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

INVESTIGATION OF CLASSIFICATION ALGORITHMS IN QUANTUM COMPUTING

Master's thesis

Author: Skalvis Paliulis
VU email address: skalvis.paliulis@mif.stud.vu.lt
Supervisor: Assoc. Prof. Linas Petkevičius

Vilnius

2024

Acknowledgments:

First, I would like to thank my supervisor Dr. Linas Petkevičius, for introducing me into quantum computing field and support during preparation and analysis of this master thesis. Also, I want to thank Vilnius University Faculty of Mathematics and Informatics for giving chance to perform this study. Lastly, I want to extend sincere thank you for all lecturers, for giving support during my studies.

Abstract

This master's thesis aims to identify and empirically investigate classification algorithms implementation in quantum computing, focusing on PennyLane framework utility. Due to substantial increase in computing power via quantum computers in recent years, actual application of quantum algorithms and their performance against classical algorithms are relevant. Moreover, given that classical algorithms optimization functions rely on non-convex objective functions, they tend to be stuck within local maxima/minima. With quantum algorithms, we can either utilize quantum bits/superposition/entanglement to represent multiple states as their respective possibilities to escape local maxima or finding global minima. The benchmark results indicate that quantum classification algorithms perform similarly to their classical algorithms counterparts, however are time inefficient and not well-suited for large data sets.

There are many similar frameworks for quantum computing. PennyLane is compatible with any gate-based quantum simulator or hardware, whereas others are not (i.e., Qiskit). Compatibility importance cannot be understated.

Keywords: Classification, Quantum Computing, PennyLane, Benchmark

Santrauka

Šiame magistriniame darbe pagrindiniai siekiniai buvo empiriškai ištirti klasifikavimo algoritmų pateikimą naudojant kvantinės komputerijos metodus, specifiškai, naudojant PennyLane karkasą. Pastaruoju metu augant kvantinių kompiuterių pajėgumams, kvantinių algoritmų galimybės lyginant su klasikiniais algoritmais tampa vis labiau aktualios. Atsižvelgiant į tai, kad klasikiniuose algoritmuose optimizavimo fukcijos priklauso nuo tiesinių tikslo pasiekimo funkcijų, dažnai užstringama vietiniam spendinio minimumame arba maksimumame. Naudojant kvantinius algoritmus ir tuo pačius pasitelkiant jiems budingas konstrukcines savybes, kvantinį bitą - kubitą, superpoziciją, bei kvantinį supynimą, mes galime tuo pačiu metu atvaizduoti keletą būsenų, bei susieti tikimybes su tomis būsenomis, kas leidžia ištrūkti iš vietinio minimumo arba maksimumo. Palyginimo analizė ir rezultatai rodo, kad kvantiniai klasifikavimo algoritmai klasifikavimo užduotis atlieka panašiu tikslumu kaip ir klasikiniai, tačiau reikalauja didelių laiko resursų kubitų operacijų atlikimui ir procesoriaus simuliacijai, bei nepritaikyti dideliems duomenų rinkiniams.

Šiuo metu yra pakankamai daug karkasų skirtų kvantiniams skaičiavimams atlikti. PennyLane yra vienas iš jų, turintis tiesioginį suderinamumą su betkuriuo plačiau žinomu vartu pagrindu sukurtu simuliatoriumi ar kvantiniu prietaisu. Tuo pačiu, kiti karkasai, tokios galimybės neturi, todėl žinant tai jog sritis plėtojasi labai greitai, pritaikomumo suderinamumas negali būti nuvertinamas.

Raktažodžiai: Klasifikavimas, Kvantiniai skaičiavimai, PennyLane, Lyginamoji analizė

Notation

We outlined the following notation used in this thesis:

- To compare performance, we use notation $\mathcal{O}(N)$, which defines the upper limit of operation order.
- In general, several cost function notations are used: $C(\theta), \chi^2, f(\theta)$, where θ usually in all cases denote parameters.
- $f(\cdot)$ denotes specific function described in the text, unless specified otherwise.
- $|a\rangle$, is used to portray a Ket 1 of state, where vector is $\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$, whereas for general Ket notation we use $|\psi\rangle$.
- respectively, Bra is used to denote a linear form of Ket from complex plane to numerical one and is denoted by $\langle a|$, where Bra a is row vector of $\begin{pmatrix} a_1^* & a_2^* \end{pmatrix}$ and corresponds to complex conjugate transpose of Ket a. For general Bra notation we use $\langle \phi|$.
- $\langle \phi|\psi\rangle$ denotes an inner product of two orthonormal vectors in a braket notation, where result is complex number and measure of their similarity between their computational states.
- Unitary operator is denoted as $U(\theta)$.
- probabilities associated basis states are usually denoted as α, β, therefore computational basis can be expressed as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$.
- Probabilities in traditional sense are denoted as $P(\cdot)$

Contents

Introduction

In recent years quantum computing power has been increasing steadily which in turn resulted in increased of interest for quantum algorithms. Easily accessible frameworks and tools are necessary for growing community to further the advance of emerging technology. Qubit count is considered as one of the limiting factors for computational tasks, given that as of now IBM has the largest quantum computer of 433 qubits[1]. Term NISQ - Noisy intermediate-scale quantum is used to describe current era of quantum processors that go up to 1000 qubits, which are not advanced enough to be fault tolerant or large enough to achieve quantum supremacy.

IBM Quantum, Google Quantum AI, Azure Quantum, Amazon Braket and Nvidia CuQuantum are of few better-known publicly available solutions with various capabilities accessible to public running on different Quantum computers. Important, distinction to consider that quantum computations can be either done via code ran in simulator on your local/cloud environment or processes through actual quantum computer. This requires software solution either compatible to do both or specialised to particular task. For example Qiskit, a SDK (software development Kit) provides support for native primitives, cloud services or quantum hardware or local simulators. Even though it offers integration with multiple hardware solutions, Qiskit is still highly tied to IBM, rendering it dependent. Therefore, for this masters thesis we have opted to use PennyLane framework that is hardware independent.

The aim of the thesis is to empirically investigate classification algorithms in quantum computing using PennyLane framework. Focus to be directed towards already available quantum classification algorithms, compared against well-established classical algorithms. Moreover, we ought to cover supervised learning algorithms. In terms of classification task, single-class/binary and multi-class classification algorithm benchmarks assessed using standard metrics. Throughout experiment, mostly locally held simulators to be used. Overall objectives of thesis are:

1. Carry out the scientific literature review of classification algorithms which are implementable in quantum computing.

2. Identify the algorithms for further analysis.

3. Identify and select data sets for bench-marking and classification algorithms.

4. Investigate and compare the selected algorithms on benchmarks setup using quantum simulators (or quantum computer).

5. Provide the results, insights and recommendations on using quantum classification algorithms in practice.

Well-know and established data sets applicable for classification tasks to be used for analysis. Expected final result and output would be clear comparison between classification performance of classical and quantum algorithms.

[1]`https://www.ibm.com/quantum/roadmap` accesed on 2023-11-16

1 Quantum Machine Learning Algorithms for Classification

1.1 Literature Review

Any physical object, if analyzed at a deep enough level, is a quantum object. Be it any simple transistor, screwdriver or electronic insulator. Any object can be assessed from their properties of electron composition, bond structure or electron wave functions. Similarly, we couldn't imagine any classical computer or classical information theory if not for knowledge about quantum physics. Therefore, any computer that is not quantum computer, does function according principles of quantum mechanics [4]. Michel Le Bellac outlines that quantum behaviour is also a collective behaviour, where simple example from computational point of view is: any classical bit can have two states of 1 and 0. Such values are charged or uncharged state of a capacitor, respectively. The difference between states is a displacement of 10^4 to 10^5 electrons. With constant advancement, physicists have found a way how to observe and manipulate quantum objects not only as collective, but also as a singular objects (ions, atoms, photons), which is a foundation of quantum computing and quantum bit - qubit [4].

In field of semiconductors *Moore's law* states that [28]:

> Transistors on a integrated circuit doubles approximately every two years, leading to an exponential increase in computational power.

There have been multiple guesses how long law would apply, given that ongoing miniaturization of microchips would eventually hit a limit where law could no longer be applied. Current consensus seems to refer that final usable limit of transistor's gate length will be 5 nm, where fundamental physical limit of the size of an atom would becomes an inhibiting factor [35]. With this in mind, continued geometrical growth of computational power reaches limit. Therefore, even if classical computers and their computational power are reaching a road block, there is an option to divert attention to quantum computers and algorithms.

Quantum computing and machine learning have been put together since the beginning of quantum computing in 1980s, whereas quantum machine learning (QML) came into use in 2013-2014. Since then, QML have established itself as an active sub-discipline of quantum computing research [33]. We can say that quantum computing itself is undergoing transition, form only being a theoretical to more applicable discipline. Notwithstanding the possible quantum advantage (or quantum speedup in other literature) to be achieved in theoretical sense, fact remains that current devices are "non-fault-tolerant", "small scale" or "near term", meaning that the order of operations available are from 1000 to 10000 elementary operations given number of 50-100 qubits [32, 6]. Therefore, this it not enough yet to move from classical to quantum approach yet. Also, given high error rates and costly error correction, for now, small scale algorithms can be of use as more of an illustration to display power of quantum computing. Whereas, the long term goal of any quantum algorithm is to achieve quantum speed up, meaning that with given quantum algorithm with M being data input entries and N denoting dimension of given data set, the number of elementary applications are $\mathcal{O}(NM)$, resulting in linear growth of elementary operations, where for classical counterparts growth amount of elementary operations is quadratic, therefore goal of quantum algorithms is to achieve quantum speed up or exponential speedup [33]. This can be rephrased by applying *Church-Turing thesis* used to describe classical computational systems [4, 8]:

any computational model can be simulated on a probabilistic Turing machine with at most a polynomial increase in the number of computational steps.

In this context, if all computational systems can be simulated only with polynomial increase in elementary operations, then Shor‘s algorithm contradicts this. Integer number factorization task is considered "intractable" problem (there are no known algorithm for given task for which running time would be polynomial in terms of size of an input), however using Shor‘s algorithm quantum computer can decompose integer into primes applying number of steps which is polynomial, whereas classical counterparts can only do the same which exponential increase in steps [4]. Grover‘s algorithm is a prime example of quantum speedup and advantage over classical search algorithms in unstructured data. Assuming that we ought to find accurate one-to-one equivalent item and given N that denotes number of entries in database, classical algorithms, on average might have to perform $\mathcal{O}(N)$ comparisons to find given item and probability of finding particular entry after k entries becomes k/N. Whereas, Grover‘s algorithm, can achieve high matching probability of two same entries by only checking $\mathcal{O}\sqrt{N}$ amount of entries in database [25, 12]. Therefore, assumed upper bound of $\mathcal{O}(N)$ is minimized by square factor.

First cloud based computer became available in 2016 released by IBM. However, due to high error-rates and qubit limitations, practical applications were limited [6]. Following, new similar devices were released and even though with apparent limits, quantum computers already show promise, to outperform some of classical supercomputers in certain mathematical tasks [6]. This raised a general term to describe current era of quantum computing: Noisy Intermediate-Scale Quantum computers (NISQ). Even though actual fault-tolerant quantum computers seems to be far away, the aim is to utilize currently available NISQ devices to achieve albeit limited advantage. Variational Quantum Algorithms as term and proof-of-concept were introduced by Alberto Peruzzo *et al.* in article "A variational eigenvalue solver on a quantum processor" [27], where authors introduced re-configurable quantum processing unit (QPU) that calculates expectation value ran on classical processing unit (CPU). In a sense, we can consider this as a quantum circuit that is utilized via layers together with classical subroutine. In similar matter, Quantum Circuits were introduced in 2019 by Mitarai *et al.*, where they published their article "Quantum Circuit Learning" (QCL) describing quantum-classical algorithm [23]. Idea behind such algorithm was to show that QCL can learn non-convex patterns, for supervised and unsupervised learning. Algorithm consists of five stages [23]:

1. Encode data x_i to into some quantum state by applying a unitary operation $U(x_i)$ to initialized qubits $|0\rangle$.

2. By applying parameterized unitary $U(\theta_i)$ to the input we generate output state.

3. Expectations values are being measured using Pauli operators.

4. Minimization of cost function $L(f(x_i), y(x_i, \theta)$, where $f(x_i)$ is the optimization function (referred as teacher in article) and y_i is the output. Circuit parameters θ_i are being optimized iteratively.

5. Performance is checked by assessing cost function, using training - test data set concept.

Such structure is conceptual example of circuit based algorithms explored and used on NISQ devices. As discussed in Perruzo *et al.* [27] and Mitarai *et al.* [23], algorithms utilize quantum feature space for data encoding into quantum feature spaces, perform inference, whereas optimization cost function is calculated by classical approach, such concept is recurring and popular [31, 27, 23, 15]. Schuld and Petruccione [33] discusses the so called four intersections in terms how quantum computing and machine learning can be combined and is outlined in Figure 1. Fundamentally, four approaches can be split if we consider whether data is being generated by classical (C) or quantum (Q) system and the corresponding data is processed by classical (C) or quantum (Q) processing device. The CC part of expects that both, processing system and generative process are classical, which are fundamentally mostly all conventional algorithms used in machine learning. Even if some degree of "dequantified" or "quantum inspired" algorithms were discovered, in essence, we assume that data is processed in a classical way [33]. In QC, quantum generated data is fed into classical machine learning algorithms.

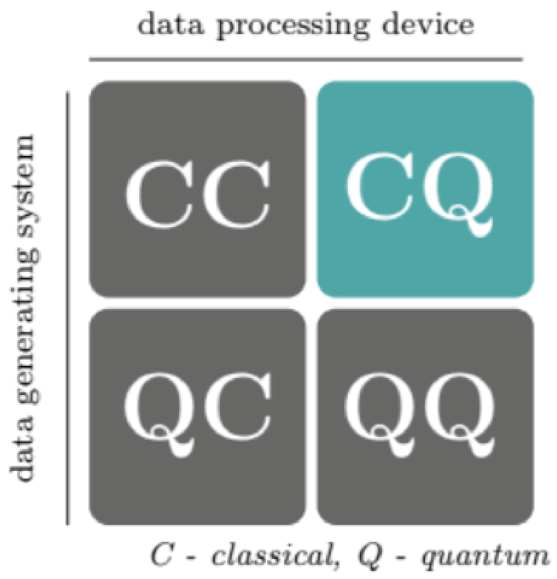

Figure 1: Four approaches for quantum computing and machine learning [33].

For example, neural networks used to simplify quantum states or used with other machine learning algorithms to understand differences between quantum states before and after an experimental step [33]. Lastly, CQ is all conventional data being processed by quantum computer. Authors, emphasize that the link (or interface) is crucial at this step, moreover, exponential speedup is likely impossible as well, due to the nature of setup as data loading from classical memory into quantum memory (quantum able) requires linear time (each data point has to be read and processed) [33]. For QQ, even though research is still rather sparse, authors explain that if data loading routine issue from CQ, then CQ and QQ essentially collapse to the same thing [33].

Quantum machine learning algorithms for classification task are still in early stages of development. Currently, favored approach is to use classical-quantum approach, CQ. In the following sections we will discuss most recent quantum classification algorithms for supervised learning. Most suitable design used in NISQ era is utilization of variational circuits, where gate parameters are learnt with low-depth (small qubit count with arbitrary low number of operations performed before measuring result) [32]. The general structure of supervised circuit-classifier, as outlined in Figure 2 according Shuld *et al.* [32], has model inference performed by QPU $f(x, \theta) = y$, here θ are learnable model parameters that can be trained/optimized, S_x is *state preparation circuit* which encodes data inputs x into quantum states that can be processed by U_θ *model circuit* that performs controlled arbitrary gate operations on qubit level. Lastly, y is model prediction. It is relevant to mention that such architecture for model circuit

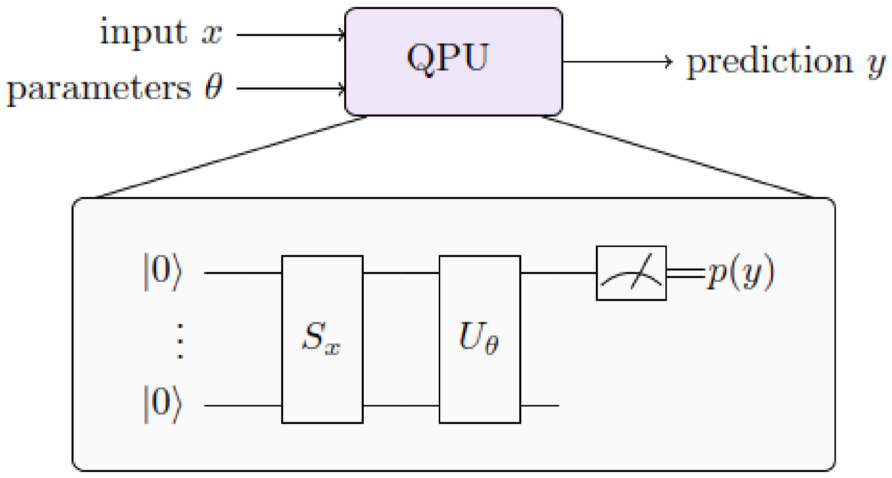

Figure 2: Circuit-centric quantum classifier scheme, outlined by Shuld *et al.* [32].

scales on polynomial level with number of qubits, meaning that number of parameters θ increases on poly-logarithmic scale in input dimension [32]. In broader terms, when we discuss U_θ, the consecutive unitary operations performed, resembles neural network architecture [32, 33, 23, 29, 6].

Classical machine learning algorithms have a multitude of performance measure, for example Sokolova and Lapalme [22] outlined 24 combined measures being used for comparison, specifically classification tasks. Empirical comparison between different algorithms is usually done by using multiple data sets with given algorithms and then doing their performance assessment with criteria outlined beforehand. This allows to perform a benchmark and with evaluated performance modellers can tailor adequate algorithms for specific classification tasks or data sets. We should keep in mind, that in context of supervised classification tasks, granularity of labels assigned by model output is important. For example, given categorical labels, if input is being classified only by two non-overlaping labels, then *binary* classification is performed. When input is being classified into one of many non-overlapping labels, then *multi-class* classification and finally, if output given multiple labels from whole set of non-overlapping labels, then *multi-label* classification is performed [22]. In context of this these, we specifically will create a multi-class classifiers only, whilst review of classification algorithms cover a wider range of classification problems. One of most comprehensive classification performance representation for binary tasks is confusion matrix, which is also applicable for multi-class as well. Authors mention, that origins of wider spectrum of measures are based on values coming from confusion matrix, such as true positive (TP), true negative (TN), false positive (FN) and false positive (FN) [22]. False positive is also considered as type one error, whereas false negative as type two error. For example, for binary classification tasks we can use area-under-curve (AUC), which cannot be simply applied for multi-class due to the fact that it quantifies overall ability to distinguish between positive and negative observations (correct and incorrect classification). Whereas, another graphical representation of reception-operating-characteristic curve (ROC), shows a trade-off between True positive rate (TPR) and false positive rate (FPR) which represents how well model can distinguish between two given classes [22]. Therefore, for multi-class we have to derived measures from confusion matrix that can be calculated for multiple class labels and choose to take One vs Rest or One vs One class comparison when calculating AUC and ROC. We can use the multiple measures as outlined by Sokolova and Lapalme [22], however for the context of this thesis, for numerical measures we will focus on the following ones:

1. Average accuracy - average per-class effectiveness of a classifier

$$\frac{\sum_{i=1}^{l} \frac{tp_i+tn_i}{tp_i+fn_i+fp_i+tn_i}}{l}$$

2. Error rate - average per-class classification error

$$\frac{\sum_{i=1}^{l} \frac{fp_i+fn_i}{tp_i+fn_i+fp_i+tn_i}}{l}$$

3. Precision - average per-class agreement of the data class labels within a classifier

$$\frac{\sum_{i=1}^{l} \frac{tp_i}{tp_i+fp_i}}{l}$$

4. Recall - average per-class effectiveness of a classifier to identify class labels

$$\frac{\sum_{i=1}^{l} \frac{tp_i}{tp_i+fn_i}}{l}$$

5. Fscore - relations between data‘s positive labels and those given by a classifier based on a per class-average

$$\frac{(\beta^2+1)\cdot Precision\cdot Recall}{\beta^2\cdot Precision+Recall}$$

In these measures, l represents number of class labels available, whereas β is parameter weight assigned to precision in combined metric, whilst controlling trade-off between precision and recall. In this thesis $\beta = 1$, meaning that our Fscore is defined as harmonic mean of precision and recall, where both, recall or precision are treated equally. On side note, we must note that, for multi-class classification micro and macro averaging can be taken as two separate approaches when calculating aforementioned measures, where micro-averaging aggregates contributions of all classes to compute performance metrics and only then with given totals you calculate precision, Fscore and recall, whereas macro-averaging performance metrics are calculated for each class independently, allowing equal consideration for each class. Metrics showed above are for macro-averaging as our data sets are balanced in terms of class label appearance. Additionally, for quantum algorithms, we have decided to discuss and include *time elapsed* as another metric, to understand if models used are viable and perform on similar manner as classical counterparts.

1.2 Simulators and Quantum Computers

Before discussing quantum-able simulators and computers, we must understand differences between *near-term* and *fault-tolerant* approaches. As mentioned earlier, goal of quantum computing is to potentially reach quantum speedup in machine learning algorithms, which refers to slower asymptotic growth of upper bound run time with increasing input size. Hypothetically, such could be achieved, if fully error-correction quantum computer was available, however in current NISQ era this is not available and will be out of reach for time being and essentially is called Fault-tolerant computing approach [33]. Near-term approach refers to more viable solution in current era, where quantum computing is considered more of a model than an actual application of quantum computer hardware acceleration and focusing on small scale and low depth computations [33].

There no fault-tolerant and scalable quantum computer yet built, however in the future one could be fully implemented by one of the physical technologies: **trapped ions**, **superconductors** or **photons** [30]. All mentioned technologies share the same issue - quantum noise, as quantum mechanical states are fragile, require isolation from outside environment [30]. Shielding from any outside interference is very hard and expensive to achieve as one has to isolate physical quantum state (wave) from outside influence. In general multiple companies have invested in building their own quantum computers, including numerous startups solely dedicated for specific quantum development area [2]. For example, tech giants like Google, Microsoft, Amazon and IBM, have branched out to hardware and software research, to name a few: IBM - quantum computing [3] and qiskit [4]; Google - hardware research [5] and software/Cirq [6]; Microsoft - stable quantum machine for Azure [7] and Azure Quantum software framework [8] and so on. From smaller companies like Riggeti [9] and Xanadu [10], both solutions are also researched.

In this theses we will review three physical implementations for quantum hardware devices. We note, that there are other possible approaches not discussed (Quantum Dot, Solid-State Spin, Topological and similar). Quantum hardware, or different physical implementations differ in their physical representation of qubit. For example, Dutta *et al.*, present data-reuploding classification algorithm build on single-qubit Ion-trap quantum device [9]. Qubit is realized in charged states of $^{138}\mathrm{Ba}^{+}$ trapped ion[9], where:

$$|0\rangle \equiv \mathrm{S}_{\frac{1}{2},-\frac{1}{2}} \quad |1\rangle \equiv \mathrm{D}_{\frac{5}{2},-\frac{1}{2}}$$

Given ion is trapped (trapped in a sense that they are being isolated by electromagnetic fields by *Linear Paul trap*), where actual values qubit of are represented by internal energy of ion state. Here S represent the initialized state, which then is manipulated quadrapole transition by quantum gates - by applying narrow linewidth laser at 1762 nm. Actual measurement of qubit state is done by laser light at 493 nm and final D-states are pumped out by two lasers at 650 and 614 nm [9]. Actual interaction time between

[2]`https://thequantuminsider.com/2023/12/29/quantum-computing-companies/` last accessed on 2024-01-03
[3]`https://www.ibm.com/quantum` last accessed on 2024-01-03
[4]`https://www.ibm.com/quantum/qiskit` last accessed on 2024-01-03
[5]`https://quantumai.google/hardware` last accessed 2024-01-03
[6]`https://quantumai.google/software` last accessed 2024-01-03
[7]`https://azure.microsoft.com/en-us/solutions/quantum-computing/technology/` last accessed on 2024-01-03
[8]`https://quantum.microsoft.com/` last accessed 2024-01-03
[9]`https://www.rigetti.com/` last accessed 2024-01-03
[10]`https://xanadu.ai/` last accessed on 2024-01-03

laser-qubit sets the rotation angle which allows to take measurement of qubit [9]. Measurement itself is done by taking excited photons over time of 2ms (with overall photon threshold of 15 counts/ 2 ms) where discrimination between bright state results in $|0\rangle \equiv \mathrm{S}_{\frac{1}{2}}$ and dark $|1\rangle \equiv \mathrm{D}_{\frac{5}{2}}$, taking repeated same state measurements gives probability, which then in turn can be represented as a projection of state along σ_z axis [9]. Main advantage of such physical device is the accurate control of small state systems [9].

Superconductor quantum devices are favored physical implementation used by big tech companies [5]. An example case for quantum device could be a programmable superconducting processor *Sycamore* [2] introduced by Google. Device has 54 qubits in two-dimnensional space, where each qubit is linked together with 4 more qubits in rectangular manner which is depicted in Figure 3. According authors, in

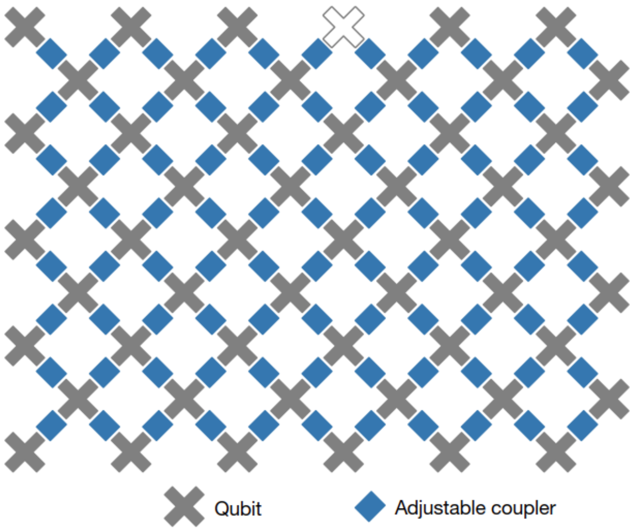

Figure 3: *Sycamore* processor qubit arrangement scheme [2].

given superconducting circuit electrons are condensed into quantum state, which in turn makes voltages and currents to act in quantum mechanical relevant way [2]. The given processor uses transmon qubits (transmonium qubit), where each qubit can be considered as a nonlinear resonators at 5-7Gz, here qubits are encoded as lowest quantum eigenstates of the circuit [2]. Each qubit here is connected to linear resonator which is used to read out qubit state [2]. Processor itself is put under 20mK temperature and is connected to attenuators to room-temperature electronics that give control signals [2]. Gates in such processor are executed as 25 ns microwave pulses that resonate with qubit frequency, whereas a state of qubit can be read in real time using frequency-multiplexing technique [2]. Authors in article review, that it is possible to have different measurement systems: single qubit, two-qubit. For actual output, they use cryogenic amplifiers to boost the signal by digitizing of 8 bits at 1Gz and then read using digital-to-analog converters of 14 bits at 1 Gz [2]. Key advantages of such quantum device is high-fidelity rate for few qubit operations in succession [2]. Currently, some of the highest qubit count available quantum devices are related to supercoducting physical architecture, for example, IBM's Heron r1 with 133 is based on its predecessor Eagle which is a superconductor quantum device [11].

[11]`https://docs.quantum.ibm.com/run/processor-types` last accessed 2024-01-03

As for photon based quantum devices we can take example from Hacker *et al.* [13]. Idea of using photon-photon for deterministic gate information transmission is limited by our ability to freely interact with two photons without photons shifting each other [13]. Authors outline one strategy to enable qubits based on photons for information transfer by taking two photons separately, affect first photon by medium, take this change to affect the second photon and then make first photon interact with the same medium again for gate reciprocity [13]. Furthermore, fundamental idea of using photons as qubits relies on analysing their polarization direction, Figure 4 [13]. In general sense, two photon carrying pulses (p_1 and p_2) enter a cavity (in this a three-dimensional lattice) cavity that has one ^{87}Rb (Rubidium) atom trapped inside, then consequently photons are reflected from given cavity by directing them to delay fibre [13]. While photons are in a delay fiber (time of 6μs), atom is read-out using fluorescence photons (blue arrows) by being directed towards single-photon detector (SPD), which is then moves towards field programmable gate array (FPGA) [13]. FPGA applies a conditional phase feedback to p_1, in the end initial photons leave the gate setup towards polarization analysers [13]. When fluorescence photons are measured, based on the result of their polarization, photon p_1 is phase shifted by π in its $|\mathrm{R}\rangle$ component (R here corresponds to right handed photon) by electro-optical modulator (EOM) [13]. Further qubit state encoding and usage depends on architecture of polarization of given photons [30].

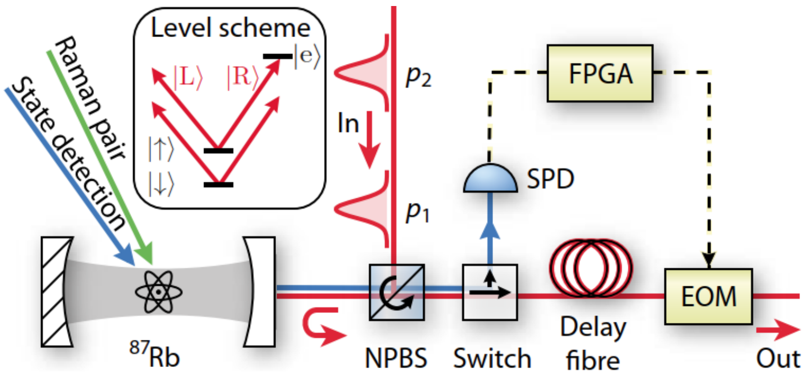

Figure 4: Setup for photon based qubit analyzer [13].

The other part of hardware used for quantum computation are simulators, where applying classical bits quantum behaviour is being simulated on classical computer. While one of the main advantages of such approach is noise-free environment, on the other hand simulators can simulate small quantum systems efficiently but their computational power is limited by classical hardware, as simulating large quantum systems becomes computationally very costly very fast. Johnson *et al.*, explains that both, simulators and computers are physical devices that reveal information about mathematical function, whereas difference flows out when we consider that function only as part of the physical model, then we are more likely to call device a simulator [17]. Model that describes the system and then emulates by taking output, not only describes model itself but the system of interest, if such simulator is considered as good, then we ought to assume it being good descriptor of real system of interest as well. Figure 5, shows simulator defining scheme, the outside comparison between real physical system and physical model could elevate consideration of simulator towards being actual computer. Fundamental difference is actual comparison and how well simulator represents system at hand and if accuracy of simulator can be relatively controlled and guaranteed then we can consider it as a computer [17].

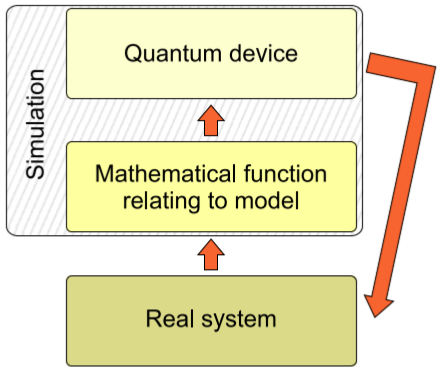

Figure 5: Role of quantum simulator [17].

Currently, there are multiple available simulators with varying approaches accessible through cloud computing: from IBM (*Statevector, Stabilizer, Matrix Product State* and others given, with a range of simulated qubits from 32 to 5000) [12]; Microsoft provides *Full state, Sparse, Toffoli* simulators with varying numbers of qubits [13]. Another thing that has to be noted is that each simulator is tied to software needed to simulate the given backend solutions; in this case, IBM has *Qiskit*, Microsoft has *Q#*, Google has *Cirq* [14]. In our case, we opt to discuss the PennyLane framework [15] that has multiple integrations with other simulators [5]. PennyLane is open-source Pyhton 3 framework performs the optimization of quantum and hybrid QC algorithms through differentiable quantum programming with extensions to several machine learning libraries: *Autograd, TensorFlow, PyTorch* and *Jax* [5]. This framework can be used with any gate based quantum computing platform as backend, whereas the goal of optimization in PennyLane is to find local of a cost function that is used to quantify a solution for given task via gradient-based algorithms, such as gradient descent and its variations [5]. PennyLane is not only designed to easily integrate with external quantum devices, but also has five in-build simulator devices:

1. *default.qubit* - A Python-based qubit statevector simulator, written in NumPy and other machine learning frameworks, therefore supports backpropagation ans is suited to work best with 0-20 qubits [5].

2. *default.mixed* - A Python-based qubit mixed-state simulator, written in NumPy [5].

3. *default.gaussian* - A Python-based continuous variable simulator, written using NumPy, and designed to support photonic-based quantum nodes. This device supports continuous-variable quantum circuits with Gaussian gates and measurements [5].

4. *lightning.qubit* - A high-performance qubit statevector simulator, written in C++. This device supports the adjoint method enabling extremely efficient optimization for quantum nodes with 20 or more qubits [5].

[12]`https://docs.quantum.ibm.com/verify/cloud-based-simulators` last accessed 2024-01-04
[13]`https://learn.microsoft.com/en-us/azure/quantum/machines/` last accessed on 2024-01-04
[14]`https://quantumai.google/qsim/cirq_interface` last accessed 2024-01-04
[15]`https://pennylane.ai/` last accessed 2024-01-04

5. *lightning.gpu* - A high-performance qubit statevector simulator, written using NVIDIA's cuQuantum SDK for GPU accelerated circuit simulation. As with lightning.qubit, adjoint differentiation is supported [5].

In this framework qubits are considered as wires in variational circuit on which gate operations are performed to manipulate quantum states of qubit, including logical operations, classical data embedding to quantum computational states and measurment calculations [5].

1.3 Qubit and Fundamental Gate Operations

Qubit is a unit of information in quantum computing just like bit is in classical computing. In classical computing a bit has two states of 0 and 1, whereas qubit also represent two states, but in addition to that it also exists in both states at the same time due to superposition [30]. In Dirac notation two state vectors, are called *basis states* and are expressed as $|0\rangle$ and $|0\rangle$ [33]. Together, two basis states represent the information stored by single qubit and form an orthonormal basis of a two-dimensional Hilbert space (in a generalized way, this is inner product space of two vectors that measure angle between them) which is called a *computational basis* [33]. In most general way a linear combination of two basis states form a superposition and qubit then is

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where $\alpha, \beta \in \mathbb{C}$ and are known as *amplitudes* [30, 33]. These amplitudes give an expression of "how much" qubit is in either state when it is measured [30] and because of normalization, amplitudes define probability of finding qubit in either state [30, 33].

$$\begin{aligned} |\alpha|^2 + |\beta|^2 &= 1 \\ p_{|0\rangle} &= |\alpha|^2 \\ p_{|1\rangle} &= |\beta|^2 \end{aligned}$$

Moreover, α and β can be negative and contain imaginary parts that do not influence probability of measurement, but do in fact allow to define phase differences between two states [30]. The Hermitian conjugate of above ket-state $|\psi\rangle$, the bra-state, can be expressed as

$$\langle\psi| = \alpha^*\langle 0| + \beta^*\langle 1|,$$

where * denotes complex conjugation [33]. Using Dirac vector notation, a single qubit can be expressed as

$$\alpha = \begin{pmatrix} \alpha_0 \\ \alpha_1, \end{pmatrix}$$

while Hermitian conjugate of such amplitude column vector is transposed and conjugated as row vector

$$\alpha^\dagger = \begin{pmatrix} a_0^*, & a_1^* \end{pmatrix} \in \mathbb{C}^2$$

moreover, two states can be represented as basis vectors of $\mathbb{C}^2$ as

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \in \mathbb{C}^2,$$
$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \in \mathbb{C}^2$$

[33]. System of two combined independent qubits (unentangled) can be algebraically represented as tensor product, taking for example one qubit from above, expressed as $|\psi_1\rangle = \alpha|0\rangle + \beta|1\rangle$ and second one, expressed as $|\psi_2\rangle = \gamma|0\rangle + \delta|1\rangle$ (here $\alpha, \beta, \gamma, \delta \in \mathbb{C}$), then the two-qubit system can be represented as

$$(\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle)$$

in such system where two qubit are independent two complex numbers per qubit are enough to describe the system, however in case of two qubit that have entangled states, meaning that amplitudes of each qubit are not independent, kets have to be represented in combined system as

$$\alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$$

and in such state kets represent states of both qubits, with four coefficients now representing four different possible states of combined quantum system [30]. Dirac notation also allows for simple description of inner product of two qubits in Hilbert space, taking for example same qubits $|\psi_1\rangle, |\psi_2\rangle$ mentioned earlier and taking into account that $|\alpha|^2 + |\beta|^2 = 1$ and $|\gamma|^2 + |\delta|^2 = 1$ where basis states of $|0\rangle$ and $|1\rangle$ are considered orthonormal, then we have that

$$\langle 0|0\rangle = \langle 1|1\rangle = 1, \quad \langle 0|1\rangle = \langle 1|0\rangle = 0,$$

therefore for inner product between $|\psi_1\rangle, |\psi_2\rangle$ is given by

$$\langle \psi_1|\psi_2\rangle = \alpha^*\gamma + \beta^*\delta$$

which is equivalent to the scalar product of two corresponding amplitude vectors [33]. In similar manner, we can represent outer product of two state compactly by

$$|\psi_1\rangle\langle\psi_2| = \begin{pmatrix} \alpha\gamma^* & \alpha\delta^* \\ \beta\gamma^* & \beta\delta^* \end{pmatrix}$$

where result is outer product of amplitude vectors [33]. Amplitudes act like variables of quantum algorithms where their magnitudes and relative phases shift, therefore considering a system of n qubits there would be 2^n amplitudes providing computational power whilst maintaining need of only n qubits to be manipulated [30]. Amplitudes themselves cannot be directly measured, meaning that amplitudes are forced to collapse to either 1 or 0, in addition, during measurement of qubits all quantum information is lost and the final output of quantum algorithm is at most n classical bits [30]. In addition, geometric representation of qubit can be parametrised as

$$|\psi\rangle = e^{i\gamma}\left(\cos\tfrac{\theta}{2}|0\rangle + e^{i\phi}\sin\tfrac{\theta}{2}|1\rangle\right)$$

where θ, ϕ, γ are real numbers satisfying $0 \leq \theta \leq \pi$ and $0 \leq \phi \leq 2\pi$ and $e^{i\phi}$ is global phase factor that has no observable effect. Moreover, angles θ and ϕ are spherical coordinates that can be used to interpret $|\psi\rangle$ as $\mathbb{R}^3$ vector $(\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\phi)$ and visualized using *Bloch sphere* as show in Figure 6 [33].

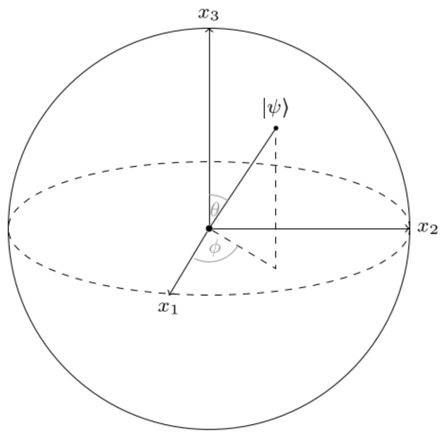

Figure 6: Representation of qubit using Block sphere [33].

Quantum logic gates are one of two of basic operation performed on qubits that are central to quantum computing which are realized by unitary transformations [33]. Unitary operation U is a linear transformation that preserves the inner product of vectors, that is important in order to describe quantum state evaluation and maintain consistency of amplitudes (probabilities) of each basis state) [33]. Single-qubit gates are 2x2 unitary operations which can be written as matrices, for example NOT gate that perform negation and return opposite output

$$NOT = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

applied to state $|0\rangle$ swaps its amplitudes to $|1\rangle$

$$NOT|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle,$$

such operation is unitary [33]. Gates X, Y and Z that are equivalent to *Pauli matrices* are used to rotate qubit along corresponding axis and are denoted as

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \sigma_y = \begin{pmatrix} 0 & -\imath \\ \imath & 0 \end{pmatrix}, \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

[33]. Hadamard gate, H has an effect on the basis states by creating a superposition of qubits

$$|0\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad |1\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

whilst its matrix form is represented as

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

[33]. Gate operations working on more than one qubit are imperative in order to change value of one qubit based on value of another qubit. The equivalent of NOT gate for one qubit, for two qubits is called $CNOT$ which can be represented by

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ , & & & \end{pmatrix}$$

such gate is only initialized when the first qubit is in state $|1\rangle$

$$|00\rangle \mapsto |00\rangle, |01\rangle \mapsto |01\rangle, |10\rangle \mapsto |11\rangle, |11\rangle \mapsto |10\rangle,$$

namely $CNOT$ are also used to fully entangle two qubits within variational circuit [33]. Such state can be achieved by taking Hadamard gate and applying to two qubit in states of $|0\rangle_1, |0\rangle_2$

$$CNOT(\frac{1}{\sqrt{2}}|0\rangle_2 \otimes |0\rangle_1 + \frac{1}{\sqrt{2}}|1\rangle_2 \otimes |0\rangle_1) = \frac{1}{\sqrt{2}}(|0\rangle_2 \otimes |0\rangle_1 + |1\rangle_2 \otimes |1\rangle_1)$$

such state is also called a *Bell state* [33]. Aside from CNOT there are also SWAP (exchanges states between qubits), Toffoli (three-qubit CNOT operation) and Fredrik (three-qubit operation gate that swaps the second and third qubits if the first qubit is in the state of $|1\rangle$) gates [33]. Gate opertions mentioned above have fixed parameters, however there are three *Pauli rotation* parameterized gates that are dependent on θ parameter:

$$R_x(\theta) = e^{-\imath\frac{\theta}{2}\sigma_x} = \begin{pmatrix} \cos\frac{\theta}{2} & -\imath\sin\frac{\theta}{2} \\ -\imath\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} R_y(\theta) = e^{-\imath\frac{\theta}{2}\sigma_y} = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} R_z(\theta) = e^{-\imath\frac{\theta}{2}\sigma_z} = \begin{pmatrix} e^{-\imath\frac{\theta}{2}} & 0 \\ 0 & e^{\imath\frac{\theta}{2}} \end{pmatrix}$$

[33]. General parameterized single-qubit rotation can be expressed as

$$R(\theta_1, \theta_2, \theta_3) = \begin{pmatrix} e^{\imath(-\frac{\theta_1}{2}-\frac{\theta_3}{2})}\cos\frac{\theta_2}{2} & -e^{\imath(-\frac{\theta_1}{2}+\frac{\theta_3}{2})}\sin\frac{\theta_2}{2} \\ e^{\imath(\frac{\theta_1}{2}-\frac{\theta_3}{2})}\sin\frac{\theta_2}{2} & e^{\imath(\frac{\theta_1}{2}+\frac{\theta_3}{2})}\cos\frac{\theta_2}{2} \end{pmatrix}$$

such gate rotations can be used as learn able parameters when building variational circuits and QML models [33]. By applying unitary Pauli gates, for example Pauli-z σ_z we can get estimate for qubit computational basis within range of [-1,1] [33].

To encode classical input data to encode data into amplitudes of quantum states for qubit systems we can apply several different approaches. We will review two approaches of *Basis embedding* and *Amplitude embedding*, however there are more strategies that can be applied such as: Qsample, Angle embedding, Time-evolution embedding, Hamiltonian embedding and possibly others [6, 33].

Basis embedding is a direct denomination of classical bits in to qubits in associative way, where information is portrayed in binary. Therefore, quantum state $|x\rangle, \text{where} x \in \mathbb{R}$, will refer to a binary representation of x with n number qubits that are embedding $|x\rangle$ computational state. Authors Schuld and Petruccione [33], outline a binary fraction representation example where each floating point number is within [0,1) and is represented as τ-bit string based on

$$x = \sum_{k=0}^{\tau-1} b_k \frac{1}{2^k}$$

given vector $x = (0.1, -0.6, 1.0)$ with binary fraction precision of $\tau = 4$, embedding results in the following

$$\begin{aligned} 0.1 &\rightarrow 0 \quad 0001\ldots \\ -0.6 &\rightarrow 1 \quad 1001\ldots \\ 1.0 &\rightarrow 0 \quad 1111\ldots \end{aligned}$$

with resulting concatenated binary vector being [00001 11001 01111], whereas quantum state
$|00001\ 11001\ 01111\rangle$ [33]. This example illustrates that in order to use binary embedding high number of qubits is necessary and it is hard to apply on near-term algorithms [33].

Amplitude embedding associates classical information by its wavefunction which defines the measurement probabilities. This way actual input values can be used to represent amplitudes in multi-qubit computational states[33]. In order to do that all input values at first have to be normalized to unit length and padded to a dimension of 2^n where n is the number of qubits used in the embedding (meaning, if input feature vector consist of 5 instances, then given $\log_2 5 = 2.321 \mapsto 3$ qubits are needed, then $2^3 = 8$ input vector needed to encode data [16]). Actual example by taking the same vector from basis embedding example would, needs to be normalized first and padded assuming two-qubit system $x = (0.073, -0.438, 0.730, 0.000)$, whereas quantum system can be represented as [33]

$$|\psi_x\rangle = 0.073|00\rangle - 0.438|01\rangle + 0.730|10\rangle + 0|11\rangle.$$

General mathematical notation for such embedding can be expressed as [33]

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_{2^n} \end{pmatrix} \leftrightarrow |\psi_x\rangle = \sum_{j=0}^{2^n-1} x_j |j\rangle.$$

[16]`https://docs.pennylane.ai/en/stable/code/api/pennylane.AmplitudeEmbedding.html` last accessed on 2024-01-04

1.4 Variational Quantum Algorithms

In NISQ era Variational Quantum algorithms (VQA) have emerged as leading strategy to in developing algorithms to achieve quantum advantage. VQAs were developed in mind to account for general constraints that arise from NISQ devices [6]:

1. variational circuits have adjustable parameters which allows to perform optimization tasks where number of such parameters are usually lower than number of qubits [33, 30].

2. CQ integration allows to process classical data into quantum computational basis states interchangeably, allowing for hybrid between quantum-classical algorithm combination (or even and extensions of such - meta learning, hybrid learning or transfer learning) [18, 34, 21, 6].

3. Shallow circuit depth usually is utilized in VQA, which helps limit errors arising from gate fidelities and coherence times using quantum hardware [33, 6, 4].

4. Given that any variational circuit can be tailored for specific task at the starting point of circuit itself, this helps to make NISQ more efficient [31, 32, 6].

One of main advantages of VQAs is that general structure is robust and is applicable for wide range of problems, in Figure7 we display general scheme provided by Cerezo *et al.* [6] which could be considered as extension of Figure 2. Here $C(\theta)$ is a cost function with parameters θ that encodes solution to the

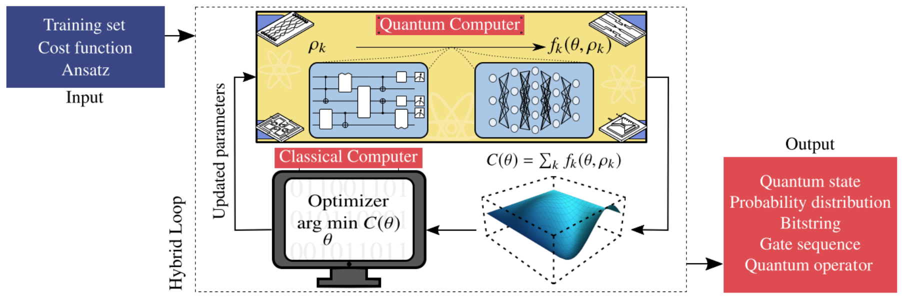

Figure 7: General VQA schematic diagram [6].

problem, ansatz (*german* "Ansatz" that translates to approach) depicting variational circuit structure (or so called parameterized quantum circuit) formed by gate operations whose parameters are trained to minimize the cost and ρ_k that denotes set of training data, such are considered as inputs [6]. Hybrid loop, with each iteration quantum computer (or simulator) calculates the cost which consecutively fed into classical computer that tries to optimize parameters whilst solving optimization problem [6]. Final output depends on task at hand where few possible types are depicted in red box. Cost function for variational circuit can be displayed in generalized manner

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

where f represent some function to minimize, $U(\theta)$ is a parameterized unitary, ρ_k are input states from training set and O_k is a set of observables [6]. Aforementioned parts can be further modified to in order to achieve required optimization tasks.

1.4.1 Universal Quantum Classifier with Data re-uploading

No-cloning theorem in quantum computing refers to quantum states being non-recoverable after measurement is performed, therefore data is lost. Algorithm with data re-uploading tries to solve such issue in quantum computing. Universal quantum classifier introduced by Perez-Salinas *et al.*, is constructed in a bit different way compared to standard parameterized quantum circuit structure - in standard circuit structure is divided in separate data encoding, uploading and processing parts, whereas in this particular algorithm proposal, all parts implemented multiple times [29]. Authors argue that there is no to use complex encoding circuits or high number of qubits in order to uncover non-linearities in data as single-qubit rotations applied multiple times along the circuit can generate sufficient amount of functions of data values [29]. In essence, such classifier‘s structure is similar to neural network structure, where each data point is introduced to each hidden layer. Figure 8 outlines similarity explained [29]. Single-qubit and multi-qubit ansatz are viable to be used with such strategy,

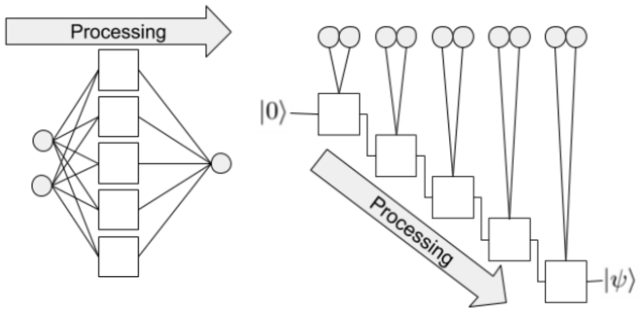

Figure 8: Simplified working schemes of Neural network (left) and Quantum classifier (right) [29].

where performance of the circuits are quantified by minimizing some objective function χ^2. Taking example of single-qubit classifier authors introduce processing gates starting with single-qubit special unitary group (SU) rotations $U(\phi_1, \phi_2, \phi_3) \in \mathrm{SU}(2)$, where $U(\vec{\phi})$, $\vec{\phi} = (\phi_1, \phi_2, \phi_3)$, here SU(2) are complex 2x2 matrices that are unitary, whereas vector of $\vec{\phi}$ are determine angle rotations that are performed by Pauli matrices σ [29]. Then structure of single-qubit classifier can be denoted as

$$\mathbf{U}(\vec{\phi}, \vec{x}) \equiv U(\vec{\phi_N})U(\vec{x}) \dots U(\vec{\phi_1})U(\vec{x}),$$

here data is re-uploaded each time on $U(\vec{\phi_i}, \vec{x})$, therefore concept of processing layer is introduced as $L(i) \equiv U(\vec{\phi_i}, \vec{x})$ [29]. Classifier then acts on single qubit in the following way

$$|\psi\rangle = \mathbf{U}(\vec{\phi}, \vec{x})|0\rangle.$$

Ansazt of such classifier then can be depicted as a combination of processing layers. Figure 9 shows such circuit structure, where layers are classifier building blocks [29]. Quantum circuit can be made more

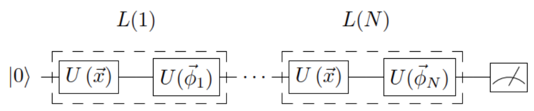

Figure 9: Single qubit scheme with data re-uploading through processing layers [29].

compact, if processing angles and data inputs are incorporated in a single processing step, including weights w_i, where weights are applied to each data point through by taking Hadamard product of two vectors $\vec{w_i} \circ \vec{x} = (w_i^1 x^1, w_i^2 x^2, w_i^3 x^3)$ [29]. This way, weights play similar role as they do in artificial neural networks and processing layer can be expressed as

$$L(i) = U(\vec{\theta_i} + \vec{w_i} \circ \vec{x}),$$

θ_i here represent series of processing angles mentioned above. In case data input dimension is less than three, then components can be set to 0, reducing circuit‘s depth, on the other hand, if higher dimensionality data is ought to be used, then processing layer definition can be expanded by dividing each observation into k vectors of dimension three, resulting in processing layer structure to

$$L(i) = U(\vec{\theta_i}^k + \vec{w_i}^k \circ \vec{x}^k) \dots U(\vec{\theta_i}^1 + \vec{w_i}^1 \circ \vec{x}^1).$$

With such setup, after k iterations all data points have been put through from one observation [29].

Results of each measurement are used to compute χ^2 in order to minimize it, however it is important to understand how assignment of classes is performed. Authors say that fundamental guiding principle is maximum orthogonality between received outputs [29]. This is easily done for binary classification, where it can be done by comparing probabilities of each class as $P(0)$ for $|0\rangle$ and $P(1)$ for $|1\rangle$ by comparing $P(0) > P(1)$ and assigning class A or B respectively [29]. Moreover, bias λ can also be introduced resulting in $P(0) > \lambda$ if true, class A or B otherwise. For multi-class classification, authors defined two strategies, similarly as comparing probabilities between themselves, it is possible to compare $P(0)$ with respective number of sectors $0 \leq \lambda_1 \leq \dots \leq \lambda_i \leq 1$, where number of classes is $i+1$ [29]. Second strategy is to compute overlapping states of class set, that is based on maximum orthogonality, expressed in Bloch sphere [29].

Cost function is minimized by tuning $\vec{\phi_i}$ parameters. Two cost function are introduced: simple fidelity function that ought to be maximized and is based on measuring angular distance the class label state and the data state; and weighted fidelity cost function that is refined first fidelity function that has to be minimized [29]. Simple fidelity function introduced as average fidelity between states at the end of circuit

$$\chi_f^2(\vec{\theta}, \vec{w}) = \sum_{\mu=1}^{M} (1 - \langle \tilde{\psi_s} | \psi(\vec{\theta}, \vec{w}, \vec{x_\mu}) \rangle |^2),$$

here $|\tilde{\psi_s}\rangle$ corresponds to correct label state of observation μ [29]. Modified weighted fidelity function separates from simple one, by having additional weights introduced that are assigned to each class and can be optimized together with $\vec{\theta}$ and $\vec{w}$ parameters [29].

Results introduced by authors show that in general increasing number of processing layers increased circuit performance until model comes into stationary regime. Also, introducing more qubits and entangling, stationary regime is reached with less processing layers [29].

1.4.2 Quantum Support Vector Machines

Earliest works related QSVMs were proposed in 2014 and since various modifications were introduced [16]. Classical SVMs are well-known algorithms that are widespread and applicable even with small data sets. Main idea for SVMs is based on maximum-margin hyperplanes (decision boundaries) that separate data points between different data classes, however with increasing data size, computational complexity raises, therefore reaching optimal hyperplane become extensively hard, Quantum SVM aims to overcome such limitation [16]. Since QSVM are popular there multiple theoretical and already implementable solutions or approaches for QSVMs, such as *SVM with Quantum Kernel Estimation, SVM with quantum matrix inversion method, the feature mapping method*, we will review two most recent ones - QSVM with quantum kernel approach and QSVM variational approach [16, 20].

In general approach, standard QSVM structure is outlined in Figure 10, here data is encoded by appropriate embedding technique or custom circuit [16]. Quantum Kernel Support Vector machine

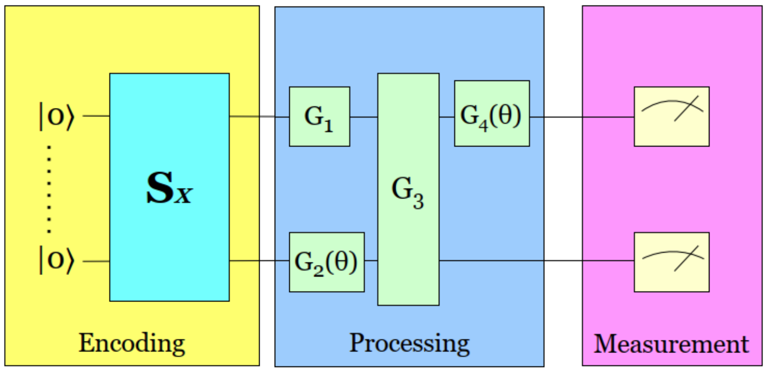

Figure 10: Generalized model description for QSVMs [16].

(QKSVM) in essence utilizes $SWAP$ tests to evaluate the quantum kernel, which helps to reduce qubits needed due to similarity measures. If we were to consider Figure 10, then this model would only use encoding and measurement parts only. The purpose of kernel function in such structure is to measure similarity between two data points in high-dimensional feature space and do it by mapping data to space where linear classifier can separate classes [16]. Mathematically, such kernel can be represented as

$$k(x_1, x_2) = |\langle\phi(x_1)|\phi(x_2)\rangle|^2,$$

where x_1, x_2 are input feature vectors and $\phi_{1,2}$ are quantum embedding procedures mapping data features to quantum states, Innan *et al.* explicitly outlines angle embedding technique [16]. Important to note, embedding technique and then inverse embedding for each quantum state is needed in order to compute overlap between two states using SWAP test [16]. Swap test can be represented by the following equation

$$\langle SWAP\rangle = |\langle\phi(x_1)\otimes\phi(x_2)|SWAP|\phi(x_1)\otimes\phi(x_2)\rangle|^2,$$

$|\langle SWAP\rangle|^2$ represents probability of measuring two quantum embedded states in the same state [16]. Hermitian observable is used as projector to initial state.

Quantum Variational Support Vector Machine (QVSVM), differs from previous model by having data training procedure done within model ansatz. Model is created through multiple layers, however as in standard parameterized circuits cost function has to be computed outside ansatz by classical means. In case of this model all parts are used from Figure 10. Given parameterized circuit, with unitary operator, we need to find such values of θ that minimize the cost function that can be expressed as

$$C(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, U(\theta)x_i).$$

Here unitary operator is represented by $U(\theta)$, where θ is vector with optimizable model parameters x_i and y_i are inputs and outputs, respectively [16]. We measure difference between predicted and actual value $L(y, y^{'})$ [16]. Cost function typically is calculated classical computer in conjunction, optimization algorithms can be used together.

1.4.3 Quantum K Nearest Neighbour Algorithm

The most simple example of Quantum K Nearest Neighbour (QKNN) algorithm can be taken from Basheer *et al.* [3] QKNN with fidelity as similarity measure. Main idea of such algorithm is to find k nearest neighbours of $|\psi\rangle$ using training data and then assign some label to same $|\psi\rangle$ based on majority voting [3]. Fidelity function for such model is calculated between $|\psi\rangle$ and test set $|\phi_j\rangle$ and can be expressed as

$$F_j \equiv |\langle\psi|\phi_j\rangle|^2,$$

assuming that j^{th} some train state number [3]. In case k is unknown then quantum search algorithm is needed to find optimal k. To do that, randomly initialized matrix A can be taken for initial step to select some T_y as threshold index, then using some search algorithm (Grover‘s search algorithm is applicable here), we search for elements that are nearest to the T_j state (such state can be initialized randomly at first), then if $T_j > T_y$ value of 1 is assigned, 0 otherwise [3]. Given that $T_j > T_y$ y is satisfied, value from test set is replaced with new one identified during test set comparison in A. Fidelity between two arbitrary states are performed by utilizing CSWAP two gates for two basis states, which is three register gate using $|0\rangle$ and then two compared states $|\psi\rangle, |\phi\rangle$, afterwards measurement of probabilities is possible [3].

$$P(0) = \frac{1}{2} + \frac{1}{2}|\langle\psi|\phi\rangle|^2,$$
$$P(1) = \frac{1}{2} - \frac{1}{2}|\langle\psi|\phi\rangle|^2.$$

$P(0) - P(1)$ gives fidelity value, which then can be used as measure to map $|\psi\rangle$ to new set by maximizing said fidelity using majority voting [3].

1.4.4 Quantum Neural Networks

Farhi and Neven [10] outline a general structure for Neural Network model used for supervised binary classification learning. In essence, quantum circuit consists of sequence of unitary transformations with

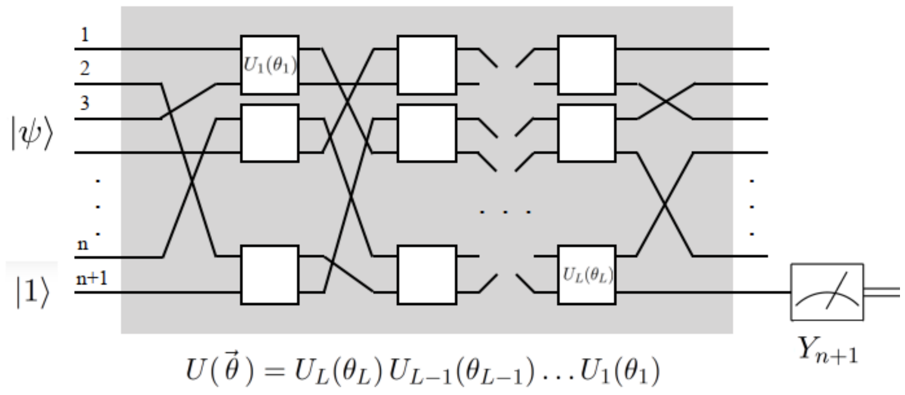
1
2
3
$|\psi\rangle$
n
n+1
$|1\rangle$
$U_1(\theta_1)$
$U_L(\theta_L)$
Y_{n+1}
$U(\vec{\theta}\,) = U_L(\theta_L)\,U_{L-1}(\theta_{L-1})\ldots U_1(\theta_1)$

Figure 11: General scheme for QNN [10].

dependent parameters. Figure 11 shows general scheme of such model. In figure, we see that model works on $n + 1$, n being total number of observations, last qubit acts as read-out. Basic unitary operation can take predefined and required number of parameters θ, in this example only one is taken [10]. L is set by user in order to set length of unitary operation network, in turn as well length of $\vec{\theta} = \theta_L, \theta_{L-1}, \ldots, \theta_1$. $U(\vec{\theta})$ denotes unitary operations performed with taken parameters, as it seems from Figure 11, all unitary operations are intertwined. Computational basis is constructed by appending readout bit and setting it to value of 1, resulting in $|x, 1\rangle = |x_1, x_2, \ldots, x_n, 1\rangle$, altogether authors denote combined unitary as $U(\vec{\theta})|x, 1\rangle$ [10]. θ_i are adjusted during learning in such way that measurement qubit aims to produce desired label for given input state $|\psi\rangle$, also, measurement itself is performed by utilizing Pauli operators (in this particular example, σ_y is used) [10]. Predicted label value will be number in range of [-1,1].

2 Experiment and Analysis

In order to test classification capabilities of quantum variational circuit algorithms we have created a benchmark setup of quantum support vector machine model with margin loss for 4 different data set, to test against classical classification algorithms. Benchmark setup was focused at supervised multi-class classification tasks.

Classical algorithms were created using Pycaret library [17]. Quantum algorithm was implemented using PennyLane framework [5] and with help of its several templates [18].

Four heterogeneous data sets were fitted to each model with data pre-processing and normalization steps outlined in below section (2.2). Performance was tested using standard metrics outlined in section (1.1) utilizing mainly scikit-learn library [26].

Expected experiment outcomes were:

- to have a consolidated table per data set, outlining selected measures returned for direct comparison between classical and quantum;
- utilize PennyLane for quantum model creation and manipulation;
- based on returned measures assess performance results and provide recommendations.

2.1 Programming Environment and Carcass for Experimentation

All codes can be accessed through Vilnius University, faculty of Mathematics and Informatics GitLab repository [19]. Experiment was carried in Python 3 programming language using Jupyter Notebooks. Codes themselves were ran on Google Colab environment [20], Linux based VM with integrated Python 3 environment. CPU consisted of two Intel(R) Xeon(R) CPU @ 2.20GHz processors.

2.2 Data sets

In this experiment we have used 4 data sets for all models. Table 1 refers to original data set parameters. We have used similar layout to present our data sets as was done by Pakrashi *et al.* [24]. In Table 1, *Observations* and *Inputs* refer to overall available data set size and input size used in this experiment, respectively. *Labels* denote overall available distinct label classes available. *Features* show miscellaneous information about available predictor variables. *ClassIR* is class imbalance ratio, that describes individual class distribution throughout the given data set and then average is taken from all classes. *MaxIR* is maximum imbalance ratio, the describes a ratio between most frequent specific label count recurrence of class in data set and least frequent occurrence count, which helps to understand imbalance between class recurrence. Class imbalance ratios and maximum imbalance ratios, were calculated before data preprocessing pipeline. Data preprocesing, generally performed in the same way for all data sets. The steps were as follows:

1. Import data sets according input size mentioned above.

[17]`https://pycaret.gitbook.io/docs/` last accessed on 2024-01-06
[18]`https://pennylane.ai/qml/demonstrations/` last accessed on 2024-01-06
[19]`https://git.mif.vu.lt/skpa8400/masters_23_skalvis/-/tree/master`
[20]`https://colab.research.google.com/` last accessed on 2024-01-06

2. Dependent variables (classes) were converted to natural numbers $\mathbb{N} = [0, 1, 2, \dots, N-1.]$. Where each number represents separate class and N represents number of class labels in data set. No additional modifications were performed on label sets.

3. Only in MNIST data set. Singular Value Decomposition (SVD) and t-Distributed Stochastic Neighbor Embedding (t-SNE) were taken from scikit-learn machine learning library [26] and were used to reduce 784 dimension feature vectors to 2 dimension feature vectors [21]. Figure 13, shows class distribution pattern after dimension reduction.

4. All predictor variable vectors (features) were normalized - divided by their L2 norms:

$$norm = \sqrt{\sum_{i=1}^{N} X_i^2} \quad \text{and} \quad X_{norm} = \frac{X}{norm} \tag{1}$$

   here N and X_i are a number of features and an instance in feature row vector, respectively. Whereas, X_{norm} and X are normalized and original feature row vectors.

5. Split data sets with train ratio set being at 0.75 and class distribution equally.

Data set	Observations	Inputs	Labels	Features	ClassIR	MaxIR
Iris	150	150	3	4(Continuous)	0.333	1
Wine Quality	6497	1000	2	11(Continuous)+1(Ordinal)	0.5	2.802
Circles	2000	2000	2	2D X and Y coordinates	0.5	1.027
MNIST	70000	1000	10	28 x 28 gray-scaled images reduced to 2 Continuous features	0.1	1.395

Table 1: Summary of Data sets.

Iris flower or Fisher‘s data set is a well-known multivariate data set introduced by Ronald Aylmer Fisher in 1936 [11]. Data set contains 50 observations for 3 different species. There are 4 columns representing length and width of petals and sepals in centimeters. Based on these features data set is widely used to test classification algorithms. Iris data set was taken as-is with only applying train-test split and L2 norm normalization. Data was taken from scikit-learn machine learning library [26]. *Wine Quality* data set is a combination of two data sets - red and white variants of the Portuguese "Vinho Verde" wine data [7]. It is used mostly for regression and classification tasks. Data set was taken from UCI Machine Learning Repository (ML repo) [7]. Sample input size reduced from 6497 to 1000 observations and then general train-test split, L2 norm normalization applied. *Modified National Institute of Standards and Technology (MNIST)* database data set is collection of 70000 handwritten digits. Each image is in grayscale and represented by 28 by 28 pixels where pixel represents how dark or light pixel is. The original NIST data was preprocessed by Yann LeCun and colleagues at AT&T Bell Labs [19]. Data set was downloaded from UCI ML repo [22], originally introduced by [19]. Overall,

[21] example for dimensionality reduction taken from `https://github.com/Qiskit-Challenge-India/2020blob/1ae3c7cb7819a1f2d3d26acd497069c1d118e92f/Day\%206\%2C\%207\%2C8/VQC\_notebook.ipynb`, last accessed on 2023-12-27

[22] Hyperlink (`https://yann.lecun.com/exdb/mnist/`) provided by the original authors, was not accessible, last checked 2023-12-27

input size reduced from 70000 to 1000, then SVD and t-SNE used to reduce dimensionality. Lastly, Train-test split and normalization was applied. *Circle* data set was manually generated reusing code provided by PennyLane tutorial [1]. Data set consists of 2 features that denote coordinates in 2D space. Each observation can be assigned to one of two classes, 0 or 1, red or blue, respectively. Moreover, classification label is given based on location of observation: if observation is found within circle, blue label (1) is assigned, if not, red (0). Circle is centered on $[0.0, 0.0]$ with radius of $\sqrt{\frac{2}{\pi}}$. Coordinate values are (pseudo) randomly generated 2 dimensional vector, which is subtracted by center vector and Frobenius norm [23] calculated on the result. Norm is then compared against radius as denoted in equation 2. Here $[x_{i,1}, x_{i,2}], y_i$ are i observation feature vector x and label y. The Frobenius norm of a vector A is denoted as $\|A\|_F$.

$$\|A\|_F = \begin{cases} y_i = 0, \|A([x_{i,1}, x_{i,2}] - [0.0, 0.0])\|_F > \sqrt{\dfrac{2}{\pi}}, \\ y_i = 1, \|A([x_{i,1}, x_{i,2}] - [0.0, 0.0])\|_F < \sqrt{\dfrac{2}{\pi}} \end{cases} \tag{2}$$

2000 observations were generated with 75:25 train - test split ratio as depicted in Figure 12. L2 norm normalization was applied in same manner as in previous data sets. Due to nature of this data set, we consider classification task performed with this data set as non-linear as we will see later in results.

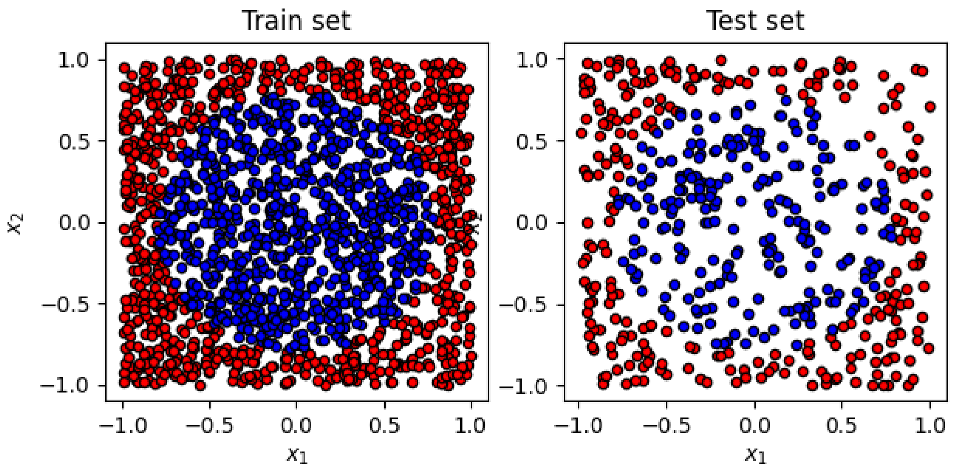

Figure 12: Circles data set split.

2.3 Quantum Support Vector Machine

Model structure inspired and taken from Shuld *et al.* [32]. The implemented model is a supervised multi-class classifier with margin loss function. Model code backbone was taken from PennyLane Multi-class margin classifier tutorial [14]. Final code can be found in MIF GitLab [24].

Classifier performs as multiple one-vs-all classifier with margin loss [32, 14]. Classifier to each class

[23]`https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html` last accessed 2023-12-28
[24]`https://git.mif.vu.lt/skpa8400/masters_23_skalvis/-/blob/master/Qsvm_marginloss.ipynb`

is constructed using QNodes [25]. The output of the ith classifier, c_i on input x is interpreted as a score, s_i between [-1,1] [14]. Where score calculated by

$$s_i = c_i(x; \theta).$$

Loss function, Multi-class Support Vector Machine loss (SVM loss or Margin loss) [26], for observations x can be denoted as

$$L(x, y) = \sum_{j \neq y} \max(0, s_j - s_y + \Delta),$$

here x denotes observation feature vector, y class label. Δ denotes margin (by standard, we have applied value of 0.15).

QSVM model for each data set was adjusted based on parameters listed in Table 2, therefore each data set had slightly different ansatz. This was required in order to fit all different input features, whilst using amplitude embedding. Number of qubits had to be rounded up to next integer by taking $\log_2$(Feature size), each feature for observation vector is represented as amplitude value. Layers, in this model context, correspond to each gate operation can but not necessarily be used in circuit. Good example would be differences between Iris ansatz in Figure 14 and MNIST ansatz in Figure 16, where Iris data set is ran using two qubit system with entanglement layer and MNIST being ran only on single-qubit system, therefore entaglement layer is skipped. Remainig ansatz for Wine Quality and Circle data sets can be found in Figures 15 and 17, respectively. For all data sets we applied Adaptive Moment Estimation (Adam) optimization algorithm from PyTorch framework [27] with learning rate of 0.01. Number of parameters, were derived as general rule, based on input parameters: $3 \cdot \text{layers} \cdot \text{qubits} \cdot 3 + 3$, giving number of learnable model parameters needed for each data set. Further results are discussed in (2.4).

Parameters	Iris	Wine Quality	MNIST	Circles
Classes	3	2	10	2
Sample size	150	1000	1000	2000
Features	4	12	2	2
Qubits	2	4	1	1
Iterations	100	10	10	50
Layers	6	6	6	6
Batch size	10	10	10	10
Train Split	0.75	0.75	0.75	0.75
Learning rate	0.01	0.01	0.01	0.01
Number of parameters	111	219	57	57

Table 2: QSVM model parameters.

[25]`https://docs.pennylane.ai/en/stable/code/api/pennylane.QNode.html` last accessed on 2024-01-07
[26]`https://cs231n.github.io/linear-classify/` last accessed on 2024-01-07
[27]`https://pytorch.org/docs/stable/generated/torch.optim.Adam.html` last accessed on 2024-01-06

2.4 Benchmark Results

Macro average test set accuracy was considered as main parameter to select best model. In addition to already outlined measures in (1.1) elapsed **training time** (TT) (in seconds, s) for training and inference combined, was included to assess model time efficiency between classical and quantum models, with intention to consider less time needed as more favorable measure. Pycaret setup allowed to construct and extract best classical models with simple pipeline setup, whereas best quantum model had to be manually checked for highest accuracy.

QSVM model best accuracy was found with Iris data set, accuracy reaching 0.9737 after 52 iterations. Table 3 shows overall comparison. Ansatz for Iris was ran for 100 iterations, where relative model stationary is hard to notice as swings in accuracy are substantial and can be seen in Figure 18, on the other hand, swings in cost were more contained throughout training procedure. Given small sample size and data set characteristics we can say that in contrast to classical algorithms model performed well, in terms of accuracy performing better than 14 out of 15 classical algorithms. Moreover, model outperformed all classical algorithms in terms of correctly identifying true cases as, recall measure was highest and highest F1score indicated best balance between precision and recall.

Model	Accuracy	Recall	Precision	F1score	TT (s)
QSVM	0.9737	**0.9762**	0.9744	**0.9743**	1510.351
Ada Boost Classifier	0.9371	0.9371	0.9568	0.9354	0.170
Decision Tree Classifier	0.9561	0.9561	0.9658	0.9554	0.028
Dummy Classifier	0.3394	0.3394	0.1164	0.1731	0.028
Extra Trees Classifier	**0.9742**	0.9742	**0.9794**	0.9738	0.165
Extreme Gradient Boosting	0.9553	0.9553	0.9671	0.9553	0.065
Gradient Boosting Classifier	0.9553	0.9553	0.9611	0.9548	0.362
K Neighbors Classifier	0.9470	0.9470	0.9585	0.9462	0.043
Light Gradient Boosting Machine	0.9280	0.9280	0.9402	0.9276	0.141
Linear Discriminant Analysis	0.9561	0.9561	0.9653	0.9552	0.029
Logistic Regression	0.8402	0.8402	0.8716	0.8207	0.669
Naive Bayes	0.9644	0.9644	0.9724	0.9639	0.030
Quadratic Discriminant Analysis	0.9652	0.9652	0.9726	0.9647	0.030
Random Forest Classifier	0.9735	0.9735	0.9788	0.9728	0.198
Ridge Classifier	0.7144	0.7144	0.6892	0.6422	**0.027**
SVM - Linear Kernel	0.7152	0.7152	0.5893	0.6317	0.033

Table 3: Benchmark results on Iris data set.

However, that is not the case for other data sets. Wine Quality data set for QSVM model outperformed only one classical algorithm in terms of accuracy, but overall had the worst performance based on all remaining measures, given that model was trained only on 10 iterations (same as classical counterparts), it is possible that increasing iteration count we could have seen model becoming stationary, due to best model performance being at iteration 10 as can be seen in Figure 19, therefore assuming that stationary model was not reached. Only 10 iterations were selected, as each iteration on average took 2 minutes to train. Table 4 depicts received results. It is worth to mention that Wine Quality data set had the highest dimensionality in terms of feature size, however ansatz for training the model had to create only 2 classifiers with 4 qubits each (resulting in the end with the highest amount learnable parameters needed of 219).

Model	Accuracy	Recall	Precision	F1score	TT (s)
QSVM	0.7920	0.6338	0.8874	0.6477	1199.929
Ada Boost Classifier	0.9667	0.9768	0.979	0.9777	0.149
Decision Tree Classifier	0.9573	0.9697	0.9734	0.9712	0.035
Dummy Classifier	0.7493	**1**	0.7493	0.8567	**0.023**
Extra Trees Classifier	0.9720	0.9839	0.9790	0.9813	0.189
Extreme Gradient Boosting	**0.9800**	0.9911	0.9825	**0.9867**	0.121
Gradient Boosting Classifier	0.9720	0.9857	0.9773	0.9814	0.270
K Neighbors Classifier	0.9280	0.9679	0.9393	0.9529	0.045
Light Gradient Boosting Machine	0.9773	0.9857	**0.9842**	0.9849	0.412
Linear Discriminant Analysis	0.9360	**1**	0.9230	0.9595	0.028
Logistic Regression	0.8773	0.9893	0.8671	0.9239	0.653
Naive Bayes	0.9413	0.9768	0.9470	0.9615	0.027
Quadratic Discriminant Analysis	0.9587	0.9679	0.9770	0.9723	0.028
Random Forest Classifier	0.9707	0.9840	0.9772	0.9805	0.361
Ridge Classifier	0.8880	0.9893	0.8787	0.9302	0.043
SVM - Linear Kernel	0.9160	0.9821	0.9132	0.9461	0.044

Table 4: Benchmark results on Wine Quality data set.

MNIST data set ran on QSVM showed unsatisfactory results and compared to classical counterparts outperformed only one classical model. From Table 5, we can see that model inference power was slightly better than random guessing as accuracy was only slightly above 0.1610 considering classification task for 10 classes. Comparing with Quadratic Discriminant analysis model, which had highest accuracy, QSVM for MNIST was three times worse based on all performance measures. Based on Figure 20, we can see best performance was reached at 5th iteration, while maintaining similar levels of accuracy afterwards. Circles data set from its own structure, having hardly any linear trends, was complex

Model	Accuracy	Recall	Precision	F1score	TT (s)
QSVM	0.1640	0.1861	0.1158	0.0585	1624.068
Ada Boost Classifier	0.3213	0.3213	0.1581	0.2070	0.138
Decision Tree Classifier	0.4800	0.4800	0.4894	0.4714	0.035
Dummy Classifier	0.1227	0.1227	0.0151	0.0268	0.027
Extra Trees Classifier	0.4773	0.4773	0.4806	0.4687	0.220
Extreme Gradient Boosting	0.4760	0.4760	0.4935	0.4703	0.461
Gradient Boosting Classifier	0.4853	0.4853	0.4879	0.4761	1.474
K Neighbors Classifier	0.4960	0.4960	**0.4936**	**0.4809**	0.074
Light Gradient Boosting Machine	0.4787	0.4787	0.4859	0.4722	1.402
Linear Discriminant Analysis	0.4720	0.4720	0.3871	0.3891	0.032
Logistic Regression	0.4880	0.4880	0.3856	0.3983	0.077
Naive Bayes	0.5133	0.5133	0.4580	0.4495	0.048
Quadratic Discriminant Analysis	**0.5200**	**0.5200**	0.4740	0.4625	0.030
Random Forest Classifier	0.4787	0.4787	0.4812	0.4697	0.259
Ridge Classifier	0.4080	0.4080	0.2780	0.3071	**0.027**
SVM - Linear Kernel	0.3720	0.3720	0.2415	0.2739	0.044

Table 5: Benchmark results on MNIST data set.

task for both: classical and QSV models. From Figure 21, we can see that QSVM model struggled to converge and best model accuracy was reached at 49th iteration. Increasing iteration count could have improved model performance, however similarly as with Iris data set QSVM model, each iteration took around 46 seconds. Table 6 shows that QSVM outperformed 12 out of 15 based on accuracy and all based on recall. Important to note, classical models‘ inference was also lacking and only slightly better than random guessing.

Model	Accuracy	Recall	Precision	F1score	TT (s)
QSVM	0.5547	**0.5562**	0.5658	0.5386	2313.797
Ada Boost Classifier	0.5767	0.4985	**0.5793**	0.5350	0.138
Decision Tree Classifier	0.5473	0.5462	0.5362	0.5406	0.029
Dummy Classifier	0.5107	0	0	0	**0.023**
Extra Trees Classifier	0.5480	0.5353	0.5362	0.5345	0.335
Extreme Gradient Boosting	0.5493	0.5315	0.5410	0.5354	0.071
Gradient Boosting Classifier	**0.5773**	0.5093	0.5785	0.5411	0.218
K Neighbors Classifier	0.5607	0.5408	0.5521	**0.5459**	0.051
Light Gradient Boosting Machine	0.5540	0.5204	0.5466	0.5328	0.677
Linear Discriminant Analysis	0.4913	0.3925	0.4771	0.4297	0.046
Logistic Regression	0.4913	0.3925	0.4771	0.4297	0.048
Naive Bayes	0.4967	0.4210	0.4839	0.4492	0.029
Quadratic Discriminant Analysis	0.4953	0.4306	0.4809	0.4535	0.025
Random Forest Classifier	0.5527	0.5489	0.5411	0.5440	0.329
Ridge Classifier	0.4913	0.3925	0.4771	0.4297	0.025
SVM - Linear Kernel	0.5213	0.3930	0.4134	0.3924	0.027

Table 6: Benchmark results on Circle data set.

Lastly, we notice that training time is equally linearly interdependent between feature dimension and class count, as comparing training time needed for Wine Quality and MNIST, we notice the same amount of learnable model parameters, where class label count (2 vs 10) and feature count (12 vs 2) were fundamental differences. Interesting to note that based on training time QSVM for Wine Quality working on 4 qubits took 25% less time than QSVM for MNIST working on 1 qubit ansatz, considering that QSVM Wine Quality had 4 time more learnable parameters (in turn higher number of operations to perform).

Results and Conclusions

Aim of this thesis was to investigate and understand currently available quantum classification algorithms. Moreover, in order to test rather new quantum PennyLane framework, experiments were carried using only available material from PennyLane library. Apart from quantum algorithms of well known classical counterparts, we found universal classifier with novel approach using data re-uploading that is exclusive to PennyLane framework. The overall achieved results are:

1. Created Quantum Support Vector variational classifier and applied to 4 heterogeneous data sets, creating 4 separate 2-qubit, 4-qubit and two 1-qubit systems/ansatz.
2. QSVM with 219 learnable parameters and 12 features took 424.139 (s) less time to train than QSVM model with 57 parameters and 2 features. TT 1199.929 (s) and 1624.068 (s), respectively.
3. Best QSVM performance was noticed for Iris data set (Accuracy: 0.9737, Recall : 0.9762, Precision : 0.9744 and F1Score : 0.9743), same quantum model outperformed also 14 out of 15 classical algorithms for given data set.
4. Worst QSVM performance was noticed for MNIST data set, which was outperformed by 12 out of 15 classical algorithms and QSVM for Wine Quality data set which was outperformed by 14 out of 15 classical algorithms.
5. QSVM for non-linear Circle data set returned similar results comparing with classical algorithms.

From the results we could draw such conclusions:

1. In general classical models outperform QSVM model with margin loss.
2. No one classical algorithm outperforms other classical counterparts on all data sets.
3. Running model on near-term quantum device could improve model performance.
4. Increasing iteration count for training procedure might improve overall QSVM accuracy.

References

[1] Shahnawaz Ahmed. Data-reuploading classifier. `https://pennylane.ai/qml/demos/tutorial_data_reuploading_classifier/`, 09 2019. Date Accessed: 2023-12-27.

[2] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandra, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510, 2019.

[3] Afrad Basheer, A. Afham, and Sandeep K. Goyal. Quantum k-nearest neighbors algorithm, 2021.

[4] Michel Le Bellac. *A SHORT INTRODUCTION TO QUANTUM INFORMATION AND QUANTUM COMPUTATION*. Cambridge University Press, Cambridge, United Kingdom, 2006. 179 p., translated by Patricia de Forcrand-Millard.

[5] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M. Sohaib Alam, Guillermo Alonso-Linaje, B. AkashNarayanan, Ali Asadi, Juan Miguel Arrazola, Utkarsh Azad, Sam Banning, Carsten Blank, Thomas R Bromley, Benjamin A. Cordier, Jack Ceroni, Alain Delgado, Olivia Di Matteo, Amintor Dusko, Tanya Garg, Diego Guala, Anthony Hayes, Ryan Hill, Aroosa Ijaz, Theodor Isacsson, David Ittah, Soran Jahangiri, Prateek Jain, Edward Jiang, Ankit Khandelwal, Korbinian Kottmann, Robert A. Lang, Christina Lee, Thomas Loke, Angus Lowe, Keri McKiernan, Johannes Jakob Meyer, J. A. Montañez-Barrera, Romain Moyard, Zeyue Niu, Lee James O'Riordan, Steven Oud, Ashish Panigrahi, Chae-Yeun Park, Daniel Polatajko, Nicolás Quesada, Chase Roberts, Nahum Sá, Isidor Schoch, Borun Shi, Shuli Shu, Sukin Sim, Arshpreet Singh, Ingrid Strandberg, Jay Soni, Antal Száva, Slimane Thabet, Rodrigo A. Vargas-Hernández, Trevor Vincent, Nicola Vitucci, Maurice Weber, David Wierichs, Roeland Wiersema, Moritz Willmann, Vincent Wong, Shaoming Zhang, and Nathan Killoran. Pennylane: Automatic differentiation of hybrid quantum-classical computations, 2022.

[6] 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.

[7] Paulo Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. Wine quality. UCI Machine Learning Repository, 2009. DOI: https://doi.org/10.24432/C56S3T.

[8] David Deutsch. Quantum theory, the church–turing principle and the universal quantum computer. In *Proc. R. Soc. Lond. A 400*, pages 97–117, 1985.

[9] Tarun Dutta, Adrián Pérez-Salinas, Jasper Phua Sing Cheng, José Ignacio Latorre, and Manas Mukherjee. Single-qubit universal classifier implemented on an ion-trap quantum device. *Phys. Rev. A*, 106:012411, Jul 2022.

[10] Edward Farhi and Hartmut Neven. Classification with quantum neural networks on near term processors, 2018.

[11] R. A. FISHER. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

[12] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing*, pages 212–219, New York, USA, 1996. Association for Computing Machinery.

[13] Bastian Hacker, Stephan Welte, Gerhard Rempe, and Stephan Ritter. A photon–photon quantum gate based on a single atom in an optical resonator. *Nature*, 536(7615):193–196, July 2016.

[14] Safwan Hossein. Multiclass margin classifier. `https://pennylane.ai/qml/demos/tutorial_multiclass_classification/`, 03 2020. Date Accessed: 2023-11-20.

[15] Hsin-Yuan Huang, Michael Broughton, Masoud Mohseni, Ryan Babbush, Sergio Boixo, Hartmut Neven, and Jarrod R. McClean. Power of data in quantum machine learning. *Nature Communications*, 12(1), May 2021.

[16] N. Innan, M.A.Z. Khan, B. Panda, and M. Bennai. Enhancing quantum support vector machines through variational kernel training. *Quantum Information Processing*, 22(10), October 2023.

[17] T. H. Johnson, S. R. Clark, and D. Jaksch. What is a quantum simulator? *EPJ Quantum Technol.*, 1:10, 2014.

[18] Yunseok Kwak, Won Joon Yun, Soyi Jung, Jong-Kook Kim, and Joongheon Kim. Introduction to quantum reinforcement learning: Theory and pennylane-based implementation, 2021.

[19] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[20] Yunchao Liu, Srinivasan Arunachalam, and Kristan Temme. A rigorous and robust quantum speed-up in supervised machine learning. *Nature Physics*, 17(9):1013–1017, July 2021.

[21] Andrea Mari, Thomas R. Bromley, Josh Izaac, Maria Schuld, and Nathan Killoran. Transfer learning in hybrid classical-quantum neural networks. *Quantum*, 4:340, October 2020.

[22] Sokolova Marina and Lapalme Guy. A systematic analysis of performance measures for classi[U+FB01]cation tasks. *Information Processing and Management*, 45:427–437, 2009.

[23] K. Mitarai, M. Negoro, M. Kitagawa, and K. Fujii. Quantum circuit learning. *Physical Review A*, 98(3), September 2018.

[24] Arjun Pakrashi, Derek Greene, and Brian Mac Namee. Benchmarking multi-label classification algorithms. In *Conference: 24th Irish Conference on Artificial Intelligence and Cognitive Science (AICS'16)*, Dublin, Ireland, 09 2016.

[25] Tadas Paulauskas and Julius Ruseckas. *Kvantinė Kompiuterija*. last updated: 2023-08-17.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1), July 2014.

[28] James R. Powell. The quantum limit to moore's law. *Proceedings of the IEEE*, 96(8):1247–1248, 2008.

[29] Adrián Pérez-Salinas, Alba Cervera-Lierta, Elies Gil-Fuster, and José I. Latorre. Data re-uploading for a universal quantum classifier. *Quantum*, 4:226, February 2020.

[30] Salonik Resch and Ulya R. Karpuzcu. Quantum computing: An overview across the system stack, 2019.

[31] Maria Schuld. Supervised quantum machine learning models are kernel methods, 2021.

[32] Maria Schuld, Alex Bocharov, Krysta M. Svore, and Nathan Wiebe. Circuit-centric quantum classifiers. *Physical Review A*, 101(3), March 2020.

[33] Maria Schuld and Francesco Petruccione. *Machine Learning with Quantum Computers*. Springer Cham, Cham, Switzerland, 2021. 312 p.

[34] 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.

[35] R. Stanley Williams. What's next? [the end of moore's law]. *Computing in Science Engineering*, 19(2):7–13, 2017.

Appendix

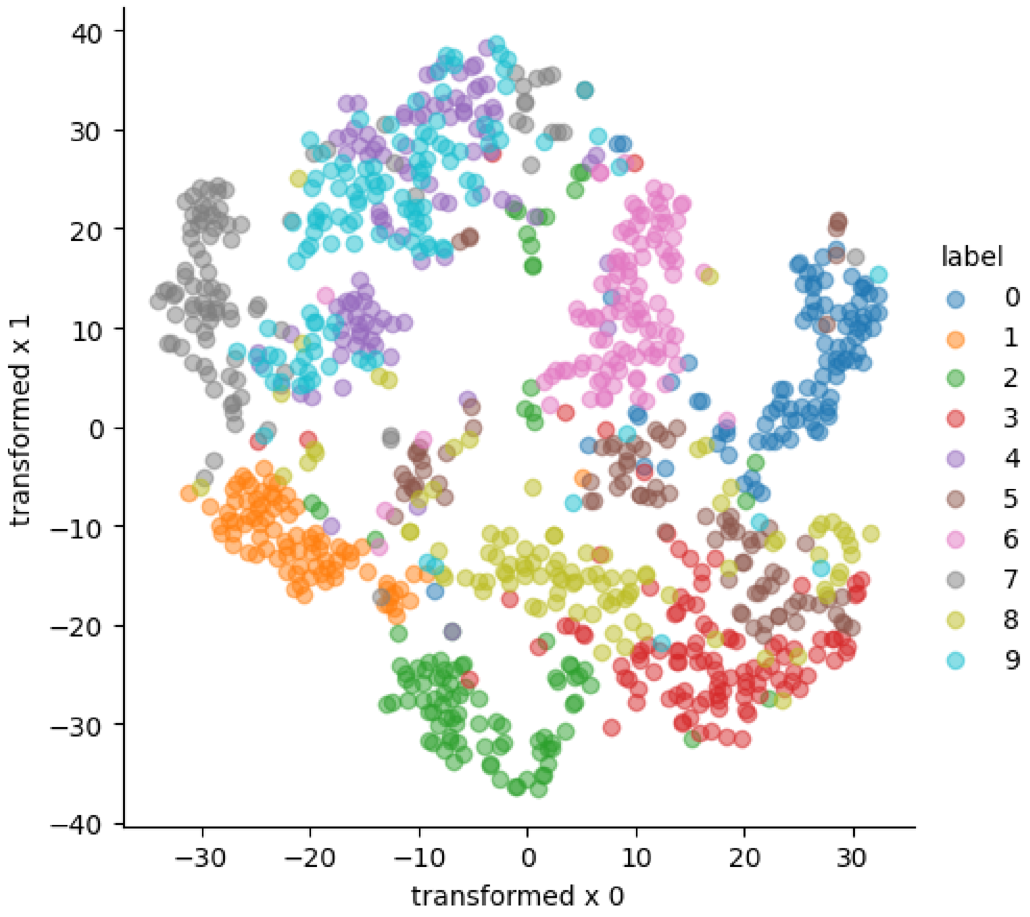

Figure 13: MNIST data set after dimensionality reduction class distribution.

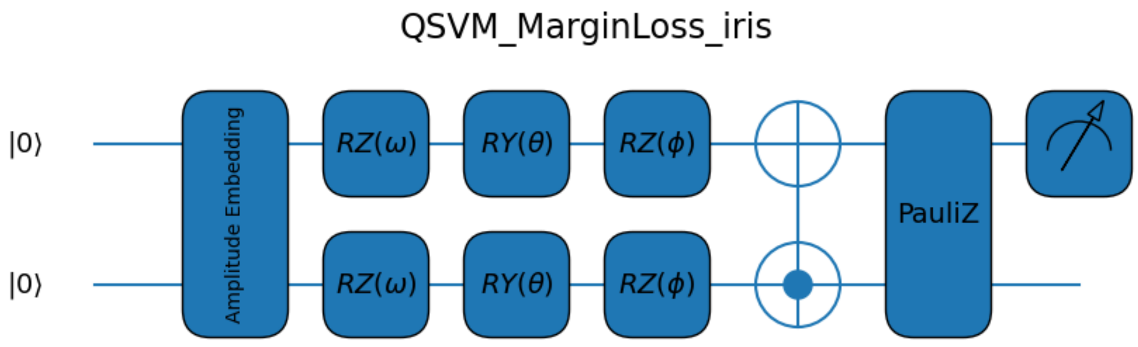

Figure 14: QSVM ansatz for Iris data set.

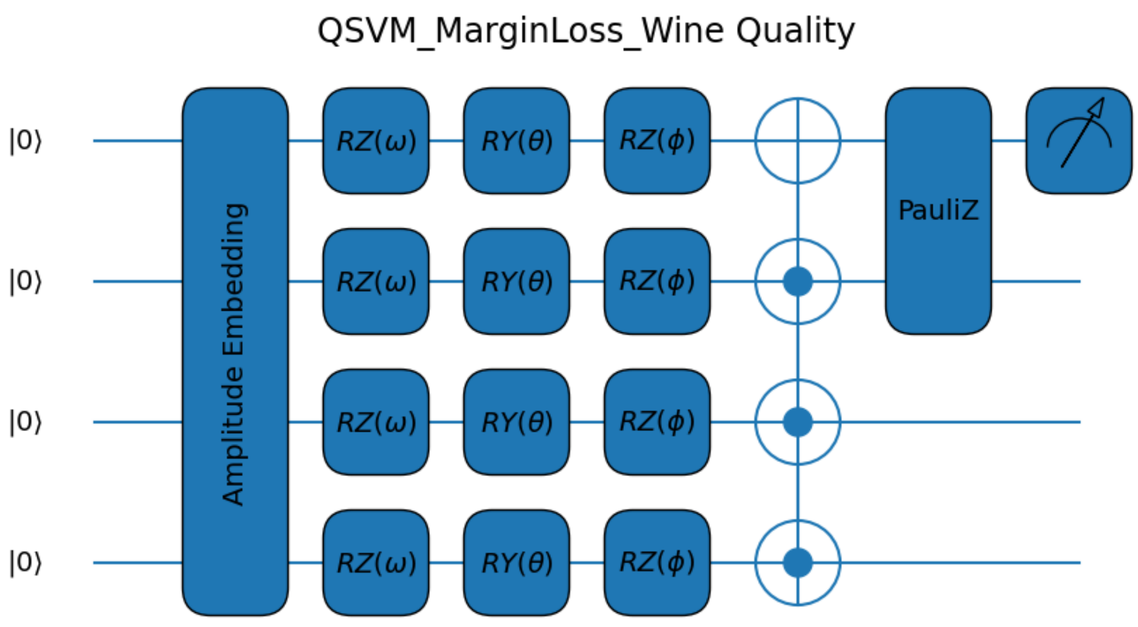

Figure 15: QSVM ansatz for Wine Quality data set.

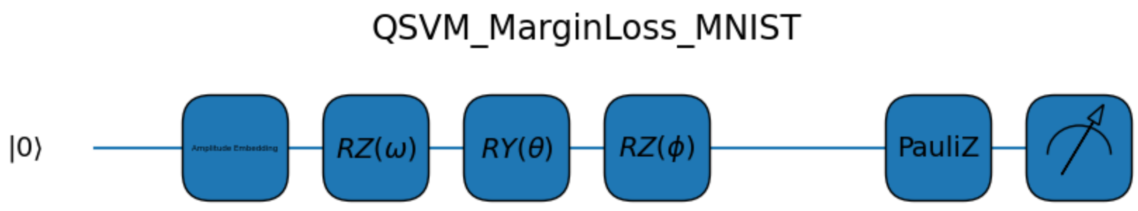

Figure 16: QSVM ansatz for MNIST data set.

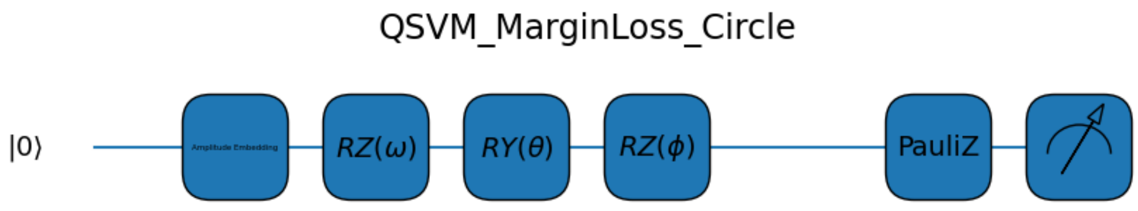

Figure 17: QSVM ansatz for Circle data set.

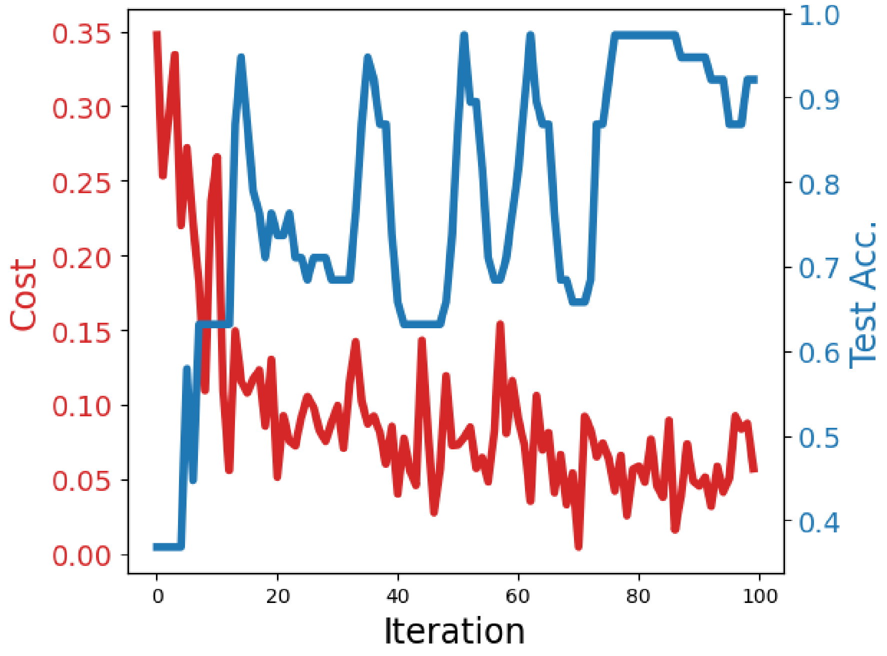

Figure 18: Iris data set cost and test accuracy trade-off results.

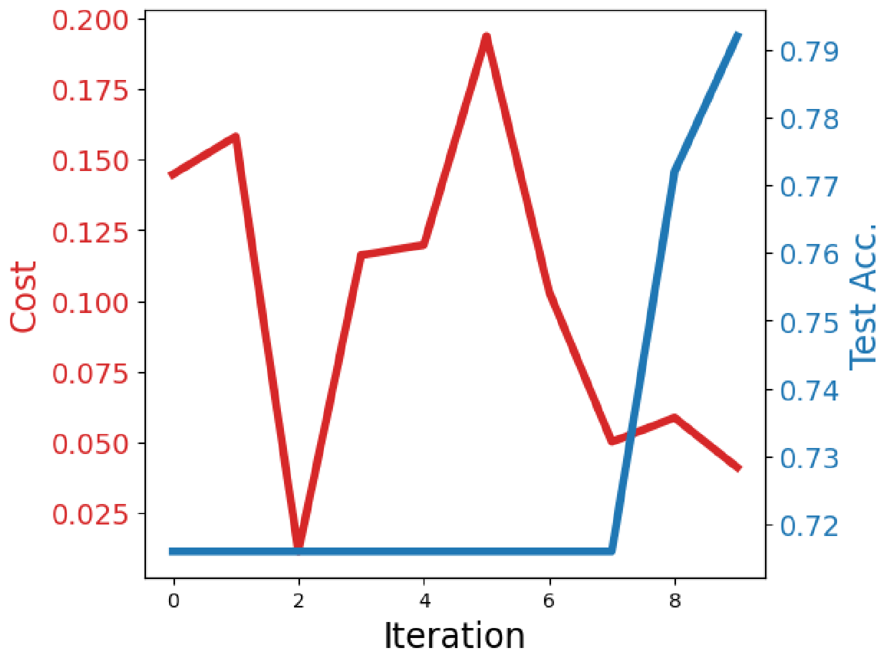

Figure 19: Wine Quality data set cost and test accuracy trade-off results.

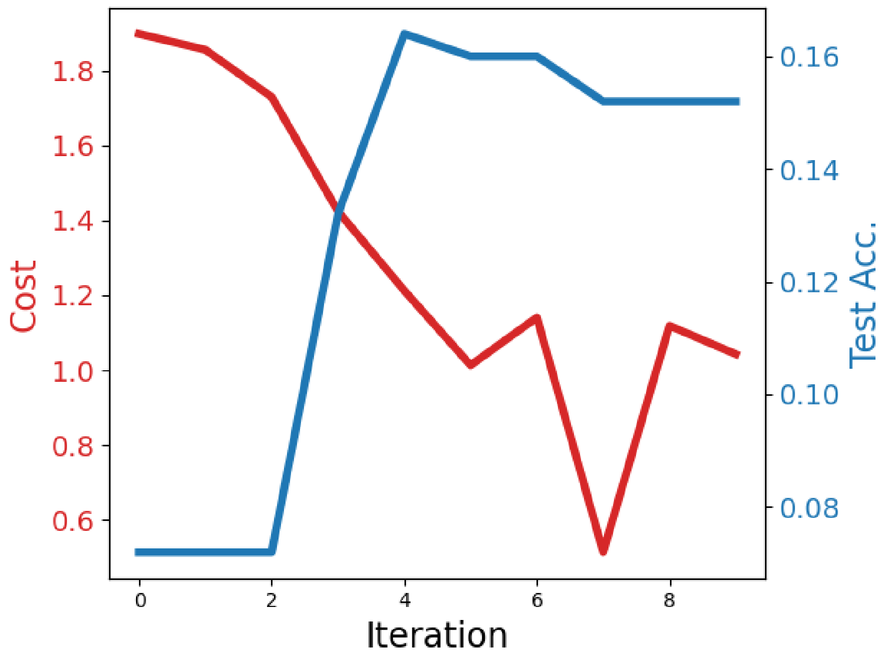

Figure 20: MNIST data set cost and test accuracy trade-off results.

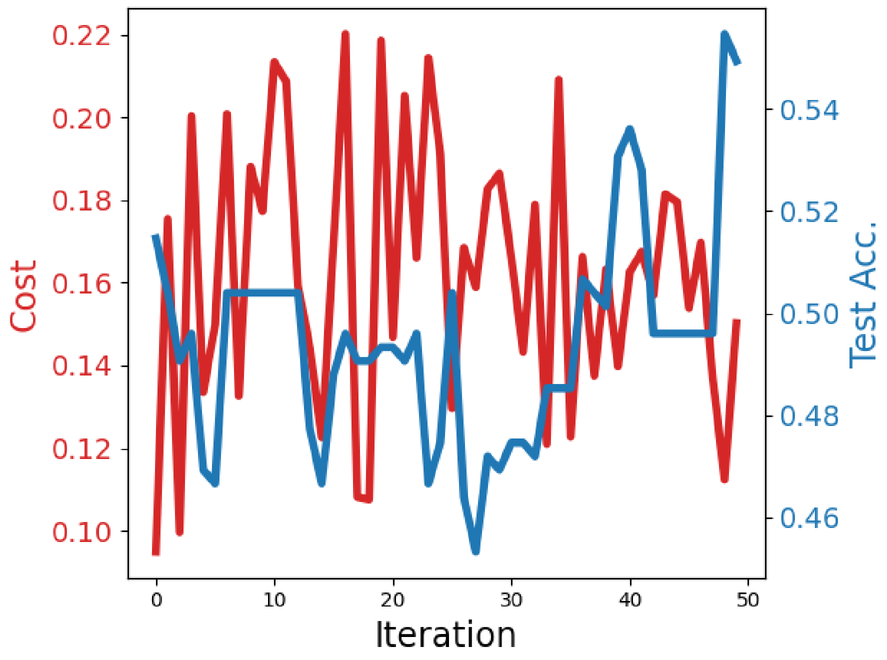

Figure 21: Circle data set cost and test accuracy trade-off results.

```
import os

# Check CPU information
cpu_info = !cat /proc/cpuinfo  # For Linux-based systems

# Print CPU information
for line in cpu_info:
    print(line)
```

```
!pip install pennylane
```

```
!pip install ucimlrepo
```

```
#update scikit - learn for precision
!pip install scikit-learn --upgrade
```

Multiclass Margin Classifier

a quantum variation of SVM (uses multiclass margin loss function)

```
import pennylane as qml
import torch
import numpy as np
import pandas as pd
from torch.autograd import Variable
import torch.optim as optim
from sklearn.preprocessing import LabelBinarizer
import time
import matplotlib.pyplot as plt

#Iris dataset
from sklearn.datasets import load_iris

#Wine quality
from ucimlrepo import fetch_ucirepo

#import dataset from OpenML
from sklearn.datasets import fetch_openml

#dimensionality reduction
from sklearn.decomposition import TruncatedSVD
from sklearn.manifold import TSNE

#metrics
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score

#set seed and parameters
np.random.seed(0)
torch.manual_seed(0)
```

Layer and Circuit

```
#layer
def layer(W):
    for i in range(num_qubits):
        qml.Rot(W[i, 0], W[i, 1], W[i, 2], wires=i)
    for j in range(num_qubits - 1):
        qml.CNOT(wires=[j, j + 1])
    if num_qubits >= 2:
        # Apply additional CNOT to entangle the last with the first qubit
        qml.CNOT(wires=[num_qubits - 1, 0])

def circuit(weights, feat=None):
    qml.AmplitudeEmbedding(feat, range(num_qubits), pad_with=0.0, normalize=True)
    for W in weights:
        layer(W)

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

def variational_classifier(q_circuit, params, feat):
    weights = params[0]
    bias = params[1]
    return q_circuit(weights, feat=feat) + bias
```

Loss function

```
#Loss function
def multiclass_svm_loss(q_circuits, all_params, feature_vecs, true_labels):
    loss = 0
    num_samples = len(true_labels)
    for i, feature_vec in enumerate(feature_vecs):
        s_true = variational_classifier(
            q_circuits[int(true_labels[i])],
            (all_params[0][int(true_labels[i])], all_params[1][int(true_labels[i])]),
            feature_vec,
        )
        s_true = s_true.float()
        li = 0

        # Get the scores computed for this sample by the other classifiers
        for j in range(num_classes):
            if j != int(true_labels[i]):
                s_j = variational_classifier(
                    q_circuits[j], (all_params[0][j], all_params[1][j]), feature_vec
                )
                s_j = s_j.float()
                li += torch.max(torch.zeros(1).float(), s_j - s_true + margin)
        loss += li

    return loss / num_samples
```

Classification function

```
#classification function
def classify(q_circuits, all_params, feature_vecs, labels):
    predicted_labels = []
    for i, feature_vec in enumerate(feature_vecs):
        scores = np.zeros(num_classes)
        for c in range(num_classes):
            score = variational_classifier(
                q_circuits[c], (all_params[0][c], all_params[1][c]), feature_vec
            )
            scores[c] = float(score)
        pred_class = np.argmax(scores)
        predicted_labels.append(pred_class)
    return predicted_labels

def accuracy(labels, hard_predictions):
    loss = 0
    for l, p in zip(labels, hard_predictions):
        if torch.abs(l - p) < 1e-5:
            loss = loss + 1
    loss = loss / labels.shape[0]
    return loss
```

Data and preprocessing

```
def load_and_process_data(X,Y):
    X = torch.tensor(X)
    print("First X sample, original  :", X[0])

    # standartize each input
    normalization = torch.sqrt(torch.sum(X ** 2, dim=1))
    X_norm = X / normalization.reshape(len(X), 1)
    print("First X sample, normalized:", X_norm[0])

    Y = torch.tensor(Y)
    return X, Y


# Create a train and test split.
def split_data(feature_vecs, Y):
    num_data = len(Y)
    num_train = int(train_split * num_data)
    index = np.random.permutation(range(num_data))
    feat_vecs_train = feature_vecs[index[:num_train]]
    Y_train = Y[index[:num_train]]
    feat_vecs_test = feature_vecs[index[num_train:]]
    Y_test = Y[index[num_train:]]
    return feat_vecs_train, feat_vecs_test, Y_train, Y_test
```

Training

```
def training(features, Y):
    num_data = Y.shape[0]
    feat_vecs_train, feat_vecs_test, Y_train, Y_test = split_data(features, Y)
    num_train = Y_train.shape[0]
    q_circuits = qnodes

    # Initialize the parameters
    all_weights = [
        Variable(0.1 * torch.randn(num_layers, num_qubits, 3), requires_grad=True)
        for i in range(num_classes)
    ]
    all_bias = [Variable(0.1 * torch.ones(1), requires_grad=True) for i in range(num_classes)]
    optimizer = optim.Adam(all_weights + all_bias, lr=lr_adam)
    params = (all_weights, all_bias)
    print("Num params: ", 3 * num_layers * num_qubits * 3 + 3)

    costs, train_acc, test_acc, itr_time = [], [], [], []
    macro_a,macro_r,macro_p,mf1, error_s = [],[],[],[],[]

    start_time = time.time()

    # train the variational classifier
    for it in range(total_iterations):

        batch_index = np.random.randint(0, num_train, (batch_size,))
        feat_vecs_train_batch = feat_vecs_train[batch_index]
        Y_train_batch = Y_train[batch_index]

        optimizer.zero_grad()
        curr_cost = multiclass_svm_loss(q_circuits, params, feat_vecs_train_batch, Y_train_batch)
        curr_cost.backward()
        optimizer.step()

        # Compute predictions on train and validation set
        predictions_train = classify(q_circuits, params, feat_vecs_train, Y_train)
        predictions_test = classify(q_circuits, params, feat_vecs_test, Y_test)
        acc_train = accuracy(Y_train, predictions_train)
        acc_test = accuracy(Y_test, predictions_test)

        #sklearn metrics
        macro_accuracy = accuracy_score(Y_test,predictions_test) #   y_true, y_pred)
        macro_recall = recall_score(Y_test,predictions_test, average='macro') #y_true, y_pred, average='macro')
        macro_precision = precision_score(Y_test,predictions_test, average='macro',zero_division = np.nan)
        macro_f1 = f1_score(Y_test,predictions_test, average='macro')
        error_rate = 1 - macro_accuracy

        iteration_time = time.time() - start_time
        print(
            "Iter: {:5d} | Cost: {:0.7f} | Acc train: {:0.7f} | Acc test: {:0.7f} | Iter time: {:0.4f} "
            "".format(it + 1, curr_cost.item(), acc_train, acc_test, iteration_time)
        )
        )

        costs.append(curr_cost.item())
        train_acc.append(acc_train)
        test_acc.append(acc_test)
        itr_time.append(iteration_time)
        macro_a.append(macro_accuracy)
        macro_r.append(macro_recall)
        macro_p.append(macro_precision)
        mf1.append(macro_f1)
        error_s.append(error_rate)

        # Reset the timer for the next iteration
        start_time = time.time()

    return costs, train_acc, test_acc, itr_time, macro_a, macro_r, macro_p, mf1, error_s
```

Results

```
def result(total_iterations,costs, train_acc, test_acc, iteration_t,macro_a,macro_r,macro_p,mf1,error_s):
    #error rate 1-accuracy
    i=0
    test_error =[]
    train_error= []
    iterations= np.arange(0, total_iterations, 1)
    for i in range(0,len(iterations)):
        test_error.append(1-test_acc[i])
        train_error.append(1-train_acc[i])

    res= { "iterations" : [item + 1 for item in iterations],
         "test error" : test_error,
         "train error" : train_error,
         "costs" : costs,
         "training accuracy" : train_acc,
         "testing accuracy" : test_acc,
        "iteration time" : iteration_t,
        "macro accuracy" : macro_a,
          "macro recall" : macro_r,
          "macro precision" : macro_p,
          "macro F1score" : mf1,
          "error_s" : error_s
    }
    df= pd.DataFrame(res)
    return df
```

Iris

```
# Load the Iris dataset
iris = load_iris(as_frame=False)

# Access the features (bdata) and target (labels)
X_iris = iris.data  # Features (attributes)
Y_iris = iris.target  # Target (class labels)


#class imbalance ratio f

def ClassIR(Y_label):
    unique_class, class_count = np.unique(Y_label, return_counts = True)
    ClassIRatio = class_count / len(Y_label)
    aver = np.average(ClassIRatio)

    return aver

def maxIR(Y_label):
    unique_class, class_count = np.unique(Y_label, return_counts = True)
    ratio= np.max(class_count)/np.min(class_count)
    return ratio

ClassIR(Y_iris)
```

```
maxIR(Y_iris)
```

Parameters

```
num_classes = 3
margin = 0.15
feature_size = 4
batch_size = 10
lr_adam = 0.01
train_split = 0.75
# the number of the required qubits is calculated from the number of features
num_qubits = int(np.ceil(np.log2(feature_size)))
num_layers = 6
total_iterations = 100

dev = qml.device("default.qubit", wires=num_qubits)
```

```
qnodes = []
for iq in range(num_classes):
    qnode = qml.QNode(circuit, dev, interface="torch")
    qnodes.append(qnode)
```

```
#circuit scheme
drawer = qml.drawer.MPLDrawer(n_wires=num_qubits, n_layers=num_layers)
drawer.label([r"$|0\rangle$",r"$|0\rangle$"] )
drawer.box_gate(layer=0, wires=[0, 1], text="Amplitude Embedding")
drawer.box_gate(layer=1, wires=[0], text=r"$RZ(\omega)$")
drawer.box_gate(layer=2, wires=[0], text=r"$RY(\theta)$")
drawer.box_gate(layer=3, wires=[0], text=r"$RZ(\phi)$")
drawer.CNOT(layer=4, wires=(0, 1))
drawer.box_gate(layer=1, wires=[1], text=r"$RZ(\omega)$")
drawer.box_gate(layer=2, wires=[1], text=r"$RY(\theta)$")
drawer.box_gate(layer=3, wires=[1], text=r"$RZ(\phi)$")
drawer.CNOT(layer=4, wires=(1, 0))
drawer.box_gate(layer=5, wires=[0,1], text =r"PauliZ" )
drawer.measure(layer=6, wires=0)
drawer.fig.suptitle('QSVM_MarginLoss_iris', fontsize='xx-large')
```

```
%%time

features, target = load_and_process_data(X_iris,Y_iris)
start_t=time.time()
costs, train_acc, test_acc,itr_time,macro_a,macro_r,macro_p,mf1,error_s = training(features, target)
end_t=time.time()
elapsed= end_t - start_t

fig, ax1 = plt.subplots()
iters = np.arange(0, total_iterations, 1)
colors = ["tab:red", "tab:blue"]
ax1.set_xlabel("Iteration", fontsize=17)
ax1.set_ylabel("Cost", fontsize=17, color=colors[0])
ax1.plot(iters, costs, color=colors[0], linewidth=4)
ax1.tick_params(axis="y", labelsize=14, labelcolor=colors[0])

ax2 = ax1.twinx()
ax2.set_ylabel("Test Acc.", fontsize=17, color=colors[1])
ax2.plot(iters, test_acc, color=colors[1], linewidth=4)

ax2.tick_params(axis="x", labelsize=14)
ax2.tick_params(axis="y", labelsize=14, labelcolor=colors[1])

plt.grid(False)
plt.tight_layout()
plt.show()

print("time elapsed for training & classification: ",elapsed)
```

```
df_iris = result(total_iterations,costs, train_acc, test_acc,itr_time,macro_a,macro_r,macro_p,mf1, error_s)
```

```
df_iris

max_row_index = df_iris["macro accuracy"].idxmax()

# Get the row with the maximum value
max_value = df_iris.loc[max_row_index]
max_value
```

Wine quality dataset

```
# fetch dataset
wine_quality = fetch_ucirepo(id=186)

# data (as np array)
X_w = np.array(wine_quality.data.features)
#y_w = wine_quality.data.targets
Y_w = wine_quality.data.original["color"]
lb = LabelBinarizer()
Y_w=np.array(lb.fit_transform(Y_w))

# variable information
print(wine_quality.variables)
```

```
X_w.shape
```

```
ClassIR(Y_w)
```

```
maxIR(Y_w)
```

Parameters

```
num_classes = 2
margin = 0.15
feature_size = 12
batch_size = 10
lr_adam = 0.01
train_split = 0.75
# the number of the required qubits is calculated from the number of features
num_qubits = int(np.ceil(np.log2(feature_size)))
num_layers = 6
total_iterations = 2

dev = qml.device("default.qubit", wires=num_qubits)
```

```
num_qubits
```

```
qnodes = []
for iq in range(num_classes):
    qnode = qml.QNode(circuit, dev, interface="torch")
    qnodes.append(qnode)

#circuit scheme
drawer = qml.drawer.MPLDrawer(n_wires=num_qubits, n_layers=num_layers)
drawer.label([r"$|0\rangle$",r"$|0\rangle$",r"$|0\rangle$",r"$|0\rangle$"])
drawer.box_gate(layer=0, wires=[0, 1, 2, 3], text="Amplitude Embedding")

drawer.box_gate(layer=1, wires=[0], text=r"$RZ(\omega)$")
drawer.box_gate(layer=2, wires=[0], text=r"$RY(\theta)$")
drawer.box_gate(layer=3, wires=[0], text=r"$RZ(\phi)$")
drawer.CNOT(layer=4, wires=(0, 1))

drawer.box_gate(layer=1, wires=[1], text=r"$RZ(\omega)$")
drawer.box_gate(layer=2, wires=[1], text=r"$RY(\theta)$")
drawer.box_gate(layer=3, wires=[1], text=r"$RZ(\phi)$")
drawer.CNOT(layer=4, wires=(1, 2))

drawer.box_gate(layer=1, wires=[2], text=r"$RZ(\omega)$")
drawer.box_gate(layer=2, wires=[2], text=r"$RY(\theta)$")
drawer.box_gate(layer=3, wires=[2], text=r"$RZ(\phi)$")
drawer.CNOT(layer=4, wires=(2, 3))

drawer.box_gate(layer=1, wires=[3], text=r"$RZ(\omega)$")
drawer.box_gate(layer=2, wires=[3], text=r"$RY(\theta)$")
drawer.box_gate(layer=3, wires=[3], text=r"$RZ(\phi)$")
drawer.CNOT(layer=4, wires=(3, 0))

drawer.box_gate(layer=5, wires=[0,1], text =r"PauliZ" )
drawer.measure(layer=6, wires=0)
drawer.fig.suptitle('QSVM_MarginLoss_Wine Quality', fontsize='xx-large')
```

smaller sample size run

```
total_iterations = 10
random_index = np.random.choice(X_w.shape[0], size = 1000, replace = False)
X_ws=X_w[random_index]
Y_ws=Y_w[random_index]
```

```
ClassIR(Y_ws)
```

```
maxIR(Y_ws)
```

```
%%time
# We now run our training algorithm and plot the results. Note that

features, target = load_and_process_data(X_ws,Y_ws)
start_t=time.time()
costs, train_acc, test_acc,itr_time,macro_a,macro_r,macro_p,mf1,error_s = training(features, target)
end_t=time.time()
elapsed= end_t - start_t

fig, ax1 = plt.subplots()
iters = np.arange(0, total_iterations, 1)
colors = ["tab:red", "tab:blue"]
ax1.set_xlabel("Iteration", fontsize=17)
ax1.set_ylabel("Cost", fontsize=17, color=colors[0])
ax1.plot(iters, costs, color=colors[0], linewidth=4)
ax1.tick_params(axis="y", labelsize=14, labelcolor=colors[0])

ax2 = ax1.twinx()
ax2.set_ylabel("Test Acc.", fontsize=17, color=colors[1])
ax2.plot(iters, test_acc, color=colors[1], linewidth=4)

ax2.tick_params(axis="x", labelsize=14)
ax2.tick_params(axis="y", labelsize=14, labelcolor=colors[1])

plt.grid(False)
plt.tight_layout()
plt.show()

print("time elapsed for training & classification: ",elapsed)
```

```
df_w_s1 = result(total_iterations,costs, train_acc, test_acc,itr_time,macro_a,macro_r,macro_p,mf1,error_s)
df_w_s1
```

```
max_row_index = df_w_s1["macro accuracy"].idxmax()

# Get the row with the maximum value
max_value = df_w_s1.loc[max_row_index]
max_value
```

MNIST

```
# Load MNIST dataset
mnist = fetch_openml("mnist_784")

# Extract features and labels as NumPy arrays
```

```
X_m, y_m = mnist.data.to_numpy(), mnist.target.to_numpy()
```

preparing MNIST

```
#overall 70k observations
ClassIR(y_m)

#overall 70k observations
maxIR(y_m)

#function fo
import seaborn as sns
import pandas as pd

# function to help plot the 2-D dataset
def plot2d(X, Y, c1, c2, N):
    lbl1 = f'transformed x {c1}'
    lbl2 = f'transformed x {c2}'
    df = pd.DataFrame({lbl1:X[:N,c1], lbl2:X[:N,c2], 'label':Y[:N]})
    sns.lmplot(data=df, x=lbl1, y=lbl2, fit_reg=False, hue='label', scatter_kws={'alpha':0.5})
```

Parameters

```
num_classes = 10
margin = 0.15
feature_size = 2
batch_size = 10
lr_adam = 0.01
train_split = 0.75
# the number of the required qubits is calculated from the number of features
num_qubits = int(np.ceil(np.log2(feature_size)))
num_layers = 6
total_iterations = 10

dev = qml.device("default.qubit", wires=num_qubits)

int(np.ceil(np.log2(feature_size)))


qnodes = []
for iq in range(num_classes):
    qnode = qml.QNode(circuit, dev, interface="torch")
    qnodes.append(qnode)
```

Smaller sample size shuffled

```
random_index_m = np.random.choice(X_m.shape[0], size = 1000, replace = False)
X_ms=X_m[random_index_m]
Y_ms=y_m[random_index_m]

Y_ms=Y_ms.astype(int)
print(Y_ms.shape)
Y_ms

#actual
ClassIR(Y_ms)

#actual
maxIR(Y_ms)

%%time
# Using SVD to bring down the dimension to 10
tsvd = TruncatedSVD(n_components=10)
X_SVD = tsvd.fit_transform(X_ms)

# Further using t-SNE to bring the dimension down to 2
tsne = TSNE(n_components=2)
X_ms_pf = tsne.fit_transform(X_SVD)
```

```
plot2d(X_ms_pf, Y_ms, 0, 1, N=1000)
```

```
#circuit scheme
drawer = qml.drawer.MPLDrawer(n_wires=num_qubits, n_layers=num_layers)
drawer.label([r"$|0\rangle$"] )
drawer.box_gate(layer=0, wires=[0], text="Amplitude Embedding")

drawer.box_gate(layer=1, wires=[0], text=r"$RZ(\omega)$")
drawer.box_gate(layer=2, wires=[0], text=r"$RY(\theta)$")
drawer.box_gate(layer=3, wires=[0], text=r"$RZ(\phi)$")

drawer.box_gate(layer=5, wires=[0], text =r"PauliZ" )
drawer.measure(layer=6, wires=0)
drawer.fig.suptitle('QSVM_MarginLoss_MNIST', fontsize='xx-large')
```

```
%%time
# We now run our training algorithm and plot the results. Note that
# for plotting, the matplotlib library is required

features, target = load_and_process_data(X_ms_pf,Y_ms)
start_t=time.time()
costs, train_acc, test_acc,itr_time,macro_a,macro_r,macro_p,mf1,error_s = training(features, target)
end_t=time.time()
elapsed= end_t - start_t

fig, ax1 = plt.subplots()
iters = np.arange(0, total_iterations, 1)
colors = ["tab:red", "tab:blue"]
ax1.set_xlabel("Iteration", fontsize=17)
ax1.set_ylabel("Cost", fontsize=17, color=colors[0])
ax1.plot(iters, costs, color=colors[0], linewidth=4)
ax1.tick_params(axis="y", labelsize=14, labelcolor=colors[0])

ax2 = ax1.twinx()
ax2.set_ylabel("Test Acc.", fontsize=17, color=colors[1])
ax2.plot(iters, test_acc, color=colors[1], linewidth=4)

ax2.tick_params(axis="x", labelsize=14)
ax2.tick_params(axis="y", labelsize=14, labelcolor=colors[1])

plt.grid(False)
plt.tight_layout()
plt.show()

print("time elapsed for training & classification: ",elapsed)
```

```
df_m_s1 = result(total_iterations,costs, train_acc, test_acc,itr_time,macro_a,macro_r,macro_p,mf1,error_s)
df_m_s1
```

```
max_row_index = df_m_s1["macro accuracy"].idxmax()

# Get the row with the maximum value
max_value = df_m_s1.loc[max_row_index]
max_value
```

Circle data set

```
# Make a dataset of points inside and outside of a circle
def circle(samples, center=[0.0, 0.0], radius=np.sqrt(2 / np.pi)):

    Xvals, yvals = [], []

    for i in range(samples):
        x = 2 * (np.random.rand(2)) - 1
        y = 0
        if np.linalg.norm(x - center) < radius:
            y = 1
        Xvals.append(x)
        yvals.append(y)
    return np.array(Xvals), np.array(yvals)
```

```
def plot_data(x, y, fig=None, ax=None):

    if fig == None:
        fig, ax = plt.subplots(1, 1, figsize=(5, 5))
    reds = y == 0
    blues = y == 1
    ax.scatter(x[reds, 0], x[reds, 1], c="red", s=20, edgecolor="k")
    ax.scatter(x[blues, 0], x[blues, 1], c="blue", s=20, edgecolor="k")
    ax.set_xlabel("$x_1$")
    ax.set_ylabel("$x_2$")


# Generate training and test data
num_training = 1500
num_test = 500

Xdata, ydata = circle(num_training)
Xtest, y_test = circle(num_test)

X_train_ploting, Y_train_ploting = load_and_process_data(Xdata,ydata)
X_test_ploting, Y_test_ploting = load_and_process_data(Xtest,y_test)
fig, axes = plt.subplots(1, 2, figsize=(7, 3))
plot_data(Xdata, ydata, fig, axes[0])
plot_data(X_test_ploting, Y_test_ploting, fig, axes[1])
axes[0].set_title("Train set")
axes[1].set_title("Test set")
plt.show()

ClassIR(ydata)


maxIR(ydata)
```

Parameters

```
num_classes = 2
margin = 0.15
feature_size = 2
batch_size = 10
lr_adam = 0.01
train_split = 0.75
# the number of the required qubits is calculated from the number of features
num_qubits = int(np.ceil(np.log2(feature_size)))
num_layers = 6
total_iterations = 50

dev = qml.device("default.qubit", wires=num_qubits)

num_qubits


qnodes = []
for iq in range(num_classes):
    qnode = qml.QNode(circuit, dev, interface="torch")
    qnodes.append(qnode)

#circuit scheme
drawer = qml.drawer.MPLDrawer(n_wires=num_qubits, n_layers=num_layers)
drawer.label([r"$|0\rangle$"] )
drawer.box_gate(layer=0, wires=[0], text="Amplitude Embedding")

drawer.box_gate(layer=1, wires=[0], text=r"$RZ(\omega)$")
drawer.box_gate(layer=2, wires=[0], text=r"$RY(\theta)$")
drawer.box_gate(layer=3, wires=[0], text=r"$RZ(\phi)$")
#drawer.CNOT(layer=4, wires=(0, 1))

drawer.box_gate(layer=5, wires=[0], text =r"PauliZ" )
drawer.measure(layer=6, wires=0)
drawer.fig.suptitle('QSVM_MarginLoss_Circle', fontsize='xx-large')
```

circles training

```
%%time

features, target = load_and_process_data(Xdata,ydata)
start_t=time.time()
costs, train_acc, test_acc,itr_time,macro_a,macro_r,macro_p,mf1,error_s = training(features, target)
end_t=time.time()
elapsed= end_t - start_t

fig, ax1 = plt.subplots()
iters = np.arange(0, total_iterations, 1)
colors = ["tab:red", "tab:blue"]
ax1.set_xlabel("Iteration", fontsize=17)
ax1.set_ylabel("Cost", fontsize=17, color=colors[0])
ax1.plot(iters, costs, color=colors[0], linewidth=4)
ax1.tick_params(axis="y", labelsize=14, labelcolor=colors[0])

ax2 = ax1.twinx()
ax2.set_ylabel("Test Acc.", fontsize=17, color=colors[1])
ax2.plot(iters, test_acc, color=colors[1], linewidth=4)

ax2.tick_params(axis="x", labelsize=14)
ax2.tick_params(axis="y", labelsize=14, labelcolor=colors[1])

plt.grid(False)
plt.tight_layout()
plt.show()

print("time elapsed for training & classification: ",elapsed)


df_c = result(total_iterations,costs, train_acc, test_acc,itr_time,macro_a,macro_r,macro_p,mf1,error_s)
df_c


max_row_index = df_c["macro accuracy"].idxmax()

# Get the row with the maximum value
max_value = df_c.loc[max_row_index]
max_value
```

```
!pip install mlflow
```

```
!pip install pycaret
```

```
import torch
import numpy as np
import pandas as pd
from torch.autograd import Variable
import torch.optim as optim
from sklearn.preprocessing import LabelBinarizer
import time
import matplotlib.pyplot as plt

#import dataset from OpenML
from sklearn.datasets import fetch_openml

from sklearn.decomposition import TruncatedSVD
from sklearn.manifold import TSNE

from numpy import linalg as la

from pycaret.classification import *

#set seed and parameters
np.random.seed(0)
torch.manual_seed(0)
```

Data import

Iris

```
#Iris dataset
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris(as_frame=True)

# Access the features (bdata) and target (labels)
X_iris = iris.data  # Features (attributes)
Y_iris = iris.target  # Target (class labels)
```

```
Data_iris = X_iris.div(la.norm(X_iris.values, axis = 1), axis = 0)
Data_iris["target"] = Y_iris

Data_iris
```

Wine

```
!pip install ucimlrepo
```

```
#Wine quality
from ucimlrepo import fetch_ucirepo

# fetch dataset
wine_quality = fetch_ucirepo(id=186)

# data (as np array)
X_w = np.array(wine_quality.data.features)
#y_w = wine_quality.data.targets
Y_w = wine_quality.data.original["color"]
lb = LabelBinarizer()
```

```
Y_w=np.array(lb.fit_transform(Y_w))

# variable information
print(wine_quality.variables)

random_index = np.random.choice(X_w.shape[0], size = 1000, replace = False)
X_w=X_w[random_index]
Y_w=Y_w[random_index]

Data_wine = pd.DataFrame(X_w)
Data_wine = Data_wine.div(la.norm(Data_wine.values, axis = 1), axis = 0)
Data_wine["target"] = Y_w

Data_wine
```

MNIST

```
# Load MNIST dataset
mnist = fetch_openml("mnist_784")

# Extract features and labels as NumPy arrays
X_m, y_m = mnist.data.to_numpy(), mnist.target.to_numpy()

random_index = np.random.choice(X_m.shape[0], size = 1000, replace = False)
X_ms=X_m[random_index]
Y_ms=y_m[random_index]
Y_ms=Y_ms.astype(int)

%%time

# Using SVD to bring down the dimension to 10
tsvd = TruncatedSVD(n_components=10)
X_SVD = tsvd.fit_transform(X_ms)

# Further using t-SNE to bring the dimension down to 2
np.random.seed(0)
tsne = TSNE(n_components=2)
X_ms_pf = tsne.fit_transform(X_SVD)

X_ms_pf


Data_mnist=pd.DataFrame(X_ms_pf)
Data_mnist = Data_mnist.div(la.norm(Data_mnist.values, axis = 1), axis = 0)
Data_mnist['target']= Y_ms

Data_mnist
```

Circles

```
# Make a dataset of points inside and outside of a circle
def circle(samples, center=[0.0, 0.0], radius=np.sqrt(2 / np.pi)):

    Xvals, yvals = [], []

    for i in range(samples):
        x = 2 * (np.random.rand(2)) - 1
        y = 0
        if np.linalg.norm(x - center) < radius:
            y = 1
        Xvals.append(x)
        yvals.append(y)
    return np.array(Xvals), np.array(yvals)


# Generate training and test data
num_training = 2000
Xdata, ydata = circle(num_training)
```

```
Xdata, ydata = circle(num_training)
```

```
Data_circles = pd.DataFrame(Xdata)
Data_circles = Data_circles.div(la.norm(Data_circles.values, axis = 1), axis = 0)
Data_circles["target"] = ydata

Data_circles
```

Models

Based on https://nbviewer.org/github/pycaret/examples/blob/main/PyCaret%202%20Classification.ipynb

```
# check version
from pycaret.utils import version
version()
```

Comparison Iris

```
#%%time
clf1 = setup(Data_iris, target = 'target', session_id=123, log_experiment=True, train_size = 0.75, experiment_name='iris1')
```

```
models()
```

```
best_iris = compare_models()
```

```
#get_config('target_param')
get_logs()
```

Comparison Wine

```
clf2 = setup(Data_wine, target = 'target', session_id=123, log_experiment=True,train_size = 0.75, experiment_name='wine1')
```

```
models()
```

```
best_wine = compare_models()
```

Comparison MNIST

```
clf3 = setup(Data_mnist, target = 'target', session_id=123, log_experiment=True,train_size = 0.75, experiment_name='wine1')
```

```
best_mnist = compare_models()
```

```
models()
```

Comparison Circles

```
clf4 = setup(Data_circles, target = 'target', session_id=123, log_experiment=True,train_size = 0.75, experiment_name='circle
```

```
best_circles = compare_models()
```

```
models()
```