Faculty of
Mathematics
and Informatics

VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
MODELLING AND DATA ANALYSIS
MASTER'S STUDY PROGRAMME

Analysis of Quantum Circuits for Classification Tasks

Master's thesis

Prasham Bhuta
prasham.bhuta@mif.stud.vu.lt
Supervisor: Assoc. Prof. Dr. Linas Petkevicius

Vilnius

2024

Summary

In recent years, governments, universities, and private companies have invested heavily in quantum computing. It is hoped that practical quantum technology can be developed that can perform classical computing on quantum computers. As Preskill writes in his articles, quantum research is entering a new frontier, where the simulation of quantum states in classical systems is expected to speed up the solution of practical problems significantly (super-polynomial or exponential).

In this work, an attempt is made to analyze the behavior of some quantum circuits in a simulated quantum-classical system. The experiments generate multiclass data with a fixed number of classes and a variable number of features. Classical and quantum machine learning algorithms are implemented. Both approaches are compared. In addition, it describes how quantum circuits are constructed and how they can be modeled using some popular Python libraries. A study of algorithm timing and circuit complexity related to quantum circuit algorithms is also presented.

Keywords: Classification, Data Science, Machine Learning, Quantum Computing, Quantum Circuits, Quantum Algorithms.

Santrauka

Pastaraisiais metais valstybinės institucijos, universitetai ir privačios įmonės daug investuoja į kvantinius skaičiavimus. Tikimasi, jog gali būti sukurta praktiška kvantinė technologija, kuri galėtų atlikti klasikinius skaičiavimus kvantiniuose kompiuteriuose. Kaip savo straipsniuose rašo Preskillas, kvantiniai tyrimai yra įžengimo į naują ribą, tikimasi, kad kvantinių būsenų modeliavimas klasikinėse sistemose labai (super-polinomiškai arba eksponentiškai) pagreitins praktinių problemų sprendimą.

Šiame darbe bandoma išanalizuoti kai kurių kvantinių grandinių veikimą imituojamoje kvantinėje-klasikinėje sistemoje. Eksperimentuose generuojami daugiaklasiai duomenys su fiksuotu klasių skaičiumi ir kintančiu požymių skaičiumi. Realizuojami klasikiniai ir kvantiniai mašininio mokymo algoritmai. Abu priėjimai palyginami. Be to, aprašoma, kaip konstruojamos kvantinės grandinės ir kaip jas galima modeliuoti naudojant kai kurias populiarias Python bibliotekas. Taip pat pateikiamas su kvantinių grandinių algoritmais susijusęs algoritmo laiko ir grandinės sudėtingumo tyrimas.

Raktažodžiai: Klasifikavimas, duomenų mokslas, mašininis mokymasis, kvantiniai skaičiavimai, kvantinės grandinės, kvantiniai algoritmai.

Contents

Introduction **7**

1 Notations & Definitions **9**
1.1 Quantum Algorithms . . . 10
1.1.1 Grover's algorithm . . . 10
1.1.2 Shor's algorithm . . . 10

2 Quantum Classification Algorithms **11**
2.1 Classical Algorithms . . . 11
2.2 Basic Quantum Functions . . . 14
2.2.1 Feature Map . . . 14
2.2.2 Ansatz . . . 15
2.3 Quantum Neural Networks . . . 16
2.4 Variational Quantum Circuits . . . 16
2.5 Quantum Kernel Methods . . . 17
2.6 Quantum Convolutional Neural Network . . . 18

3 Related Works & Frameworks **20**
3.1 Qiskit Library . . . 20

4 Methods **22**
4.1 Classification . . . 22
4.1.1 Classical Statistical Classification . . . 22
4.1.2 Quantum Based Classification . . . 23

5 Experiments **26**
5.1 Data . . . 27
5.2 Model Training . . . 28

Results **30**

Conclusions **36**

List of Figures

1 Figure shows a single qubit quantum circuit with a Hadamard gate, and input & weight parameters. 9

2 Figure represents how k-Nearest Neighbors works with data plotted in two dimensions, and class assigned based on classes of 'k' nearest neighbors [9]. . . . 11

3 Figure represents a simplified instance of random forest classification [9]. n random forests are constructed and the popular class decision from each forest is used to decide the final class. 12

4 Figure represents the concept behind boosting [2]. Multiple learners are generated each improving the classification based on previous learners, which are called weak learners. 13

5 Figure shows a kernel-based, in this case, RBF-based, support vector classifier (SVM) that can create complex decision boundaries for linearly non-separable data. 13

6 Figure shows the schematic of a quantum neural network (QNN) structure, with feature map, ansatz and measurement gate components. [26] 14

7 Figure shows that in original space (left) decision boundary is not possible, but mapping to a higher dimension (right) makes it possible to have a decision boundary [31]. 15

8 Figure shows the schematic of ansatz describing the Unitary $U(\theta)$ as a product of L unitaries $U_l(\theta_l)$ acting on the input state 15

9 Figure illustrates the design of Quantum Neural Network (QNN), where the first layer acts for input coding, the next entangled layers are for parameter training, and with a final measurement output layer. 16

10 Figure shows a schematic representation of the Variational Quantum Algorithm (VQA) process from Cerezo et al., [10] article. The input to the VQA is the cost function $C(\theta)$, the ansatz, and the input vector. 17

11 Figure shows the variety of applications and uses of variational quantum algorithms (VQA) as stated by Cerezo et al., [10]. 17

12 Figure illustrates the Convolutional Neural Network (CNN) and Quantum Convolutional Neural Network (QCNN); similar to CNN, a sequence of learning layers of quantum gates is applied to QCNN as well. 18

13 Figure illustrates a parameterized two-qubit unitary circuit with unitary gate as in equation 7 3. 19

14 Figure illustrates a constructed pooling layer for a two-qubit circuit, as per the details in the Figure 13 19

15 Figure shows support vector classifier with an 'RBF' kernel. Training with kernel functions results in complex decision boundaries, making class separation possible. 23

16 Figure shows the hybrid training using QVC, for an input vector $\vec{x}$, the QVC is used to calculate the output and the gradient. The probabilistic measurement of the first qubit q_0 is calculated as λ_1 24
17 Figure highlights the difference between two approaches of variational quantum classifier, and quantum support vector classifier. In the QSVC example, a kernel is used to transform the data before passing it for training. 25
18 Figure shows the two-dimensional scatter plot of the generated data, with two features $X1$, *and* $X2$, and several transformed labels $[0, 1, ..7]$ 26
19 Figure shows the 3D dimensional representation of the generated data with transformed classes $[7, 6, 1]$. 27
20 Figure plots the original targets and predicted values for seven (7) observations, after transforming the predictions to the original multiclass shape. 30
21 Figure visualizes the performance, F1 score, of the classification models, with varying numbers of input features. 32
22 Figure visualizes the performance, mean F1 Score with error bars, of the classification models, with varying numbers of classes. 33
23 Figure visualizes the training time, in log-seconds scale, of the classification models, with varying numbers of input features. 33
24 Figure visualizes the variational quantum classifier (QVC) performance, Mean F1 score, with increasing pairs of $feature\_map$ and $ansatz$ repetitions. 34

List of Tables

1 Table represents the generated data (X, Y) and transformed labels YT. 26
2 Table shows the parameters used for classification and quantum circuit experimentation. 29
3 Table shows the parameters used for classification problem complexity experimentation. 29
4 Table shows the performance, Mean F1 score, for all the experiment models (quantum & classical) for varying number of features. 31
5 Table shows the performance, mean F1 score, for all the experiment models (quantum & classical) for varying number of classes. 31
6 Table shows the computation time, in *seconds*, for all the experiment models (quantum & classical) for varying number of features. 31
7 Table shows the variational quantum classifier (VQC) performance, F1 score, for varying numbers of feature map and ansatz repetitions, i.e. varying circuit complexity. 31

List of Algorithms

1 Classical training 28
2 Quantum training 28

Introduction

Data science is the art of extracting information and knowledge from data. Data science is applied to understand customer relationships or behavior and to calculate customer value; it is further used in finance to do credit scoring, calculate risks, fraud detection, and workplace management. Retailers use it to target advertisements specific to a user, to recommend products, and to manage their supply chain [24]. Neural networks have led to breakthroughs in data science and machine learning. Tasks of computer vision, and natural language processing, form a majority of areas where active research is done and new fronts are explored [15]. Famous algorithms such as Shor's algorithm [34], and Grover's algorithm [16] show the benefits of quantum states to solve classically-hard problems. We will briefly describe the algorithms while reviewing the basic principles behind quantum circuits in Chapter 1.

In his paper, Biamonte et al., [5] describe the principle of applying quantum algorithms to problems that are computationally complex to solve in classical systems. A quantum system that uses quantum algorithms could either speed up solving speeds as in the case [34] [16], or could lead to better performance metrics. The advantage of quantum computing comes from quantum algorithms that perform tasks, which were beyond the scope of classical computers. The most famous example is Shor's algorithm [34] for finding the prime factors of an integer. On a quantum computer, Shor's algorithm runs in a polynomial time of scale $log(N)$, where N is the size of the input integer. In comparison, the best classical method of finding prime factors of an integer runs with a complexity of $exp(N)$ [38]. In 2019, a team of IBM quantum computing was able to factor integer 35, on an IBM Quantum System One computer [1]. Attempts have been made to factorize even larger numbers on a quantum computer with interesting results [18].

In his paper Preskill (2018) [23] mentions three advantages of quantum computing over classical computers. To solve classically intractable and hard problems such as finding prime factors of large integers, sampling from a correlated probability distribution at any time, and no classical algorithm can simulate a quantum computer. Though all that way is done in a pure quantum computing state, progress has also been made in the field of classically simulated quantum algorithms. Within just a few years from 2018, using Variational Quantum Circuits (VQA) [10], Quantum Kernel Algorithms [17] [29], Quantum Neural Networks (QNN) [37], or Quantum Convolutional Neural Networks (QCNN) [11] a quantum circuit can be trained with a combination of classical optimizers. VQAs, Quantum Kernels, & QNNs have emerged as a leading way to address the quantum constraints of a limited number of qubits, and noise in quantum circuits that limit circuit depth. VQAs, QNNs are the quantum equivalent of machine learning methods like neural networks [10], and serve as a base for all quantum-classical computing. The underlying formulation and principles behind each of the algorithms are explained further in Chapter 2.

In this study, Chapter 4 refers to the construct behind the models in detail, while Chapter 5 will help us understand the basics of the experiments. Various methods of gate-complexity and data-complexity are applied to make the experiments equal for both systems, which are

explained and detailed in the results (Chapter 5.2) and conclusions (Chapter 5.2) part of this study.

The aim of this study is to determine the advantages, disadvantages, and contrasts between quantum algorithms and classical machine learning. For that, we set up a classification task and perform them under a similar construct for classic machine learning classification models and quantum algorithm models, from now on called quantum models.

Objectives

1. To perform systematic exploration of the development and underlying principles governing the formation of quantum circuits.
2. To provide a detailed breakdown of some of the quantum circuit architectures, and to showcase the use of current technologies that are used to initialize and train quantum architectures.
3. To perform an investigation on quantum performance with varying quantum circuit complexities, and classification problem complexities.
4. To perform machine learning task of classification with quantum circuits, and compare their performance with classical statistical methods.

1 Notations & Definitions

This section aims to cover the basic notations and definitions used throughout this study. It is provided to lay an understanding of terms, and keywords behind the science of quantum computing, and machine learning. Kopczyk [19], Beer [4], and Phalak et al., [22] provide the necessary knowledge required to understand the basic concepts of quantum theory in their respective texts.

Qubit: Qubits are the fundamental units behind quantum computing. Unlike classical bits, qubits can represent both states 0, and 1 simultaneously. A qubit with states 0, and 1, can be represented by a two-dimensional vector such as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α, β are the coefficients of the basic states of the qubit.

Quantum gates: Quantum gates are analogical to the classical logic gates in classical computing. The quantum gates are applied on qubit states. As mentioned above the vector representation of a single-qubit is $|\psi\rangle$, now quantum gates are generated by applying a unitary gate U such that $U|\psi_1\rangle = |\psi_2\rangle$, where $|\psi_2\rangle$ is the new quantum state. Pauli-X, Pauli-Y, Pauli-Z, CNOT, and Hadamard are typically some of the quantum gates, though there are infinitely many more.

Quantum circuit: Quantum circuits are a collection of quantum gates connected and working together. The first step is the initialization of qubits, then the initialization of appropriate quantum gates to change the qubits to the required state. A quantum circuit is an envelope of such gates, for output a measurement gate is constructed which converts the quantum state into a classical state.

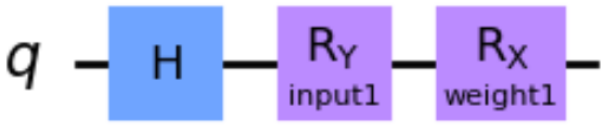

Figure 1: Figure shows a single qubit quantum circuit with a Hadamard gate, and input & weight parameters.

Quantum Superposition & Entanglement: Though this study doesn't detail the physics behind quantum computing, it is necessary to have some understanding of superposition and entanglement to understand the contrast between classical computing and quantum computing.

Now the state of the qubit is represented by vectors in Hilbert space, and if the basis state is $|0\rangle$, *and* $|1\rangle$ then the principle of superposition means that a qubit can be in both the states simultaneously. When two or more qubits are used to interact with each other somehow, then

it is not possible to know the state of one qubit without knowing the state of the other, this phenomenon is quantum entanglement. Superposition and entanglement are the core concepts behind quantum physics.

1.1 Quantum Algorithms

The strength of quantum computing has been showcased by many algorithms such as Grover's algorithm [16], Shor's algorithm [34], and many more. In the few paragraphs below, we provide a brief introduction to such quantum algorithms.

1.1.1 Grover's algorithm

Grover's algorithm [16] is a quantum search algorithm that was devised by Lov Grover in 1996. In an unsorted database, to find a value with a probability of $\frac{1}{2}$ the classical computing algorithm has a complexity of $\frac{N}{2}$ where N is the number of entries in the database. By Grover's algorithm, the problem can be solved using a complexity of $\frac{\sqrt{N}}{2}$. At its core, Grover's algorithm uses quantum states that are in superposition and multiple entries can be examined simultaneously.

As input for Grover's algorithm, suppose we have a function $f : 0, 1, ..., N - 1 - > 0, 1$. In the "unstructured database" analogy, the domain represents indices to a database, and $f(x) = 1$ if and only if the data that x points to satisfies the search criterion. We additionally assume that only one index satisfies $f(x) = 1$, and we call this index ω. Our goal is to identify ω.

1.1.2 Shor's algorithm

Shor's algorithm [34] is a quantum algorithm for finding the prime factors of an integer, Peter Shor developed it. Factoring an integer is considered to be a hard problem for classical computers, and as such is the basis for many cryptography schemes such as the RSA cryptosystem. The RSA system is widely used today to secure data transmission. Shor's algorithm allows finding a factor of a number N in computational complexity of polynomial $\log(N)$, where classical factoring algorithms are in the order of exponential $O(e^{\log(N)})$, thus showcasing the polynomial speed behind quantum computing. Such algorithms threaten the robustness of cryptography and other data-securing methods.

2 Quantum Classification Algorithms

In statistics, classification is the problem of identifying which class an observation belongs to. For example, an email is "$spam$" or "$notspam$", here "$spam$" and "$notspam$" are classes. This example is called binary classification as there are only two classes an observation can belong to. If there number of classes is higher than two, then such problems are called mutli-class classification [3].

Multilabel classification is the generalization of multi-class classification by classifying the observation into more than one mutually exclusive class. Thus in multilabel classification, there is no limit to how many classes an observation can belong to. For example, when classifying a news article, it can be classified as one of "$politics$", "$football$" or can be classified as both "$politics$", and "$football$" [32].

2.1 Classical Algorithms

The most common approach is that of probabilistic classification, where, the probability of an observation belonging to each class is calculated. Such an approach provides the confidence associated with the predictions. Some of the most popular classification algorithms are decision tree learning like the random forest, K-nearest neighbor (KNN), support vector machines (SVM), linear classifiers, and quadratic classifiers, like linear or quadratic discriminant analysis, and many more. In this section, we briefly introduce a few of them that are relevant to this study.

K Neighbors Classifier: As developed by Cover et al., [12] in their article, the nearest neighbor decision rule assigns an unclassified observation to a class based on the set of previously classified points. The class is assigned based on the popularity of the class most common among the k neighbors. To find the neighbors, a distance metric is used, in most cases it is Euclidean distance, but other distance metrics are Manhattan distance, Minkowski distance, and more. Figure 2 shows how a simple kNN with data plotted in two dimensions.

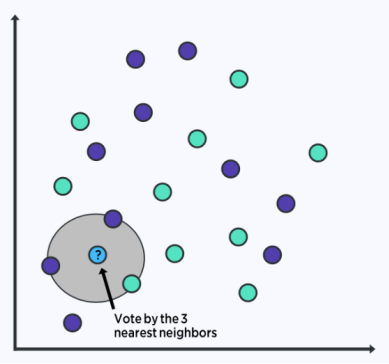

Figure 2: Figure represents how k-Nearest Neighbors works with data plotted in two dimensions, and class assigned based on classes of 'k' nearest neighbors [9].

Quadratic Discriminant Analysis: In statistics a linear classifier creates a linear decision boundary, while a quadratic classifier uses a quadratic decision boundary to separate measurement between classes. That means the separation boundary could be circular, elliptical, parabolic, or hyperbolic. This technique assumes that data has a Gaussian distribution, and there is no assumption about the covariance of each class [35]. Discriminant analysis is derived from the formula based on Bayes' rule, for an observation $(x) \in R^d$, given by,

$$P(y = k|x) = \frac{P(x|y = k)P(y = k)}{P(x)} \tag{1}$$

where k is the class of observation.

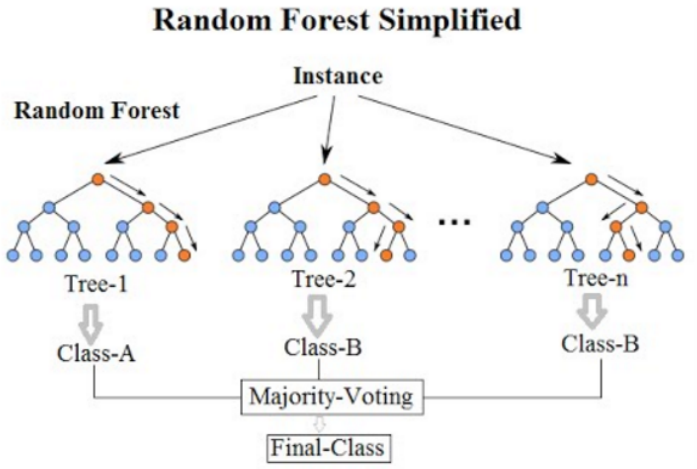

Figure 3: Figure represents a simplified instance of random forest classification [9]. n random forests are constructed and the popular class decision from each forest is used to decide the final class.

Random Forest Classifier: Random decision forests are a method of constructing decision forests, and in classification, the output of the random forest is the class selected by the most trees. A random forest is made up of tree-structured classifiers such as $\{h(x, \Omega_k),\ (k = 1, ...)\}$, where Ω_k are independently identically distributed random tree vectors and each tree outputs a vote for the most popular class at for observation x [7]. Figure 3 showcases a simplified instance of a random forest classifier.

Gradient Boosting Machine: Boosting is a method of iterative learning by using knowledge of the classifier from the previous iteration and re-weighting the observations to iteratively develop a stronger classifier. Figure 4 shows the iterative approach behind the boosting algorithms. Gradient boosting algorithm developed by Friedman et al., [14], views the boosting algorithms as iterative functional gradient descent algorithms, i.e. optimization of a cost function, by applying a steep descent to the minimization problem [2].

Figure 4: Figure represents the concept behind boosting [2]. Multiple learners are generated each improving the classification based on previous learners, which are called weak learners.

Support Vector Machines (SVM) with Kernel: A support vector machine (SVM) creates a separation plane which creates a good separation by the largest distance between observations of different classes. When a problem is stated in an original finite-dimension space, it may same that there is no linear space that can separate the classes. SVM with kernel [6], performs a kernel trick where the original finite-dimensional data is mapped to a higher dimension to create a clear, and concise decision boundary. The kernel function $k(x, y)$ is selected to fit the problem. The kernel function can be linear, polynomial, sigmoid, or any other type; as long as the computed function can create a separation between classes. Figure 5 shows a kernel-based SVM classifier that can create a decision boundary for complex, linearly non-separable data.

Since analyzing classical machine learning algorithms is not a part of this study, we will not delve much into the mathematics behind each of the classification models.

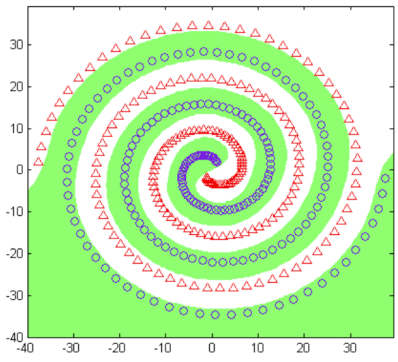

Figure 5: Figure shows a kernel-based, in this case, RBF-based, support vector classifier (SVM) that can create complex decision boundaries for linearly non-separable data.

2.2 Basic Quantum Functions

As mentioned earlier classical neural networks are based on interconnected nodes or neurons, organized in a layered structure. The motivation behind Quantum Machine Learning (QML) is to merge quantum computing and classical machine learning. Quantum Neural Networks (QNN) lie at a merging of these two fields and can be described from both perspectives [26]. From the machine learning side, QNNs are algorithmic models that need to be trained to find patterns in data. From the quantum side, the QNNs are made up of a Quantum Circuit that contains a feature map and an ansatz, as well as quantum gates. Figure 6 represents the basic structure of a QNN.

In the figure, you can see the QNN is composed of feature maps & ansatz. To better understand quantum circuits, we define and discuss feature maps, and ansatz in the following passages.

2.2.1 Feature Map

In machine learning, kernel methods are a well-established field used to embed data into a higher-dimensional feature space to make it easier to analyze [31] [21]. For example, a support vector machine (SVM) draws a decision boundary between classes by mapping the data into a feature space. In quantum computing the input data is transformed to a higher-dimensional Hilbert space through manipulation. Feature map creates the data loader state to feed the input data into a quantum state.

Figure 7 represents the basic purpose of the feature map, which is to transform the data into new space. Formally, if χ is an input set, then a feature map is a map $\phi : \chi \to F$ from input to the vector in Hilbert space. The vector $\phi(x)$ for all $x \in \chi$ are called feature vectors. F is a vector space which is also called the Hilbert space and the feature vectors are quantum states. The map transform $x \to |\phi(x) >$ is performed by a unitary transformation $U_\phi(x)$ which is a part of the quantum variational circuit explained further in the document.

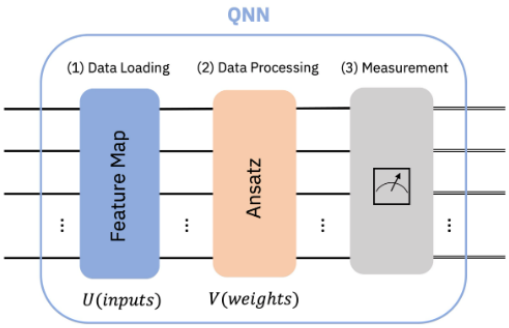

Figure 6: Figure shows the schematic of a quantum neural network (QNN) structure, with feature map, ansatz and measurement gate components. [26]

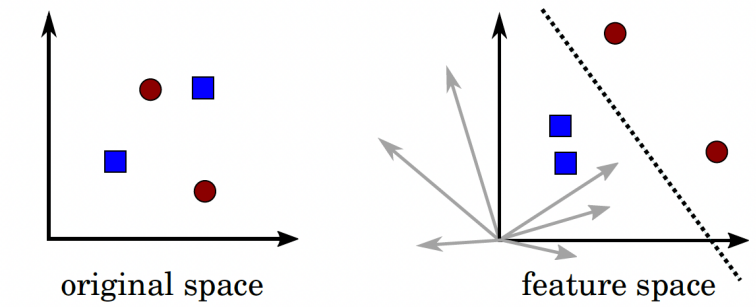

Figure 7: Figure shows that in original space (left) decision boundary is not possible, but mapping to a higher dimension (right) makes it possible to have a decision boundary [31].

2.2.2 Ansatz

A parameterized quantum circuit is passed to the featured data, which has a set of tunable weights for training. This parameterized quantum circuit has many names such as parameterized trial state, variational form, or ansatz; where *ansatz* is the most commonly used name [27].

Ansatz means "initial placement of a tool at a workpiece", and is initialized with an educated guess or initial assumption to solve a problem. It is the starting assumptions of the parameter θ and how they can be trained to minimize the cost. Ansatz is usually problem-specific, but some ansatz architectures are problem-agnostic and ansatz can be used generically [10].

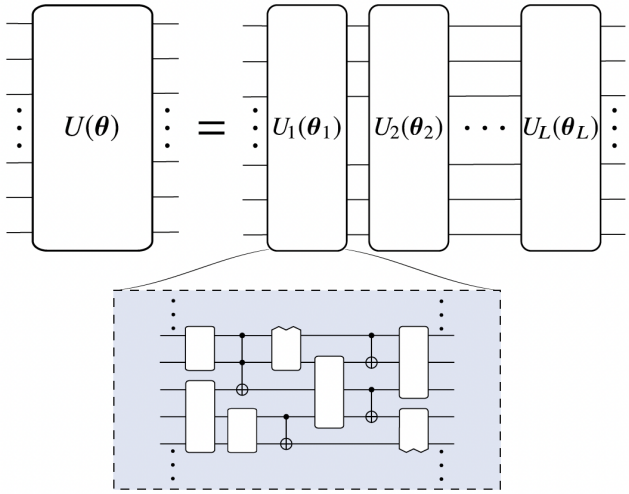

Figure 8: Figure shows the schematic of ansatz describing the Unitary $U(\theta)$ as a product of L unitaries $U_l(\theta_l)$ acting on the input state

The unitary $U(\theta)$ is generally expressed as a product of L sequentially applied unitaries

$$U(\theta) = U_L(\theta_L)...U_2(\theta_2)U_1(\theta_1) \tag{2}$$

2.3 Quantum Neural Networks

Classical neural networks are networks of interconnected nodes (called neurons) in a layered structure. Quantum Neural Network (QNN) integrates quantum circuits, to form layered structures and classical machine learning for different optimizing methods. QNN combines the classical neural network with parameterized quantum circuits [26] [37], as shown in Figure 9.

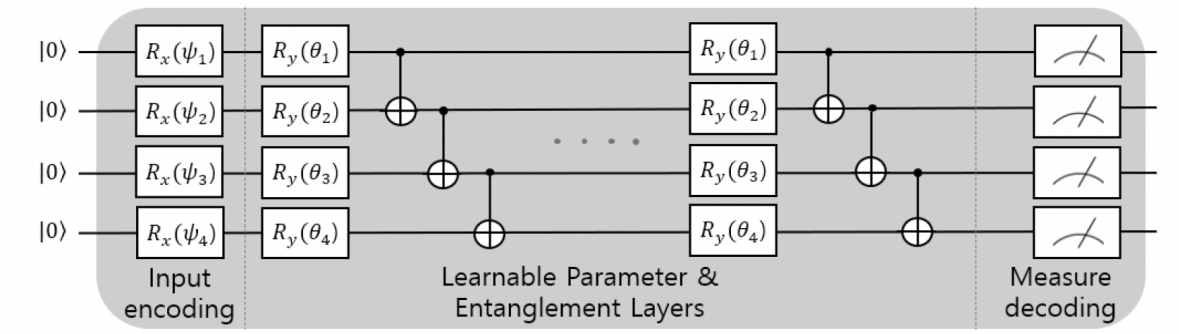

Figure 9: Figure illustrates the design of Quantum Neural Network (QNN), where the first layer acts for input coding, the next entangled layers are for parameter training, and with a final measurement output layer.

QNN works in the following way, first, the input data is encoded and transformed using a feature map and fed into the corresponding layers of the quantum circuit with unitary operations (ansatz). The output of the quantum circuit is measured through the output of a quantum gate. One of the ways to implement a QNN for problem-solving is through the use of variational quantum circuits (VQC) as stated by Beer [4].

2.4 Variational Quantum Circuits

The introduction of Variational Quantum Circuits (VQC) [10] has made a lot of advancements in quantum computations. The algorithm used by VQC is called Variational Quantum Algorithm (VQA) which is a quantum-classical algorithm where the classical computer often performs the parameter optimization. VQA is an extension of quantum circuits that uses a classical co-part to train parameterized quantum algorithms [30]. VQC is one such algorithm. VQC is similar to artificial neural networks in that it estimates the function through parameter learning. In artificial neural networks, we use activation functions to pass information between the layers while VQC uses entanglement layers to create a multilayer structure [10].

The first step with VQA is to define the cost (loss) function C, which is constructed as,

$$C(\theta) \;=\; f(\{\rho_k\}, \{O_k\}, U(\theta)) \tag{3}$$

where f is some function, ρ_k is the input states, O_k is set of observables, and $U(\theta)$ is unitary function.

$$\theta^* = argmin_\theta \mathcal{C}(\theta) \tag{4}$$

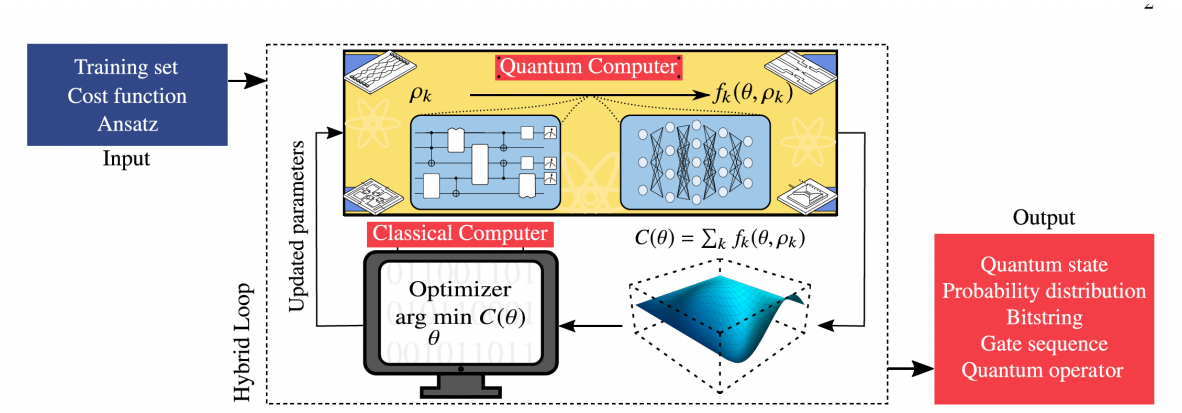

Figure 10: Figure shows a schematic representation of the Variational Quantum Algorithm (VQA) process from Cerezo et al., [10] article. The input to the VQA is the cost function $C(\theta)$, the ansatz, and the input vector.

At each loop, VQA uses quantum computing to estimate the cost function but uses classical computing to train the parameters θ. The cost function defines a hyper-surface, called the cost-landscape, and the optimizer navigates through this landscape to find the global minima. NISQ devices are error-prone and have limited qubit counts, therefore it is necessary to have efficient VQA construction. Figure 11 displays the various use cases of Variational Quantum Algorithms (VQAs).

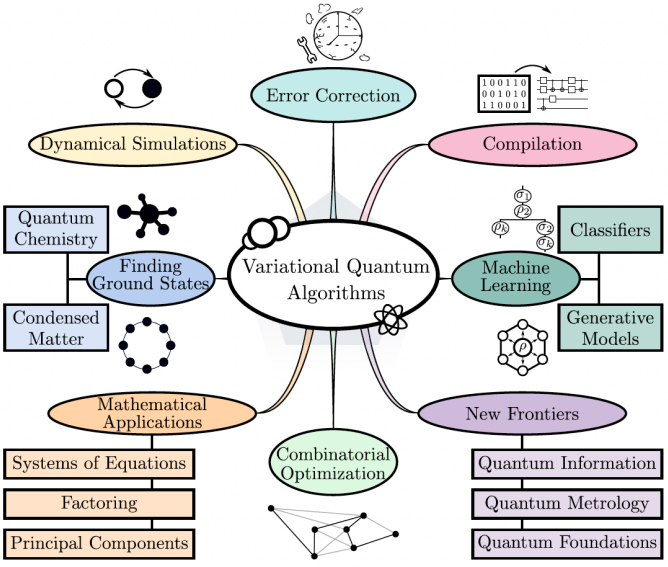

Figure 11: Figure shows the variety of applications and uses of variational quantum algorithms (VQA) as stated by Cerezo et al., [10].

2.5 Quantum Kernel Methods

Kernel methods are used in machine learning for pattern recognition, and the support vector machines (SVM) method leverages the use of kernel functions. In SVM, the data

gets mapped to a higher dimension space called feature space, where a decision boundary is constructed to separate the labeled samples [17]. The kernel transformation helps in creating a decision boundary which could be linear, polynomial, exponential or any complex function. As per Havlicek, a quantum version of this approach was proposed by Rebentrost, in his paper "Quantum support vector machine for big data classification" [29], but the method at that time did not work on hybrid systems, such as VQA, (i.e. quantum-classical methods).

Kernel functions are defined mathematically as, $k(\vec{x_i}, \vec{x_j}) = \langle f(\vec{x_i}), f(\vec{x_j}) \rangle$, where k is the kernel function, $(\vec{x_i}, \vec{x_j})$ are n dimensional inputs, f is the map to higher dimension, and $\langle a, b \rangle$ denotes the inner product [13]. Quantum kernel methods leverage the quantum feature map to perform kernelisation. In such cases, the quantum kernel is created by mapping the input vectors $\vec{x_i}, \vec{x_j}$ to a Hilbert space using quantum feature map $\phi(\vec{x})$. Mathematically the generated quantum kernel matrix can be represented as:

$$K_{ij} = \left| \langle \phi(\vec{x_i}) | \phi(\vec{x_i}) \rangle \right|^2 \tag{5}$$

where K_{ij} is the kernel matrix, $(\vec{x_i}, \vec{x_j})$ are n dimensional inputs, $\phi(\vec{x})$ is the quantum feature map, and $|\langle a|b \rangle|^2$ denotes the overlap of two quantum states a and b [13].

2.6 Quantum Convolutional Neural Network

Quantum Convolutional Neural Network (QCNN) was first proposed by Cong et al., [11]. Figure 12 from the study illustrates the architecture of QCNN and Convolutional Neural Network (CNN). The first step involves encoding the data into a qubit state with proper operator gates. The convolution layer filters the data from the input channel into the feature map such as $ZFeatureMap$ or $ZZFeatureMap$, pooling layer combines the information

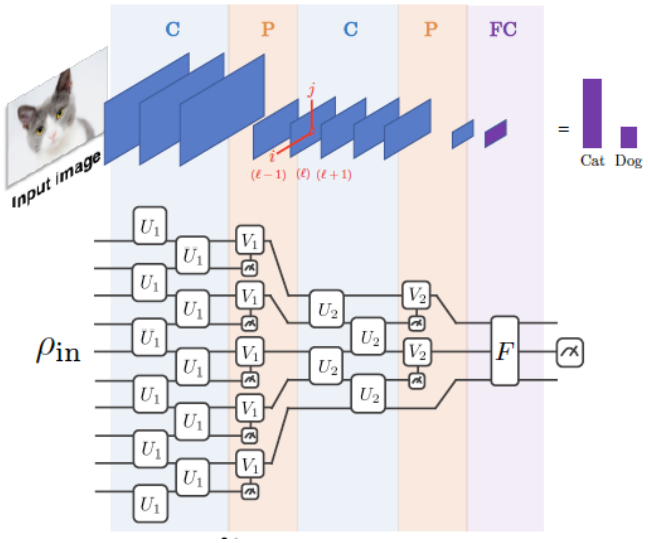

Figure 12: Figure illustrates the Convolutional Neural Network (CNN) and Quantum Convolutional Neural Network (QCNN); similar to CNN, a sequence of learning layers of quantum gates is applied to QCNN as well.

between two qubits and reduces the number of qubits in each layer by half. In the end, the output is calculated by measuring the output of the one remaining qubit.

The convolutional layer consists of two-qubit unitary operators that combine the information between the two qubits, based on the two-qubit unitary as proposed in [36]. The proposed solution with the equation 6, where $\otimes$ is the tensor product and α, β, γ are the parameters that can be adjusted. Tuning of a large number of such parameters would lead to longer training times, the trick is to restrict the ansatz formation to a smaller subspace. Figure 13 illustrates a parameterized two-qubit circuit,

$$U = (A_1 \otimes A_2).(N(\alpha, \beta, \gamma)).(A_3 \otimes A_4) \tag{6}$$

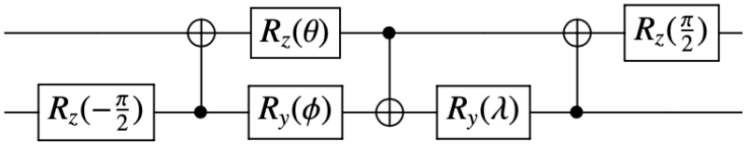

Figure 13: Figure illustrates a parameterized two-qubit unitary circuit with unitary gate as in equation 7 3.

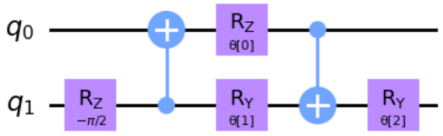

Figure 14: Figure illustrates a constructed pooling layer for a two-qubit circuit, as per the details in the Figure 13

In QCNN implementation, the ansatz is restricted to a particular subspace and defined as $(N(\alpha, \beta, \gamma))$, where $N(\alpha, \beta, \gamma) = exp(i[\alpha\sigma_x^2 + \beta\sigma_y^2 + \gamma\sigma_z^2)$. Thus the final two-qubit unitary equation 7 and structure, in Figure 13, are depicted accordingly. The two-qubit unitary is applied to all even pairs of qubits and then to all odd pairs of qubits. Finally, the first and final qubit is also coupled through the two-qubit unitary gate.

$$U = N(\alpha, \beta, \gamma) = exp(i[\alpha\sigma_x^2 + \beta\sigma_y^2 + \gamma\sigma_z^2) \tag{7}$$

As with CNN, the purpose of the pooling layer is to reduce the dimensionality of the Quantum Circuit. It means to reduce the number of required qubits while keeping most of the information or learning intact. We apply a two-qubit unitary to each pair as discussed earlier to reduce the number of qubits in a quantum circuit. After applying the operation we ignore one qubit from each pair. The unitary operation combines the information between two qubits, and there is no further use of the discarded qubit. Thus at each step of applying the pooling layer, the number of qubits is reduced by half. Figure 14 depicts the schematic for a two-qubit pooling layer. After the application of two-qubit unitary circuits, one of the qubits (q0) is discarded in the future layers.

3 Related Works & Frameworks

In this Chapter, we lay out and describe the libraries, and documentation, currently in public, required for this study. This will help to get familiarized with some of the frameworks, how they work, and how to run quantum circuit algorithms using such a library. For this study, we worked with Python libraries that offered an extension to run or simulate quantum systems with well-written, organized, and continuously updated documentation.

TensorFlow Quantum Library: TensorFlow Quantum (TFQ) is a quantum machine learning library for rapid prototyping of hybrid quantum-classical ML models [8]. It leverages Google's quantum computing frameworks all within Tensorflow. It focuses on quantum data and building hybrid quantum-classical models. It is an integration of logic designed in Cirq and existing TensorFlow APIs along with high-performance quantum circuit simulators. TFQ offers an abstraction for designing and training quantum models under TensorFlow and supports quantum circuit simulators, it offers to simulate quantum systems with 30+ qubits and multiple GPUs.

Pennylane: Pennylane.ai is an open-source framework for quantum computing and quantum machine learning. With Pennylane researchers can connect to quantum hardware using machine learning frameworks, like $Keras$, $PyTorch$, $NumPy$, and others. Researchers can connect with quantum devices like Strawberry Fields, Amazon Braket, IBM Q, Google Cirq, and more.

Others: We provide a list of other similar libraries that are used to run, or simulate quantum circuits. Cirq, an open-source framework from Google; Qibo, an opensource quantum ecosystem with the potential to deploy quantum applications; Amazon Braket, an Amazon framework for quantum computing (connected with Amazon Braket Python SDK); PyQuil, a python quantum framework based on the Quil language that is used to interact with quantum systems.

3.1 Qiskit Library

Qiskit is an open-source programming language used for programming quantum computers, developed and managed by IBM. Qiskit has a production-ready circuit compiler, and quantum circuits can be deployed based on the available list of tutorials. Qiskit can be run on

- Native primitives - services or software packages that provide native support to Qiskit Primitives. Qiskit primitives provide a set of interfaces for performing operations such as sampling, and estimation that form the fundamental building blocks of quantum algorithm development.
- Quantum hardware - cloud services that allow users to execute quantum programs in specialized hardware that leverages quantum mechanical phenomena for quantum computation.

- Local simulators - software packages that allow you to simulate quantum programs in your local computer.
- Cloud simulators or multi-platforms - cloud services that allow to simulation of quantum programs in high-performance computers, or with a combination of cloud services.

In our experiments of this study, we use IBM's qiskit library, to instantiate, deploy, and train quantum circuit models.

4 Methods

This Chapter describes the methods, and algorithms available and used to perform the experiments. The study aims to perform classification using quantum algorithms and to draw a performance comparison between quantum methods, and classical machine learning methods. We separate the methods into the classical side, and quantum side as we have done throughout this study.

4.1 Classification

We have discussed the concept of classification, which is to classify an observation into either one or more than one class. The multiclass classification is a classification task with more than two classes, however, each observation can only be labeled as one class, while, the concept of classifying an observation into more than one class is called multilabel classification.

In this study, we focus on combining the multilabel, and multiclass classification, to conform to our quantum circuit needs and limitations. The generated data and predictions are of multilabel type, but we use a transformation to convert multiple labels into a multiclass classification problem to fit the training models. For example for a $n\_class = 3$, and $n\_labels = 3$, and observation could be such as $[1, 0, 1]$. To convert the problem to multiclass classification we transform the generated data to an integer. For such transformation, we assume the original data to be of a binary $0s$ and $1s$, and convert the binary representation to an integer of the base 10.

For the example of $[1, 0, 1]$, the converted label would be $1 * (2^2) + 0 * (2^1) + 1 * (2^0) = 5$. Thus, for $n\_class = 3$, we would have 8 classes of $[0, 1, .., 7]$. For predictions, the models output a single class, and we transform it back to the array of $0s$, and $1s$, so a prediction of 3 will convert to $[0, 1, 1]$.

Such a step was necessary to reduce quantum complexity and run times by increasing the number of parameters to train, drastically increasing computational time. Additionally, for some of the methods, this could be seen as a case of dimension reduction, i.e. converting a multiple-dimensional array into a single dimension can help speed up the training process.

The downside is the loss of co-variant information, which could be critical for better performance. But at the same time models such as quadratic discriminant analysis do not assume the co-variance between classes, thus its performance should not be affected.

For this study, it doesn't matter much as our aim is not to find the best-performing model but to set up classification tasks for quantum circuits, and check the performance against classical algorithms. Since the setup for both the approaches is same, the relative performance should change. However, in our results and conclusions, we would mention this as a multiclass classification problem and not a multilabel classification problem.

4.1.1 Classical Statistical Classification

We use methods from the *scikit* package of Python, which is a popular framework to run machine learning tasks. For the k-nearest neighbors (kNN) classifier, the number of neighbors

($n\_neighbors$) is five (5), and the distance metric is $Minkowski$ metric with a power factor (p) of two (2). The distance is defined by

$$D(X,Y) \;=\; (\sum_{i=1}^{n} \left|x_i \;-\; y_i\right|^p)^{\frac{1}{p}}$$

, where if $p = 2$ it becomes Euclidean distance. For quadratic discriminant analysis (QDA), a classifier is created that can fit quadratic decision boundary using the Bayes' rule described in equation 1.

Random forest classifier (RFC) is created with a classifier of hundred (100) decision trees, fitted on sub-samples of the data, and uses averaging to improve the accuracy. The default loss function used in $gini$ loss is calculated as $(1 \;-\; \sum_j p_j^2)$, where p_j is the probability of class j. Light gradient boosting machine (LGBM) is created similarly with $n \;=\; 100$ number of boosted trees, and boosting_type as Gradient boosting decision tree. Support vector classifier (SVC) is created with a default radial basis (rbf) kernel. The advantage of the rbf is that it can generate complex decision boundaries, as seen in Figure 15.

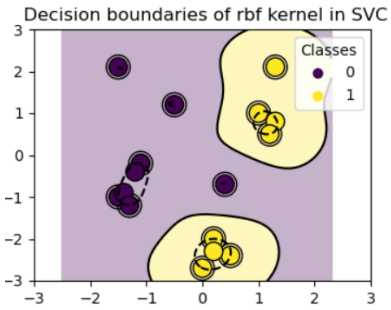

Figure 15: Figure shows support vector classifier with an 'RBF' kernel. Training with kernel functions results in complex decision boundaries, making class separation possible.

Scikit's page [33] on multiclass and multioutput algorithms provides a more detailed breakdown of all the methods and the construction behind each of them.

4.1.2 Quantum Based Classification

In this study, we explore a class of variational quantum algorithms (VQA) namely the variational quantum classifier (VQC), and quantum kernel learning namely the quantum support vector classifier (QSVC) with a quantum kernel. Figure 17 shows the difference between the two methods.

Classification using Quantum Variational Algorithm Variational quantum classifiers (VQA) are parameterized quantum circuits that can be trained with a classical co-processor. Chapter 2.4 describes the algorithm in detail, and the formulation of the method. Figure 16 shows the setup of a hybrid VQA.

With Qiskit, first, a simple one-qubit quantum circuit is created as shown in Figure 1. A Quantum observable, which observes the output of the quantum circuit is created which is

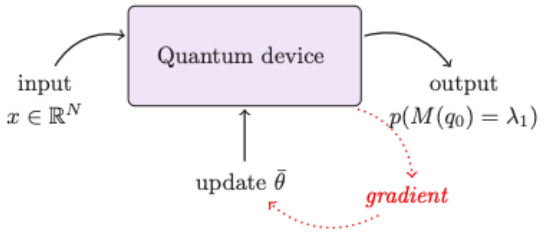

Figure 16: Figure shows the hybrid training using QVC, for an input vector $\vec{x}$, the QVC is used to calculate the output and the gradient. The probabilistic measurement of the first qubit q_0 is calculated as λ_1

based on the number of qubits of the circuit. In our experiments, the Pauli gate is used to calculate the value of the quantum circuit.

Next, a neural network classifier is created based on SamplerQNN or EstimatorQNN, with generally a "cross-entropy" loss or any other kind. The optimizer function is stated, and the standard classical machine learning methods are used to fit the data in the model, and predict and score functions to get the score.

Implementation using qiskit learning, there are different approaches to instantiate QNNs. Firstly the NeuralNetwork class - The abstract class all QNNs inherit from. EstimatorQNN - a QNN with a measurement for the quantum observable at the end. SamplerQNN - a QNN based on the samples resulting from the measurement of the quantum circuit. Both QNN estimators are based on the qiskit primitives. EstimatorQNN has at its base a BaseEstimator class that estimates the expected value of quantum circuits and observables. SamplerQNN has BaseSampler at its base which calculates the quasi-probabilities from quantum circuits [25].

The estimator class is initialized with an empty parameter set and a $JobV1$ job is created with $...Estimator.run()$ method. The method parameters are quantum circuits ($\psi(\theta)$): list of QuantumCircuit objects, observables (H_j): a list of quantum observable (SparsePauliOp) objects, parameter values (θ_k): list of sets of values to be bound to the parameters of the quantum circuits. Calling $...JobV1.result()$ yields a list of expectation values plus optional metadata information, based on the formula:

$$< \psi_i(\theta_k)|H_j|\psi_i(\theta_k) > \tag{8}$$

The sampler class is run the same way with $sampler.run()$ and $JobV1.result()$ commands. The implementation differs in that the Sampler class doesn't require a list of quantum observable (SparsePauliOp) objects. The sampler class calculates the probabilities of bitstrings from the quantum circuits.

For more advanced circuits, based on the given tutorial [25], a Quantum Neural Network (QNN) is taken as input and leveraged for the context of classification. The QNN is constructed from a feature map and ansatz. In our experiments we use a Variational Quantum Classifier (VQC), which is derived from the SamplerQNN, SamplerQNN returns $d - dimensional$ probability vector as output, where d denotes the number of classes. The special variant

VQC applies a mapping or extends to the multiple classification problem. By standard, it applies the Cross-Entropy loss which is discussed further in the section below. We generate a $ZZFeatureMap$ feature map and $RealAmplitudes$ ansatz, with num_qubits taken based on the number of features, and construct a VQC based on them. Alternatively, an Estimator QNN is constructed based on the quantum circuit, and the quantum observable. Estimator QNN takes into account the input parameters R_y and the weight parameter R_x.

Classification using Quantum Kernel Machine Learning A quantum kernel is implemented based on the class FidelityStatevectorKernel, the kernel function is defined as an overlap of two quantum states defined by the feature map which is of type ZZFeatureMap as stated in the equation 5. The feature map is created based on the number of features in the input. The kernel transformation of the input data is performed based on the formula 5, and the model fitting and predictions are done after the kernel transformation.

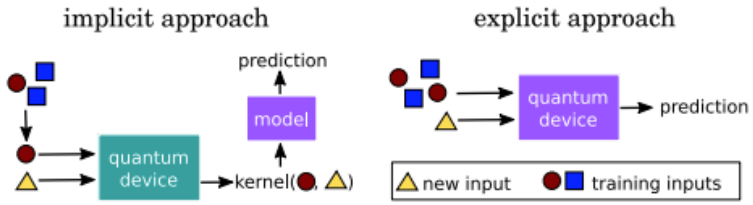

Figure 17: Figure highlights the difference between two approaches of variational quantum classifier, and quantum support vector classifier. In the QSVC example, a kernel is used to transform the data before passing it for training.

5 Experiments

Our goal is to perform classification tasks using classical machine-learning methods and quantum circuit-based algorithms of variational quantum algorithm (VQA), and quantum support vector classifier (QSVC).

X1	X2	X3	X4	Y1	Y2	Y3	YT
15.0	11.0	18.0	7.0	1	1	1	7
14.0	12.0	16.0	5.0	1	1	1	7
14.0	17.0	16.0	6.0	0	0	1	1
14.0	13.0	17.0	6.0	1	1	1	7
25.0	3.0	17.0	5.0	1	0	1	5

Table 1: Table represents the generated data (X, Y) and transformed labels YT.

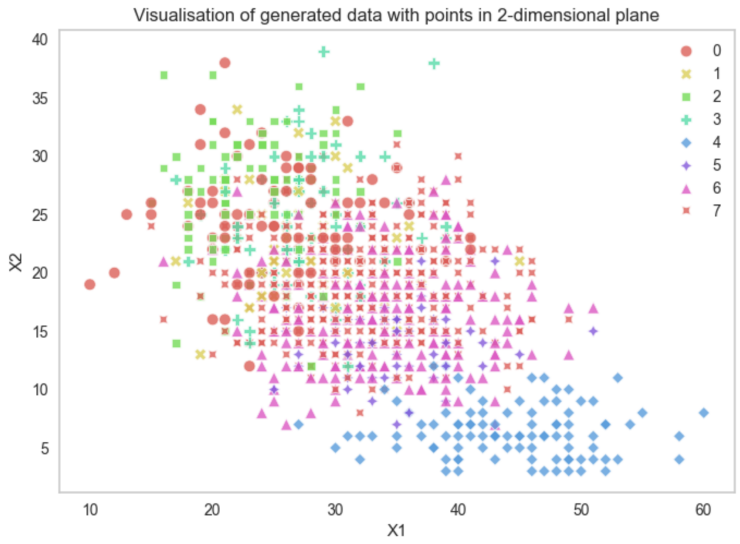

Figure 18: Figure shows the two-dimensional scatter plot of the generated data, with two features $X1$, *and* $X2$, and several transformed labels $[0, 1, ..7]$

For classification with classical machine learning, a *pycaret* training model is set with models ['k Nearest Neighbors (kNN)', 'Random Forest Classifier (RFC)', 'Quadratic Discriminant Analysis (QDA)', 'Light Gradient Boosting Machine (lGBM)', and 'Support Vector Machine with RBF Kernel (SVM)']. All these methods are popular in machine learning tasks, and provide a good performance benchmark to test the quantum algorithm against. $pycaret.classification.ClassificationExperiment$ class initializes the training process and creates the training pipeline.

For classification with quantum methods, our experiments use a Variational Quantum Classifier (VQC), which is a special variant of Quantum Neural Network (QNN) with a $ZZFeatureMap$ & $RealAmplitudes$ ansatz. It uses the Cross-Entropy loss [39] [28]. For binary classification, where the number of classes equals 2, the cross-entropy is calculated as

$$C.E. \ = \ -y(log(p)) + (1-y)log(1-p) \tag{9}$$

For multiclass problems, i.e., the number of classes is >2, cross-entropy is calculated for each class label and the result is the sum of all losses. The formula behind it is, $-\sum y_{o,c} log(p_{o,c})$, where o is observation, p is predicted probability observation o is of class c. It is constructed using the VQC class and passing feature map, ansatz, and optimizer values as attributes for construction.

The optimizer function used is $COBYLA$, which stands for Constrained Optimization By Linear Approximation. It is a gradient-free optimization that uses a linear approximation of a function in the neighborhood of its current value and determines the next optimum point. Now due to this, there is no steep descent to the optima, and thus COBYLA is considered slower but requires fewer weights for training [20].

The library $qiskit\_machine\_learning.algorithms.classifiers$ has the VQC, and QSVC classes required for training the quantum circuits.

5.1 Data

We use the $make\_multilabel\_classification$ class of sklearn to generate multilabel data of varying classes. To transform the data into a single dimension, we use the trick of converting the n-array of 0s, and 1s into integers. The approach is how we convert binary to integers of base 10, for example [1, 0, 1] will convert to $1 * 2^2 \ + \ 0 * 2^1 \ + \ 1 * 2^0$, which is equal to 5. The transformed data into passed to the respective training pipeline of *pycaret* (classical ML training), SVC & $QSVC$ (quantum-based training).

Table 1 shows a sample of the generated 4-dimensional $X1, \ X2, \ X3, \ X4$ feature data, with 3 class labels $Y1, \ Y2, \ Y3$ and the transformed label YT. Figure 18, and Figure 19 show

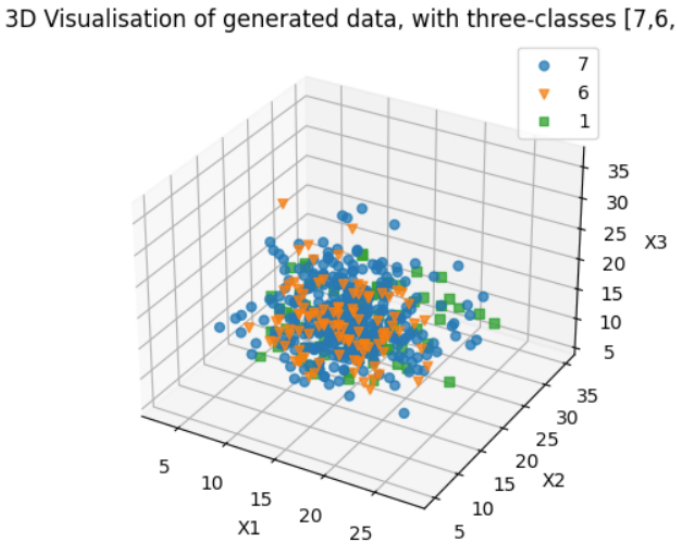

Figure 19: Figure shows the 3D dimensional representation of the generated data with transformed classes $[7, 6, 1]$.

a scatter plot of the generated data, and the transformed classes in two-dimensional (2D) and three-dimensional (3D) planes.

5.2 Model Training

The generated data and the transformed labels are passed to respective model systems for training. *Pycaret* is an open-source machine learning library that simplifies the model training and workflow. The selected models ['knn', 'random forest classifier', 'light gradient boosting machine', 'quadratic discriminant analysis', and 'support vector machine'] are passed as a list to be trained. The hyperparameters, optimization functions, epochs, and regularisation techniques for each model are managed by pycaret pipeline itself.

Algorithm 1 Classical training

Load data from sklearn.datasets.make_multilabel_classification
Setup PyCaret:
 # Specify target variable(Y) and features (X)
 # Transform target variable(Y)
`exp1 = setup(data, target='YT')`
Create and Train Models:
 # Create different classification models
`create_model(models=['knn', 'qda', 'rfc', 'svm', 'lgbm'])`
Make Predictions:
 # Make predictions on new data
`predictions = predict_model(best_model, X)`
 # Transform predictions to original shape
 # Calculate F1 score

Algorithm 2 Quantum training

Load data from sklearn.datasets.make_multilabel_classification
Setup Feature map, Ansatz & Quantum Kernel
Setup QSVC or VQC:
 # Specify target variable(Y) and features (X)
 # Transform target variable(Y) to (YT)
Compare and Train Models:
`QSVC.fit(X,YT) or VQC.fit(X,YT)`
Make Predictions:
 # Make predictions on new data
`predictions = QSVC.predict(X) or VQC.predict(X)`
 # Transform predictions to original shape
 # Calculate F1 score

For quantum model training, a VQC, and $QSVC$ classifiers are instantialized with $COBYLA$ optimizing function, and a max_iter of 50. That means the model is trained for 50 epochs. The loss function for both cases is cross-entropy loss based on the formula 9. Table 2, and Table 3 show the parameters used for experimentation. We run experiments to test the performance as well as the time complexity of the quantum algorithms on clas-

Parameters	Values
No of samples	1024
No of features	[2, 4, 8, 12, 16]
No of classes	3
No of labels	3
No of feature map reps (Quantum)	[2,3,4,5,6,8,12]
No of ansatz reps (Quantum)	[2,3,4,5,6,8,12]

Table 2: Table shows the parameters used for classification and quantum circuit experimentation.

Parameters	Values
No of samples	1024
No of features	[2, 4, 8, 12, 16]
No of classes	[3,4,5,6,8,12]
No of labels	[3,4,5,6,8,12]
No of feature map reps (Quantum)	3
No of ansatz reps (Quantum)	3

Table 3: Table shows the parameters used for classification problem complexity experimentation.

sically simulated systems. Algorithm 1, and Algorithm 2 show the algorithms used in the experimentation.

To check for quantum performance with increasing quantum complexity, we set up a variational quantum classifier (VQC) with different pairs of feature maps and ansatz repetitions. Feature maps and ansatz repetitions increase the number of entanglement layers among qubit layers. Increasing the number of repeated circuits increases the depth of the circuit, which will change the transfer of information between qubits. In our experiments, we try different pairs of the feature map and ansatz repetitions. The values are $[(2,2),\ (3,3),\ (4,4),\ (5,5),\ (6,6),\ (8,8),\ (12,12)]$ for $feature\_map$, and $ansatz$ repetitions respectively.

To measure the quality of the classification tasks, the metric used is the F1 score. The F1 score, is a harmonic mean of precision and recall, ranging from the best value of 1 to the worst value of 0. It is given by the formula,

$$F_1 \ = \ 2\frac{precision\ .\ recall}{precision\ +\ recall}$$

In the multiclass classification case, the final score is obtained by macro-weighted averaging (based on class frequency). Particularly F1 score is useful for class imbalance when a certain class has very few observations. Such cases do occur in the case of multiclass classification and in our study we use a 'weighted' average, weighted based on the number of instances for each label. This helps account for label imbalance.

Results

In this Chapter, we present the key findings of our experiments, aiming to address the objectives defined in the introduction chapter. One of the objectives, of this study, is to perform classification tasks with quantum circuits and compare their performance with popular classical machine learning methods. We provide a detailed theoretical description of quantum architecture in Chapter 2, while the latter chapters of methods (Chapter 4), and experiments (Chapter 5) detail how the circuits were used for the classification task. The parameters used for training, are mentioned in Table 2 and are fixed for all classifier models. We present the results for time complexity and quantum model (QVC) performance for circuit complexity to provide an understanding of the computing complexity involved with quantum circuits as the number of training parameters increases.

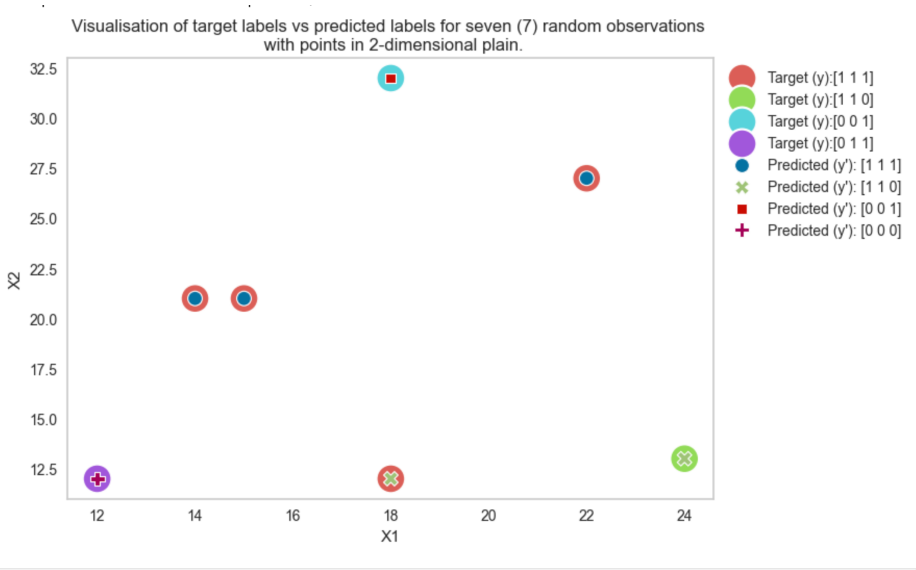

Figure 20: Figure plots the original targets and predicted values for seven (7) observations, after transforming the predictions to the original multiclass shape.

The experiment results in predicting the classes of observation, and the plot 20 shows the distribution of seven (7) observations, highlighted with their original labels, and predicted labels. The labels are transformed as a single integer-value before training and are transformed back to the original shape to measure and quantify performance.

Performance Results

In this section, we present the classification results for the models. As mentioned earlier, we will use the F1 score to evaluate how the models perform, across an increasing number of input features. We argue that increasing the number of input features provides a better chance of having a decision boundary for separation. Table 4 displays the performance score (F1 score) for all the models in the experiments. Figure 21 visualizes the model performance with a changing number of input features.

The performance of the classification task from Figure 21, and Table 4 shows that the

Model Name	No of features (M)				
	2	**4**	**8**	**12**	**16**
VQC	0.199	0.395	0.586	0.589	0.545
QSVC FidelityStateKernel	0.803	0.848	**0.987**	**1.000**	**1.000**
K Neighbors Classifier	0.703	0.736	0.751	0.706	0.797
Light Gradient Boosting Machine	0.820	0.951	0.955	0.962	0.956
Quadratic Discriminant Analysis	0.751	0.839	0.835	0.860	0.886
Random Forest Classifier	**0.821**	**0.972**	0.978	0.998	0.956
SVM - Linear Kernel	0.615	0.692	0.699	0.823	0.6961

Table 4: Table shows the performance, Mean F1 score, for all the experiment models (quantum & classical) for varying number of features.

Model Name	No of classes (N)					
	3	**4**	**5**	**6**	**8**	**12**
VQC	0.502	0.333	0.298	0.232	0.178	0.179
QSVC FidelityStateKernel	0.903	0.899	0.891	0.880	0.817	0.772
K Neighbors Classifier	0.849	0.803	0.760	0.705	0.656	0.564
Light Gradient Boosting Machine	**0.953**	**0.947**	**0.936**	**0.921**	**0.968**	**0.943**
Quadratic Discriminant Analysis	0.888	0.870	0.842	0.773	0.802	0.720
Random Forest Classifier	0.950	0.942	0.933	0.895	0.950	0.936
SVM - Linear Kernel	0.856	0.832	0.810	0.783	0.704	0.507

Table 5: Table shows the performance, mean F1 score, for all the experiment models (quantum & classical) for varying number of classes.

Model Name	No of features (M)				
	2	**4**	**8**	**12**	**16**
VQC	68.729	192.284	732.892	45223.159	131429.982
QSVC FidelityStateKernel	3.096	4.406	6.810	27.492	4174.854
K Neighbors Classifier	2.449	2.025	1.882	1.794	0.139
Light Gradient Boosting Machine	29.447	33.578	38.154	33.527	46.722
Quadratic Discriminant Analysis	0.167	0.169	0.179	0.084	0.130
Random Forest Classifier	0.462	0.511	0.519	0.475	0.468
SVM - Linear Kernel	0.214	0.229	0.241	0.137	0.139

Table 6: Table shows the computation time, in *seconds*, for all the experiment models (quantum & classical) for varying number of features.

Model Name	No of feature map and ansatz repetitions (R)						
	2	**3**	**4**	**5**	**6**	**8**	**12**
VQC	0.296	0.294	0.194	0.213	0.182	0.201	0.191

Table 7: Table shows the variational quantum classifier (VQC) performance, F1 score, for varying numbers of feature map and ansatz repetitions, i.e. varying circuit complexity.

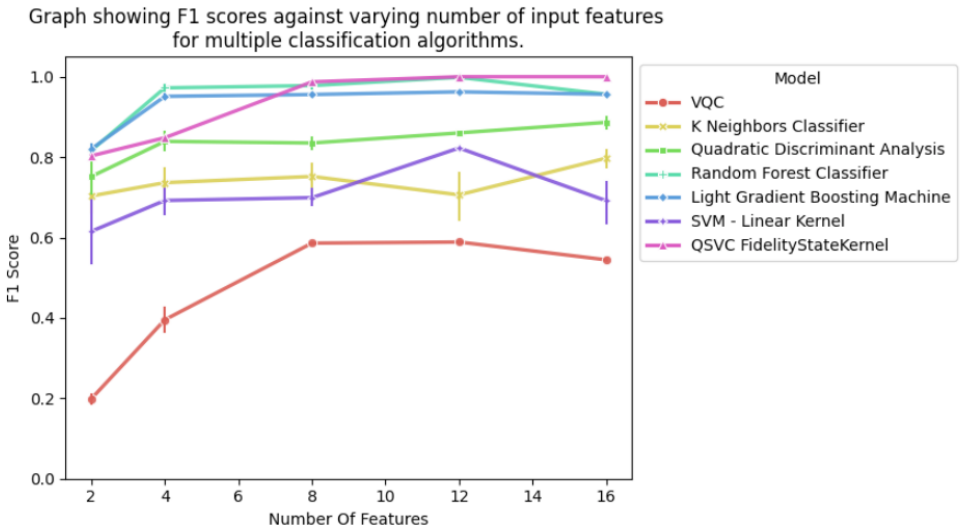

Figure 21: Figure visualizes the performance, F1 score, of the classification models, with varying numbers of input features.

kernel-based quantum support vector classifier (QSVC) performs on par/above to classical methods. For a higher number of input features, it is the best-performing model among both the classical and quantum models, achieving an F1 score of 1.000 for the number of features (12, and 16). The base variant of a quantum circuit classifier(QVC), which is constructed using a $ZZFeatureMap$ and $RealAmplitudes$ ansatz repetitions, is underperforming in terms of F1 score. Without any optimization, or feature engineering trick, it is not a shock, as other models in the experiments implement some sort of optimized training trick to be more precise.

Increasing the number of input features introduces more complexity to training; in the case of quantum circuits, it means more qubits, and more weight parameters to train. This results in an increase in computation time as well, which is presented in the next section.

Classification Problem Complexity Results

In the second part of our experiments, we check for the performance of quantum circuits with varying classification problem complexity. We do that by varying the number of classes, and number of labels. With changes in the number of classes, the data becomes different each time. A higher number of classes, but a lower number of labels leads to class imbalance, as there could be rows with a lot of empty or fewer original labels. It is difficult for a model to learn when there is not much information in the generated data, but when the number of labels increases, the data becomes more complex, and it could be hard for algorithms to train. Table 5 shows the mean model performance and F1 Score for varying numbers of classes. Figure 22 visualizes the F1 Score, with error bars representing the confidence interval for models with a changing number of classes.

It is evident from Figure 22, that the performance drops with an increase in the number of classes, for mostly all models but some of the classical machine learning models perform exceptionally well. The quantum-based QSVC, model with kernel function remains somewhat

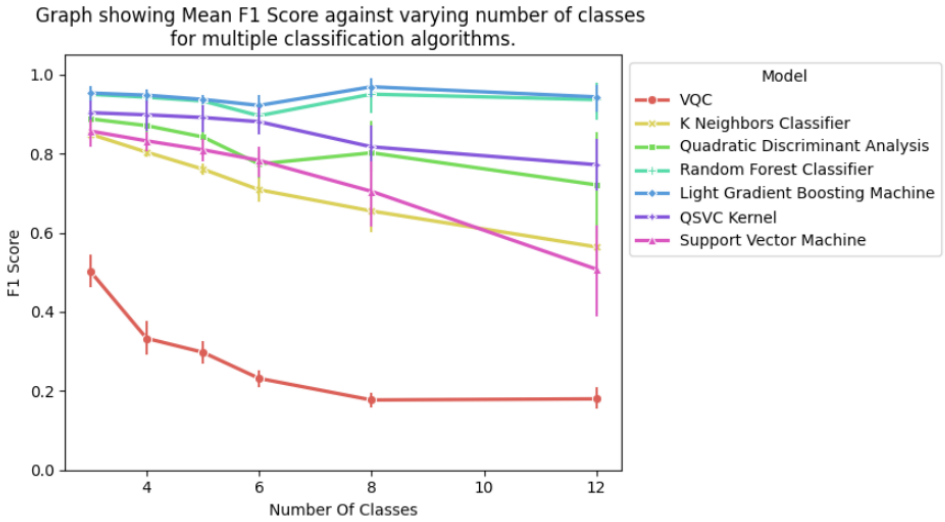

Figure 22: Figure visualizes the performance, mean F1 Score with error bars, of the classification models, with varying numbers of classes.

stable.

Computational Time Results

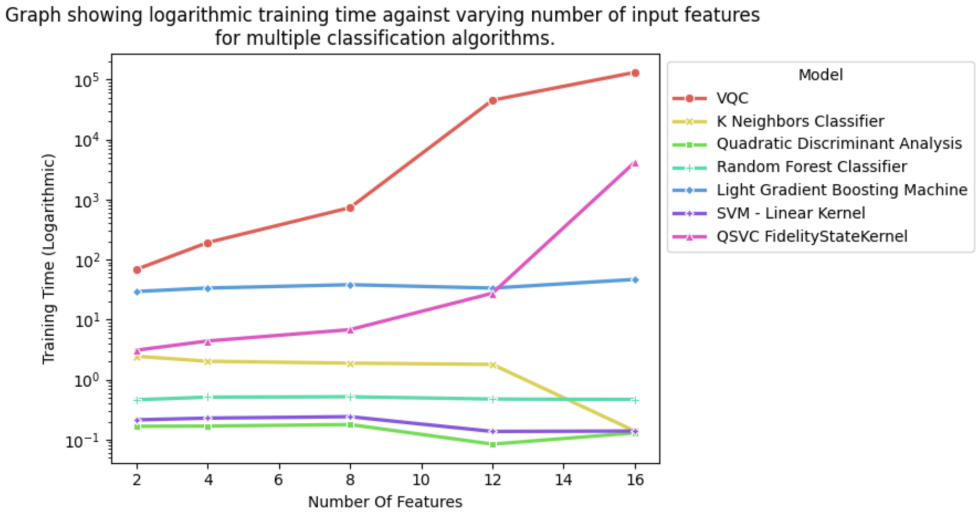

Figure 23: Figure visualizes the training time, in log-seconds scale, of the classification models, with varying numbers of input features.

In our study, we also want to check for the computational time of quantum algorithms and plot them with faster machine-learning models. For classical training, *pycaret* training models are accelerated highly to return faster-trained models. Table 6 shows the computational time for the model training, in seconds. Figure 23 visualizes the training time for models, in logarithmic-seconds scale, with a changing number of input features.

It is evident from Figure 23, that the training time for the quantum circuits is exponentially

increasing with an increase in the number of input features. We argue it is the case that for our experiments we had a fixed training iteration parameter of fifty (50), and maybe the quantum model required more iterations of training to have better results.

Quantum Complexity Results

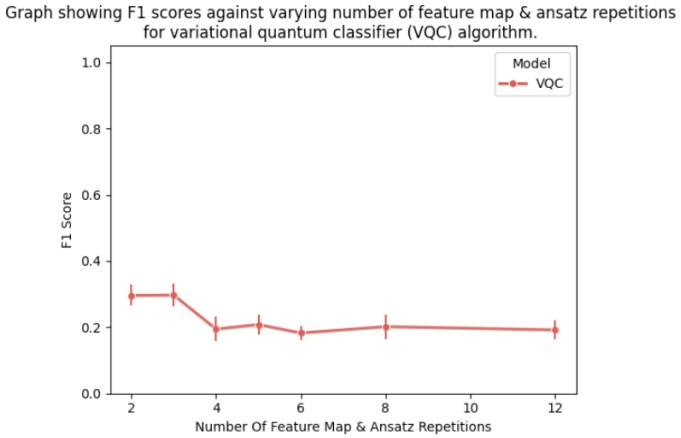

Figure 24: Figure visualizes the variational quantum classifier (QVC) performance, Mean F1 score, with increasing pairs of $feature\_map$ and $ansatz$ repetitions.

In our objectives, we defined the aim of checking for quantum performance changes with increasing quantum complexity. Figure 24 shows the quantum circuit performance with varying degrees of quantum complexity. Table 7 shows the F1 score for varying levels of feature map and ansatz repetitions. Figure 24 shows no relation between the performance and the quantum circuit complexity. There is a belief that increasing the circuit complexity should improve performance, as complex circuits mean more entanglement layers and better sharing of information between qubits. In our experiments, an increase in feature map, and ansatz repetitions did not lead to any improvement in performance.

To summarize the results,

1. The best-performing models for varying numbers of input features (M) are Random forest ($M = 2 : F1 = 0.821$, $M = 4 : F1 = 0.972$), Quantum SVC ($M = 8 : F1 = 0.987$, $M = 12 : F1 = 1.000$, $M = 16 : F1 = 1.000$). The results for the remaining model are in the Table 4.

2. The performance of the quantum QSVC model, and best classical model for varying numbers of classes (N) is, for Light Gradient Boosting Machine is ($N = 3 : Mean - F1 = 0.953$, $N = 4 : Mean - F1 = 0.947$, $N = 5 : Mean - F1 = 0.936$, $N = 6 : Mean - F1 = 0.921$, $N = 8 : Mean - F1 = 0.968$, $N = 12 : Mean - F1 = 0.943$), and for Quantum SVC is ($N = 3 : Mean - F1 = 0.903$, $N = 4 : Mean - F1 = 0.899$, $N = 5 : Mean - F1 = 0.891$, $N = 6 : Mean - F1 = 0.880$, $N = 8 : Mean - F1 = 0.817$, $N = 12 : Mean - F1 = 0.772$)

3. The quantum model training time on the quantum-classical simulator for the number of input features (M) for QVC is ($M = 2 : T = 68s,\ M = 4 : T = 192s,\ M = 8 : T = 732s,\ M = 12 : T = 45223s,\ M = 16 : T = 131429s$), and for Quantum SVC is ($M = 2 : T = 3s,\ M = 4 : T = 4s,\ M = 8 : T = 7s,\ M = 12 : T = 27s,\ M = 16 : T = 4174s$).

4. The performance, F1 score, for QVC with varying numbers of feature map and ansatz repetitions (R) is $R = 2 : F1 = 0.296,\ R = 3 : F1 = 0.294,\ R = 4 : F1 = 0.194,\ R = 5 : F1 = 0.213,\ R = 6 : F1 = 0.182\ R = 8 : F1 = 0.201,\ R = 12 : F1 = 0.191$).

Conclusions

As we bring this study to an end, we build this section to share the experimental, and theoretical findings. To briefly recall, the objective of this study is to explore and present the science of quantum computing by constructing quantum circuit architectures to perform the machine learning task of classification. At the same time to have a benchmark to compare it with, we used popular classical machine learning algorithms, with an accelerated and simplified training pipeline. The aim is not to have a direct one-on-one comparison, as quantum computing and machine learning advancements are at different stages of growth; such comparisons would be unfair. When considering the domain of time complexity, the field of quantum computing is exploring all domains and is not optimized for machine learning tasks. The variational quantum algorithm on which the quantum circuits are built is a generalized algorithm to approach problem-solving using quantum computing.

To conclude,

1. Quantum support vector classifier (QSVC) based on a linear kernel outperforms the classical machine learning classifiers. Among the classical ml classifiers, the random forest classifier, and light gradient boosting machine perform the best.

2. The performance of the quantum SVC classifier drops by a small margin with increasing the number of classes, this is in contrast with the good classical models whose performance remains stable.

3. Increasing the complexity of the quantum circuit algorithms, exponentially increases computation time on the quantum-classical simulator.

4. Varying the number of feature map repetitions, or ansatz repetitions to vary the variational quantum classifier (QVC) complexity does not lead to improved performance of the classifier.

5. The performance, F1 score, of quantum models could change if the experiments were repeated on a physical quantum computer. While the training time would not increase exponentially due to the nature of the quantum algorithm.

Acknowledgments: We would like to thank, firstly, Vilnius University, and the Mathematics & Informatics Faculty, for providing a chance to perform, and present this study. We would like to thank Dr. Linas Petkevicius, and Dr. Virginijus Marcinkevicius, without your input this would not have been possible. I would like to extend my thanks to my parents, brother, friends, colleagues, IBM, and the city of Vilnius.

References

[1] Mirko Amico, Zain H. Saleem, and Muir Kumph. Experimental study of shor's factoring algorithm using the ibm q experience. *Physical Review A*, 100(1), July 2019.

[2] Wikipedia article on Boosting (machine learning). Wikipedia boosting (ml): https://en.wikipedia.org/wiki/boosting_(machine_learning), 2023.

[3] Wikipedia article on classification. Wikipedia classification: https://en.wikipedia.org/wiki/statistical_classification, 2023.

[4] Kerstin Beer. Quantum neural networks. 2022.

[5] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, September 2017.

[6] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, page 144–152, New York, NY, USA, 1992. Association for Computing Machinery.

[7] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[8] Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J. Martinez, Jae Hyeon Yoo, Sergei V. Isakov, Philip Massey, Ramin Halavati, Murphy Yuezhen Niu, Alexander Zlokapa, Evan Peters, Owen Lockwood, Andrea Skolik, Sofiene Jerbi, Vedran Dunjko, Martin Leib, Michael Streif, David Von Dollen, Hongxiang Chen, Shuxiang Cao, Roeland Wiersema, Hsin-Yuan Huang, Jarrod R. McClean, Ryan Babbush, Sergio Boixo, Dave Bacon, Alan K. Ho, Hartmut Neven, and Masoud Mohseni. Tensorflow quantum: A software framework for quantum machine learning, 2021.

[9] builtin.com. Machine learning algorithms: https://builtin.com/data-science/tour-top-10-algorithms-machine-learning-newbies, 2023.

[10] M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and Patrick J. Coles. Variational quantum algorithms. *Nature Reviews Physics*, 3(9):625–644, August 2021.

[11] Iris Cong, Soonwon Choi, and Mikhail D. Lukin. Quantum convolutional neural networks. *Nature Physics*, 15(12):1273–1278, aug 2019.

[12] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[13] Qiskit ecosystem. Quantum kernels: https://qiskit.org/ecosystem/machine-learning/tutorials/03_quantum_kernel.html, 2023.

[14] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[15] Siddhant Garg and Goutham Ramakrishnan. Advances in quantum deep learning: An overview, 2020.

[16] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996.

[17] Vojtěch Havlíček, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, and Jay M. Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209–212, March 2019.

[18] Amir H. Karamlou, William A. Simon, Amara Katabarwa, Travis L. Scholten, Borja Peropadre, and Yudong Cao. Analyzing the performance of variational quantum factoring on a superconducting quantum processor. *npj Quantum Information*, 7(1), October 2021.

[19] Dawid Kopczyk. Quantum machine learning for data scientists, 2018.

[20] Avery Leider, Gio Abou Jaoude, Abigail E Strobel, and Pauline Mosley. Quantum machine learning classifier. In *Future of Information and Communication Conference*, pages 459–476. Springer, 2022.

[21] pennylane.com. Feature map: https://pennylane.ai/qml/glossary/quantum_feature_map/, 2023.

[22] Koustubh Phalak, Avimita Chatterjee, and Swaroop Ghosh. Quantum random access memory for dummies, 2023.

[23] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.

[24] Foster Provost and Tom Fawcett. Data science and its relationship to big data and data-driven decision making. *Big Data*, 1(1):51–59, 2013. PMID: 27447038.

[25] Qiskit. Quantum neural network classifier: https://qiskit.org/ecosystem/machine-learning/tutorials/02_neural_network_classifier_and_regressor.html, 2023.

[26] Qiskit. Quantum neural networks: https://qiskit.org/ecosystem/machine-learning/tutorials/01_neural_networks.html, 2023.

[27] Qiskit. Quantum vqc: https://qiskit.org/ecosystem/machine-learning/tutorials/02a_training_a_quantum_model_on_a_real_dataset.html, 2023.

[28] readthedocs.io. Cross-entropy: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html, 2023.

[29] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. Quantum support vector machine for big data classification. *Physical Review Letters*, 113(13), September 2014.

[30] Maria Schuld, Alex Bocharov, Krysta M. Svore, and Nathan Wiebe. Circuit-centric quantum classifiers. *Phys. Rev. A*, 101:032308, Mar 2020.

[31] Maria Schuld and Nathan Killoran. Quantum machine learning in feature hilbert spaces. *Physical Review Letters*, 122(4), February 2019.

[32] Scikit. Scikit: https://scikit-learn.org/stable/modules/multiclass.html, 2023.

[33] scikit learn.org. Mutliclass and multioutput algorithms: https://scikit-learn.org/stable/modules/multiclass.html, 2023.

[34] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[35] Alaa Tharwat. Linear vs. quadratic discriminant analysis classifier: a tutorial. *International Journal of Applied Pattern Recognition*, 3(2):145–180, 2016.

[36] Farrokh Vatan and Colin Williams. Optimal quantum circuits for general two-qubit gates. *Physical Review A*, 69(3), mar 2004.

[37] Guillaume Verdon, Michael Broughton, Jarrod R. McClean, Kevin J. Sung, Ryan Babbush, Zhang Jiang, Hartmut Neven, and Masoud Mohseni. Learning to learn with quantum neural networks via classical neural networks, 2019.

[38] Wikipedia. Wikipedia article: https://en.wikipedia.org/wiki/shor2023.

[39] wikipedia.com. Cross-entropy: https://en.m.wikipedia.org/wiki/cross-entropy, 2023.

Classical ML Running,

Multiclass Performance with Single Label Training & Prediction,

In [ ]:
```python
# Imports
import numpy as np
import pandas as pd
import time
```

In [ ]:
```python
# generate data imports
from sklearn.datasets import make_classification, make_multilabel_classification
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score, f1_score
```

In [ ]:
```python
# for data split
from sklearn.model_selection import train_test_split
from qiskit_algorithms.utils import algorithm_globals
algorithm_globals.random_seed = 123
from pycaret.classification import *
```

In [ ]:
```python
class classicalMultiLabelAlgoTraining:
    """
    Class to train algorithms."""
    def __init__(self, no_of_features, no_of_samples, no_of_classes, no_of_labels, models = ['knn', 'qda', 'rf', 'l
        self.no_of_features = no_of_features
        self.no_of_samples = no_of_samples
        self.no_of_labels = no_of_labels
        self.no_of_classes = no_of_classes
        self.models = models
        print(f"the no of classes is {self.no_of_classes}")

    def data_generation(self):
        """
        Generate classification data using sklearn's data generation.\
        """
```

```
        n_samples=self.no_of_samples
        n_features=self.no_of_features
        n_classes=self.no_of_classes
        n_labels=self.no_of_labels
        print(n_samples)
        X, y = make_multilabel_classification(n_samples=n_samples,
                                              n_features=n_features,
                                              n_classes=n_classes,
                                              n_labels=n_labels,
                                              random_state=algorithm_globals.random_seed
                                             )
        y_new = np.array([self.conv_to_int(val) for val in y])
        return X, y, y_new

    def int_to_hot(self, x):
        "convert to labels of [0s, 1s]"
        format = '{' + '0:0{:d}b'.format(self.no_of_classes) + '}'
        result = format.format(x)
        list_of_ints = [int(x) for x in result]
        return np.array(list_of_ints)

    def get_mapping(self):
        dictionary = {}
        numbers = list(range(0, 2**self.no_of_classes))
        for i in numbers:
            dictionary[i] = self.int_to_hot(i)
        return dictionary

    def conv_to_int(self, vector):
        "convert d-dimensional array to integer"
        val = "".join(vector.astype(str))
        value = int(val, 2)
        return value

    def pycaret_training(self):
        X, y, y_new = self.data_generation()
        print(X.shape, y.shape, y_new.shape)
        exp_name = setup(data = X,  target = y_new.reshape(-1), train_size=0.8)
        mapping = self.get_mapping()
        for model in self.models:
            start = time.time()
```

```
            model_created = create_model(model, verbose=False)
            end = time.time()
            time_elapsed = end - start
            predicted_model = predict_model(model_created)
            predictions = model_created.predict(X)
            short_results = pull(model_created)
            # Use list comprehension to create the new array
            predictions_labels_final = np.array([mapping[val] for val in predictions])
            accuracy_scores = accuracy_score(y, predictions_labels_final)
            f1_scores = f1_score(y, predictions_labels_final, average='weighted')
            results.setdefault('Model', []).append(short_results['Model'].values[0])
            results.setdefault('No of features', []).append(self.no_of_features)
            results.setdefault('No of samples', []).append(self.no_of_samples)
            results.setdefault('No of classes', []).append(self.no_of_classes)
            results.setdefault('No of labels', []).append(self.no_of_labels)

            results.setdefault('Accuracy', []).append(accuracy_scores)
            results.setdefault('F1', []).append(f1_scores)
            results.setdefault('Time taken', []).append(time_elapsed)
```

In [ ]:
```
results = {}
no_of_features = [2,4,8,12,16] # as much as quantum side allows
no_of_samples = 1024 # only 1024
no_of_classes = [3] # look for quantum side
no_of_labels = [3] # starting from 2, till max no of classes.
```

In [ ]:
```
for no_feature in no_of_features:
    for no_class in no_of_classes:
        for no_label in no_of_labels:
            if no_label <= no_class:
                # print(f"the setup is {no_feature}, {no_class}, {no_label}")
                training_object = classicalMultiLabelAlgoTraining(no_feature, no_of_samples, no_class, no_label)
                training_object.pycaret_training()
```

Quantum VQC Running,

Multiclass Performance with Single Label Training & Prediction,

Here are the experiments with the quantum side of things.

In [ ]:
```
# For dataset.
from sklearn.datasets import make_multilabel_classification, make_classification
from qiskit_algorithms.utils import algorithm_globals
from qiskit.algorithms.optimizers import COBYLA, L_BFGS_B
from qiskit_machine_learning.algorithms.classifiers import NeuralNetworkClassifier, VQC, QSVC
from qiskit_machine_learning.kernels.algorithms import QuantumKernelTrainer
from qiskit_machine_learning.kernels import BaseKernel, TrainableFidelityQuantumKernel
```

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
import time
import pandas as pd
import abc
```

In [ ]:
```
# for data split
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
```

In [ ]:
```
from qiskit.circuit.library import ZFeatureMap, ZZFeatureMap
```

In [ ]:
```
from qiskit.circuit.library import RealAmplitudes
```

In [ ]:
```
class quantumMultiLabelAlgoTrainingVQC:
    """
    Class to train quantum algorithms for multiclass multilabel.
```

```
    """
    def __init__(self, no_of_features, no_of_samples, no_of_classes, no_of_labels, no_of_feature_map_reps, no_of_an
        self.no_of_features = no_of_features
        self.no_of_samples = no_of_samples
        self.no_of_labels = no_of_labels
        self.no_of_classes = no_of_classes
        self.models = models
        self.featuremap_reps = no_of_feature_map_reps
        self.ansatz_reps = no_of_ansatz_reps
        self.objective_func_vals = []
        print(f"the no of classes is {self.no_of_classes}")

    def data_generation(self):
        """
        Generate classification data using sklearn's data generation.\
        """
        n_samples=self.no_of_samples
        n_features=self.no_of_features
        n_classes=self.no_of_classes
        n_labels=self.no_of_labels
        print(n_samples)
        X, y = make_multilabel_classification(n_samples=n_samples,
                                              n_features=n_features,
                                              n_classes=n_classes,
                                              n_labels=n_labels,
                                              random_state=algorithm_globals.random_seed
                                             )
        y_new = np.array([self.conv_to_int(val) for val in y])
        print(X.shape, y.shape, y_new.shape)
        print(X[:5], y[:5], y_new[:5])
        return X, y, y_new

    def int_to_hot(self, x):
        format = '{' + '0:0{:d}b'.format(self.no_of_classes) + '}'
        result = format.format(x)
        list_of_ints = [int(x) for x in result]
        return np.array(list_of_ints)

    def get_mapping(self):
        dictionary = {}
        numbers = list(range(0, 2**self.no_of_classes))
```

```
        for i in numbers:
            dictionary[i] = self.int_to_hot(i)
        return dictionary

    def conv_to_int(self, vector):
        val = "".join(vector.astype(str))
        value = int(val, 2)
        return value

    def generate_featuremap(self):
        """ Generating the feature map."""
        feature_map = ZZFeatureMap(feature_dimension=self.no_of_features, reps=self.featuremap_reps)
        return feature_map

    def generate_ansatz(self):
        """Generating ansatz."""
        ansatz = RealAmplitudes(num_qubits=self.no_of_features, reps=self.ansatz_reps)
        return ansatz

    def callback_graph(self, weights, obj_func_eval):
        clear_output(wait=True)
        self.objective_func_vals.append(obj_func_eval)
        plt.title("Objective function value against iteration")
        plt.xlabel("Iteration")
        plt.ylabel("Objective function value")
        plt.plot(range(len(self.objective_func_vals)), self.objective_func_vals)
        plt.show()

    def quantum_training(self):
        X, y, y_new = self.data_generation()
        mapping = self.get_mapping()
        optimizer = COBYLA(maxiter=50)
        model = 'VQC'
        vqc = VQC(
            feature_map=self.generate_featuremap(),
            ansatz=self.generate_ansatz(),
            loss='cross_entropy',
            optimizer=optimizer,
            callback=self.callback_graph,
        )
```

```
        # clear objective value history
        objective_func_vals = []

        start = time.time()
        vqc.fit(X, y_new)
        elapsed = time.time() - start

        print(f"Training time: {round(elapsed)} seconds")

        # testing the QSVC scores
        predictions = vqc.predict(X)

        # Use list comprehension to create the new array
        print(f"The type of predictions is: {type(predictions)}")

        predictions_labels_final = np.array([mapping[val] for val in predictions])
        print(y.shape, predictions_labels_final.shape)
        print(predictions_labels_final[:5])
        accuracy_scores = accuracy_score(y, predictions_labels_final)
        f1_scores = f1_score(y, predictions_labels_final, average='weighted')
        results.setdefault('Model', []).append(model)
        results.setdefault('No of features', []).append(self.no_of_features)
        results.setdefault('No of samples', []).append(self.no_of_samples)
        results.setdefault('No of classes', []).append(self.no_of_classes)
        results.setdefault('No of labels', []).append(self.no_of_labels)
        results.setdefault('No of feature map reps', []).append(self.featuremap_reps)
        results.setdefault('No of ansatz reps', []).append(self.ansatz_reps)
        results.setdefault('Accuracy', []).append(accuracy_scores)
        results.setdefault('F1', []).append(f1_scores)
        results.setdefault('Time taken', []).append(elapsed)
```

In [ ]:

```
results = {}
no_of_features = [2,4,6,8,12,16]
no_of_samples = [1024]
no_of_classes = [3]
no_of_labels = [3]
no_of_feature_map_reps = [3]
no_of_ansatz_reps = [3]
```

In [ ]:
```
for feature in no_of_features:
    for sample in no_of_samples:
        for featuremap_reps in no_of_feature_map_reps:
            for ansatz_reps in no_of_ansatz_reps:
                for clas in no_of_classes:
                    for lab in no_of_labels:
                        if lab <= clas:

                            training_object = quantumMultiLabelAlgoTrainingVQC(feature, sample, clas, lab, featurem
                            training_object.quantum_training()
```

Quantum QSVC Training,

Multiclass Performance with Single Label Training & Prediction,

In [ ]:
```
# For dataset.
from sklearn.datasets import make_multilabel_classification, make_classification
from qiskit_algorithms.utils import algorithm_globals
from qiskit.algorithms.optimizers import COBYLA, L_BFGS_B
from qiskit_machine_learning.algorithms.classifiers import NeuralNetworkClassifier, VQC, QSVC
from qiskit_machine_learning.kernels.algorithms import QuantumKernelTrainer
from qiskit_machine_learning.kernels import BaseKernel, TrainableFidelityQuantumKernel
```

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
import time
import pandas as pd
import abc
```

In [ ]:
```
# for data split
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
```

In [ ]:
```
from qiskit.circuit.library import ZFeatureMap, ZZFeatureMap
from qiskit.circuit.library import RealAmplitudes
```

In [ ]:
```
from qiskit.primitives import Sampler
from qiskit_algorithms.state_fidelities import ComputeUncompute
from qiskit_machine_learning.kernels import FidelityQuantumKernel, FidelityStatevectorKernel
```

In [ ]:
```
def conv_to_int(vector):
        val = "".join(vector.astype(str))
```

```
        value = int(val, 2)
        return value
```

In [ ]:
```
def int_to_hot(x, no_of_classes):
        format = '{' + '0:0{:d}b'.format(no_of_classes) + '}'
        result = format.format(x)
        list_of_ints = [int(x) for x in result]
        return np.array(list_of_ints)
```

In [ ]:
```
def get_mapping(no_of_classes):
        dictionary = {}
        numbers = list(range(0, 2**no_of_classes))
        for i in numbers:
            dictionary[i] = int_to_hot(i, no_of_classes)
        return dictionary
```

In [ ]:
```
# get data
def data_generation(n_samples, n_features, n_classes, n_labels):
        """
        Generate classification data using sklearn's data generation.\
        """
        X, y = make_multilabel_classification(n_samples=n_samples,
                                              n_features=n_features,
                                              n_classes=n_classes,
                                              n_labels=n_labels,
                                              random_state=algorithm_globals.random_seed
                                             )
        y_new = np.array([conv_to_int(val) for val in y])
        print(X.shape, y.shape, y_new.shape)
        print(X[:5], y[:5], y_new[:5])
        return X, y, y_new
```

In [ ]:
```
results = {}
model = 'QSVC FidelityStateKernel'
no_of_samples = 1024
no_of_features = [2,4,6,8,12,16] # as much as quantum side allows
no_of_classes = [3]
no_of_labels = [3]
no_of_featuremap_reps = [3]
```

In [ ]:

```
for no_feature in no_of_features:
    for reps in no_of_featuremap_reps:
        for no_class in no_of_classes:
            for no_label in no_of_labels:
                if no_label <= no_class:
                    X, y, y_new = data_generation(no_of_samples, no_feature, no_class, no_label)
                    adhoc_feature_map = ZZFeatureMap(feature_dimension=no_feature, reps=reps, entanglement="linear"
                    adhoc_kernel = FidelityStatevectorKernel(feature_map=adhoc_feature_map)
                    optimizer = COBYLA(maxiter=50)
                    qsvc = QSVC(quantum_kernel=adhoc_kernel)
                    start = time.time()
                    qsvc.fit(X, y_new)
                    elapsed = time.time() - start
                    predictions = qsvc.predict(X)
                    mapping = get_mapping(no_class)
                    predictions_labels_final = np.array([mapping[val] for val in predictions])
                    print(predictions_labels_final[:5])
                    accuracy_scores = accuracy_score(y, predictions_labels_final)
                    f1_scores = f1_score(y, predictions_labels_final, average='weighted')
                    results.setdefault('Model', []).append(model)
                    results.setdefault('No of features', []).append(no_feature)
                    results.setdefault('No of samples', []).append(no_of_samples)
                    results.setdefault('No of classes', []).append(no_class)
                    results.setdefault('No of labels', []).append(no_label)
                    results.setdefault('No of feature map reps', []).append(reps)
                    results.setdefault('Accuracy', []).append(accuracy_scores)
                    results.setdefault('F1', []).append(f1_scores)
                    results.setdefault('Time taken', []).append(elapsed)
```