediction Logic:**

Go through the tree using recursion to assign a label.

❖ **STEP 4: FORM THE DATA FOR THE MODEL**

1. Filter and use the Iris dataset to do binary classification.

2. Change features to a uniform scale by using **MinMaxScaler**.

3. Transform the data into only 2 features using PCA to work with 2-qubit circuits.

4. Separate the data into two groups: training and test data.

❖ **STEP 5: TRAIN THE MODEL**

```
qdt = QuantumDecisionTree(max_depth=3)
qdt.fit(X_train, y_train)
```

❖ **STEP 6: MAKE PREDECTIONS AND EVALUATE**

```
y_pred_train = qdt.predict(X_train)
y_pred_test = qdt.predict(X_test)

print(f"Train Accuracy: {np.mean(y_pred_train == y_train) * 100:.2f}%")
print(f"Test Accuracy: {np.mean(y_pred_test == y_test) * 100:.2f}%")
```

TESTING AND EVALUATION

1. Dataset Preparation

- Focus on binary classification of the species in the Iris dataset (**Setosa vs Versicolor**).
- Convert the two features into scaling's and proceed with the 2-qubit circuit.

Example :

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA

# Load dataset
data = load_iris()
X, y = data.data, data.target

# Binary classification (classes 0 and 1 only)
X, y = X[y != 2], y[y != 2]

# Normalize
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Dimensionality reduction
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=42)
```

2. Train the QDT Model

Example:

```
qdt = QuantumDecisionTree(max_depth=3)
qdt.fit(X_train, y_train)
```

3. Make Predictions

```
y_pred_train = qdt.predict(X_train)
y_pred_test = qdt.predict(X_test)
```

4. Evaluate Accuracy

```
train_acc = np.mean(y_pred_train == y_train)
test_acc = np.mean(y_pred_test == y_test)

print(f"Quantum Decision Tree Train Accuracy: {train_acc * 100:.2f}%")
print(f"Quantum Decision Tree Test Accuracy: {test_acc * 100:.2f}%")
```

5. Output of the Example

```
Quantum Decision Tree Train Accuracy: 100.00%
Quantum Decision Tree Test Accuracy: 90.00%

Confusion Matrix (Test):
[[10  0]
 [ 1  9]]

Classification Report (Test):
              precision    recall  f1-score   support

           0       0.91      1.00      0.95        10
           1       1.00      0.90      0.95        10

    accuracy                           0.95        20
   macro avg       0.95      0.95      0.95        20
weighted avg       0.95      0.95      0.95        20
```

METRICS OR BENCHMARKS

1. Accuracy

- It describes the share of predictions that were right.
- It is straightforward to use and reliable for dividing two categories.

```
accuracy = np.mean(y_pred == y_true)
```

2. Precision, Recall and F1 Score are used to measure performance.

- These tactics work well when there are more of one class on the field.

```
from sklearn.metrics import precision_score, recall_score, f1_score

precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)
```

3. Confusion Matrix

- Shows the numbers for TP, FP, TN, FN.
- Enables a person to identify various mistakes.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true, y_pred)
```

4. For probabilistic output, use ROC-AUC.

- Not utilized in this case, but you can use them if the model gives confidence scores.

5. Time to train the network / Time to use the network

- Evaluating a quantum circuit takes longer because real hardware is involved.
- Running benchmarks allows you to measure if your app can be scaled well.

6. Benchmarks to Compare Against

❖ Classical Models for Comparison

Use standard models to benchmark the QDT:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
```

- Compare QDT's accuracy and F1 score with these models

❖ **Quantum Benchmarks**

- Examine them against other quantum models (such as Quantum SVMs or Quantum Kernels).
- Analyze how working with increasing quantum resources affects QDT's performance.

TIMELINE AND COMPLEXITY

1. Timeline for Implementation and Execution

Step	Estimated Time
Library setup and data preprocessing	5–10 minutes
Defining QDT class and components	20–30 minutes
Training the tree (fit method)	5–30 minutes, depending on:
	➢ **Data Size**
	➢ **Tree Depth**
	➢ **Qubit Count**
Inference/prediction	1–2 minutes
Evaluation (accuracy, metrics)	<1 minute

2. Computational Complexity Analysis

1. Training Complexity

Each non-leaf node in the decision tree trains a **quantum circuit** using gradient descent. Assuming:

- d: tree depth
- n: number of samples
- m: number of features (qubits = 2 here)
- t: iterations per node (e.g., 300)
- k: number of trainable weights ($\approx 2 * n\_qubits$)

➢ **Worst-Case Node Count:**

- For binary trees, up to 2^d - 1 nodes.

➢ **Cost per Node:**

- For each node, circuit evaluation for all samples + gradient optimization $\Rightarrow \boldsymbol{O(n * t * k)}$

➢ **Total Tree Training Cost:**

- $\boldsymbol{O(2\text{^}d * n * t * k)}$

2. Prediction Complexity

Making predictions about a single game:

- For every d node, an instance of the circuit is run.
- Therefore, each sample requires O(d) steps in the evaluation process.
- For n_test samples: $O(n\_test * d)$

3. Quantum-Specific Overhead

- It takes a long time to run quantum circuits on simulators using PennyLane and NumPy.
- Photos using real gear usually take longer since several photos are taken, it is louder and there's a wait to be seen.
- The depth and number of qubits are restricted on noisy intermediate-scale quantum (NISQ) devices**.**

REQUIREMENTS

Software Requirements

❖ Programming Language

- Python 3.7+

❖ Required Libraries

Library	Purpose	Install Via
$Pennylane$	Quantum circuit construction/simulation	$pip\ install\ pennylane$
$Numpy$	Numerical operations	$pip\ install\ numpy$
$Scikit-learn$	Data preprocessing and evaluation	$pip\ install\ scikit-learn$

Hardware Requirements

❖ Lower Option (Local Simulator)

- At least 4 GB of RAM should be present.

- You only need a standard CPU for your laptop or desktop.
- A GPU is not necessary because the computations are handled by the CPU.
- When simulating more qubits, the computer's memory and CPU are pushed to the limit.

Requirements

❖ **Quantum Resource Requirements**

Resource	Value used in first step
No of Qubits	2 qubits
Circuit Depth	Moderate (3–4 entangling layers)
Shots	1 (expectation value used)

❖ **Data Requirements**

- Input features: The features have two dimensions after being processed by PCA.
- The target values are limited to 0 or 1.
- Data provided: Iris (Setosa vs Versicolor)

Tools And Libraries

1. PennyLane

❖ Aim: Creating and simulating quantum circuits.
❖ Features Used:

- $qml.device$ provides the options to create and use a quantum device (for example, simulators or real backends).
- $qml.QNode$: Creates a function that can be called to run the quantum circuit
- Quantum gates: $qml.RY, qml.CNOT$ and so on.
- The selected method for gradient descent is $qml.AdamOptimizer$.

❖ Install :

```
pip install pennylane
```

2. PennyLane relies upon NumPy.

❖ Application: Operations on mathematical data.

❖ Use:

- When working with auto-grad , $pennylane.numpy$ is used for differentiable functions.
- It is $numpy.random$ that facilitates setting up random weights.

```
from pennylane import numpy as np
import numpy.random as npr
```

3. scikit-learn (also called sklearn)

❖ Desired outcomes: Deal with data preprocessing, dataset split and assessment.

❖ Modules Used:

- $datasets.load\_iris()$: Get the Iris dataset.
- $model\_selection.train\_test\_split$: Divide your data into parts for training and testing.
- $preprocessing.MinMaxScaler$: Scale all the features between 0 and 1.
- PCA: Cut down the number of features by relying on fitted qubits.

Examples include accuracy, precision, recall, etc.

4. Built-in libraries included with Python

- Class definitions and control structures are explained in this chapter (you do not need to import anything).
- Both $QuantumTreeNode$ and $QuantumDecisionTree$ have been written using common Python OOP methods.

Team Requirements

1. Team Roles and Responsibilities

Intern Role	Responsibilities	Skills to Learn or Have
Quantum Coding Intern	Implement and test quantum circuits using PennyLane	Python, PennyLane basics, quantum gates

ML & Data Intern	Preprocess data, apply PCA/scaling, handle input/output pipelines	scikit-learn, NumPy, data cleaning
QML Research Intern	Study and explain the theory behind QDT, assist in model design choices	QML concepts, decision trees, optimization
Testing & Evaluation Intern	Write unit tests, evaluate accuracy/F1, compare QDT with classical models	Metric evaluation, debugging, visualization
Documentation Intern	Maintain clear docs on QDT implementation, explain each function and component	Technical writing, Markdown, Python docstrings
Quantum Cloud Intern (optional)	Help run circuits on real hardware (e.g., via IBM Q, Amazon Bracket)	PennyLane plugins, cloud tools

2. Minimum Skills Required (Entry-Level)

- Knowledge of Python programming (from basic to intermediate).
- Basic knowledge of quantum computing (superposition, entanglement).
- Having some experience with NumPy and the ability to pick up ML fast.
- Being willing to review documents, fix errors and practice new things.

3. Development Goals for Interns

➢ Interns should:

- Develop an entire QDT pipeline that is fully operational.
- Look at the differences between quantum and classical models.
- Grasp how quantum circuits allow for non-linear dividing of the circuit.
- Join forces in creating and reporting on your findings.

4. Suggested Team Size and Workflow

- 3–5 interns is ideal for a short-term project .
- Weekly stand-ups or sync meetings.
- Code reviews and paired programming to ensure learning.

REFERENCES AND RESOURCES

Following is a prepared list of References and Resources to assist interns during their work on the QDT project. These are sorted according to topics and how they are applied in real life.

Foundational Reading

❖ **A Beginner's Guide to Quantum Computing**

- Learning in Quantum Country is interactive and helps you remember.
- "Quantum Computation and Quantum Information" is the book everyone uses in this field.
- IBM Quantum's "Learn Quantum": https://quantumcomputing.ibm.com/lab/en/docs/iql

❖ **Head to the PennyLane Documentation.**

- The documentation pages are available at https://docs.pennylane.ai.
- Tutorials: https://pennylane.ai/qml/demonstrations.html
- I recommend working with PennyLane, Quantum Classifier and VQC tutorials.

Machine Learning & Classical Counterparts

- Visit the Scikit-learn User Guide page here: https://scikit-learn.org/stable/user_guide.html.
- Aurélien Géron's book "Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow" covers topics such as decision trees, PCA and preprocessing data.

Quantum Machine Learning

- Machine Learning as it relates to quantum mechanics
- Xanadu Blog can be found at https://pennylane.ai/blog/.
  - Used to write readable guideposts on QML theory and practice
- Paper: "Quantum Machine Learning" by Maria Schuld & Francesco Petruccione (to learn in more depth)
- Paper: "Quantum Decision Trees" (should be available only if it exists currently)

Video Lectures & MOOCs

- **MIT Open Course Ware – Quantum Computation**: https://ocw.mit.edu/courses/6-845-quantum-complexity-theory-fall-2010/
- **Qiskit YouTube Channel**: Many beginner-to-advanced quantum videos
- **PennyLane YouTube Channel**: Includes workshops and coding demos

Tools and Environments

- **Google Colab** – For running and sharing Python + PennyLane notebooks
- **Jupyter Notebook** – Local or cloud IDEs for experiment tracking
- **GitHub** – Version control, collaborative code reviews

- **Slack/Discord** – For team communication and Q&A with mentors

LINK TO PAPERS, ARTICLES, VIDEOS (OR) GitHub REPOSITRIES USED FOR RESEARCH

Here are some **key links to papers, articles, videos, and GitHub repositories** that can support interns or researchers studying and building the **Quantum Decision Tree (QDT)**:

Academic Papers & Articles

1. **Quantum Decision Tree Architectures**
   Not an exact match for classical decision trees, but relevant theoretical insights:
   - ["A Quantum Algorithm for Decision Tree Induction" – Arunachalam et al. (2017)]
     https://arxiv.org/abs/1705.10364
   - ["Quantum Algorithms for Decision Problems" – Kerenidis et al.]
     https://arxiv.org/abs/quant-ph/0306073
2. **Quantum Machine Learning General References**
   - ["Quantum Machine Learning" – Maria Schuld et al. (Review Paper)]
     https://arxiv.org/abs/1409.3097
   - ["Quantum-enhanced machine learning" – Biamonte et al. (Nature 2017)]
     https://www.nature.com/articles/nature23474
   -

Documentation & Articles

1. **PennyLane Documentation (Core for This Project)**
   https://docs.pennylane.ai/
2. **PennyLane Quantum Classifier Tutorial**
   https://pennylane.ai/qml/demos/tutorial_qubit_rotation/
3. **Blog Post – "Quantum Decision Trees" (by Xanadu or others)**
   May appear in forums or blogs, e.g.:
   - https://discuss.pennylane.ai/
   - https://medium.com/tag/quantum-computing

Videos and Lectures

1. **PennyLane YouTube Channel** – Demos and workshop recordings
   https://www.youtube.com/c/XanaduAI
2. **Qiskit YouTube Playlist** – Quantum circuit basics (general knowledge)
   https://www.youtube.com/c/qiskit

3. **Quantum Machine Learning Explained** – Simplified videos by educators like
    - Two Minute Papers (YouTube)
    - Brilliant.org's Quantum Courses

GitHub Repositories

1. **PennyLane Examples & Demos**
   🔗 https://github.com/PennyLaneAI/qml
   (Check /demos, e.g., VQC or QML tutorials)
2. **XanaduAI Main PennyLane Repository**
   🔗 https://github.com/PennyLaneAI/pennylane
3. **Quantum Machine Learning Implementations (Community)**
    - Example repo with custom QML models:
      🔗 https://github.com/kris7011/Quantum-Machine-Learning
    - Other examples might be found by searching GitHub: quantum decision tree