EP-PQM: Efficient Parametric Probabilistic Quantum Memory With Fewer Qubits and Gates

MUSHAHID KHAN[1], JEAN PAUL LATYR FAYE[2], UDSON C. MENDES[2], AND ANDRIY MIRANSKYY[1] (Member, IEEE)

[1]Department of Computer Science, Ryerson University, Toronto, ON M5B 2K3, Canada
[2]CMC Microsystems, Sherbrooke, QC J1K 1B8, Canada

Corresponding author: Andriy Miranskyy (e-mail: avm@ryerson.ca)

ABSTRACT Machine learning (ML) classification tasks can be carried out on a quantum computer (QC) using probabilistic quantum memory (PQM) and its extension, parametric PQM (P-PQM), by calculating the Hamming distance between an input pattern and a database of r patterns containing z features with a distinct attributes. For PQM and P-PQM to correctly compute the Hamming distance, the feature must be encoded using one-hot encoding, which is memory intensive for multiattribute datasets with $a > 2$. We can represent multiattribute data more compactly by replacing one-hot encoding with label encoding; both encodings yield the same Hamming distance. Implementing this replacement on a classical computer is trivial. However, replacing these encoding schemes on a QC is not straightforward because PQM and P-PQM operate at the bit level, rather than at the feature level (a feature is represented by a binary string of 0's and 1's). We present an enhanced P-PQM, called efficient P-PQM (EP-PQM), that allows label encoding of data stored in a PQM data structure and reduces the circuit depth of the data storage and retrieval procedures. We show implementations for an ideal QC and a noisy intermediate-scale quantum (NISQ) device. Our complexity analysis shows that the EP-PQM approach requires $O(z \log_2(a))$ qubits as opposed to $O(za)$ qubits for P-PQM. EP-PQM also requires fewer gates, reducing gate count from $O(rza)$ to $O(rz \log_2(a))$. For five datasets, we demonstrate that training an ML classification model using EP-PQM requires 48% to 77% fewer qubits than P-PQM for datasets with $a > 2$. EP-PQM reduces circuit depth in the range of 60% to 96%, depending on the dataset. The depth decreases further with a decomposed circuit, ranging between 94% and 99%. EP-PQM requires less space; thus, it can train on and classify larger datasets than previous PQM implementations on NISQ devices. Furthermore, reducing the number of gates speeds up the classification and reduces the noise associated with deep quantum circuits. Thus, EP-PQM brings us closer to scalable ML on an NISQ device.

INDEX TERMS Efficient encoding, label encoding, quantum memory.

I. INTRODUCTION

Nowadays, classical algorithms play an important role in finding information in data [1]. However, due to the large amount of data that we have today, some problems require a lot of resources to be solved [1]. Recently, quantum computing has emerged as a promising candidate for solving large-scale data problems. Quantum computers (QCs) use quantum mechanical properties, such as superposition and entanglement, to perform computations. Machine learning (ML) is a promising field where QCs, specifically near-term intermediate scale [noisy intermediate-scale quantum (NISQ)] devices, can have potential applications. The purpose of quantum ML is to build quantum-enhanced ML models [2], [3]. These algorithms have proved to be faster than classical algorithms for a variety of tasks, including supervised and unsupervised ML [4]–[6], reinforcement learning [7], and support vector machine [8]. For a review on quantum ML, see [9] and [10].

In this article, we propose an enhancement to a quantum ML model that belongs to the quantum associative memory family of models [11]–[14]. Specifically, we focus on probabilistic quantum memory (PQM) [12], [13], which computes

the Hamming distance, deemed D, between an n-bit input pattern and a database of r patterns of length n.

PQM has strong characteristics, such as storing and simultaneously analyzing r patterns while using only n qubits. In other words, on a QC, one needs $O(n)$ qubits as opposed to $O(rn)$ bits of associative memory on a classical computer.

Despite this, PQM is not perfect. For example, suppose we have multiple input patterns to compare to the database of patterns. The comparison is made one input string at a time. This means that we need to reload[1] the database into a QC after every comparison because measuring D collapses the state of the QC [18], [19]. Since the reloading cost is $O(rn)$, it may outweigh the benefits of PQM, making it more efficient to compute D on a classical computer.

PQM can be useful for specialized tasks, e.g., one PQM execution may be sufficient for evaluating the artificial neural network architectures probabilistically without having to initialize the weights [20].

PQM can be used in supervised ML for pattern classification tasks. This is done by computing the probability that an input pattern belongs to a given database of patterns [14], [21]. A variety of datasets were used to demonstrate the applicability of this approach, ranging from detecting breast cancer to finding winning strategies in Tic-Tac-Toe [14], [21].

However, the current approach for codifying patterns as binary strings may be inefficient. Let us look at two examples.

Example I.1: Suppose that each of the pattern's features has only two attributes. In this case, we can efficiently codify the value of a feature using a single bit per feature. ∎

Example I.2: What if we have more than two attributes? Let us use a to represent the number of distinct attributes. To codify attributes, we can either use one-hot encoding or label encoding. In one-hot encoding, a single feature is represented by a binary variables, while label encoding maps the attributes to a integers. Therefore, one-hot encoding requires a bits to represent a feature, while label encoding only needs $\lceil \log_2(a) \rceil$ bits, where $\lceil \cdot \rceil$ denotes ceiling function. Thus, label encoding is more memory-efficient than one-hot encoding. This difference is significant as a real-world dataset (e.g., NSL-KDD [22]) may contain hundreds of distinct attributes. For example, one-hot encoding requires 100 bits for a feature with 100 distinct attributes, whereas label encoding requires only 7 bits. This exponential decrease in the number of qubits and gates may make NISQ devices capable of processing complex multiattribute patterns.

Consider patterns containing three features: an input pattern $\mu_0 = $ "A A A" and a database with two patterns $\mu_1 = $ "B B B" and $\mu_2 = $ "C C A." We would like to compute the distance between μ_0 and two patterns in the database. It is obvious that $D(\mu_0, \mu_1) = 3$ and $D(\mu_0, \mu_2) = 2$.

Let us now demonstrate how the above example is implemented using PQM. As PQM operates at the bit level, the patterns must be converted to bit strings. We will use the label encoding to represent the three attributes, which requires two bits per attribute. Suppose that "A" is codified as "00," "B" as "01," and "C" as "11." This encoding converts the patterns to the following bit strings: $\mu_0 = $ "00 00 00," $\mu_1 = $ "01 01 01," and $\mu_2 = $ "11 11 00." PQM operates on individual bits, thus $D_{\text{label}}(\mu_0, \mu_1) = 3$ and $D_{\text{label}}(\mu_0, \mu_2) = 4$.

As we can see, $D(\mu_0, \mu_1) > D(\mu_0, \mu_2)$, but $D_{\text{label}}(\mu_0, \mu_1) < D_{\text{label}}(\mu_0, \mu_2)$. Consequently, label encoding, while efficiently utilizing memory, may lead to erroneous results.

To overcome this problem, we must resort to one-hot encoding, in which attributes are codified as follows: "A" as "100," "B" as "010," and "C" as "001." The patterns will now be represented by $\mu_0 = $ "100 100 100," $\mu_1 = $ "010 010 010," and $\mu_2 = $ "001 001 100." PQM will yield $D_{\text{one-hot}}(\mu_0, \mu_1) = 6$ and $D_{\text{one-hot}}(\mu_0, \mu_2) = 4$. Note that $D_{\text{one-hot}} = 2D$ (see Appendix A for details); thus, one-hot encoding yields the desired values of D.

Therefore, one-hot encoding allows PQM to compute D correctly, whereas label encoding does not. However, one-hot encoding consumes more space: nine qubits will be required instead of six. ∎

Based on the above examples, to compute D correctly, it appears that PQM can only use one-hot encoding for features with multiple attributes. Consequently, the pattern must be represented using a large amount of space. Modern QCs, however, have a small number of qubits and limited coherence. Therefore, we need to find a way to use label encoding to reduce space constraints and utilize qubits more efficiently.

In this article, our contribution is as follows. We generalize the PQM algorithm to compute the Hamming distance D for features codified with label encoding. Specifically, we improve the parametric PQM (P-PQM) classification algorithm designed for modern NISQ computers [14] to use fewer qubits and quantum gates. This efficient version of P-PQM, which we call EP-PQM, results in

1) space savings, as for z features the number of required qubits reduces from $O(za)$ to $O(z \log_2(a))$ for $a > 2$;
2) improved computational efficiency, as the amount of quantum gates needed to execute the PQM algorithm is proportional to the number of bits required to represent a pattern. The number of gates reduces from $O(rza)$ to $O(rz \log_2(a))$.

With EP-PQM requiring fewer qubits and gates to implement PQM than P-PQM, it becomes a better candidate for applications in NISQ devices.

The rest of this article is structured as follows. Section II recaps the PQM and P-PQM algorithms [14]. In Section III,

---

[1]This concern may be alleviated by probabilistic cloners for QC [13], which may create approximate copies of the states for QC. There are currently no practical probabilistic cloners for QC, but research is ongoing in this area [15]–[17].

we introduce the EP-PQM. Section IV provides sample experiments with ML datasets. Finally, in Section V, we present our final remarks and conclusions.

II. PROBABILISTIC QUANTUM MEMORY

In this section, we review the PQM and P-PQM algorithms. The PQM [12], [13] algorithm introduces a data structure that allows computing the D distance at the bit level between a binary input pattern and all other binary patterns stored in a superposition state. PQM has two parts: storing information and retrieving information, discussed in Sections II-A and II-B, respectively. The core quantum computing fundamentals needed to implement these algorithms are introduced in Appendix A.

To store information, PQM will take a dataset of r binary patterns, each of n bits and store them in a superposition state with equal probability. Given an input, to retrieve information, the memory quantum state is rotated within the subspace defined by the stored patterns. The resulting amplitudes are peaked on the stored patterns which are closest in Hamming distance D to the input.

P-PQM is similar in nature to PQM, but adds an extra parameter that may improve ML classification.

A. PQM AND P-PQM: STORING INFORMATION

This section recaps the existing PQM and P-PQM data storage processes, leveraging the same algorithm.

Formally, the storing information part of the algorithm receives a dataset of r binary patterns, each containing n bits: data $= \cup_{i=1}^{r}\{p^i\}$. To store the patterns, three quantum registers are needed: the input register p, the memory register m, and the auxiliary two-qubit register u. The input register state $|p\rangle$ will hold every pattern of length n. $|m\rangle$ is the memory register that will store each pattern p^i by the end of the algorithm. The auxiliary two-qubit register state $|u\rangle$ is used to keep tabs on which patterns are stored in memory and which ones need to be processed. The first qubit in $|u\rangle$ is used to change the second qubit in $|u\rangle$. The second qubit in $|u\rangle$ indicates whether a pattern has been already stored or not. In this case, $|u\rangle = 1$ indicates that the pattern has not been stored yet. To make a copy of the n bits from the pattern in $|p\rangle$ to the respective register $|m\rangle$, the algorithm checks if the pattern has been stored on $|m\rangle$ by checking if the second qubit in $|u\rangle$ is 1. If it is 1, the pattern will be copied to $|m\rangle$. All three quantum registers are initialized in the state $|0\rangle$ and the algorithm initial state is

$$|\psi_0\rangle_1 = |\underbrace{0_1 0_2 \ldots 0_n}_{|p\rangle}; \underbrace{01}_{|u\rangle}; \underbrace{0_1 0_2 \ldots 0_n}_{|m\rangle}\rangle. \quad (1)$$

The storage process loads each pattern (represented as a binary string) into $|p\rangle$ and then stores them in $|m\rangle$ in a superposition with equal probabilities. Once the patterns are loaded into $|m\rangle$, they will be processed to find the closest pattern match, as we will discuss in Section II-B.

The storage process is given in Algorithm 1; see [14] for an in-depth explanation of the storing algorithm. This algorithm

---

Algorithm 1: PQM and P-PQM storage algorithm [14]. In control gate $C^\alpha NOT_{m_1 \cdots m_n, u_1}$ qubits $m_1 \cdots m_n$ act as controls and qubit u_1 is a target. For more details, see Appendix B.

---

input : The initial state $|\psi_0\rangle_1$ given in Eq. (1)
output : Return $|\psi_8^r\rangle_1$ defined in Eq. (2)

1   $|\psi_1^1\rangle = |\psi_0\rangle_1$
2   **foreach** $p^i \in data$ **do**
3   Load p^i into quantum register $|p\rangle_n$
4     $|\psi_2^i\rangle = \prod_{j=1}^{n} C^2 NOT_{p_j^i, u_2, m_j}|\psi_1^i\rangle$
5     $|\psi_3^i\rangle = \prod_{j=1}^{n} X_{m_j} CNOT_{p_j^i, m_j}|\psi_2^i\rangle$
6     $|\psi_4^i\rangle = C^n NOT_{m_1 \cdots m_n, u_1}|\psi_3^i\rangle$
7     $|\psi_5^i\rangle = CS_{u_1, u_2}^{r+1-i}|\psi_4^i\rangle$
8     $|\psi_6^i\rangle = C^n NOT_{m_1 \cdots m_n, u_1}|\psi_5^i\rangle$
9     $|\psi_7^i\rangle = \prod_{j=1}^{n} CNOT_{p_j^i, m_j} X_{m_j}|\psi_6^i\rangle$
10    $|\psi_8^i\rangle = \prod_{j=1}^{n} C^2 NOT_{p_j^i, u_2, m_j}|\psi_7^i\rangle$
11    $|\psi_1^{i+1}\rangle = |\psi_8^i\rangle$
12   Unload p^i from quantum register $|p\rangle_n$
13 The final state after completion of the for-loop will be $|\psi_8^r\rangle_1$

---

uses common quantum computing gates recapped in Appendix A. In addition to these gates, Step 7 of the algorithm uses a two-qubit control gate CS^j, which adds a pattern p^i to the memory register with uniform amplitudes. It is defined as follows:

$$CS^j = |0\rangle\langle 0| \otimes 1 + |1\rangle\langle 1| \otimes S^j$$

where $j \in \mathbb{Z}$ and $j = 1, 2, ..., r$, and

$$S^j = \begin{bmatrix} \sqrt{\frac{j-1}{j}} & \frac{1}{\sqrt{j}} \\ \frac{-1}{\sqrt{j}} & \sqrt{\frac{j-1}{j}} \end{bmatrix}.$$

Thus

$$CS^j = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{\frac{j-1}{j}} & \frac{1}{\sqrt{j}} \\ 0 & 0 & \frac{-1}{\sqrt{j}} & \sqrt{\frac{j-1}{j}} \end{bmatrix}.$$

The final result of the algorithm is the state

$$|\psi_8^r\rangle_1 = \frac{1}{\sqrt{r}} \sum_{k=1}^{r} |0_1 0_2 \ldots 0_n; 01; m_1^k m_2^k \ldots m_n^k\rangle. \quad (2)$$

Note that in the retrieval phase, we are only concerned with the memory-register state $|m\rangle$.

B. PQM AND P-PQM: RETRIEVING INFORMATION

PQM and P-PQM retrieval processes—described in Sections II-B1 and II-B2, respectively—are similar. PQM retrieval algorithm only has an implementation for fault-tolerant QCs, while P-PQM also has an implementation designed for NISQ devices [14].

Algorithm 2: PQM retrieval algorithm — fault-tolerant implementation [14].

input : The initial state $|\psi_0\rangle_2$ given in Eq. (4)
output : 1) The value of c and 2) one of the stored
patterns that has minimum D with the target
pattern in a given run

1 $|\psi_1\rangle = \mathrm{H}_c |\psi_0\rangle_2$
2 $|\psi_2\rangle = \prod_{j=1}^{n} \mathrm{X}_{m_j} \mathrm{CNOT}_{t_j,m_j} |\psi_1\rangle$
3 $|\psi_3\rangle = \prod_{e=1}^{n} \left(G\tilde{U}^{-2} \right)_{c,m_e} \prod_{j=1}^{n} \tilde{U}_{m_j} |\psi_2\rangle$
4 $|\psi_4\rangle = \mathrm{H}_c \prod_{j=n}^{1} \mathrm{CNOT}_{t_j,m_j} \mathrm{X}_{m_j} |\psi_3\rangle$
5 Measure qubit $|c\rangle$
6 **if** $c == 0$ **then**
7   $\quad$ Measure the memory to obtain the desired state.

---

1) PQM: RETRIEVING INFORMATION

The algorithm for retrieving information relies on the memory register m in state $|\psi_8^r\rangle_1$ (i.e., the output of the storage algorithm). This state is then further manipulated to perform pattern analysis as described in the following.

PQM uses the Hamming distance D between a target pattern and all patterns, which are stored in a superposition, to indicate probabilistically the chances of the target pattern being in the memory. This algorithm uses three quantum registers, namely t, m, and c

$$\underbrace{t_1 t_2 \dots t_n}_{t}; \underbrace{m_1 m_2 \dots m_n}_{m}; \underbrace{c}_{c}. \tag{3}$$

The target pattern, deemed T and represented by bits $\tau_1 \tau_2 \dots \tau_n$, is loaded into register state $|t\rangle$; $|m\rangle$ contains all the stored patterns from the storage algorithm; and $|c\rangle$ contains a control qubit initialized in state $|0\rangle$. Once the input has been loaded on to $|t\rangle$ and the stored patterns are in $|m\rangle$, the full initial quantum state is

$$|\psi_0\rangle_2 = \frac{1}{\sqrt{r}} \sum_{k=1}^{r} |t_1 t_2 \dots t_n; m_1^k m_2^k \dots m_n^k; 0\rangle \tag{4}$$

where r is the total number of stored patterns, $t_1 t_2 \dots t_n$ are qubits used to store corresponding bits $\tau_1 \tau_2 \dots \tau_n$ of the target pattern T, and $m_1^k m_2^k \dots m_n^k$ is the kth stored pattern.

The retrieval process is summarized in Algorithm 2. In Step 1, we apply the Hadamard gate on to the control qubit $|c\rangle$ to get

$$|\psi_1\rangle = \frac{1}{\sqrt{2r}} \left(\sum_{k=1}^{r} |t_1 t_2 \dots t_n; m_1^k m_2^k \dots m_n^k; 0\rangle \right.$$
$$\left. + \sum_{k=1}^{r} |t_1 t_2 \dots t_n; m_1^k m_2^k \dots m_n^k; 1\rangle \right). \tag{5}$$

Step 2 sets the jth qubit in register $|m\rangle$ to $|1\rangle$ if the jth qubit of $|t\rangle$ and $|m\rangle$ are the same or to $|0\rangle$ if they differ.

Step 3 computes D between the target pattern and all patterns in $|m\rangle$. The number of zeros in $|m\rangle$ (representing the qubits that differ between memory and target string) is

computed. Operator $\tilde{U}$, used in this step, is defined as

$$\tilde{U} = \begin{bmatrix} \exp\left(\frac{i\pi}{2n}\right) & 0 \\ 0 & 1 \end{bmatrix} \tag{6}$$

where i denotes the unit imaginary number. First, $\tilde{U}$ is applied to each qubit in $|m\rangle$. Then $\tilde{U}^{-2}$ (which is $\tilde{U}$ to the power of -2) is applied to each qubit in $|m\rangle$ if the qubit is in state $|c\rangle = |1\rangle$. This control operator is denoted by G. As per [12], $G\tilde{U}^{-2}$ is formally defined as

$$G\tilde{U}^{-2} = |0\rangle\langle 0| \otimes 1 + |1\rangle\langle 1| \otimes \tilde{U}^{-2}.$$

Step 4 reverts register $|m\rangle$ to its original state and the Hadamard gate H is applied to the control qubit in $|c\rangle$; this operation is denoted by H_c. After Step 4, the state will be

$$|\psi_4\rangle = \frac{1}{\sqrt{r}} \left(\sum_{k=1}^{r} \cos\frac{\pi}{2n} d_k |t_1 t_2 \dots t_n; m_1^k m_2^k \dots m_n^k; 0\rangle \right.$$
$$\left. + \sum_{k=1}^{r} \sin\frac{\pi}{2n} d_k |t_1 t_2 \dots t_n; m_1^k m_2^k \dots m_n^k; 1\rangle \right) \tag{7}$$

where d_k is D between the target pattern t and stored pattern m^k.

Step 5 measures register $|c\rangle$. A target pattern similar to the stored patterns increases the probability of measuring $|c\rangle = |0\rangle$. Otherwise, if the target pattern is dissimilar, then the probability of $|c\rangle = |1\rangle$ increases. If $|c\rangle$ is measured in $|0\rangle$ (see Step 6), then measuring the qubits in the memory register (in Step 7) will return the binary pattern from the set of stored patterns that produced the minimum D with the target pattern in a given run of the algorithm, with the following probability:

$$P\left(m^k\right) = \begin{cases} 0, & \text{if } |c\rangle = |1\rangle \\ \frac{1}{rP(|c\rangle=|0\rangle)} \cos^2\left(\frac{\pi}{2n} d_k\right), & \text{otherwise.} \end{cases} \tag{8}$$

This probability peaks around the patterns which have smallest D to t.

2) P-PQM: RETRIEVING INFORMATION

In a nutshell, P-PQM operates as the PQM, but with the addition of a scale parameter $\nu \in (0, 1]$ in the retrieval algorithm. The parameter ν is used to compute weighted D, which may improve the performance of the classifier [14].

To integrate ν into the retrieval algorithm, (6) is redefined as

$$U = \begin{bmatrix} \exp\left(\frac{i\pi}{2n\nu}\right) & 0 \\ 0 & 1 \end{bmatrix}. \tag{9}$$

Note that the quantum circuit depth and complexity are independent of ν, and that P-PQM reduces to PQM when $\nu = 1$. Let us give a brief summary of both fault-tolerant and NISQ implementations (see [14] for additional details).

2) a) Fault-tolerant implementation

The algorithm for the quantum implementation of the P-PQM retrieval procedure is almost identical to the PQM one (see Algorithm 2). The only difference is that (6) is replaced with (9). The input into the algorithm, as in the PQM case, is given by (5).

2) b) NISQ implementation

The NISQ implementation of the P-PQM retrieval algorithms is shown in Algorithm 3. This implementation requires only two registers, namely, m and c

$$\underbrace{m_1 m_2 \ldots m_n}_{m}; \underbrace{c}_{c} \qquad (10)$$

and the input is stored classically. Once the stored patterns from the storing part of the algorithm are in $|m\rangle$, the full initial quantum state is

$$|\psi_0\rangle_3 = \frac{1}{\sqrt{r}} \sum_{k=1}^{r} |m_1^k m_2^k \ldots m_n^k; 0\rangle. \qquad (11)$$

Because the input is stored classically, all the control operators from the input register to the memory register are removed from the circuit and replaced with a x operator. Essentially, the input pattern is processed dynamically without the need for a dedicated register. In the NISQ implementation [14], Steps 2 and 4 of the fault-tolerant implementation (see Algorithm 2) are modified. Specifically, they are replaced by Steps 2–4 and 6–8 of Algorithm 3. In this NISQ implementation, rather than applying x based on CNOT to each qubit, we examine each τ_j bit in T on the classical computer and apply x only if $\tau_j = 1$. This leads to "inversion" of logic; thus, we are now interested in measuring $|m\rangle$ when $c = 1$ rather than when $c = 0$ (compare Steps 6–7 of Algorithm 2 with Steps 11–12 of the Algorithm 3).

C. POSTPROCESSING ON CLASSICAL COMPUTER

PQM- and P-PQM-based ML have probabilistic nature. To get N measurements, we need to run the storage and retrieval algorithms N times (because after the memory registers are measured, we need to reinitialize the database).

We collect N measurements of $|c\rangle$ for PQM-based ML classification (whether PQM or P-PQM). Suppose we measure $M \leq N$ instances when $|c\rangle = 0$ (for fault-tolerant Algorithm 2) or when $|c\rangle = 1$ (for NISQ Algorithm 3). Then the affinity of an input pattern belonging to a given database of patterns, deemed ρ, is given by $\rho = M/N$. The closer ρ is to 1, the closer the input pattern is to the database of patterns.

To infer the class/label of an input pattern, an analyst needs to construct individual pattern databases for each class/label. Next, the analyst will compute ρ for each database. Finally, the analyst will assign a label to the input pattern based on the label of the database with the highest ρ. In this article, we do not compute ρ as our goal is to improve the storage and retrieval process used to obtain individual measurements. To learn more about PQM-based ML classification, see [14].

Algorithm 3: P-PQM retrieval algorithm — NISQ implementation [14].

> **input** : The initial state $|\psi_0\rangle_3$ given in Eq. (11)
> **output** : 1) The value of c and 2) one of the stored patterns that has minimum D with the target pattern in a given run

1   $|\psi_1\rangle = \mathrm{H}_c |\psi_0\rangle_3$
2   **foreach** $\tau_j \in T$ **do**
3   **if** $\tau_j == 1$ **then**
4       $|\psi_2\rangle = \mathrm{X}_{m_j} |\psi_1\rangle$
5   $|\psi_3\rangle = \prod_{e=1}^{n} \left(GU^{-2} \right)_{c, m_e} \prod_{j=1}^{n} U_{m_j} |\psi_2\rangle$
6   **foreach** $\tau_j \in T$ **do**
7   **if** $\tau_j == 1$ **then**
8       $|\psi_4\rangle = \mathrm{X}_{m_j} |\psi_3\rangle$
9   $|\psi_5\rangle = \mathrm{H}_c |\psi_4\rangle$
10   Measure qubit $|c\rangle$
11   **if** $c == 1$ **then**
12   Measure the memory register to obtain the desired state.

Note that the last two steps of Algorithms 2 and 3 suggest to measure the values of specific patterns in the database. On an ideal QC, we can get rid of these steps, as all the required information needed to compute ρ would be given to us by measuring qubit $|c\rangle$. However, for NISQ devices, the situation is different: we may end up with measuring a pattern that has not been stored in the database. This happens due to the noisy nature of NISQ devices. In this case, we may need to implement an additional postprocessing scheme. For example, one may assume that information about the value of $|c\rangle$ is important and should be included in the computation of ρ independent of the value of the pattern. Another approach would involve discarding the measurements of patterns that have not been present in the database. Answering this question is outside of the scope of this article.

III. EFFICIENT PARAMETRIC PROBABILISTIC QUANTUM MEMORY

In this section, we cover our extension of P-PQM. Storing of information is discussed in Section III-A and retrieval in Section III-B. For both storage and retrieval, we design fault-tolerant and NISQ implementations. Both of these implementations of EP-PQM typically require fewer qubits and gates than PQM and P-PQM implementations, as we will demonstrate as follows.

A. EP-PQM: STORING INFORMATION

1) FAULT-TOLERANT IMPLEMENTATION

For a fault-tolerant implementation, we will reuse the PQM and P-PQM storage procedures shown in Algorithm 1.

2) NISQ IMPLEMENTATION

EP-PQM storage procedure suitable for NISQ device is given in Algorithm 4. The latter requires two registers, the memory

---

Algorithm 4: EP-PQM storage algorithm — NISQ implementation.

   input : The initial state $|\psi_0\rangle_4$ given in Eq. (12)
   output : Return $|\psi_6^r\rangle_4$ defined in Eq. (13)

1 Prepare the initial state $|\psi_1^1\rangle = |\psi_0\rangle_4$
2 **foreach** $p^i \in data$ **do**
       `/* `p_j^i` denotes the j-th bit of the i-th pattern `p_i`.                          */`
3   **foreach** $p_j^i \in p^i$ **do**
4   **if** $p_j^i == 1$ **then**
5           $|\psi_2^i\rangle = \text{CNOT}_{u_2,m_j}|\psi_1^i\rangle$
6   **else**
7           $|\psi_2^i\rangle = \text{X}_{m_j}|\psi_1^i\rangle$
8     $|\psi_3^i\rangle = \text{C}^n\text{NOT}_{m_1\cdots m_n,u_1}|\psi_2^i\rangle$
9     $|\psi_4^i\rangle = \text{CS}_{u_1,u_2}^{r+1-i}|\psi_3^i\rangle$
10    $|\psi_5^i\rangle = \text{C}^n\text{NOT}_{m_1\cdots m_n,u_1}|\psi_4^i\rangle$
11   **foreach** $p_j^i \in p^i$ **do**
12   **if** $p_j^i == 1$ **then**
13          $|\psi_6^i\rangle = \text{CNOT}_{u_2,m_j}|\psi_5^i\rangle$
14   **else**
15          $|\psi_6^i\rangle = \text{X}_{m_j}|\psi_5^i\rangle$
16    $|\psi_1^{i+1}\rangle = |\psi_6^i\rangle$
17 The final state after completion of the for-loop will be $|\psi_6^r\rangle_4$

---

register $|m\rangle$ with n qubits and an auxiliary two-qubit register $|u\rangle$, as defined in Section II-A. The algorithm starts with the following initial state:

$$|\psi_0\rangle_4 = |\underbrace{01}_{|u\rangle}; \underbrace{0_10_2\ldots0_n}_{|m\rangle}\rangle \quad (12)$$

where all the qubits in memory register are in state $|0\rangle$ and the auxiliary register is in state $|0, 1\rangle$. Since in EP-PQM the input patterns do not require a dedicated register, as they can be stored in a classical computer, this algorithm requires less qubits for the storage part. Indeed, comparing (12) and (1), we see that EP-PQM storage part uses $n + 2$ qubits versus $2n + 2$ qubits for PQM and P-PQM. After the execution of Algorithm 4, the final state is a superposition of the input patterns with equal probabilities

$$|\psi_6^r\rangle_4 = \frac{1}{\sqrt{r}}\sum_{k=1}^{r}|01; m_1^k m_2^k \ldots m_n^k\rangle. \quad (13)$$

In Algorithm 4, since the input pattern is stored classically, all the control operators acting on the input register are removed. Instead, we can classically check if the input pattern consists of 0 or 1. As a result, Steps 4, 5, 9, and 10 of Algorithm 1 are changed. These changes can be seen in Steps 3–7 and 11–15 of Algorithm 4. Steps 3–7 are used to make a copy of the n bits of p^i to the memory register if the u_2 is flagged as 1, and then fill with 1's all the bits in the memory register which are equal to the respective bits in p^i. Steps 11–15 will reverse the work done in Steps 3–7.

B. EP-PQM: RETRIEVING INFORMATION
1) FAULT-TOLERANT IMPLEMENTATION

As explained in Section II-C, after N measurements, PQM outputs the probability of a target pattern being close to patterns in the database at the bit level. As discussed in Section I, this approach is ineffective if we compute D for symbols represented by multiple bits. For this reason, we extend the information retrieval part of PQM given in Algorithm 2.

Our extension generalizes the PQM algorithm to compute D for features codified with label encoding. The extension requires four registers, namely, t, m, c, and h

$$\underbrace{t_1t_2\ldots t_n}_{t}; \underbrace{m_1m_2\ldots m_n}_{m}; \underbrace{c}_{c}; \underbrace{h_1h_2\ldots h_z}_{h}. \quad (14)$$

Registers t, m, and c are the same as in (3), i.e., the PQM case discussed in Section II-B1.

The register h is used to compare features. The number of qubits in h is equal to the number of features in the pattern, deemed z. Note that $z = n/d$, where d is the number of bits required to represent an attribute of a feature, i.e., $d = \lceil\log_2(a)\rceil$. The jth qubit in register h is set to $|1\rangle$ if the binary string of length d that represents the jth feature of T is the same as the corresponding binary string in m. With $|h\rangle$ included, the initial quantum state for retrieval of information is

$$|\psi_0\rangle_5 = \frac{1}{\sqrt{r}}\sum_{k=1}^{r}|t_1t_2\ldots t_n; m_1^k m_2^k \ldots m_n^k; 0; 1_11_2\ldots 1_z\rangle \quad (15)$$

where all qubits in register $|h\rangle$ are initially in state $|1\rangle$ and the register $|c\rangle$ in state $|0\rangle$.

Algorithm 5: EP-PQM retrieval algorithm — fault-tolerant implementation.

input : The initial state $|\psi_0\rangle_5$ given in Eq. (15)
output : 1) The value of c and 2) one of the stored patterns that has minimum D with the target pattern in a given run

1   $|\psi_1\rangle = \mathrm{H}_c |\psi_0\rangle_5$
2   $|\psi_2\rangle = \prod_{j=1}^{n} \mathrm{X}_{m_j} \mathrm{CNOT}_{t_j, m_j} |\psi_1\rangle$
3   $|\psi_3\rangle = \prod_{j=1}^{z} \mathrm{X}_{h_j} \mathrm{C^d NOT}_{m_{d(j-1)+1} m_{d(j-1)+2} \ldots m_{dj}, h_j} |\psi_2\rangle$
4   $|\psi_4\rangle = \prod_{e=1}^{z} \left(GW^{-2} \right)_{c, h_e} \prod_{j=1}^{z} W_{h_j} |\psi_3\rangle$
5   $|\psi_5\rangle = \prod_{j=z}^{1} \mathrm{C^d NOT}_{m_{d(j-1)+1} m_{d(j-1)+2} \ldots m_{dj}, h_j} \mathrm{X}_{h_j} |\psi_4\rangle$
6   $|\psi_6\rangle = \mathrm{H}_c \prod_{j=n}^{1} \mathrm{CNOT}_{t_j, m_j} \mathrm{X}_{m_j} |\psi_5\rangle$
7   Measure qubit $|c\rangle$
8   **if** $c == 0$ **then**
9 | Measure the memory register to obtain the desired state.

With this extension, the retrieval algorithm is changed, as shown in Algorithm 5. The first and second steps are the same as the original Algorithm 2. In Step 3, results from Step 2 are used to update register $|h\rangle$. Given that each symbol is represented by a binary string of length d and each input is of length n, Step 3 will set the jth qubit in $|h\rangle$ to $|0\rangle$. This will happen if the binary string of length d, which represents the jth symbol of T, is not the same as the corresponding binary string in $|m\rangle$. With register $|h\rangle$ updated in Step 3, $|h\rangle$ can be used in place of $|m\rangle$ for Step 4. In Step 4, we use operator W instead of U, which is defined as

$$W = \begin{bmatrix} \exp\left(\frac{i\pi}{2zv}\right) & 0 \\ 0 & 1 \end{bmatrix}.$$

We use W to calculate D at the feature level. In Steps 5 and 6, inverse transformations of Steps 2 and 3 are applied and H gate is applied to the control qubit. In Step 7, $|c\rangle$ is measured. If $|c\rangle$ is measured in state $|0\rangle$, it means that the input is close to all stored patterns in the dataset. This probability is peaked around those patterns which have the smallest D to the input at the feature level. The highest probability of retrieval, thus, occurs for patterns that are most similar to the input at the feature level. Finally, Steps 7–9 are identical to Steps 5–7 of Algorithm 2.

2) NISQ IMPLEMENTATION

Akin to the NISQ implementation [14], we decided to store the input pattern classically. This implementation requires only three registers: m, c, and h, with n, 1, and z qubits, respectively

$$\underbrace{m_1 m_2 \ldots m_n}_{m}; \underbrace{c}_{c}; \underbrace{h_1 h_2 \ldots h_z}_{h}. \quad (16)$$

The initial quantum state for retrieval of information is

$$|\psi_0\rangle_6 = \frac{1}{\sqrt{r}} \sum_{k=1}^{r} |m_1^k m_2^k \ldots m_n^k; 0; h_1 h_2 \ldots h_z\rangle. \quad (17)$$

Also, to further reduce the complexity of the circuit, we removed the x gates from Steps 3 and 5 of Algorithm 5. As in Algorithm 3, this results in measuring $|m\rangle$ when $|c\rangle$ is in state $|1\rangle$. As a result, Algorithm 5 changes to Algorithm 6.

Algorithm 6: EP-PQM retrieval algorithm — NISQ implementation.

input : The initial state $|\psi_0\rangle_6$ given in Eq. (17)
output : 1) The value of c and 2) one of the stored patterns that has minimum D with the target pattern in a given run

1   $|\psi_1\rangle = \mathrm{H}_c |\psi_0\rangle_6$
2   **foreach** $\tau_j \in T$ **do**
3 | **if** $\tau_j == 0$ **then**
4 | | $|\psi_2\rangle = \mathrm{X}_{m_j} |\psi_1\rangle$
5 | **else**
6 | | $|\psi_2\rangle = |\psi_1\rangle$
7   $|\psi_3\rangle = \prod_{j=1}^{z} \mathrm{C^d NOT}_{m_{d(j-1)+1} m_{d(j-1)+2} \ldots m_{dj}, h_j} |\psi_2\rangle$
8   $|\psi_4\rangle = \prod_{e=1}^{z} \left(GW^{-2} \right)_{c, h_e} \prod_{j=1}^{z} W_{h_j} |\psi_3\rangle$
9   $|\psi_5\rangle = \prod_{j=z}^{1} \mathrm{C^d NOT}_{m_{d(j-1)+1} m_{d(j-1)+2} \ldots m_{dj}, h_j} |\psi_4\rangle$
10   **foreach** $\tau_j \in T$ **do**
11 | **if** $\tau_j == 0$ **then**
12 | | $|\psi_6\rangle = \mathrm{X}_{m_j} |\psi_5\rangle$
13 | **else**
14 | | $|\psi_6\rangle = |\psi_5\rangle$
15   $|\psi_7\rangle = \mathrm{H}_c |\psi_6\rangle$
16   Measure qubit $|c\rangle$
17   **if** $c == 1$ **then**
18 | Measure the memory register to obtain the desired state.

C. POSTPROCESSING ON CLASSICAL COMPUTER

The postprocessing for EP-PQM is the same as that for PQM and P-PQM. We refer the reader to Section II-C for details.

D. IMPLEMENTATION

NISQ version of the EP-PQM can be implemented on any modern NISQ architecture. We introduce a reference implementation on QisKit [23], a Python-based open-source software development kit for coding in OpenQASM and leveraging the IBM QCs. The code is given in [24]; it is based on our PQM-based string comparison approach [25].

Similar to the QisKit Aqua [26] library, we wrap Open-QASM invocations into a Python class so that a programmer without any QC coding experience can leverage the algorithm from any Python program. The code can be executed in a simulator on a personal computer or on the actual IBM QC.

E. COMPLEXITY ANALYSIS

1) SPACE COMPLEXITY (QUBITS COUNT)

As discussed above, all the algorithms operate on n-bit patterns, but n depends on the value of a and the type of encoding (one-hot or label). Suppose we want to encode a pattern containing z features with a distinct attributes.

The case when $a = 1$ is trivial because $D = 0$ for any string. For the case when $a > 1$, formalizing Examples I.1 and I.2, we may say that for PQM and P-PQM, which have to use one-hot encoding

$$n = n_o = \begin{cases} z & \text{if } a = 2 \\ za & \text{if } a > 2 \end{cases} \quad (18)$$

and for EP-PQM, which can use label encoding

$$n = n_l = z \lceil \log_2(a) \rceil \quad \text{if } a \geq 2. \quad (19)$$

In other words, when $a = 2$, all algorithms require $n = z$ bits to represent a pattern, and when $a > 2$, EP-PQM requires fewer bits and qubits.

1) a) Fault-tolerant implementation

On an ideal QC, based on (1) and (3), PQM and P-PQM require $2n + 2$ qubits for storage and $2n + 1$ qubits for retrieval, making a total of $4n + 3$ qubits [12]. In practice, one can perform the storage and retrieval algorithms for PQM and P-PQM on a single circuit. This will lead to a reduction in the number of qubits to $2n + 2$ [14].

In EP-PQM, we stick to the conceptually similar approach and, based on (12) and (14), use $2n + z + 1$ qubits. In the fault-tolerant implementation, EP-PQM requires $2n + 2$ qubits for storage and $2n + z + 1$ for retrieval. During the storage procedure, we need register $|u\rangle$ with two qubits. During retrieval, register $|u\rangle$ is not needed, thus its qubits can be reassigned. We can reuse one of these qubits for register $|c\rangle$ and one qubit as part of the register $|h\rangle$. Thus, we need only $2n + z + 1$ qubits in total.

Based on (18) and (19), in the $a = 2$ case, PQM and P-PQM will require $2z + 2$ bits, while EP-PQM will need $3z + 1$ bits. When $a > 2$, the savings start to emerge. EP-PQM needs $z(2\lceil \log_2(a) \rceil + 1) + 1$ qubits, while PQM and P-PQM encoding need $2za + 2$ qubits. Asymptotically, this gives a reduction from $O(za)$, in the PQM and P-PQM cases, to $O(z \log_2(a))$, in the EP-PQM case.

1) b) NISQ implementation

If we use one circuit for both the storage and retrieval algorithms, P-PQM's NISQ implementation,[2] based on (1)

---

[2]As mentioned in Section II-B, PQM does not have an NISQ implementation.

and (10), requires $2n + 2$ qubits. EP-PQM, based on (12) and (16), uses $n + z + 1$ qubits (as in the fault-tolerant implementation above, we reassign qubits).

Based on (18) and (19), if $a = 2$, EP-PQM will require one less qubit than P-PQM: $2z + 1$ instead of $2z + 2$. And when $a > 2$, P-PQM require $2za + 2$ qubits, while EP-PQM needs only $z(\lceil \log_2(a) \rceil + 1) + 1$. Asymptotically, this again gives a reduction from $O(za)$ to $O(z \log_2(a))$.

1) c) Summary

In the fault-tolerant implementation, EP-PQM requires a lesser number of bits and qubits than PQM and P-PQM when $a > 2$. In the NISQ implementation, EP-PQM is more efficient than PQM and P-PQM for all values of a.

2) TIME COMPLEXITY—OVERALL NUMBER OF GATES

Our algorithm's time complexity is proportional to the depth of the circuit for storage and retrieval. For now, we assume that different gates have the same time complexity (we will examine each gate type in Section III-E3).

2) a) Fault-tolerant implementation

In the fault-tolerant implementation, all three algorithms use Algorithm 1 to store the data, which requires $O(rn)$ gates.

PQM and P-PQM retrieval, based on Algorithm 2, needs $O(n)$ gates. EP-PQM retrieval, based on Algorithm 5, needs $O(n + z)$ gates.

Consequently, the combined number of gates for storage and retrieval is $O(rn)$ for all three algorithms. As in the case of space complexity, discussed above, the savings are driven by the encoding schema. Since for one-hot encoding $n = za$ and for label encoding $n = z\lceil \log_2(a) \rceil$, the number of gates for PQM and P-PQM is $O(rza)$, while for EP-PQM it is only $O(rz \log_2(a))$.

2) b) NISQ implementation

PQM does not have an NISQ implementation, thus we focus on P-PQM and EP-PQM. The storage part of P-PQM, as in the fault-tolerant case, is governed by Algorithm 1, which, as discussed above, requires $O(rn)$ gates. EP-PQM storage is given in Algorithm 4; it still needs $O(rn)$ gates. As in the fault-tolerant case, the savings will come from the encoding schema, yielding $O(rza)$ gates for P-PQM and $O(rz \log_2(a))$ gates for EP-PQM.

2) c) Summary.

In both fault-tolerant and NISQ cases, EP-PQM becomes more efficient than PQM and P-PQM with the growth of a.

3) SPECIFIC GATES

Complexitywise, not all gates are created equal. Some gates increase complexity more than others. For example, on an NISQ device, $c^2\text{NOT}$ gate requires more native gates than x gate, which increases complexity.

TABLE 1. Gate Count for Storage Algorithms

Gate	All Fault-tolerant and P-PQM NISQ Algorithm 1	EP-PQM NISQ Algorithm 4
CNOT	$2n_o r$	$2\gamma n_l r$
C^2NOT	$2n_o r$	0
C^nNOT	$2r$	$2r$
X	$2n_o r$	$2(1-\gamma)n_l r$
CS	r	r

3) a) Storage

Algorithm 1 governs the data storage procedures for all three algorithms, except for the EP-PQM NISQ implementation, which is governed by Algorithm 4. Table 1 lists how many gates are required to implement these algorithms. When comparing the number of individual gates, we have to be mindful of the encoding, as n varies as per (18) and (19), where n_o and n_l denote n in one-hot and label encoding cases, respectively.

The parameter $\gamma \in [0, 1]$ denotes a fraction of 1 bits in the database. The largest number of gates will be needed when $\gamma = 1$.

Algorithm 4 requires fewer control gates than Algorithm 1. Table 1 lists that it does not need any C^2NOT gates. The number of CNOT gates will vary with n, r, and γ. Let us analyze this variation by exploring under what conditions the number of CNOT gates of EP-PQM is smaller than the number of P-PQM gates. That is, when does the following inequality holds?

$$2n_o r > 2\gamma n_l r \tag{20}$$

For $a = 2$ and using (18) and (19), the inequality (20) becomes

$$2n_o r > 2\gamma n_l r \implies 2zr > 2\gamma zr \implies 1 > \gamma.$$

This inequality never holds, as $\gamma \leq 1$. However, P-PQM and EP-PQM have the same number of CNOT gates when $\gamma = 1$. In the $a > 2$ case, (20) becomes

$$2n_o r > 2\gamma n_l r \implies za > \gamma z \lceil \log_2(a) \rceil$$

$$\implies a > \gamma \lceil \log_2(a) \rceil \implies \frac{a}{\lceil \log_2(a) \rceil} > \gamma.$$

This inequality holds when $a > 2$ for all γ. Thus, EP-PQM requires a lesser number of CNOT gates when $a > 2$.

To understand x gates count, we need to analyze the following inequality:

$$2n_o r > 2(1-\gamma)n_l r. \tag{21}$$

When $a = 2$, using (18) and (19), (21) simplifies to

$$2n_o r > 2(1-\gamma)n_l r \implies 2zr > 2(1-\gamma)zr \implies 1 > 1-\gamma.$$

This implies that P-PQM and EP-PQM have the same number of x gates when $\gamma = 0$. For other values of γ EP-PQM outperforms P-PQM (when $a = 2$).

TABLE 2. Gate Count for Retrieval Algorithms

	Fault-tolerant		NISQ	
Gate	(P-)PQM Algorithm 2	EP-PQM Algorithm 5	P-PQM Algorithm 3	EP-PQM Algorithm 6
CNOT	$2n_o$	$2n_l$	0	0
C^dNOT	0	$2z$	0	$2z$
X	$2n_o$	$2n_l + 2z$	$2\delta n_o$	$2(1-\delta)n_l$
U or W	n_o	z	n_o	z
GU or GW	n_o	z	n_o	z
H	2	2	2	2

[2](P-)PQM stands for "PQM and P-PQM."

Let us now explore $a > 2$ case, where (21) becomes

$$2n_o r > 2(1-\gamma)n_l r \implies za > (1-\gamma)z \lceil \log_2(a) \rceil$$

$$\implies \frac{a}{\lceil \log_2(a) \rceil} + \gamma > 1. \tag{22}$$

This inequality will hold for all γ. Thus, EP-PQM needs a lesser number of x gates than P-PQM (when $a > 2$).

In summary, we can say that EP-PQM will use the same number of gates as P-PQM when $a = 2$ and $\gamma = 0$ (which is an extreme and rare case). For all other scenarios, EP-PQM will need a lesser number of x gates than P-PQM.

3) b) Retrieval

The comparison of the number of specific gate required to retrieve data based on Algorithms 2, 3, 5, and 6 is given in Table 2. The parameter $\delta \in [0, 1]$ denotes the fraction of the bits in the input pattern that are 1's.

A few observations of Table 2 suggest that the retrieval phase requires a lesser number of gates than the storage phase (as it is independent of r). NISQ versions of the algorithms require the same or lesser number of gates than their fault-tolerant counterparts; this is expected as NISQ approaches are designed with the focus on performance.

For both fault-tolerant and NISQ, EP-PQM requires $2z$ additional C^dNOT gates than PQM and P-PQM. EP-PQM needs a times less W and GW gates than corresponding PQM and P-PQM counterparts (U and GU) for $a > 2$.

For the *fault-tolerant implementations*, when $a = 2$, PQM and EP-PQM will need the same number of CNOT gates [based on (18) and (19)]. When $a > 2$, EP-PQM will need a lesser number of CNOT gates

$$2n_o > 2n_l \implies 2za > 2z \lceil \log_2(a) \rceil \implies a > \lceil \log_2(a) \rceil.$$

For the x gates, when $a = 2$, based on (18) and (19), P-PQM is more efficient than EP-PQM

$$2n_o > 2n_l + 2z \implies 2z > 4z.$$

When $a > 2$, the inequality becomes

$$2n_o > 2n_l + 2z \implies 2za > 2z \lceil \log_2(a) \rceil + 2z$$

$$\implies a - \lceil \log_2(a) \rceil > 1.$$

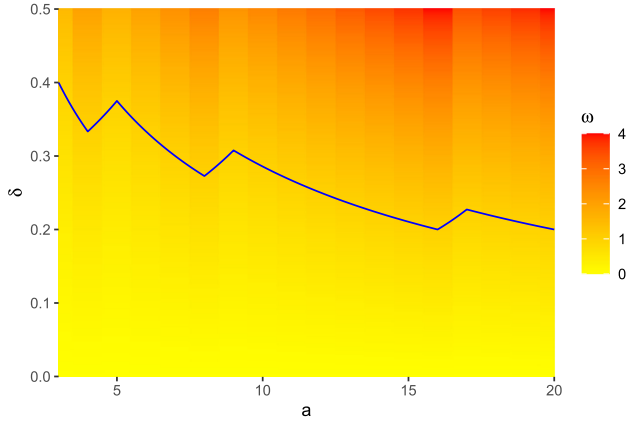This inequality holds for $a < 4$. Thus, EP-PQM needs a lesser number of x gates when $a \geq 4$.

FIGURE 1. z-axis of this contour plot depicts the values of ω, defined in (23). Blue line denotes the boundary where $\omega = 1$. Above the blue line, P-PQM requires more x gates than EP-PQM; below it, P-PQM requires fewer x gates.

For the *NISQ implementations*, the number of x gates will vary with δ. For the $a = 2$ case

$$2\delta n_o > 2(1-\delta)n_l \;\Rightarrow\; 2\delta z > 2(1-\delta)z \;\Rightarrow\; \delta > (1-\delta).$$

Thus, for $a = 2$, if $\delta < 0.5$, then P-PQM needs a lesser number of gates than EP-PQM, and if $\delta > 0.5$, vice versa. If $\delta = 0.5$, the number of gates is identical.

For the $a > 2$ case, EP-PQM will need lesser number of x gates than P-PQM when

$$2\delta n_o > 2(1-\delta)n_l \;\Rightarrow\; \delta a > (1-\delta)\lceil \log_2(a) \rceil$$

$$\Rightarrow\; \underbrace{\frac{\delta}{1-\delta}\frac{a}{\lceil \log_2(a) \rceil}}_{\omega} > 1. \quad (23)$$

The values of the inequality's left-hand side are given in Fig. 1. The inequality holds for different values of δ (in general, δ nonmonotonically decreases with the increase of a). In other words, as a increases, EP-PQM will require fewer x gates for a wider variety of input patterns. For example, when $a = 3$, EP-PQM will require fewer x gates for any input pattern with more than 40% of bits set to 1, and when $a = 16$, for any input pattern with more than 20% of bits set to 1.

F. REMARKS
1) LABEL ENCODING
The principles of label encoding on a classical or a QC are the same. Label encoding can, for instance, be applied directly to nominal and ordinal data. This method can also be applied to numeric data if they are mapped to labeled intervals.

2) ATTRIBUTES
So far, we were operating under the implicit assumption that each feature in an input dataset has the same number of attributes. However, in practice, this is often not the case. Our current (implicit) workaround is to set the number of

TABLE 3. Datasets' Description (Based on 90% of the Observations) of the Class/Label Having the Maximum Number of Observations

Dataset	Class/Label name	r	z	a
Balance Scale [28], [29]	R	262	4	5
Breast Cancer [30], [31]	2	412	9	11
SPECT Heart [32], [33]	1	36	22	2
Tic-Tac-Toe Endgame [34], [35]	positive	558	9	3
Zoo [36], [37]	1	40	16	6

characters in the alphabet to the maximum distinct number of attributes of any feature in the dataset.

For example, one feature may have two distinct attributes and another feature—100 distinct attributes. In this case, we will use seven bits (i.e., $\lceil \log_2(100) \rceil$) to represent each of the features.

To use only the minimal number of bits needed to represent each feature, we need to alter Steps 5 and 7 of Algorithms 4. Specifically, we need to replace[3] the c^dNOT gate for the jth feature with c^{d_j}NOT, where d_j is the number of bits needed to represent an attribute of the jth feature.

3) OTHER USE CASES
In this article, we focus on the quantum ML classification use case. However, the same principles can be applied to any use case requiring the encoding of data with multiple attributes.

For example, by using this approach, we can efficiently codify strings where characters are drawn from an alphabet with a large number of characters. A string may represent anything from mRNA sequences in bioinformatics to software log records in software engineering [25].

IV. EXPERIMENTS
In order to compare P-PQM with EP-PQM, we show the number of qubits and gates necessary to perform classification on five datasets from the UCI Machine Learning Repository [27] (which were also used by the authors of P-PQM [14]). Summary statistics for the datasets are given in Table 3. Data are encoded using one-hot encoding in the P-PQM case, whereas label encoding is used in the EP-PQM case. We do not measure the accuracy of the models in our experiments because PQM and EP-PQM yield the same accuracy (since D is the same in both cases).

Sousa *et al.* [14] tested the performance of ML models using a tenfold cross-validation approach, which assesses how well the training results generalize to previously unexplored data. In this approach, a dataset is divided into ten equal subsets. Then, nine subsets are used to train the model and one to validate the training results. The process is repeated ten times, with each of the subsets being used for validation only once.

In order to mimic this approach, we sample 90% of observations (i.e., patterns) from each dataset, referred to as b. The

---

[3]Gate replacement is all that is needed for the ML classification task. To use this approach for computing the value of D on a classical computer, we also need to alter the trigonometric formulas [25, (2) and (3)].

TABLE 4. Comparison of the Number of Qubits and Quantum Circuit Depth for Class/Label With the Maximum Number of Observations

Dataset	Number of Qubits			Quantum Circuit Depth			Decomposed Quantum Circuit Depth		
	P-PQM	EP-PQM	Savings	P-PQM	EP-PQM	Savings	P-PQM	EP-PQM	Savings
Balance Scale	42	18	57%	12,338	2,899	77%	108,510	4,515	96%
Breast Cancer	200	47	77%	84,563	9,776	88%	821,728	12,407	98%
SPECT Heart	46	46	0%	1,862	747	60%	16,390	982	94%
Tic-Tac-Toe Endgame	56	29	48%	34,069	8,478	75%	309,188	11,847	96%
Zoo	194	66	66%	8,060	334	96%	77,554	629	99%

A comma separates groups of thousands.

observations in the dataset belong to different classes. Let us assume the dataset has l distinct classes/labels. We group b observations by l labels, creating l databases of observations. We then focus on the l databases with the most observations (as it represents the hardest task) since it will require the deepest quantum circuit for a given dataset. Let us look at a toy example.

Example IV.1: Consider a dataset with three classes/labels and 1000 observations. We sample 90% of observations, i.e., $b = 900$. Suppose 500 observations belong to label 1, 300 observations to label 2, and 100 observations to label 3. We will focus only on the database of observations for label 1, as it is the largest. ∎

We implement the quantum circuit needed to store and retrieve data using P-PQM (Algorithms 1 and 3) and EP-PQM (Algorithms 4 and 6). All quantum algorithms are implemented using Qiskit v0.33.1 [23]. The results for the quantum-circuit-based implementation are as follows; the results for the quantum simulator backend can be found in Appendix C.

For each dataset's database with the most observations, we compute the quantum circuit depth and the number of gates needed to execute NISQ storage and retrieval algorithms. Finally, we perform a shallow decomposition of this circuit (by calling "decompose()" methods of the QisKit quantum circuit) and obtain the set of gates (see Table 4).

Table 3 contains high-level statistics for the datasets under study. The datasets have a diverse structure. For instance, SPECT Heart dataset has only two features and will be stored using binary variables for both P-PQM and EP-PQM. The rest of the algorithms have a between 3 and 11. Thus, for P-PQM, we will encode the attributes using one-hot encoding and for EP-PQM using label encoding.

Table 4 lists the number of qubits and quantum circuit depths. In addition, we compute relative resource savings while treating the resources needed for P-PQM as a baseline. The number of qubits used by P-PQM and EP-PQM in SPECT Heart is the same,[4] which is expected as $a = 2$. Label encoding reduces the number of qubits required by 48% to 77% in the rest of the datasets.

Peculiarly, quantum circuit depth is reduced for all datasets by 60% to 96%, showing the effectiveness of EP-PQM even for datasets with $a = 2$. The savings are even more evident after decomposition: the decomposed quantum circuit depth is reduced by 94% to 99%. These additional savings can be explained by the fact that many gate types (especially multicontrolled ones) require a lot of native gates to be implemented, thus increasing the quantum circuit depth.

As discussed in Section III-E3, to understand the efficiency of the algorithms, we need to assess the count of specific gates. This information is given in Table 5.

We will use `typewriter font` to reference the gates used in the QisKit's implementation of the circuit. A number of gates are readily available. The CNOT gate is implemented using `cx`. The $c^{/u\,palpha}$NOT gates, where α is the number of control qubits, will be implemented using `cx` when there is one control qubit, `ccx` when there are two control qubits, `mcx` when there are three or four control qubits, and `mcx_gray` when there are more than or equal to five control qubits. The H gate is implemented using `h`, and x gate is implemented using `x`.

According to the table, EP-PQM significantly reduces the number of control gates `cx`, `ccx`, and `mcx_gray` by 58.7% to 100.0%. For three datasets in EP-PQM with $a \geq 4$, a small number of `mcx` control gates ($2z$, from 8 to 32) were added to implement c^dNOT for three datasets with $a \geq 4$; for P-PQM, no `mcx` gates were required.

The EP-PQM approach results in significant reductions in other gates (namely, `unitary` and `cunitary`[5] QisKit simulator gates which cover algorithmic U, GU, W, CW, and CS gates) from 3.1% to 90.9% for all datasets except for SPECT Heart dataset where the number of gates was identical to the P-PQM algorithm.

For all datasets, EP-PQM reduces the number of x gates by 53.2% to 76.2%.

The count of gates obtained using shallow decomposition of the quantum circuit is given in Table 6. Decomposition of the circuit of the Breast Cancer dataset yielded gates `c3sx`, `cu1`, `rcccx`, and `rcccx_dg` not found in circuits of other datasets. To facilitate comparison with other datasets, we further decomposed these gates using "decompose(['c3sx,'

---

[4]For this implementation of P-PQM and EP-PQM, to simplify the code, one qubit from $|u\rangle$ was reassigned to $|c\rangle$, but the second qubit of $|u\rangle$ was not reassigned to $|h\rangle$. EP-PQM, therefore, requires one extra qubit in comparison with the theoretical requirement: $2n + 2$ instead of $2n + 1$.

[5]`unitary` denotes a custom unitary gate and `cunitary` is a controlled version of a custom unitary gate.

TABLE 5. Gate Count in the Quantum Circuit

	ccx			cunitary			cx			h			mcx		
Dataset	P	EP	S	P	EP	S	P	EP	S	P	EP	S	P	EP	S
Balance Scale	10,480	0	100.0%	20	4	80.0%	10,480	2104	79.9%	2	2	0.0%	0	8	–
Breast Cancer	81,576	0	100.0%	99	9	90.9%	81,576	8526	89.5%	2	2	0.0%	0	18	–
SPECT Heart	1,584	0	100.0%	22	22	0.0%	1,584	654	58.7%	2	2	0.0%	0	0	–
Tic-Tac-Toe Endgame	30,132	18	99.9%	27	9	66.7%	30,132	6790	77.5%	2	2	0.0%	0	0	–
Zoo	7,680	0	100.0%	96	16	83.3%	7,680	188	97.6%	2	2	0.0%	0	32	–

	mcx_gray			measure			unitary			x		
Dataset	P	EP	S	P	EP	S	P	EP	S	P	EP	S
Balance Scale	524	524	0.0%	21	13	38.1%	282	266	5.7%	12,585	4,207	66.6%
Breast Cancer	824	824	0.0%	100	37	63.0%	511	421	17.6%	89,011	21,200	76.2%
SPECT Heart	72	72	0.0%	23	23	0.0%	58	58	0.0%	2,205	1,031	53.2%
Tic-Tac-Toe Endgame	1,116	1,116	0.0%	28	19	32.1%	585	567	3.1%	40,195	13,328	66.8%
Zoo	80	80	0.0%	97	49	49.5%	136	56	58.8%	8,993	3,757	58.2%

P stands for P-PQM, EP for EP-PQM, and S for relative savings computed as (P-EP)/P. The gates used in QisKit's implementation of the quantum circuit are as follows: ccx is a Toffoli gate c^2NOT; cunitary and unitary are custom unitary gates (which our U, GU, W, CW, and cs gates translated to); cx is a CNOT gate; h is an H gate; mcx is a c^αNOT gate where α is 3 or 4; mcx_gray is a c^αNOT gate where $\alpha \geq 5$; measure is the measurement of the qubit into classical bit; x is an x gate; see QisKit documentation for details [23]. We deliberately leave QisKit gate names as-is to differentiate "algorithmic gates" from "implementation gates."

TABLE 6. Gate Count in the Decomposed Quantum Circuit

	cu			cx			h			mcu1		
Dataset	P	EP	S	P	EP	S	P	EP	S	P	EP	S
Balance Scale	20	4	80.0%	73,883	2,739	96.3%	22,008	1,064	95.2%	524	524	0.0%
Breast Cancer	99	9	90.9%	571,855	9,997	98.3%	164,800	1,972	98.8%	824	824	0.0%
SPECT Heart	22	22	0.0%	11,159	725	93.5%	3,312	144	95.7%	72	72	0.0%
Tic-Tac-Toe Endgame	27	9	66.7%	21,2039	8,013	96.2%	62,496	2,268	96.4%	1,116	1,116	0.0%
Zoo	96	16	83.3%	53,839	715	98.7%	15,520	224	98.6%	80	80	0.0%

	measure			p			t			tdg		
Dataset	P	EP	S	P	EP	S	P	EP	S	P	EP	S
Balance Scale	21	13	38.1%	20	124	-520.0%	41,920		100.0%	31,440		100.0%
Breast Cancer	100	37	63.0%	99	9	90.9%	326,304		100.0%	244,728		100.0%
SPECT Heart	23	23	0.0%	22	22	0.0%	6,336		100.0%	4,752		100.0%
Tic-Tac-Toe Endgame	28	19	32.1%	27	9	66.7%	120,528	72	99.9%	90,396	54	99.9%
Zoo	97	49	49.5%	96	496	-416.7%	30,720		100.0%	23,040		100.0%

	u1			u2			u3		
Dataset	P	EP	S	P	EP	S	P	EP	S
Balance Scale			–	2	2	0.0%	14,175	5,781	59.2%
Breast Cancer		774	–	2	146	-7200.0%	91,580	23,679	74.1%
SPECT Heart			–	2	2	0.0%	2,441	1,267	48.1%
Tic-Tac-Toe Endgame			–	2	2	0.0%	43,568	16,683	61.7%
Zoo			–	2	2	0.0%	9,327	4,011	57.0%

P stands for P-PQM, EP for EP-PQM, and S for relative savings.

'rcccx,' 'rcccx_dg']).decompose('cu1')." This results in all datasets containing the same gates.

The savings are even more pronounced. For cu, cx, h, measure, t, tdg, and u3, the number of gates reduced by 32.1% to 100.0%. Also, the number of phase gates, p, reduced by 66.7% and 90.9% for Tic-Tac-Toe Endgame dataset and Breast Cancer dataset, respectively.

The number of gates for p in the Balance Scale dataset and Zoo dataset increased by 520.0% and 416.7%, respectively, and the number of u2 gates (where u2 is a single qubit gate) increased by 7200.0% for the Breast Cancer dataset. While the relative increase is significant, the absolute increase is relatively small—there are only a few hundred gates added. The number of single qubit gates u1 increased from 0 to 774 (as above, this number is small).

Overall, our experiments confirm our theoretical analysis and show that EP-PQM leads to a significant reduction in gate count compared to P-PQM. Consequently, EP-PQM is more applicable for NISQ devices due to a reduction in quantum circuit depth.

V. SUMMARY
In this article, we extend the PQM-based ML classification algorithm designed for use in NISQ devices. The original approach provided correct classification using one-hot encoding. We extend this approach to enable label encoding, which reduces space complexity (i.e., qubit count) from $O(za)$ to $O(z \log_2(a))$ and decreases the number of gates in the quantum circuit from $O(rza)$ to $O(rz \log_2(a))$.

By simulating ML classification on five datasets (using QisKit QC Simulator) and analyzing the resulting circuit, we verified our theoretical analysis. Depending on the dataset, EP-PQM quantum circuit depth saving ranges between 60% and 96%. Similarly, EP-PQM reduces the corresponding decomposed quantum circuit depth between 94% and 99%. Qubit count was reduced by 48% to 77% for datasets with $a > 2$.

A reduction in space requirements makes it possible to load larger datasets into a QC. Furthermore, reducing the number of gates helps speed up classification and decrease noise associated with deep quantum circuits.

APPENDIX A ONE HOT ENCODING AND D: RELATIONSHIP
Suppose there are two patterns $X = x_1, x_2, \ldots, x_z$ and $Y = y_1, y_2, \ldots, y_z$, containing z features with a attributes. Further, suppose that X and Y represented using one-hot encoding. Thus, the ith feature will be represented by a-bit string.

Let the Hamming distance be denoted by D and the Hamming distance computed when comparing patterns encoded using one-hot encoding be denoted by $D_{\text{one-hot}}$. When performing the pairwise comparison, $D(x_i, y_i) = D_{\text{one-hot}}(x_i, y_i) = 0$, when x_i and y_i are identical. The similarity ends when $x_i \neq y_i$. By definition

$$D(x_i, y_i) = 1, \text{ when } x_i \neq y_i. \tag{24}$$

However, by construction of the one-hot encoding

$$D_{\text{one-hot}}(x_i, y_i) = 2, \text{ when } x_i \neq y_i. \tag{25}$$

That is, x_i and y_i will have a single bit set to 1 at different positions, which requires two operations to convert x_i into y_i.

Extrapolating (24) and (25) to X and Y, given that k features are different, $D(X, Y) = \sum_{i=1}^{k} D(x_i, y_i) = k$, while $D_{\text{one-hot}}(X, Y) = \sum_{i=1}^{k} D_{\text{one-hot}}(x_i, y_i) = 2k$. This implies that

$$D(X, Y) = \frac{1}{2} D_{\text{one-hot}}(X, Y).$$

APPENDIX B QUANTUM COMPUTING: FUNDAMENTALS
Here, we review the fundamentals of quantum computing needed to implement the algorithms described in this article. See [38] for more information about quantum computing.

In quantum computing, a qubit represents the basic unit of information. It is formed by two-quantum states: one constitute the state $|0\rangle = \begin{bmatrix} 1 & 0 \end{bmatrix}^\mathsf{T}$ and the other the state $|1\rangle = \begin{bmatrix} 0 & 1 \end{bmatrix}^\mathsf{T}$. The vectors $|0\rangle$ and $|1\rangle$ that are the eigenvectors of the Pauli-Z matrix are known as the computational basis. The other well-known bases are the Pauli-X and Pauli-Y eigenvectors. Differently from a classical bit, a qubit can exist in a superposition state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha & \beta \end{bmatrix}^\mathsf{T}$, where α and β are, respectively, the amplitude of the qubit in state $|0\rangle$ and $|1\rangle$. The amplitudes are complex numbers that satisfy the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. The state $|\psi\rangle$ is said to be in a superposition of states $|0\rangle$ and $|1\rangle$, which reflects the fact that the qubit can be in more than one basis state at a particular time with a given probability.

To process quantum information, a sequence of quantum gates and measurements must be performed on the qubits. The quantum circuit is formed by these operations. Quantum gates are unitary operations acting on single or multiple qubits. Quantum gates are responsible for evolving the state of the QC and forming the quantum algorithm solution. To learn the QC state, qubits are measured. The measurement operation transforms the qubit state $|\psi\rangle$ in a classical bit 0 or 1, which are measured with probability $|\alpha|^2$ or $|\beta|^2$, respectively. The quantum gates used in this article are defined as follows.

The Hadamard gate, given by

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

transforms the state $|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, which are both superpositions. If the state is already in a superposition, applying the Hadamard gate on it will revert it to $|0\rangle$ or $|1\rangle$, respectively.

Pauli-X gate is defined as

$$\mathsf{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

TABLE 7 Gate Count in the Quantum Circuit for QasmSimulator When $r = 2$

	Quantum Circuit Depth			cx			measure		
	P	EP	S	P	EP	S	P	EP	S
Balance Scale	20,972,436	81,981	99.6%	12,583,496	49,274	99.6%	21	13	38.1%
SPECT Heart	83,887,088	83,886,215	0.0%	50,332,292	50,331,748	0.0%	23	23	0.0%

	u1			u2			u3			unitary		
	P	EP	S	P	EP	S	P	EP	S	P	EP	S
Balance Scale	12,583,540	49,276	99.6%	170	26	84.7%	145	61	57.9%	22	6	72.7%
SPECT Heart	50,332,340	50,331,724	0.0%	186	10	94.6%	171	161	5.8%	24	24	0.0%

P stands for P-PQM, EP for EP-PQM, and S for relative savings.

TABLE 8 Gate Count in the Decomposed Quantum Circuit for QasmSimulator When $r = 2$

	Quantum Circuit Depth			cx			measure		
	P	EP	S	P	EP	S	P	EP	S
Balance Scale	20,972,442	81,987	99.6%	12,583,499	49,277	99.6%	21	13	38.1%
SPECT Heart	83,887,094	83,886,221	0.0%	50,332,295	50,331,751	0.0%	23	23	0.0%

	u			u3		
	P	EP	S	P	EP	S
Balance Scale	145	61	57.9%	12,583,740	49,316	99.6%
SPECT Heart	171	161	5.8%	50,332,558	50,331,766	0.0%

P stands for P-PQM, EP for EP-PQM, and S for relative savings.

It is the x gate that flips the qubit state, transforming the state $|0\rangle$ into $|1\rangle$ and the state $|1\rangle$ into $|0\rangle$.

Controlled-NOT (CNOT) gate, given by

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

is a two-qubit gate, operating on control and target qubits. It applies the x gate to a target qubit whenever its control qubit is in state $|1\rangle$. In this context, we can interpret the CNOT as the classic XOR gate. CNOT gate can be extended to having α control qubits ($C^{\alpha}NOT$). In this case, the x gate is applied to the target qubit whenever each of the α control qubits is in state $|1\rangle$. The $C^{\alpha}NOT$ is $2^{\alpha+1} \times 2^{\alpha+1}$ unitary operator. All control gates have subscripts indicating the qubits on which they are applied, with the control qubits listed first. For example, in control gate $C^{\alpha}NOT_{m_1 \cdots m_n, u_1}$ qubits $m_1 \cdots m_n$ act as controls and qubit u_1 is a target.

Finally, the last operation in a quantum circuit is the qubit-state measurement. Indeed, when a quantum system is not measured, a qubit can be in a superposition of states of $|0\rangle$ and $|1\rangle$. However, after measurement, the qubit state collapses into either the state $|0\rangle$ or $|1\rangle$ with a probability of the absolute value of the amplitude squared.

APPENDIX C CIRCUIT FOR QUANTUM SIMULATOR
We have also tried to transpile the circuit for the QisKit Simulator backend (namely, "QasmSimulator"). To minimize computation efforts, no optimization of the circuit is performed ("optimization_level = 0"). The transpilation, however, requires a substantial amount of memory: empirically, we found that 1 TB of memory was insufficient for transpiling the circuits for our datasets. This was not expected, as the generation of data in Tables 4–6 required $\approx$ 3 GB of memory.

Therefore, to get a feel for the QasmSimulator-based circuits, we reduced the number of observations to two in each dataset, i.e., $r = 2$. In spite of this simplification, we still ran out of memory for the Breast-Cancer-, Tic-Tac-Toe-, and Zoo-based circuits. However, we could produce the circuits for the Balance Scale and SPECT Heart datasets.

Table 7 lists the results of transpilation without optimization, and Table 8 lists stats for decomposed versions of these circuits. According to the tables, the results are mixed. For Balance Scale dataset, the quantum circuit depth is three orders of magnitude smaller, resulting in overall savings of 99.6%. In the original circuit, the biggest savings are associated with gates cx and u1, while in the decomposed case, the biggest savings are associated with cx and u3. Savings would be even greater with the increase in r.

For SPECT Heart dataset, the quantum circuit depth is reduced, but the reduction is marginal: 0.001%, because the

number of `cx` and `u1` gates are almost identical for both approaches. This differs from the results that we saw in Tables 4–6: there the quantum circuit depth was reduced by 60% and 94% for quantum circuit and decomposed quantum circuit, respectively.

We conjecture that this stark difference may be due to inefficiencies in the Quantum Simulator's optimizer. Significant memory requirements indirectly support this conjecture.

ACKNOWLEDGMENT

REFERENCES

[1] L. Zhou, S. Pan, J. Wang, and A. V. Vasilakos, "Machine learning on Big Data: Opportunities and challenges," *Neurocomputing*, vol. 237, pp. 350–361, 2017, doi: 10.1016/j.neucom.2017.01.026.

[2] M. Schuld, I. Sinayskiy, and F. Petruccione, "An introduction to quantum machine learning," *Contemporary Phys.*, vol. 56, no. 2, pp. 172–185, 2015, doi: 10.1080/00107514.2014.964942.

[3] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, "Quantum machine learning," *Nature*, vol. 549, no. 7671, pp. 195–202, 2017, doi: 10.1038/nature23474.

[4] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Kuala Lumpur, Malaysia: Pearson, 2016.

[5] E. Yom-Tov, *Advanced Lectures on Machine Learning: ML Summer Schools.* Berlin, Germany: Springer-Verlag, 2004, doi: 10.1007/b100712.

[6] Y. Liu, S. Arunachalam, and K. Temme, "A rigorous and robust quantum speed-up in supervised machine learning," *Nature Phys.*, vol. 17, no. 9, pp. 1013–1017, 2021, doi: 10.1038/s41567-021-01287-z.

[7] D. Dong, C. Chen, H. Li, and T.-J. Tarn, "Quantum reinforcement learning," *IEEE Trans. Syst., Man, Cybern.*, vol. 38, no. 5, pp. 1207–1220, Oct. 2008, doi: 10.1109/TSMCB.2008.925743.

[8] P. Rebentrost, M. Mohseni, and S. Lloyd, "Quantum support vector machine for Big Data classification," *Phys. Rev. Lett.*, vol. 113, no. 13, 2014, Art. no. 130503, doi: 10.1103/PhysRevLett.113.130503.

[9] N. Mishra *et al.*, "Quantum machine learning: A review and current status," in *Proc. Data Manage., Analytics Innovation*, 2021, pp. 101–145, doi: 10.1007/978-981-15-5619-7_8.

[10] R. Li, J. Xu, J. Yuan, and D. Li, "An introduction to quantum machine learning algorithms," in *Proc. 9th Int. Conf. Comput. Eng. Netw.*, 2021, pp. 519–532, doi: 10.1007/978-981-15-3753-0_51.

[11] D. Ventura and T. Martinez, "Quantum associative memory," *Inf. Sci.*, vol. 124, no. 1/4, pp. 273–296, 2000, doi: 10.1016/S0020-0255(99)00101-2.

[12] C. Trugenberger, "Probabilistic quantum memories," *Phys. Rev. Lett.*, vol. 87, 2001, Art. no. 067901, doi: 10.1023/A:1024022632303.

[13] C. Trugenberger, "Quantum pattern recognition," *Quantum Inf. Process.*, vol. 1, pp. 471–493, 2002, doi: 10.1023/A:1024022632303.

[14] R. S. Sousa, P. G. M. dos Santos, T. M. L. de Veras, W. R. de Oliveira, and A. J. da Silva, "Parametric probabilistic quantum memory," *Neurocomputing*, vol. 416, pp. 360–369, 2020, doi: 10.1016/j.neucom.2020.01.116.

[15] M.-H. Wang and Q.-Y. Cai, "High-fidelity quantum cloning of two nonorthogonal quantum states via weak measurements," *Phys. Rev. A*, vol. 99, Jan. 2019, Art. no. 012324, doi: 10.1103/PhysRevA.99.012324.

[16] L.-C. Peng *et al.*, "Cloning of quantum entanglement," *Phys. Rev. Lett.*, vol. 125, Nov. 2020, Art. no. 210502, doi: 10.1103/PhysRevLett.125.210502.

[17] Z.-B. Yang *et al.*, "Experimental demonstration of entanglement-enabled universal quantum cloning in a circuit," *Neonatal Pathological Jaundice Quantum Inf.*, vol. 7, no. 1, pp. 1–5, 2021, doi: 10.1038/s41534-021-00375-5.

[18] T. Brun, H. Klauck, A. Nayak, M. Rötteler, and C. Zalka, "Comment on 'probabilistic quantum memories'," *Phys. Rev. Lett.*, vol. 91, Nov. 2003, Art. no. 209801, doi: 10.1103/PhysRevLett.91.209801.

[19] M. Schuld, I. Sinayskiy, and F. Petruccione, "Quantum computing for pattern classification," in *Proc. Pac. Rim Int. Conf. Artif. Intell., Trends Artif. Intell.*, D.-N. Pham and S.-B. Park, Eds. 2014, pp. 208–220, doi: 10.1007/978-3-319-13560-1_17.

[20] P. G. M. dos Santos, R. S. Sousa, I. C. Araujo, and A. J. da Silva, "Quantum enhanced cross-validation for near-optimal neural networks architecture selection," *Int. J. Quantum Inf.*, vol. 16, no. 8, 2018, Art. no. 1840005, doi: 10.1142/S0219749918400051.

[21] P. G. M. dos Santos, R. S. Sousa, and A. J. da Silva, "A WNN model based on probabilistic quantum memories," in *Proc. 27th Eur. Symp. Artif. Neural Netw.*, 2019, pp. 313–318. [Online]. Available: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2019-56.pdf

[22] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," in *Proc. IEEE Symp. Comput. Intell. Secur. Defense Appl.*, 2009, pp. 1–6, doi: 10.1109/CISDA.2009.5356528.

[23] H. Abraham *et al.*, "Qiskit: An open-source framework for quantum computing," 2019, doi: 10.5281/zenodo.2562111.

[24] "Qiskit implementation of the algorithm," 2021. [Online]. Available: https://github.com/miranska/qc-str

[25] M. Khan and A. Miranskyy, "String comparison on a quantum computer using hamming distance," 2021, *arXiv:2106.16173*. 2021, doi: 10.48550/arXiv.2106.16173.

[26] "Aqua (algorithms for quantum applications) (qiskit.aqua)," qiskit 0.25.3 documentation, 2021. [Online]. Available: https://qiskit.org/documentation/apidoc/qiskit_aqua.html

[27] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[28] T. Hume, "Balance scale data set," UCI Mach. Learn. Repository, 1976. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/balance scale

[29] D. Klahr and R. Siegler, "The representation of children's knowledge," *Adv. Child Develop. Behav.*, vol. 12, pp. 61–116, 1978, doi: 10.1016/S0065-2407(08)60036-1.

[30] M. Tan and J. Schlimmer, "Breast cancer data set," UCI Mach. Learn., Repository, 1986. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/breastcancer

[31] R. Michalski, I. Mozetic, J. Hong, and N. Lavrac, "The multi-purpose incremental learning system AQ15 and its testing application to three medical domains," in *Proc. 5th AAAI Nat. Conf. Artif. Intell.*, 1986, pp. 1041–1045, doi: 10.5555/2887770.2887940.

[32] K. J. C. Lukasz and A. Kurgan, "SPECT heart data set," UCI Mach. Learn. Repository, 2001. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/SPECT Heart

[33] L. Kurgan, K. Cios, R. Tadeusiewicz, M. Ogiela, and L. S. Goodenday, "Knowledge discovery approach to automated cardiac SPECT diagnosis," *Artif. Intell. Med.*, vol. 23, pp. 149–169, 2001, doi: 10.1016/s0933-3657(01)00082-3.

[34] D. W. Aha, "Tic-tac-toe endgame data set," UCI Mach. Learn. Repository, 1989. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Tic-Tac-Toe Endgame

[35] C. J. Matheus and L. A. Rendell, "Constructive induction on decision trees," in *Proc. 11th Int. Joint Conf. Artificial Intelligence (IJCAI'89)*, vol. 1, Aug. 1989, pp. 645–650. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/1623755.1623857

[36] R. S. Forsyth, "UCI machine learning repository: Zoo data set," 1990. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/zoo

[37] R. Forsyth, *PC/BEAGLE User's Guide*, 1990.

[38] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition.* New York, NY, USA: Cambridge Univ. Press, 2010, doi: 10.1017/CBO9780511976667.