University of Trento

Department of Physics

Binary classification on an IBM quantum computer

Author
Marco
Chiloiro

Supervisor
Dr. Davide
Pastorello

June 2022

Abstract

The purpose of this thesis is the implementation of a binary classifier on an IBM quantum machine. In particular, a classification algorithm based on cosine similarity is used to test different quantum processors. Given a training dataset consisting of N real d-dimensional vectors and their binary labels, the algorithm can correctly classify any other d-dimensional unlabelled input vector. To test the algorithm on quantum processors, quantum circuits, with $N = d = 2$, were implemented using IBM Quantum, which is an online platform that offers cloud access to IBM's quantum computers. The software development kit Qiskit was used to implement and run the algorithm and to save the obtained results. Different training datasets were examinated, for each of which different unlabelled input vectors were considered. The results obtained are not yet optimal. The classification is performed correctly only under certain conditions on the training and test vectors.

Contents

1 Introduction

Machine learning (ML) is the science of fitting parameterized mathematical models to data in order to realize intelligent systems that can make predictions or perform inference. The term *data* refers to collections of numeric values that can be obtained from measurements, surveys, interactions with technical devices, or comparable procedures and can be processed by computers. Note that everything that is held in the memory of a computer is encoded in terms of bit patterns and therefore in terms of numbers. In the last twenty years, machine learning has seen an immense success, with applications in many fields. However, the increasing size of available datasets and the end of the Moore's law are limiting the potential development of this field. While new developments in hardware architectures, such as GPUs, enable orders of magnitude improved performance compared to CPUs, they cannot significantly improve the performance any longer, as they also reach their physical limitations. On the other hand, quantum computing has been predicted to overcome these limitations on classical hardware. Then, *quantum machine learning* (QML), the combination of quantum mechanics and machine learning, has received particular attention, especially in this decade around the development of the first prototypes of quantum machines (eg. see [Goo], [D-W], [IBM]). Quantum machine learning is a term that usually denotes machine learning algorithms that are executed on quantum computers and applied to classical data, such as [WKS14] [LMR13]. In this context, it is also called *quantum-enhanced* machine learning [KKP20]. The purpose in designing such quantum algorithms is to achieve computational advantages over any classical algorithm for the same task, as in the case of the algorithm considered in this thesis.

Even if not covered in this thesis, there are other types of applications of QML. For example, there are classical machine learning algorithms applied to quantum data generated from quantum experiments [BAT17] [HDW18], quantum algorithms applied to quantum data [Yu19] [Gho19], hybrid methods that involve both classical and quantum processing [Ben17] [FN18].

In this thesis, a binary classification algorithm based on cosine similarity [PB21] was tested on two 7-qubit IBM quantum processors, *ibmq_jakarta* and *ibm_lagos*, which are two of the IBM Quantum Falcon Processors [Qua21]. They have quantum volume values of 16 and 32, respectively, referring to the IBM's definition of quantum volume [Cro19]. Three different training datasets, each formed by two 2-dimensional unit vectors, and 2-dimensional unlabelled test vectors are considered. Quantum circuit implementation and execution was made using *Qiskit*, an open-source software development kit (SDK) for working with quantum computers [ANI21]. The classification is not optimal yet, it is correct only under certain conditions on training datasets and test input vectors. There is an obvious noise component that compromises the results. Moreover,

the difference in performance due to quantum volume values is not appreciable, since the one with higher volume does not give better results than the one with lower volume, as it would be expected.

This thesis is organized as follows: in chapter 2 it is given an essential introduction to fundamentals of quantum computing and an overview on the used tools to implement and run the quantum classifier on IBM devices. In chapter 3 it is explained how the algorithm works mathematically. In chapter 4 it is shown in details the implementation and execution of the algorithm and the analysis of the obtained results. Finally, in chapter 5, the final conclusions are reported.

2 Quantum computing

This chapter provides an overview of the main topics of quantum computing: the fundamental concepts of qubit, its representation on the *Bloch sphere*, and a brief explanation about the quantum circuit model [NC11]. Moreover, it describes two common subroutines that form a part of many quantum machine learning algorithms and that are directly used for the considered algorithm: the *SWAP test* [Buh01] and the *amplitude encoding* [Pas20]. Finally, it describes the two main tools for implementing and running quantum circuits, i.e. *IBM Quantum Composer* and Qiskit, and it gives the general definition of quantum volume (QV) [Mol18] and its modified definition by IBM [Cro19].

2.1 Fundamental concepts

Quantum computing is a type of computation which uses quantum physics properties to perform calculations. It is built upon the *quantum bit* or *qubit*, which is the analogous concept of the classical *bit*. A qubit is a quantum two-level system described in a two-dimensional Hilbert space $\mathcal{H}_1$. Any two-level quantum-mechanical system and many multilevel systems (if they possess two states that can be effectively decoupled from the rest) can be used as a qubit. Consider two pure states of a qubit, said $|0\rangle$ and $|1\rangle$, that form a basis of $\mathcal{H}_1$ and physically are given by two eigenstates of an observable like a spin component of an electron or the rectilinear polarization of a photon. These two orthonormal basis states, $\{|0\rangle, |1\rangle\}$, together, are called the *computational basis*. Any general state $|\psi\rangle$ of a qubit can be described as a linear superposition of the basis states of $\mathcal{H}_1$ as follows:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \tag{2.1}$$

where α and β are complex numbers and $|\alpha|^2 + |\beta|^2 = 1$. As a classical bit, there are two possible outcomes for the measurement of a qubit, usually said to be 0 and 1. The main difference between bit and qubit is, whereas a bit can be either 0 or 1, the state of a qubit can be a coherent superposition of both. Another difference is, whereas a measurement of a classical bit would not disturb its state, a measurement of a qubit would disturb its superposition state, causing it to collapse in either state $|0\rangle$ or $|1\rangle$.

Qubit basis states can also be combined to form product basis states. A set of qubits taken together is called a *quantum register*. Quantum computers perform calculations by manipulating qubits within a register. For example, two qubits could be represented in a four-dimensional Hilbert space $\mathcal{H}_2$, which is the tensor product of the Hilbert spaces of two single qubits. Its basis could

be represented by the following product basis states, which is the computational basis for a 2-qubit quantum register: $|00\rangle, |01\rangle, |10\rangle$ and $|11\rangle$, where the notation of the type $|00\rangle$ is intended as the tensor product $|0\rangle \otimes |0\rangle$. A n size quantum register is represented by a superposition state vector in 2^n dimensional Hilbert space $\mathcal{H}_n$.

The *Bloch sphere* is a geometrical representation of the state space of a qubit. It is useful in visualizing the state of a single qubit. Consider a generic pure state of a qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers. Because $|\alpha|^2 + |\beta|^2 = 1$, the state $|\psi\rangle$ can be expressed as follows:

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle, \tag{2.2}$$

because a global phase factor $e^{i\gamma}$, which has no observable physical effects, is ignored. The real numbers θ and φ define a point on the unit sphere, as shown in Figure 2.1.

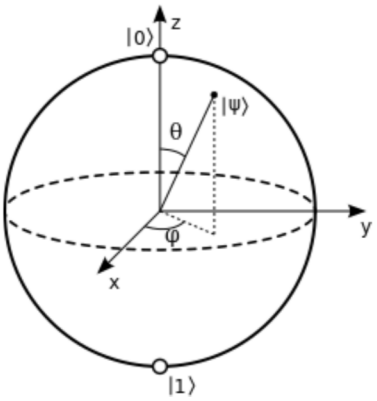


Figure 2.1: Bloch sphere representation of a qubit.

This representation is limited because the Bloch sphere may be generalized to an n-level quantum system, but then the visualization is less useful. Furthermore, there is no simple generalization of the Bloch sphere known for multiple qubits.

The *quantum circuit model* describes the computation in terms of a network of *quantum logic gates*, or simply quantum gate. A quantum computer is built from a *quantum circuit* containing wires and elementary quantum gates to carry around and manipulate the quantum information, in analogy to classical digital computing. Consider an n-qubit register, which is described in the 2^n dimensional Hilbert space $\mathcal{H}_n$. Any unitary operator on $\mathcal{H}_n$ is called *n-qubit gate*. Physically, a quantum gate is a time evolution of an isolated composite system made by qubits. Then, quantum gates are reversible, unlike many classical logic gates. Quantum gates are represented as unitary matrices relative to the computational basis.

Operation on a qubit are described by 2×2 unitary matrices. Some of the most important are the Pauli matrices:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{2.3}$$

Note that the Pauli X gate is the quantum equivalent of the NOT gate for classical computers with respect to the computational basis. Two other important quantum gates are the *Hadamard gate* H and the *phase shift gates* P_ϕ:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad P_\phi = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}. \tag{2.4}$$

Some common examples of shift gates are the $\pi/8$ gate, or T gate, and the S gate:

$$T = P_{\pi/4} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix} \qquad S = P_{\pi/2} = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}. \tag{2.5}$$

The Pauli matrices give rise to a class of unitary matrices when they are exponentiated, the *rotation operators*. If $\hat{n} = (n_x, n_y, n_z)$ is a real unit vector in three dimension, a rotation by θ about the $\hat{n}$ axis of the Bloch sphere is given by:

$$R_{\hat{n}}(\theta) \equiv \exp\left(-\frac{i\theta\hat{n}\vec{\sigma}}{2}\right) = \cos\left(\frac{\theta}{2}\right)\mathbb{I} - i\sin\left(\frac{\theta}{2}\right)(n_x X + n_y Y + n_z Z), \tag{2.6}$$

where $\vec{\sigma} = (X, Y, Z)$ denotes the three component vector of Pauli matrices, and therefore the equation $(\hat{n}\vec{\sigma})^2 = \mathbb{I}$ holds. $R_x(\theta)$, $R_y(\theta)$ and $R_z(\theta)$ denotes the rotation operators about the $\hat{x}$, $\hat{y}$ and $\hat{z}$ Bloch sphere axes.

The symbol representing a measure process on a qubit is shown in Figure 2.2.

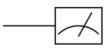


Figure 2.2: Circuit representation for measure process on a qubit.

An important class of operations on two qubits are the *controlled operations*. The most common controlled operation is the controlled-NOT, which is often referred to as CNOT. It is a 2-qubit quantum gate with two qubits in input known as the *control qubit* and *target qubit*, respectively. If the control qubit is set to $|1\rangle$, then the target qubit is flipped, otherwise the target qubit does not change. Then, the action of the CNOT gate is given by $|c\rangle\,|t\rangle \to |c\rangle\,|t \oplus c\rangle$,

where $\oplus$ means the XOR logic operation. In the computational basis, the matrix representation of CNOT is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{2.7}$$

and the symbol for the CNOT gate is represented in Figure 2.3.

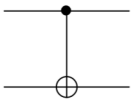


Figure 2.3: Circuit representation for the CNOT gate. The top line represents the control qubit, the bottom one represents the target qubit.

More generally, suppose U is an arbitrary single qubit unitary operation. A *controlled-U* operation is a two qubit operation with a control and a target qubit. When the control qubit is set to $|1\rangle$, then U is applied to the target qubit, otherwise the target qubit does not change. The circuit representation of the controlled-U gate is shown in Figure 2.4.

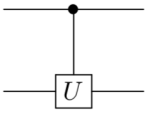


Figure 2.4: Circuit representing the controlled-U for 1-qubit gate U.

It is possible to implement circuits applying multiple qubit conditioning. Suppose to have $n + k$ qubits, and U is a k qubit unitary operator. Then the controlled operation $C^n(U)$ is applied to the last k qubits if the first n qubits are all equal to one, and otherwise, nothing is done. The notation for this type of circuit is shown in Figure 2.5.

Another important two-qubit gate is the SWAP gate. It swaps the states of the two input qubits. With respect to the computational basis, it is represented by the matrix:

$$\text{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.8}$$

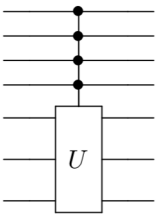


Figure 2.5: Sample circuit representation for the $C^n(U)$ operation, where U is a unitary operator on k qubits, for $n = 4$ and $k = 3$.

SWAP gate can be constructed by CNOT gates in sequence, as it is shown in Figure 2.6.

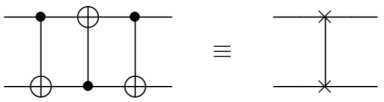


Figure 2.6: Circuit swapping two qubits, and a schematic symbol notation for this circuit.

The controlled SWAP gate (also called Fredkin gate) is used to implement the SWAP test, a useful subroutine for quantum algorithms. The circuit diagram of the SWAP test is shown in Figure 2.7.

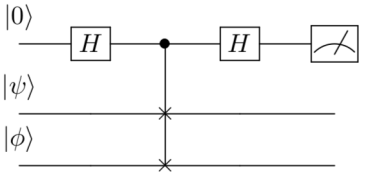


Figure 2.7: Circuit representing SWAP test.

It is possible to show that the probability of measuring 1 as the circuit output is:

$$p = \frac{1}{2} - \frac{1}{2}|\langle\psi|\phi\rangle|^2. \tag{2.9}$$

Therefore, the SWAP test allows to estimate the squared module of the inner product $\langle\psi|\phi\rangle$ from the frequencies of measure 1 (or 0) as output of the circuit. The SWAP test can be perfomed in the general case where $|\psi\rangle$ and $|\phi\rangle$ are n-qubit states, using n Fredkin gate.

Any procedure which maps a classical data string into the physical state of a considered quantum system is called *quantum encoding*. There are several types of quantum encoding (see [SK19]); *amplitude encoding* is one of the most used in QML. The amplitude encoding is a representation of classical data into amplitudes in a quantum state. A data vector $\mathbf{x} \in \mathbb{C}^d$ can be represented by the amplitudes of a quantum state $|\mathbf{x}\rangle$ with reference to an orthonormal basis $\{|\phi_j\rangle\}_j$ of the n-qubit Hilbert space $\mathcal{H}_n$, where $d = 2^n$:

$$|\mathbf{x}\rangle = \frac{1}{\|\mathbf{x}\|} \sum_{j=1}^{d} x_j \, |\phi_j\rangle \in \mathcal{H}_n. \tag{2.10}$$

The main advantage of the amplitude encoding is the space efficiency. Consider a system made by n qubits, in the amplitude encoding it's possible to store data vectors of dimension 2^n. However, there are two limitations of the amplitude encoding: the normalization requirement and the fact that quantum amplitudes cannot be directly observed, then data cannot be directly retrieved from a quantum state.

2.2 IBM quantum computing

IBM Quantum is an online platform that offers cloud access to IBM's quantum computers. It allows developing quantum algorithms through a classical terminal. It can be done using *IBM Quantum Composer*, which is a graphical user interface (GUI) by which it is possible to construct quantum circuits. In Figure 2.8 is shown the interface when a new job is initiated. Once the circuit has been constructed, it is possible to select either a device or a simulator on which it is wanted to run the circuit. Then, the job is queued for execution.

Another method to develop quantum algorithms is using Qiskit. Qiskit is an open-source software development kit (SDK) for working with quantum computers at the level of pulses, circuits, and application modules. It uses the Python programming language. It is possible to start Qiskit locally or with Jupyter Notebooks hosted in *IBM Quantum Lab*, another IBM Quantum's service. The workflow consists of following four steps:

- **Build**: designing a quantum circuit (or more than one) that represent the problem considered.
- **Compile**: compiling circuits for a specific quantum service, such as quantum system or classical simulator. This is made by the transpiler.
- **Run**: run the compiled circuits on the specified quantum service(s). These services can be cloud-based or local.
- **Analyse**: compute summary statistics and visualize the results of the experiments.

Figure 2.8: New job on IBM Quantum Composer. The three single lines represent the quantum register's qubits (in this case there are only three lines, but you can add more by pressing the + image at the bottom left), while the double line at the bottom represents the classical register (in this case there are 3 bits, but it is also possible varying the number of bit in the classical register). After a measurement on a qubits, its result is reported to the classical register in the form of bits. The code that implements the circuit using Qiskit is shown on the right of the interface. The usable gates are illustrated on the top: they can be dragged over the underlying circuit.

The transpiler translates Qiskit code into an optimized circuit using a backend's native gate set, allowing users to program for any quantum processor or processor architecture with minimal inputs. It is also possible to define the optimization level of the circuit compilation on a specific device. This fact is useful because quantum processors can have different properties, such as connectivity architectures, error rates, basis allowed gates and so on. The optimization level ranges from 0 (no optimization) to 3 (maximum level). Another important transpiler's parameter is the *seed_transpiler*, which sets random seed for the stochastic parts of the transpiler. A random seed is a number (which can assume only integer values in this case) which is used to initialize a pseudo-random number generator. The transpiler performs a different compilation of the circuit for each seed value. If no value is specified, the transpiler randomly chooses one. Therefore, at each run the same circuit is compiled in a different optimized circuit. If the seed is specified, the circuit is always compiled in the same way. In Chapter 4 is shown the implementation of the studied algorithm using Qiskit.

A way to compare different quantum computers in their performance using a single number is the concept of *quantum volume*. Quantum volume is an architecture-neutral metric that characterizes the capability of a chosen quantum computing architecture to run useful quantum circuits. It enables the comparison of hardware with widely different performance characteristics and quantifies the complexity of algorithms that can be run on such a system. The quantum computer's performance depends on five main hardware parameters:

1. Number of physical qubits N;
2. Connectivity between qubits [TQ19];

3. Number of gates that can be applied before errors or decoherence mask the result;

4. Available gate set;

5. Number of operations that can be run in parallel;

The *circuit depth* d is the necessary number of steps to run a given algorithm. For any given quantum algorithm, there is a lower bound on the number of qubits N and on the circuit depth d required to run the algorithm. Therefore, the quantum volume is defined taking into account both the number of qubits N and the allowable depth $d(N)$ of quantum circuits that can be run on a quantum device, which depends on the hardware parameter given in the list above. However, a given algorithm doesn't necessarily need all the N available qubits. When a subset of $n < N$ qubits with a good connectivity is selected, it could be favourable for an algorithm which needs n qubits to run on a N-qubit device. The quantum volume is therefore defined as:

$$V_Q = \max_{n<N} \left(\min \left[n, d(n) \right]^2 \right), \tag{2.11}$$

where the maximum is taken over an arbitrary choice of n qubits to maximize the quantum volume that can be obtained with such a subset.

The modified definition of quantum volume used by IBM is:

$$\log_2 V_Q = \underset{n \leq N}{\operatorname{argmax}} \{ \min[n, d(n)] \}. \tag{2.12}$$

This definition differs from (2.11) and loosely coincides with the complexity of classically simulating the model quantum circuits.

3 The classification algorithm

The algorithm used, based on cosine similarity, implements a model for binary classification of data vectors [PB21]. Given two vectors $\mathbf{x}$, $\mathbf{y} \in \mathbb{R}^d$, cosine similarity is defined by:

$$\cos(\mathbf{x}, \mathbf{y}) \doteq \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}. \tag{3.1}$$

Let $X = \{\mathbf{x}_i, y_i\}_{i=0,...,N-1}$, with $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}\ \forall\ i \in \{0, ..., N-1\}$, be a training set of N data instances. Let $\mathbf{x} \in \mathbb{R}^d$ be a new data instance to be classified as either 1 or -1. The classification model is defined as follows:

$$y(\mathbf{x}) \doteq \text{sign}\left(\sum_{i=0}^{N-1} y_i \cos(\mathbf{x}_i, \mathbf{x})\right). \tag{3.2}$$

In this model, any training vector contributes to the prediction of the new label. Then, time complexity of the classical calculation of the new label is $O(Nd)$. However, assuming $d = 2^n$ with $n \in \mathbb{N}$ without loss of generality, the data vector $\mathbf{x}_i \in \mathbb{R}^d$ can be encoded in the amplitudes of a quantum state of n qubits:

$$|\mathbf{x}_i\rangle = \frac{1}{\|\mathbf{x}_i\|} \sum_{j=0}^{d-1} x_{ij} |j\rangle \in \mathcal{H}_n, \tag{3.3}$$

where $\{|j\rangle\}_{j=0,...,d-1}$ is an orthonormal basis of the n-qubit Hilbert space $\mathcal{H}_n$ and x_{ij} is the jth component of $\mathbf{x}_i$. Assuming that the d components of $\mathbf{x}_i$ are stored in an array of memory cells and the norm $\|\mathbf{x}_i\|$ is given separately (or unit vectors are considered), the state (3.3) can be recovered in $O(\log_2 d)$ steps according to the *bucket-brigade qRAM* [Vit08]. Consider a $\log_2 N$-qubit register to encode the indexes of training data vectors, with Hilbert space $\mathcal{H}_i \simeq (\mathbb{C}^2)^{\otimes \log_2 N}$, and a 1-qubit one to encode the values of labels, with Hilbert space $\mathcal{H}_l$, and construct the state:

$$|X\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle |\mathbf{x}_i\rangle |b_i\rangle \in \mathcal{H}_i \otimes \mathcal{H}_n \otimes \mathcal{H}_l, \tag{3.4}$$

with $b_i = \frac{1-y_i}{2} \in \{0, 1\}$. The states $|b_i\rangle$ are eigenstates of the Pauli matrix σ_z with eigenvalue y_i, given that $\sigma_z |0\rangle = |0\rangle$ and $\sigma_z |1\rangle = -|1\rangle$. In the same registers construct also the state that represent the new data vector $\mathbf{x}$ in a quantum superposition of the two possible classes:

$$|\psi_{\mathbf{x}}\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle |\mathbf{x}\rangle |-\rangle \in \mathcal{H}_i \otimes \mathcal{H}_n \otimes \mathcal{H}_l, \tag{3.5}$$

where the label qubit is in the state $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Now consider an ancillary qubit a, with Hilbert space $\mathcal{H}_a$, and prepare the state:

$$\frac{1}{\sqrt{2}}(|X\rangle|0\rangle + |\psi_{\mathbf{x}}\rangle|1\rangle) \in \mathcal{H}_i \otimes \mathcal{H}_n \otimes \mathcal{H}_l \otimes \mathcal{H}_a, \tag{3.6}$$

that can be recovered from the qRAM in time $O(\log_2(Nd))$. Now perform a SWAP test between the qubit a and a second ancillary qubit, named b, prepared in the state $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. Consider another qubit, c, prepared in the state $|0\rangle$ to control the Fredkin gate. In Figure 3.1 is shown the circuit representation of the SWAP test just described.

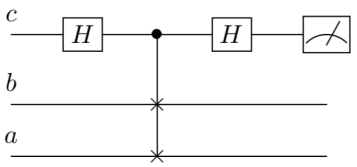


Figure 3.1: SWAP test between the ancillary qubits a and b, controlled by the ancillary qubit c.

After the action of SWAP test, the probability to obtain the outcome 1 measuring the qubit c is:

$$\mathbf{P}(1) = \frac{1}{4}(1 - \langle X|\psi_{\mathbf{x}}\rangle). \tag{3.7}$$

Within the amplitude encoding (3.3), the correspondence $\cos(\mathbf{x}, \mathbf{y}) = \langle \mathbf{x}|\mathbf{y}\rangle$ holds. Then the term $\langle X|\psi_{\mathbf{x}}\rangle$ can be written as:

$$\langle X|\psi_{\mathbf{x}}\rangle = \frac{1}{N\sqrt{2}} \sum_{i=0}^{N-1} y_i \cos(\mathbf{x}_i, \mathbf{x}). \tag{3.8}$$

Therefore, the probability $\mathbf{P}(1)$ is directly related to the prediction of the label of $\mathbf{x}$, according to the model (3.2). As a matter of fact, it is possible to rewrite (3.2) as:

$$y(\mathbf{x}) = \text{sign}(1 - 4\mathbf{P}(1)). \tag{3.9}$$

Therefore, if $\mathbf{P}(1) > 0.25$, $y(\mathbf{x}) = -1$; otherwise $y(\mathbf{x}) = 1$.

The preparation of states and SWAP test must be repeated several times for sampling the outcome of the measure of the qubit c to estimate $\mathbf{P}(1)$ as the success probability of a Bernoulli trial. An estimation within an error ϵ requires a number of repetitions on the order of ϵ^{-2} according to the binomial proportion confidence interval. Then, the time complexity of Algorithm 1 is $O(\epsilon^{-2} \log_2(Nd))$.

Algorithm 1 : Quantum implementation of the model (3.2)

Input: training set $X = \{\mathbf{x}_i, y_i\}_{i=0,\dots,N-1}$, unclassified data vector $\mathbf{x}$.
Output: y label of $\mathbf{x}$

```
repeat
    initialize the register H_i ⊗ H_n ⊗ H_l ⊗ H_a;
    initialize the ancillary qubit a in the state (3.6);
    initialize a qubit b in the state |+⟩;
    perform the SWAP test on a and b, with control qubit c prepared in |0⟩
(Figure: 3.1);
    measure qubit c;
until desired accuracy on the estimation of ℙ(1);
Estimate ℙ(1) (relative frequency ℙ̂ of outcome 1);
if ℙ̂ > 0.25 then
    return y = −1
else
    return y = 1
```

4 Implementation and results

This chapter reports the implementation of the algorithm described in chapter 3 for $N = d = 2$ and the obtained results.

4.1 Circuit setting

Consider a training dataset formed by two two-dimensional data instances, i.e. $N = d = 2$, given by $X = \{(\mathbf{x}_0, y_0), (\mathbf{x}_1, y_1)\}$, where $\mathbf{x}_0 = (1, 0)$, $y_0 = 1$, $\mathbf{x}_1 = (x_{10}, x_{11})$ and $y_1 = -1$. The model (3.2) predicts the labels for any $\mathbf{x} = (x_0, x_1) \in \mathbb{R}^2$ unit unlabelled data vector to classify as a *nearest neighbor*, for example as shown in Figure 4.1.

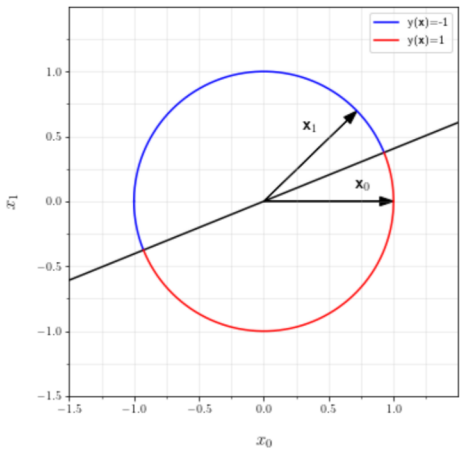


Figure 4.1: Labels assigned by model (3.2) within the training dataset $X = \{(\mathbf{x}_0 = (1, 0), y_0 = 1), (\mathbf{x}_1 = (0.714, 0.696), y_1 = -1)\}$ for any unit vectors $\mathbf{x} = (x_0, x_1) \in \mathbb{R}^2$.

Let $\mathbf{x} = \left(\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\right)$ be the parametric unlabelled data instance to classify, where θ is the parameter. The following states are prepared, according to (3.4) and (3.5), in order to run Algorithm 1:

$$|X\rangle = \frac{1}{\sqrt{2}}(|0\rangle\,|0\rangle\,|0\rangle + |1\rangle\,|\mathbf{x}_1\rangle\,|1\rangle) \tag{4.1}$$

and

$$|\psi_{\mathbf{x}}\rangle = |+\rangle\,|\mathbf{x}\rangle\,|-\rangle\,. \tag{4.2}$$

The quantum circuits for the construction of these quantum states are represented respectively in Figure 4.2 and in Figure 4.3.
The first is:

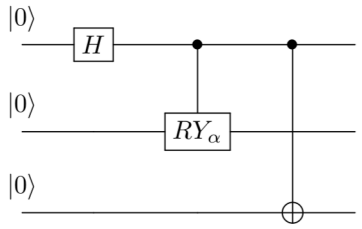


Figure 4.2: Quantum circuit implementing the state (4.1).

where RY_α is a rotation of an angle α around the y-axis of the Bloch sphere, mapping the input state of the second qubit $|0\rangle$ into the state $|\mathbf{x}_1\rangle = \cos\frac{\alpha}{2}\,|0\rangle + \sin\frac{\alpha}{2}\,|1\rangle$, which encodes the training vector $\mathbf{x}_1 = (\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2})$.
The second circuit is:

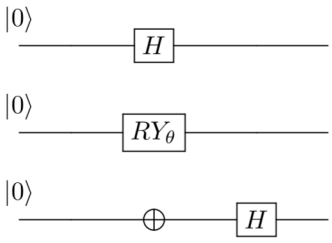


Figure 4.3: Quantum circuit implementing the state (4.2).

where the RY_θ gate rotates $|0\rangle$ into $|\mathbf{x}\rangle = \cos\frac{\theta}{2}\,|0\rangle + \sin\frac{\theta}{2}\,|1\rangle$ state. The circuit implementing Algorithm 1 for the case just introduced is represented in Figure 4.4.

The next step is to run this quantum algorithm on IBM quantum processors, with different quantum volume, varying the unclassified data vector $\mathbf{x}$, i.e. varying the parameter θ. For this purpose, it is useful to know for what angles $\theta \in [0, 4\pi]$ the probability $\mathbf{P}(1)$ assumes the maximum and minimum values.
Given any angle $\alpha \in [0, 4\pi]$, which defines the training dataset, the expression for the probability $\mathbf{P}(1)$ is:

$$\mathbf{P}(1) = \frac{1}{4}\left[1 - \frac{1}{2\sqrt{2}}\left(\cos\frac{\theta}{2} - \cos\frac{\theta}{2}\cos\frac{\alpha}{2} - \sin\frac{\theta}{2}\sin\frac{\alpha}{2}\right)\right], \tag{4.3}$$

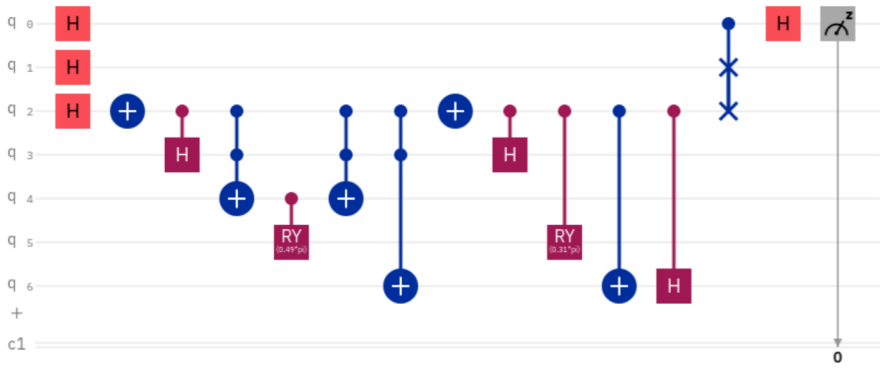


Figure 4.4: Quantum circuit implementing Algorithm 1 in the IBM Quantum Composer notation, where $N = 2$, $d = 2$.
The training set is $X = \{(\mathbf{x}_0 = (1,0), y_0 = 1), (\mathbf{x}_1 = \left(\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2}\right), y_1 = -1)\}$ and the unclassified data vector is $\mathbf{x} = \left(\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\right)$. In the figure, as an example, the values $\theta = 0.31\pi$ and $\alpha = 0.49\pi$ are reported.

where the product $\langle X|\psi_{\mathbf{x}}\rangle$ in the equation (3.7) is explicated. Hence, for a fixed angle α, the maximum value for the probability $\mathbf{P}(1)$ corresponds to the value of $\theta = \theta_{max}$, for $\theta \in [0, 4\pi]$, which is given by the following equation:

$$\theta_{max} = 2\tan^{-1}\left(\frac{\sin\frac{\alpha}{2}}{\cos\frac{\alpha}{2} - 1}\right) + 2\pi. \tag{4.4}$$

If $\theta_{max} < 2\pi$, the minimum of $\mathbf{P}(1)$ corresponds to $\theta_{min} = \theta_{max} + 2\pi$; otherwise, it corresponds to $\theta_{min} = \theta_{max} - 2\pi$.

Three training datasets were considered. The first was chosen to consider the value of $\alpha \in [0, 4\pi]$ which corresponds to a maximum range of expected probability, that is, for which $\mathbf{P}(1)_{max} - \mathbf{P}(1)_{min}$ is as large as possible. By differentiating the equation (4.3), it is found that its maximum with respect to $\alpha \in [0, 4\pi]$ and $\theta \in [0, 4\pi]$ is for $\alpha = \theta = 2\pi$. However, if $\theta = 2\pi$, the training states $|\mathbf{x}_0\rangle$ and $|\mathbf{x}_1\rangle$ differ only in a common phase factor $e^{i\pi}$, then they would be the same state for quantum mechanics. Therefore, an angle α in the neighbourhood of 2π was considered, i.e. $\alpha = 2\pi - 0.4$, to have a large range of probability $\mathbf{P}(1)$ and the two training state vectors quantistically distinguishable. The second training dataset was chosen with the two training states as quantistically different as possible, i.e. almost orthogonal. The case orthogonal was avoided because it is a limit case in which the two training vectors would be totally distinguishable. Therefore, the case with $\alpha = \pi - 0.1$ was considered. The last considered dataset is an intermediate case between the first two, i.e. with $\alpha = 1.5\pi$.

The 7-qubit IBM devices *ibmq_jakarta v1.0.33* and *ibm_lagos v1.0.27* were used, which have quantum volume of 16 and 32, respectively, referring to IBM's

definition of quantum volume. They are two of the IBM Quantum Falcon Processors. Processor types are named for the general technology qualities that go into builds, consisting of the family and revision. *Family* (e.g., Falcon) refers to the size and scale of circuits possible on the chip. This is primarily determined by the number of qubits and the connectivity graph. *Revisions* (e.g., r1) are design variants within a given family, often leading to performance improvements or tradeoffs. *Segments* are comprised of chip subsections, and are defined within a given family. For instance, segment H of a Falcon consists of seven qubits arranged as seen in Figure 4.5. Segment H on a Hummingbird (which is another family), if implemented, could be entirely different. Both processors used are part of the segment H of a Falcon.

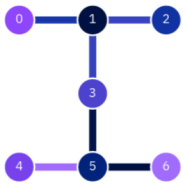


Figure 4.5: Illustration of segment H on a Falcon processor.

Both the ibmq_jakarta and ibm_lagos have as basis gates the following gates: CX, ID, RZ, SX (such that $SX^2 = X$), X.

For each probability estimate, therefore for each value of the θ angle, $2 \cdot 10^4$ repetitions were carried out. The simulator *simulator_mps* was used to have a comparison of the results from quantum machines with the results obtained from it, since the results obtained from the simulator can be considered as the expected results of the classification, within a negligible statistical error of the order of 10^{-3}.

4.2 Implementation using Qiskit

As already mentioned, Qiskit was used for programming and running the algorithm. This section reports the code step by step, commenting it.

- Importing libraries

```
#Math
import numpy as np
#Data storing
import pandas as pd
#Qiskit
from qiskit import IBMQ, transpile
from qiskit import QuantumCircuit, ClassicalRegister,
    QuantumRegister
```

```
from qiskit.providers.ibmq import least_busy
from qiskit.tools.monitor import job_monitor
```

- Functions

```
def initialize(qc, qubits):
    """Apply a H-gate to 'qubits' in qc (quantum circuit)"""
    for q in qubits:
        qc.h(q)
    return qc

def classification(results):
    """Given results (list of dictionaries with frequencies)
    returns a vector with the corrispective classifications."""
    classification = []
    for i in results:
        if i['1'] > n_iter/4:
            classification += [-1]
        else:
            classification +=[1]
    return classification
```

- Parameter definition

```
#non-modifiable parameters
pi = np.pi
circuit_list = []
#circuit's parameters
alpha = ...
theta_list = np.array([...])*pi
n_iter = 20000
#for using quantum devices
token = ...    #API token from IBM Quantum's account
prvd = ...     #provider from IBM Quantum's account
dvc = ...      #device you want to use
#saving
saving_path = '...'
```

- Circuit construction

```
#Inizialization of quantum register (7 qubits) and classical
    register (1 bit)
qreg_q = QuantumRegister(7, 'q')
creg_c = ClassicalRegister(1, 'c')

#Construction of the circuits (one for each value in theta_list);
    at the end of the for loop, circuit_list is containing all the
    circuits
```

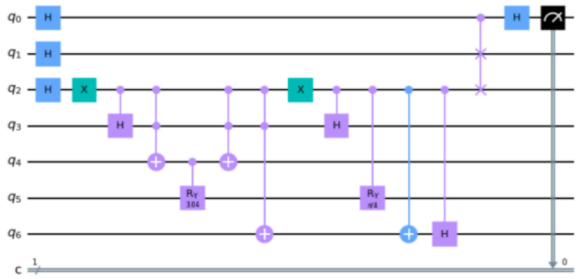
```
for theta in theta_list:
    circuit = QuantumCircuit(qreg_q, creg_c)

    circuit = initialize(circuit, [0,1,2])
    circuit.x(2)
    circuit.ch(2, 3)
    circuit.ccx(qreg_q[2], qreg_q[3], qreg_q[4])
    circuit.cry(alpha, qreg_q[4], qreg_q[5])
    circuit.ccx(qreg_q[2], qreg_q[3], qreg_q[4])
    circuit.ccx(qreg_q[2], qreg_q[3], qreg_q[6])
    circuit.x(qreg_q[2])
    circuit.ch(qreg_q[2], qreg_q[3])
    circuit.cry(theta, qreg_q[2], qreg_q[5])
    circuit.cx(qreg_q[2], qreg_q[6])
    circuit.ch(qreg_q[2], qreg_q[6])
    circuit.cswap(qreg_q[0], qreg_q[1], qreg_q[2])
    circuit.h(qreg_q[0])
    circuit.measure(qreg_q[0], creg_c[0])

    circuit_list += [circuit]

#As an example, I visualize the first circuit in the variable
    'circuit_list'
circuit_list[0].draw(output='mpl')
```

The output of the last code's line is the following:



The circuits have been constructed correctly.

- Compilation

```
#loading IBM account to have a provider of quantum devices
IBMQ.save_account(token)
provider = IBMQ.load_account()
provider = IBMQ.get_provider(prvd)
#print the list of available devices
available_cloud_backends = provider.backends()
```

```
print('\nHere is the list of cloud backends that are available to
    you:')
for i in available_cloud_backends: print(i)
#selecting the device to use
device = provider.get_backend(dvc)
#compilation of circuits with maximum optimization
transpiled_circuit_list = transpile(circuit_list, device,
    optimization_level=3)
```

- Run

```
#run
job = device.run(transpiled_circuit_list, shots = n_iter)
#monitoring of execution
job_monitor(job, interval=2)
```

- Get results

```
results = job.result()
answer = results.get_counts()
#grouping data in a dataframe
data = pd.DataFrame(answers)
data['Theta [pi]'] = theta_list/pi
columns_titles = ["Theta [pi]","1",'0']
data=data.reindex(columns=columns_titles)
data['Class'] = classification(answer)
#saving data
data.to_csv(saving_path)
```

At this point, the result is a .csv file containing the data needed for analysing the results that are discussed in the next section.

4.3 Results and discussion

This chapter reports the results obtained from the algorithm considered for the case with $N = d = 2$. The training dataset X is formed by the two vectors $\mathbf{x}_0 = (1, 0)$ and $\mathbf{x}_1 = (\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2})$. The test unlabelled vector to classify is defined by the parameter θ as $\mathbf{x} = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2})$. Three different training datasets are considered: the first is defined by $\alpha = 2\pi - 0.4$, i.e. the case with the training vectors almost antiparallel, the second by $\alpha = 1.5\pi$ which is the intermediate case between the almost antiparallel case and the almost orthogonal one, and the third by $\alpha = \pi - 0.1$, which is the case with the two training vectors almost orthogonal. For each of these training datasets, different test vectors are considered. Particular attention is given to the θ values in the range of the maximum and minimum of the probability of measure 1 as output.

Training dataset with $\alpha = 2\pi - 0.4$

Given the training dataset with $\alpha = 2\pi - 0.4$, i.e. $X = \{((1,0),1),((-0.980,0.199),-1)\}$, from the equation (4.4) the maximum for the probability $\mathbf{P}(1)$ is in correspondence with $\theta_{max} = 1.968\pi$.
The values $\{1\pi, 1.868\pi, 1.968\pi, 2.068\pi, 3\pi, 3.868\pi.3.968\pi\}$ were considered for the angle θ, which defines the unlabelled data vector. In Figure 4.6 are shown the results. As a reminder, if the estimate of probability of measure 1 as outcome is $\hat{\mathbf{P}}(1) > 0.25$, then the labels assigned to the unlabelled data vector $\mathbf{x} = \left(\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\right)$ is $y = -1$; otherwise it is $y = +1$.

Figure 4.6: Illustration of the results obtained by applying the Algorithm 1 implemented on the circuit represented in Figure 4.4 given the training dataset with $\alpha = 2\pi - 0.4$. On the x-axis, the angle θ values are reported in terms of π. The angle θ defines the unlabelled data vector $\mathbf{x} = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2})$. On the y-axis, the estimated probability $\hat{\mathbf{P}}(1)$ values are reported. The dotted line indicates the threshold $\hat{\mathbf{P}}(1) = 0.25$.

It is possible to see from Figure 4.6 that both the processors fail with the classification. In particular, both of them fail to classify the unlabelled vectors with expected label $y = 1$. The device ibmq_jakarta seems insensitive with respect to variations of angle θ, while the device ibm_lagos follow the simulator's trade, even if the variations of the estimated probability $\hat{\mathbf{P}}(1)$ are small. There is an obvious noise component covering the real probability estimate.

Training dataset with $\alpha = 1.5\pi$

It is considered now the case with the training dataset which is defined by $\alpha = 1.5\pi$, i.e. $X = \{((1,0),1),((-0.707,0.707),-1)\}$. In this case $\theta_{max} = 1.750\pi$.

The values $\{0.825\pi, 1.650\pi, 1.750\pi, 1.850\pi, 2.750\pi, 3.650\pi.3.750\pi, 3.850\pi\}$ are considered for the angle θ. In Figure 4.7 are shown the results.

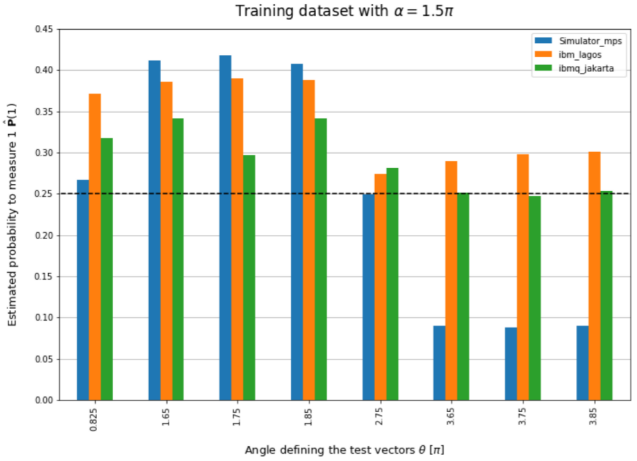


Figure 4.7: Illustration of the results obtained by applying the Algorithm 1 implemented on the circuit represented in Figure 4.4 given the training dataset with $\alpha = 1.5\pi$. On the x-axis, the angle θ values are reported in terms of π. The angle θ defines the unlabelled data vector $\mathbf{x} = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2})$. On the y-axis, the estimated probability $\hat{\mathbf{P}}(1)$ values are reported. The dotted line indicates the threshold $\hat{\mathbf{P}}(1) = 0.25$.

In this case, the ibmq_jakarta processor is not insensible to the variation of the angle θ and it succeeds in classify correctly one of the three test vectors labelled as $y = 1$. The ibm_lagos processor follows the simulator trade more markedly than in the previous case, but it fails in classification for vectors labelled as $y = 1$. Although the noise component is still present, the estimated probabilities are more precise than those of the previous case. This improvement in algorithm's performance might be explained from the fact that the two training states are more quantistically distinguishable than those of the previous case, which were almost the same state by a quantum physics point of view.

Training dataset with $\alpha = \pi - 0.1$

Now, the training dataset which is defined by the angle $\alpha = \pi - 0.1$, i.e. $X = \{((1, 0), 1), ((0.050, 0.999), -1)\}$, is considered. From the equation (4.4) the maximum for the probability $\mathbf{P}(1)$ is in correspondence with $\theta_{max} = 1.484\pi$. The values $\{0.25\pi, 1.384\pi, 1.484\pi, 1.584\pi, 2.1\pi, 3.384\pi.3.483\pi, 3.583\pi\}$ for the angle θ are considered, which defines the unlabelled data vectors. In Figure 4.8 are shown the results.

Figure 4.8: Illustration of the results obtained by applying the Algorithm 1 implemented on the circuit represented in Figure 4.4 given the training dataset with $\alpha = \pi - 0.1$. On the x-axis, the angle θ values are reported in terms of π. The angle θ defines the unlabelled data vector $\mathbf{x} = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2})$. On the y-axis, the estimated probability $\hat{\mathbf{P}}(1)$ values are reported. The dotted line indicates the threshold $\hat{\mathbf{P}}(1) = 0.25$.

It is possible to see from the illustration above that both the quantum processors predict all the labels, except the first, correctly. It can be seen that for classifications of the type $y = -1$, i.e. $\mathbf{P}(1) > 0.25$, the processors tend to underestimate the results obtained by the simulator, while for classifications of the type $y = 1$, i.e. for $\mathbf{P}(1) < 0.25$, they tend to overestimate them: it is because of this overestimation that the processors fail to label the first point. The processor ibm_lagos has the same behaviour as ibmq_jakarta, but the underestimates and overestimates are greater, except for the first test vector where ibm_lagos seems to estimate better the probability of measure 1 than ibmq_jakarta.

The origins of data fluctuations can be of various kinds. There is a statistical error due to the fact that the probabilities were estimated from Bernoulli trials, but since $2 \cdot 10^4$ repetitions were performed for each estimated probability, the statistical error is of the order of 10^{-3}, then it is negligible for the scope of this algorithm. This fact can be seen in Figure 4.9, where 100 probability estimates have been performed under the same conditions, i.e. same circuit and same transpiler's parameters. The standard deviation obtained is of the expected order of magnitude.

As it can be seen from Figure 4.9, another origin of fluctuations is the transpiler's seed. The transpiler has a stochastic part: for each value of seed, the transpiler outputs a different compiled circuit which gives different estimated probabilities. This stochastic part of the transpiler can be removed, setting a

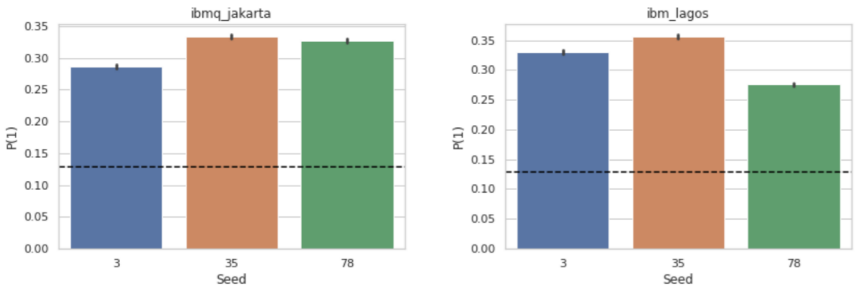


Figure 4.9: Barplot of estimated probabilities of measuring 1 for 3 different values of seed, with a training dataset defined by $\alpha = \pi - 0.1$ and for the input test vector for which the expected probability is the minimum, i.e. for $\theta = 3.484\pi$. The dotted line represents the expected value, $\mathbf{P}(1) = 0.128$. 100 repetitions were made for each value of seed. The height of bars corresponds to the mean value calculated over the 100 repetitions. The range indicated in black, at the top of the bars, indicates the standard deviation. The results concerning the ibmq_jakarta processor are reported on the left, those concerning the ibm_lagos are reported on the right.

value for the seed. Figure 4.10 reports the results for 100 different values of seed, which were taken randomly within an interval from 0 to 100.

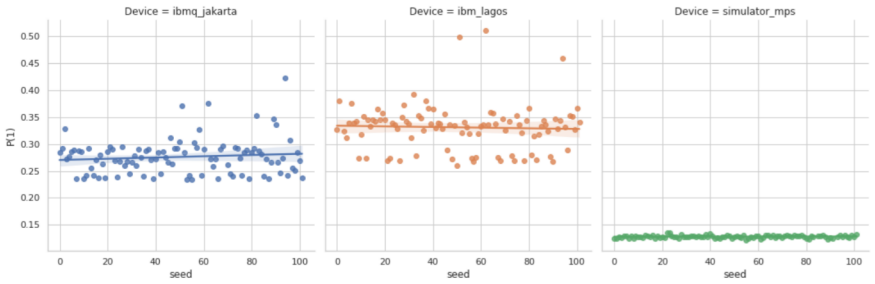


Figure 4.10: Scatterplot of estimated probabilities of measuring 1 as an output of the same circuit used for the Figure 4.9 for 100 different values of seed taken randomly from the interval $[0, 100]$. The straight line represents the linear regression made on data and the shaded region represents the 95% confidence interval, which was calculated from 10^4 bootstrap resample. The results concerning the ibmq_jakarta processor are reported on the left, those concerning the ibm_lagos are reported in the middle and those concering the simulator_mps are reported on the right.

The simulator behaves as expected, estimating correctly the probabilities except for small statistical fluctuations, independently of the seed value. In

addition to fluctuations due to different transpiler's random seed, there is an obvious noise component that leads to an overestimation of the estimated probabilities with respect to the expected value, which is $\mathbf{P}(1) = 0.128$ for the circuit considered. This last component of noise can be due to the hardware features of ibmq_jakarta and ibm_lagos processors.

The presence of the various noise components just discussed makes ibm_lagos performance worse than ibmq_jakarta, despite ibm_lagos have a higher quantum volume value than ibmq_jakarta.

5 Conclusions

For the first training dataset, the one with almost antiparallel training vectors, i.e. which is defined by $\alpha = 2\pi - 0.4$, the two devices are mostly insensitive to angle θ variations. This fact is probably due to the small difference between the training states, in addition to the noise. As proof of this fact, it can be seen that the more the two training states are quantistically distinguishable, the more the performance of the algorithm improves. Indeed, for the case with $\alpha = 1.5\pi$, there is an evident performance improvement of the algorithm, despite the classification is still wrong for almost all test vectors labelled as $y = 1$. The angle $\alpha = \pi - 0.1$ defines the training dataset with the two training states almost orthogonal, i.e. more quantistically distinguishable than the previous two cases. In this case, both the processors succeed to correctly classify all the unlabelled input vectors except the first one.

Despite the results obtained, the considered processors, ibmq_jakarta and ibm_lagos, are not accurate enough to apply the considered classifier in the general case. The main problem is a noise component that leads to overestimating the probability of measure 1, $\hat{\mathbf{P}}(1)$, resulting in difficulty in classifying input vectors as $y = 1$. Moreover, the less the training states are distinguishable, the more this noise component is evident. This noise component consists of a pure statistical part due to Bernoulli trials for estimating probability, a part due to the transpiler's random seed and another part probably due to hardware features, such as circuit depth, qubits connectivity, number of qubits and basis available gates. The first one is negligible if enough repetitions are performed. To consider the stochastic part of the transpiler, it would be possible to integrate over the seed's probability distribution. This point might be a possible start for future work.

In conclusion, due to the noise components described, the difference in results for different quantum volumes, 16 and 32, is not yet noticeable. This work may be useful for testing the same algorithm on future devices with greater quantum volume.

Bibliography

[Buh01] Harry Buhrman et al. "Quantum Fingerprinting". In: *Physical Review Letters* 87.16 (Sept. 2001).

[Vit08] Lorenzo Maccone Vittorio Giovannetti Seth Lloyd. "Architectures for a quantum random access memory". In: *Phys. Rev. Lett. 100, 160501* (2008).

[NC11] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2011. ISBN: 9781107002173.

[LMR13] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. *Quantum algorithms for supervised and unsupervised machine learning*. 2013.

[WKS14] Nathan Wiebe, Ashish Kapoor, and Krysta Svore. "Quantum Algorithms for Nearest-Neighbor Methods for Supervised and Unsupervised Learning". In: (2014).

[Ben17] Marcello Benedetti et al. "Quantum-Assisted Learning of Hardware-Embedded Probabilistic Graphical Models". In: *Physical Review X* 7.4 (Nov. 2017).

[BAT17] Peter Broecker, Fakher F. Assaad, and Simon Trebst. *Quantum phase recognition via unsupervised machine learning*. 2017.

[FN18] Edward Farhi and Hartmut Neven. *Classification with Quantum Neural Networks on Near Term Processors*. 2018.

[HDW18] Patrick Huembeli, Alexandre Dauphin, and Peter Wittek. "Identifying quantum phase transitions with adversarial neural networks". In: *Physical Review B* 97.13 (Apr. 2018).

[Mol18] Nikolaj Moll et al. "Quantum optimization using variational algorithms on near-term quantum devices". In: *Quantum Science and Technology* 3.3 (June 2018), p. 030503.

[Cro19] Andrew W. Cross et al. "Validating quantum computers using randomized model circuits". In: *Physical Review A* 100.3 (Sept. 2019).

[Gho19] Sanjib Ghosh et al. "Quantum reservoir processing". In: *npj Quantum Information* 5.1 (Apr. 2019).

[SK19] Maria Schuld and Nathan Killoran. "Quantum Machine Learning in Feature Hilbert Spaces". In: *Physical Review Letters* 122.4 (Feb. 2019).

[TQ19] Swamit Tannu and Moinuddin Qureshi. "Not All Qubits Are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers". In: Apr. 2019, pp. 987–999.

[Yu19] Shang Yu et al. “Reconstruction of a Photonic Qubit State with Reinforcement Learning”. In: *Advanced Quantum Technologies* 2.7-8 (Mar. 2019), p. 1800074.

[KKP20] Viraj Kulkarni, Milind Kulkarni, and Aniruddha Pant. *Quantum Computing Methods for Supervised Learning*. 2020.

[Pas20] Davide Pastorello. *Notes on Quantum Machine Learning*. Department of Information Engineering and Computer Science, University of Trento. 2020.

[ANI21] MD SAJID ANIS et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2021.

[PB21] Davide Pastorello and Enrico Blanzieri. “A Quantum Binary Classifier based on Cosine Similarity”. In: *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, Oct. 2021.

[Qua21] IBM Quantum. 2021. URL: `https://quantum-computing.ibm.com/`.

[D-W] D-Wave. URL: `https://www.dwavesys.com`.

[Goo] Google. URL: `https://quantumai.google/`.

[IBM] IBM. URL: `https://www.ibm.com/quantum-computing`.