POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Evaluation of quantum machine learning algorithms for cybersecurity

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Tommaso Fioravanti**

Student ID: 925858
Advisor: Prof. Stefano Zanero
Co-advisors: Armando Bellante, Michele Carminati, Alessandro Luongo
Academic Year: 2021-2022

Acknowledgements

Per la realizzazione di questo lavoro:

Desidero ringraziare il mio relatore, Prof. Stefano Zanero, per avermi dato la possibilità di affrontare questo lavoro di tesi molto impegnativo e allo stesso tempo stimolante.
Ringrazio con grande affetto Armando Bellante e Alessandro Luongo, per il costante aiuto, per la grandissima disponibilità e per l'infinita pazienza.
Ringrazio inoltre Michele Carminati per la sua grande disponibilità e per i suoi utli consigli per la stesura di questa tesi.

Un ringraziamento speciale va a tutta la mia famiglia che mi ha supportato e mi supporta sempre in tutti i momenti della mia vita.
In particolare, ringrazio mia mamma per l'amore e la forza che mi trasmette ogni giorno, per avermi aiutato nelle difficoltà ed per avermi insegnato a superarle.
Un ringraziamento particolare anche a mio zio Alberto, che mi ha permesso di intraprendere questo bellissimo percorso, credendo sempre in me e spingendomi continuamente a dare il massimo.
Questo traguardo è anche vostro.

Infine, ringrazio tutti i miei amici, da quelli sempre presenti a quelli conosciuti durante questi anni, che sono riusciti a rendere meno faticoso e più piacevole questo percorso.

Dedico questo grande traguardo a mio padre e a mio nonno. Ho cercato di affrontare questo lungo percorso nel migliore dei modi mettendoci tutto l'impegno del mondo, sia per me stesso ma anche e soprattutto per voi perchè so quanto ci avreste tenuto.
Sì, per voi. Vi voglio bene.

Abstract

Quantum machine learning is a recent discipline that brings together notions from quantum computing and machine learning, providing speedups and improved performances over classical solutions. In this work, we study possible applications of quantum machine learning in cybersecurity by analyzing its possible advantages and disadvantages. We simulate quantum machine learning algorithms to solve intrusion detection problems as we were using a fault-tolerant quantum computer. The quantum machine learning algorithms we consider are formalized by the researchers in theorems that describe their running time, the probability of failure, and the approximation error that we have to tolerate. The approximation error and the probability of failure appear as running time parameters. We study the trade-off between the approximation error and the running time. We find cases of errors to tolerate to best match classical performances and, at the same time, observe a reasonable number of samples and features to get a quantum speed-up over classical machine learning. With this study, we derive possible conditions for which quantum machine learning can play a role in cybersecurity in the next years. To simplify the process of simulation of quantum machine learning algorithms, we develop an open-source framework.

Keywords: quantum computing; unsupervised machine learning; PCA; K-Means; pure state tomography; phase estimation; consistent phase estimation; amplitude estimation; quantum machine learning; cybersecurity; anomaly detection.

Abstract in lingua italiana

Il machine learning quantistico è una disciplina recente che unisce concetti del calcolo quantistico e del machine learning classico, in modo da garantire vantaggiosi speed-ups rispetto al caso classico per risolvere vari tipi di problemi. In questo lavoro proponiamo uno studio su possibili applicazioni del machine learning quantistico per risolvere problemi di sicurezza informatica, analizzandone i relativi vantaggi e svantaggi. Per fare questo simuliamo algoritmi di machine learning quantistico per rilevare intrusioni di rete. In questa tesi, assumiamo di simulare gli algoritmi quantistici come se avessimo a disposizione un computer quantistico fault-tolerant. Gli algoritmi di machine learning quantistico che utilizziamo possono essere considerati come algoritmi di approssimazione randomizzati. Infatti essi sono formalizzati in teoremi che ne descrivono la probabilità di fallimento, l'errore di approssimazione che dobbiamo tollerare e il tempo di esecuzione. L'errore di approssimazione e la probabilità di fallimento sono parametri del tempo di esecuzione. Per questo, l'obiettivo principale è quello di studiare il trade-off tra l'errore di approssimazione e il tempo di esecuzione degli algoritmi quantistici considerati. In particolare, troviamo casi di errore che ci permettano di ottenere le stesse performance dei rispettivi algoritmi classici e, allo stesso tempo, di osservare un numero ragionevole di dati e features tali da garantire uno speed-up quantistico rispetto agli algoritmi classici. In questa tesi quindi vogliamo studiare se e come il machine learning quantistico potrà impiegare un ruolo importante nella sicurezza informatica del futuro. Per semplificare il processo di simulazione degli algoritmi quantistici, abbiamo sviluppato un framework open-source.

Parole chiave: calcolo quantistico; machine learning non supervisionato; analisi delle componenti principali; K-Means; tomografia di stati puri; stima di fase; stima di fase consistente; stima delle ampiezze; apprendimento automatico quantistico; sicurezza informatica; rilevamento di anomalie.

Contents

Introduction

Quantum machine learning is a novel discipline that combines concepts from quantum computing and machine learning, providing speed-ups and improved performances over classical solutions. The link between the fields of quantum computing and machine learning begins to become concrete in 2009, when Harrow, Hassidim, and Lloyd [20] proposed a quantum algorithm that prepares a quantum representation $|\boldsymbol{x}\rangle$ of the desired vector $\vec{\boldsymbol{x}}$ such that $\boldsymbol{A}\,|\boldsymbol{x}\rangle = |\boldsymbol{b}\rangle$, with $\boldsymbol{A} \in \mathbb{C}^{n\times d}$ and $\boldsymbol{b} \in \mathbb{C}^n$, in time $\sim O(\log(n))$. This was the major breakthrough towards the discovery of new quantum linear algebra routines that led to the advent of quantum machine learning. The first works in this direction concerned many algorithms to perform linear regression [10], support vector machine [37], K-nearest neighbors [50], or execute K-Means [26] providing exponential speed-up over its classical implementation. To date, quantum machine learning is still currently studied on a theoretical level but has not yet found practical application to solve real problems. This is mainly because, nowadays, we are in the *noisy intermediate-scale quantum* (NISQ) era [36]. *Intermediate-scale* is related to the size of quantum computers that will be available in the next years. These will be quantum computers with a maximum of 50-100 physical qubits which is enough to code interesting problems such as testing quantum error correction. However, these qubits are *noisy*, meaning that we do not have perfect control over them. The noise in the qubits affects the accuracy with which we can execute quantum gates. Moreover, since noise tends to scale with the circuit depth, it limits the maximum depth of the quantum logic circuits we can execute and, therefore, also the computational power of quantum computers. This is the main problem of modern quantum computers. Indeed, to solve complex problems and applications, we need large-scale quantum computers with more than 100 physical qubits and large depth circuits. For example, breaking 1024-bit RSA encryption would require a 2000-qubit quantum computer [31]. However, due to the noise in quantum gates, we are not yet able to perform useful circuits for complex applications. For this reason, the application of quantum machine learning is also difficult in practice and this pushes us towards a theoretical approach in this field by assuming a fault-tolerant quantum computer. Despite their still purely theoretical study, quantum machine learning and, more generally, quantum computing are very promising

in many application fields. One of these is certainly cybersecurity given that quantum computing has strongly influenced cryptography. Indeed, a quantum-enabled attacker can make classical cryptosystems, whose security relies on the difficulty of solving hard mathematical problems with a classical computer, vulnerable. For instance, using the Shor algorithm, which enables factoring integers and finding discrete logarithms in polynomial time [44], one could break the RSA encryption since it exploits the difficulty of factorizing prime numbers with classical computations. Also cryptographic protocols such as Elliptic Curve Cryptography (ECC) and Diffie-Hellman (DH) are no longer safe being based on the difficulty of finding discrete logarithms.
Another quantum algorithm that threatens symmetric-key cryptography is the Grover one, which enables searching an element with a known property in a search space of length N in time $O(\sqrt{N})$ [16]. By exploiting Grover's algorithm, an attacker can reduce the time of a bruteforce attack to its square root, therefore, for a classical n-bit block cipher, it reduces the time from 2^n to $\sqrt{2^n} = 2^{\frac{n}{2}}$. For example, it can reduce the security level of a 128-bit block cipher, such as AES-128, to a 64-bit one. Grover's algorithm can also be used to find a collision in a hash function. Indeed, given an n-hash function, a preimage attack can be performed in $2^{\frac{n}{2}}$ steps, making hash algorithms vulnerable [31].
While it is true that a quantum-enabled attacker can jeopardize the security of cryptographic systems, it is equally true that a quantum-enabled defender can build secure cryptography systems against quantum and classical computers. One of the main examples in this direction is the Quantum Key Distribution (QKD) which, based on quantum communication, provides a secure method for key exchange over an insecure channel [39]. It exploits a fundamental concept in quantum mechanics: quantum communication between two parties cannot be intercepted without producing a detectable interference. Therefore, a third party that tries to eavesdrop information about the key exchange introduces a disturbance, leaving a trace of intrusion and alerting the two parties that were exchanging keys.
Moreover nowadays, to better prepare for the quantum computers era, cybersecurity experts search for *post-quantum cryptography* algorithms that are used to build quantum-resistant cryptographic systems. An example of a quantum-resistant cryptographic algorithm is the lattice-based one which avoids the weakness of RSA by exploiting the shortest vector problem which involves finding the non-zero shortest vector in a lattice [32]. Also multivariate-based algorithms are quantum-resistant since they rely on the difficulty of solving systems of multivariate polynomials over finite fields [14].
With the previous examples, it is even more clear that quantum computing in cryptography is very promising and finds a lot of applications. However, cryptography is not the only field in cybersecurity. Indeed, there are different cybersecurity tasks such as intrusion

detection, malware detection, and many others. In these tasks, machine learning is used a lot both to perform attacks and build automated defense systems. That said, and given that quantum machine learning is a field that is becoming more and more popular, we ask ourselves: can a quantum machine learning-enabled attacker or defender change the dynamics of future cybersecurity?

In this thesis, to answer this question, we develop an open-source framework that simplifies the process of simulation of quantum algorithms. We focus on defender-side quantum machine learning cases. We simulate quantum machine learning algorithms over real intrusion detection datasets as we were using fault-tolerant quantum computers.

The quantum machine learning algorithms that we consider can be seen as randomized approximation algorithms. Indeed, they are formalized in theorems and proofs in which a theoretical analysis of their running time, probability of failure γ, and the approximation error ϵ that we have to tolerate is reported. Indeed, in these algorithms, we might not know exactly the real solution s of our problem but we can approximate it with an absolute error such that $\|s - \overline{s}\|_2 \leq \epsilon$ with a probability of $1 - \gamma$. The approximation error is expected by the quantum computational paradigm and is not due to technological limits since we assume fault-tolerant quantum computers. The approximation error and probability of failure are running time parameters. In this work, to simulate quantum machine learning algorithms means to simulate their approximation errors. We insert ϵ error in the correct steps, as described in the proofs of the corresponding algorithms, to obtain estimates ϵ-close to the exact solution, following the theoretical guarantees of the error bounds. In this way, we see how the error, and more precisely the approximated solution of quantum machine learning algorithms used to solve intrusion detection problems, affects the accuracy in detecting intrusion with respect to the exact solution of classical machine learning algorithms. Once we find the errors that we can tolerate to best match classical performances, we fix the corresponding error parameters and we evaluate the running times as the number of samples and features change. We see which is the number of samples and features for which we observe a speed-up of quantum over classical machine learning, asking ourselves in which cybersecurity tasks we deal with data of that size. Therefore the core part of this work is the study of this trade-off between approximation error and running time, searching for cases of approximation errors that allow us to match classical performances and, at the same time, to observe a reasonable number of samples and features for which we have a quantum speed-up over classical machine learning.

Main contributions

The main contributions of this thesis can be summarized as follows:

- We build the Sqlearn framework. It simplifies the process of classically simulating quantum algorithms as we were using a fault-tolerant quantum computer. The framework allows us to simulate quantum routines such as phase estimation, amplitude estimation, and pure state tomography with realistic error models. By exploiting the main quantum routines, the framework also simulates more complex algorithms such as q-PCA and q-Means [7, 26].
- We simulate quantum state tomography, phase estimation, consistent phase estimation, and amplitude estimation to demonstrate that they actually work as expected. Moreover, we show that the number of measures we need to perform of a quantum state in ℓ_2-state tomography to reach the desired accuracy in the estimates is often much less than $N = \frac{36d \log d}{\delta^2}$ in practice [24].
- We extend a classical PCA-based anomaly detection algorithm, improving its performance over the CICIDS dataset [43]. In particular, we get an improvement of $\approx 15\%$ on F1-score and $\approx 10\%$ on accuracy.
- We do an analysis of the error of quantum machine learning algorithms to see how it affects the performances of quantum intrusion detection models.
- We report a critical analysis of the running times of quantum machine learning models compared to the classical ones under a future practical perspective.

Thesis outline

The thesis is organized into six chapters:

- **Machine learning preliminaries:** first, we describe the PCA and K-Means algorithms, which are the machine learning models implemented in our framework, to give a general overview of how they work classically. Then, we report the concept of anomaly detection and how machine learning can tackle this type of problem. Finally, we describe the anomaly detection models that we use in our experiments that are based on PCA and K-Means algorithms. We do not consider state-of-the-art anomaly detection models. Instead, we start from machine learning models that are already present in literature and still achieve competitive performances.
- **Quantum preliminaries:** we introduce quantum computation. We start from the main quantum postulates and we show their application in quantum computing.

We introduce the notation used that is useful to better understand the rest of this work. We report some useful notions in quantum machine learning such as quantum data representation, quantum data loading, and the QRAM concept. Finally, we describe the main quantum algorithms used in this work (i.e., phase estimation, consistent phase estimation, pure state tomography, amplitude estimation, median evaluation, inner product estimation, singular value estimation, and spectral norm estimation).

- **The Sqlearn framework:** we describe the *Sqlearn* framework with all the implemented quantum routines and their integration with q-PCA and q-Means. Furthermore, we present experiments and numerical simulations for quantum routines such as tomography, phase estimation, consistent phase estimation, and amplitude estimation. In this way, we verify their correct functioning and show some interesting insights into tomography.
- **Application to cybersecurity:** we describe the goal of our experiments, the datasets used, and the experiments done over those datasets both in their classical and quantum version. We show the improvements in performance for the extended classical model over the CICIDS dataset. Finally, we report a comparative study between the classical and quantum running times of the proposed models.
- **Open research directions:** we summarize the main contributions of this work, discussing the main limitations.
- **Conclusions:** we summarize our conclusions and promote future works on the topic of this thesis.

1 | Machine Learning preliminaries

In this chapter, we start with a description of the PCA and K-Means algorithms since the anomaly detection models that we successively describe are based on them. Then, we report the concept of anomaly detection and the role of machine learning to tackle this problem.

1.1. Machine Learning algorithms

1.1.1. Principal Component Analysis

Principal Component Analysis (PCA) is a technique used for many applications such as dimensionality reduction, lossy data compression, feature extraction, and data visualization. It is defined as the orthogonal projection of the data onto the subspace which accounts for most of the variance. It builds new features that are obtained by linear combinations of the original ones. The basic idea of PCA is to identify the directions on which the points spread more. More formally, given a dataset $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, the goal of PCA is to project those data into a new feature space $\mathbb{R}^{n \times k}$, where $k \leq d$, maximizing the variance of the projected data. In Figure 1.1, we report, just for demonstrative example, an application of the PCA model over 200 randomly sampled data points, showing the direction in which the points spread more, that corresponds to the first 2 principal components that PCA found. With PCA, we are able to visualize and work with a lower-dimensional dataset retaining the most important information. In real scenarios, since we deal with high-dimensional datasets, the procedure of projecting the data in a lower-dimensional space is very useful.
Now, more formally, we report the most important steps of the PCA procedure, assuming to start with a data matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$.
First of all, it is important to center the data. To do this, we compute the sample mean

$$\mu = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{d} \boldsymbol{x}_{ij} \tag{1.1}$$

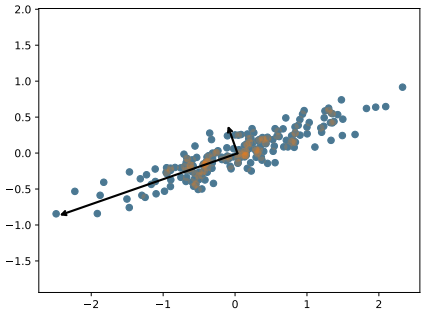


Figure 1.1: The first two principal components of 200 synthetic data points.

and remove it from all the data

$$\boldsymbol{X} = \boldsymbol{X} - \mu \mathbf{1}_n \mathbf{1}_d^T \tag{1.2}$$

where $\mathbf{1}_n$ is a column vector that contains n entries, all equal to 1.
Then, we compute the covariance matrix $\boldsymbol{S}$

$$\boldsymbol{S} = \frac{1}{n-1} \sum_{i=1}^{n} \sum_{j=1}^{d} (\boldsymbol{x}_{ij} - \mu)(\boldsymbol{x}_{ij} - \mu)^T.$$

Finally, we compute the eigenvalues and eigenvectors of $\boldsymbol{S}$. The eigenvector $\boldsymbol{e}_1$ associated with the largest eigenvalue λ_1 is the first principal component, and eigenvector $\boldsymbol{e}_k$ with the k-th largest eigenvalue λ_k is the k-th principal component. The eigenvectors of a matrix have the property of being a basis, so they are orthogonal to each other. Eigenvalues, instead, describe how much data are spread in the direction of eigenvectors. To select the number of principal components that explain the most of the variance of the data, we can use the factor score ratio given by the formula

$$\frac{\lambda_k}{\sum_j^n \lambda_j} \tag{1.3}$$

where $\sum_j^n \lambda_j = 1$. So, just to clarify, if we want to reduce the dimensionality of the data and we want to explain the 95% of the original variance, we take all the eigenvectors, starting with the one associated with the largest eigenvalue, and we try to add eigenvectors until the sum of eigenvalues reaches the 95% of the sum of all the eigenvalues.
The last step, once found the principal components, is to project the data onto the new reduced space. The projection of original data onto the first k principal components $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_k$ gives a reduced dimensionality representation of the data. This is done by

multiplying the original data matrix with the top-k principal components

$$\boldsymbol{X'} = \boldsymbol{X}\boldsymbol{E}^{(k)}$$

where $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ and $\boldsymbol{E} \in \mathbb{R}^{d \times k}$.

It is important to underline that PCA is strictly related to Singular Value Decomposition (SVD). SVD is a method to factorize a matrix such that, given any matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, we can decompose it as

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}} \tag{1.4}$$

where $\boldsymbol{U} \in \mathbb{R}^{n \times k}$ and $\boldsymbol{V} \in \mathbb{R}^{d \times k}$ are orthogonal matrices and $\boldsymbol{\Sigma} \in \mathbb{R}^{k \times k}$ is a diagonal matrix, with $k \leq \min(n, d)$. The columns of $\boldsymbol{U}$ and $\boldsymbol{V}$ are called *left* and *right singular vectors* respectively, while the values of the diagonal matrix $\boldsymbol{\Sigma}$ are the corresponding *singular values*. Now, the right singular vectors $\boldsymbol{V}$ of $\boldsymbol{A}$ are the principal components that, before introducing SVD, we indicate with $\boldsymbol{E}$.
In practice, to perform PCA, we can compute the singular value decomposition of the original matrix, after removing the mean from the data, as in Equation 1.4. Then, we can compute the percentage of explained variance using the singular values since

$$\frac{\lambda_k}{\sum_j^n \lambda_j} = \frac{\sigma_k^2}{(n-1)} \frac{(n-1)}{\sum_j^n \sigma_j^2} = \frac{\sigma_k^2}{\sum_j^n \sigma_j^2}$$

and obtaining the new feature space by multiplying the top-k left singular vectors with the singular values, since

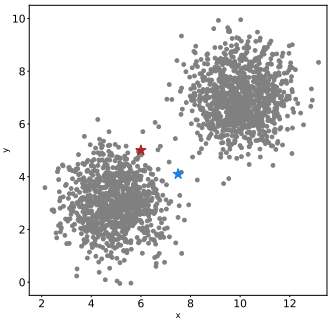
$$\boldsymbol{X'} = \boldsymbol{X}\boldsymbol{V}^{(k)} = (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}})\boldsymbol{V}^{(k)} = \boldsymbol{U}^{(k)}\Sigma^{(k)}$$

The computational complexity of PCA is $O(\min(n^2 d, nd^2))$, which is the time complexity of performing the full SVD, with n number of samples and d number of features. There are many randomized algorithms for SVD that, by computing an approximated matrix decomposition, help in extracting principal components in a faster way than the full SVD. For instance, if the input data matrix $\boldsymbol{X}$ is dense, there is a randomized algorithm that runs in $O(nd \log(k))$, with k number of principal components retained [18].

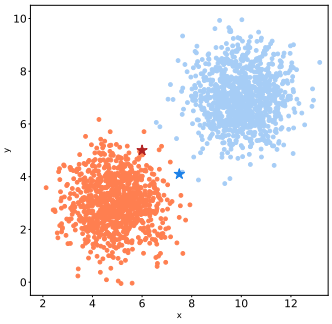
1.1.2. K-Means

K-Means is a clustering algorithm that aims to identify groups or clusters of data points in a multidimensional space. Let us suppose to have a dataset $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ of n d-dimensional observations. The goal of K-Means is to partition those data into K cluster $C_1, \ldots, C_K$,

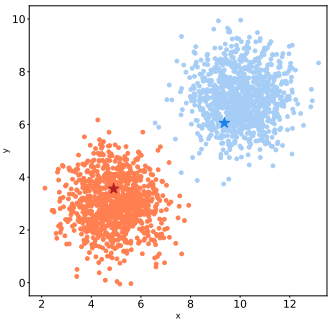
for a given K.



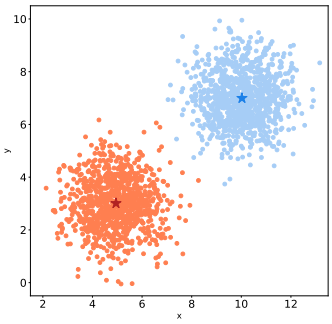
(a) K-Means initialization.



(b) K-Means initial data assignment.



(c) K-Means centroids update.



(d) Final K-Means clustering after the last centroids update.

Figure 1.2: K-Means example over synthetic data with K=2.

More formally, we define a d-dimensional vector $\boldsymbol{\mu}_k$ (also called centroid) with $k = 1, \ldots, K$, that is a representative of the k-th cluster (from here we can also define K-Means as a representative-based clustering algorithm). Basically, each of the k clusters can be represented with the corresponding $\boldsymbol{\mu}_k$. We write

$$\boldsymbol{\mu}_i = \frac{1}{n_i} \sum_{x_j \in C_i} \boldsymbol{x}_j$$

where n_i are the number of observations in the i-th cluster. Our goal, when executing K-Means is to assign each data point to a cluster, such that the sum of square distances of each data point to its closest vector $\boldsymbol{\mu}_k$ is a minimum. So giving a clustering $\mathcal{C} =$

$\{C_1, \ldots, C_k\}$, we define the SSE

$$SSE(\mathcal{C}) = \sum_{i=1}^{k} \sum_{\boldsymbol{x}_j \in C_i} \|\boldsymbol{x}_j - \boldsymbol{\mu}_i\|^2$$

and the goal is to find the clustering that minimizes the SSE score

$$\mathcal{C}^* = \arg\min_{\mathcal{C}} SSE(\mathcal{C}).$$

In Figure 1.2, we report a sketch of K-Means execution over synthetic generated data, choosing K=2. At the beginning, we initialize the two centroids, the red and blue crosses in Figure 1.2a. Then, there is the assignment step, in Figure 1.2b, where we assign every single observation to one of the two centroids based on the distance between the data and the representative object. In Figure 1.2c, we update the centroids based on the previous data assignment, making the average of all data in each cluster. In Figure 1.2d, we report the last snapshot of the K-Means algorithm, after some iterations, showing the resulting clustering.

So, the first step of the K-Means algorithm is the centroids initialization. There are many ways of doing it: the easiest one is to generate randomly k points in the data space, or using an initialization algorithm as *kmeans++* [4], or even using another clustering algorithm such that we cluster a sample data in k clusters and, from each one of those, we pick one point as a centroid.

K-Means is an iterative algorithm, where each iteration consists of two steps: cluster assignment and centroid update. So, given the k initial centroids, we assign each d-dimensional point $\boldsymbol{x}_j$ to the closest cluster C_{j^*} where

$$j^* = \arg\min_{i=1}^{k} \|\boldsymbol{x}_j - \boldsymbol{\mu}_i\|^2$$

Then, after the cluster assignment, new mean values are computed considering the points in each of the k clusters, resulting in new updated centroids. This procedure is repeated iteratively until K-Means has converged, which means until the cluster centroids do not change between two consecutive iterations. A possible stopping condition can be

$$\sum_{i=1}^{k} \|\boldsymbol{\mu}_i^t - \boldsymbol{\mu}_i^{t-1}\|^2 \leq \epsilon$$

where ϵ is the convergence threshold and t is the current iteration of the algorithm.

For what concern the computational complexity, it is given by $O(tnkd)$, with t number

of iterations, n number of samples, k number of clusters, and d number of features. K-Means clustering has some limitations since it has problems with clusters of different sizes, and densities, or with non-globular-shaped ones. Another problem that might affect K-Means clustering is the presence of outliers in the data because they could condition the centroids' computation. Indeed, if all the data are grouped together, like a globular shape, the average makes sense since it is a good representation of all those points. With the presence of outliers, instead, the centroids computation is biased and the cluster centers are pushed towards the outliers.

1.2. Machine Learning for anomaly detection

Anomaly detection is the problem of identifying patterns, events, observations, or data that differ significantly from the expected behavior. In Figure 1.3, we report a very simple example to introduce the anomaly detection problem. As we can see, the blue data are very close to each other, while the red squares are very distant from them. For this reason, the latter could be considered *anomalies* or *outliers*. The goal of anomaly detection is to find patterns that well describe the behavior of the data and, given new data, to be able to determine if they conform to the normal behavior or are anomalies. In Figure 1.3, the ellipsis that surrounds the blue data exemplifies the expected normal behavior. Clearly,

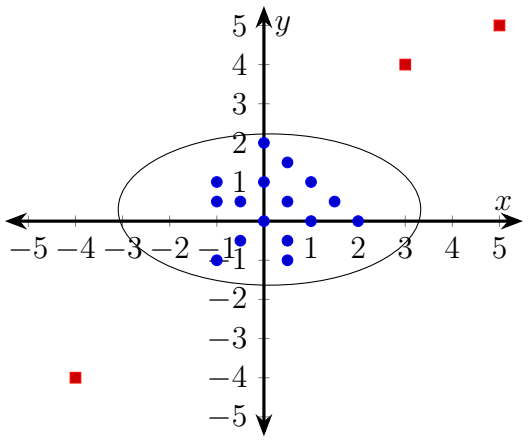


Figure 1.3: Anomaly detection example. Blue dots represent normal observations while red squares are anomalies.

this is a very straightforward approach, in which we circumscribe an area representing the normal behavior and we describe as an anomaly any observation which is not included in this normal region. But in real scenarios, the distribution of data is more complex and, very often, there are no such clear-cut regions, which means that the anomalies are not so different from the normal observations. Of course, this makes the process of finding

patterns conforming to the normal behavior more complex.
There are many types of anomaly detection applications like health anomaly detection, bank fraud, and many others.
In this work, we concentrate on *intrusion detection*, more precisely on *network intrusion detection*, whose main purpose is to analyze network traffic to detect anomalies in the network flow or intrusions by an attacker. We assume to deal with *point anomalies*: in our experiments, we consider data independent of each other and, in the classification stage, we analyze every single observation by itself, trying to figure out if it is an anomaly or not. Anomalies can be categorized also in contextual and collective anomalies, as reported by Chandola et al. [11].
In general, one of the main problems of anomaly detection intrusion detection systems (IDS), which is the family of the models that we use in this work, is that they are very prone to errors, in particular generating false positive (FP) or false negative (FN) results. FP means that the system detects as anomalous a point that, instead, is normal and legitimate. FN is the opposite, meaning that the system does not raise an alarm for a point that is anomalous. The high false-negative problem, at first sight, may seem more threatening because it means that a system exposed to attacks is unable to detect them. Anyway, also the high false-positive problem is not to be underestimated, as a system that continuously raises alarms for points that in reality are not attacks, it no longer becomes reliable and will be ignored. In all the anomaly detection IDS models, we have to deal with false-negative and false-positive problems. Indeed, one of the main objectives when building such systems is precisely that of limiting the false-negative and false-positive rates to the minimum. The latter is not an easy task since, generally, a decrease in the number of false positives leads to an increase in false negatives and viceversa. Therefore, we need to find a trade-off between the two. Generally, also in our work, the metrics used to evaluate the performance of the system are:

- precision: $p = \frac{TP}{TP+FP}$ that is, how many points classified as anomalies are actually anomalies?
- recall: $r = \frac{TP}{TP+FN}$ that is, how many points are classified as anomalies among all anomalous points?
- accuracy: $a = \frac{TP+TN}{TP+FP+TN+FN}$, that is the number of correctly predicted points out of all the points.
- $F1$-score: $F1 = \frac{2}{\frac{1}{p}+\frac{1}{r}}$, that is the harmonic mean of precision and recall.

Therefore, when we talk about the trade-off between false positive and false negative, we refer, more precisely, to the balance between precision and recall. Indeed, as we can see,

an increase in false positives results in a decrease in precision. Viceversa, an increase of false negatives corresponds to a decrease in recall. As we are going to show also in our experiments, it is important to consider all these metrics together. For instance, taking into account only the accuracy can be misleading in the case of a high-unbalanced dataset and, since one of the most frequent problems in anomaly detection is the fact that the data we deal with are unbalanced, this might be a problem. About that, let us consider a dataset with the unbalancing ratio between anomalies and normal samples of 1:100, which means that each anomaly sample has a corresponding 100 normal samples. We have seen that, in anomaly detection, we want that our systems are able to detect the anomalies, which are the minority class. If we wrongly build a system that detects always the normal samples, we would find an accuracy of 99%, which, at first sight, might seem very good. But, in this case, we would have precision and recall of 0, meaning that the model is completely useless. Of course, this is an extreme case, but it is to convey the idea that it is very important to consider all those metrics together in the evaluation of the performance of a model.

Another important challenge for intrusion detection problems is that data, in general, are not labeled. This is because there is a huge amount of data, and people should manually label every single observation, which is a very time consuming process. Moreover, labeling anomalies is not easy as the types of attacks are constantly evolving. In our experiments, the datasets used are labeled both because they are small and in some cases fictitious datasets and because, otherwise, we could not measure the performance of our models if we did not have a ground-truth to compare with. Generally, anomaly detection problems are not easy as the one depicted in Figure 1.3. Therefore, we need more complex and powerful models to solve these problems and, depending on whether we have labeled data or not, we can use three different machine learning paradigms to tackle anomaly detection problems: supervised, semi-supervised, and unsupervised anomaly detection.
In our work, despite the presence of the labels in the data that we use to have a ground-truth to compare with, the algorithms that we are going to use are part of the unsupervised anomaly detection family. Unsupervised machine learning is a good solution to discover the underlying structure of the data. Algorithms of this type try to find hidden patterns or clusters within the data, without the need for a human to label the data. This method is typically used when the data are unlabeled.
The main problems that unsupervised machine learning tackles are: clustering, which is the problem of searching for natural grouping or structure in unlabeled data according to some distance measures, dimensionality reduction, which is the problem of reducing the dimension of a dataset from a high-dimensional space to a lower one maintaining the

most important properties of the data, and association rules that is the problem of finding frequent patterns, correlations, or causal structures among sets of items in information repositories.
The anomaly detection models that we are going to introduce in the next sections and that we use in our cybersecurity experiments in Chapter 4 are already present in literature and they are based on PCA dimensionality reduction and K-Means clustering algorithms. We also extend one of the classical models, adding some novelties (Section 1.2.2) which bring improvements in the performance.

1.2.1. Principal components classifier

The first anomaly detection model that we present is the one described by Shyu et al. [45]. It is a PCA-based model for intrusion detection, where anomalies are treated as outliers. The intrusion predictive model is built over the *major* and *minor* components of the normal observations. The major (or principal) components are the ones that account for the majority of the variance, while the minor components are the ones whose corresponding eigenvalues are less than a specific value. We exploit these components, extracted by the PCA model fitted only on normal observations, to compute the outlier thresholds, which we use to classify if a sample is normal or an anomaly. More precisely, for each new incoming observation, we compute the anomaly score using, also in this case, the majors and minors components. Then, if this score is greater than the outlier threshold, we classify the corresponding observation as an anomaly, meaning that its anomaly score is suspicious compared to the normal one (which is represented by the anomaly threshold). Now let us describe more formally this model.

The first thing that we have to do is to perform PCA on the correlation matrix of the normal observations which is equivalent to perform PCA on the covariance matrix with standardized features, so features normalized to zero mean and unit variance. So, what we have done is just standardize the features before performing PCA on the input matrix. It is important to be sure that the training data do not contain outliers. A way of ensuring this is the so called *trimming*, which is a procedure designed to remove outliers. Let us suppose to have an initial data matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ that contains n observations of d variables $X_1, \dots X_d$. Let us suppose also that $\boldsymbol{S} \in \mathbb{R}^{d \times d}$ is the sample covariance matrix. Starting from the samples mean $\overline{\boldsymbol{x}} = (\overline{x}_1, \overline{x}_2, \dots, \overline{x}_d)$ and from the samples covariance matrix $\boldsymbol{S}^{-1}$, we define the Mahalanobis distance as

$$d_i^2 = (\boldsymbol{x_i} - \overline{\boldsymbol{x}})^T \boldsymbol{S^{-1}} (\boldsymbol{x_i} - \overline{\boldsymbol{x}}) \tag{1.5}$$

for each sample $\boldsymbol{x_i}$, with $i = 1, \dots n$. Once computed the Mahalanobis distance for each sample, we find the $\xi * n$ largest values of $\{d_i^2, i = 1, \dots, n\}$ and remove the corresponding observations. For example, let us suppose for simplicity that $n = 4$ and $d^2 = [1.2, 3.4, 0.3, 5.6]$. Let us suppose also that $n * \xi = 2$, so we want to find the two largest values of d^2 which, in this case, are 5.6 and 3.4 that correspond to the indexes $i = 3$ and $i = 1$. Therefore, from the initial training set, we remove the samples corresponding to that indexes: $\boldsymbol{x}_1$ and $\boldsymbol{x}_3$.
In this work, we use $\xi = 0.005$ as done in Shyu et al. [45] since we want to be able to replicate their experiments. After removing the outlier from the training set as explained before, we obtain new trimmed estimators $\overline{\boldsymbol{x}}$ and $\boldsymbol{S}$ from the remaining observations. From the elements of this new matrix $\boldsymbol{S}$, we can extract a robust estimator of the correlation matrix.

That said, let us go into more details of the models. Let $\boldsymbol{R} \in \mathbb{R}^{d \times d}$ be the correlation matrix computed from the d features $X_1, \dots, X_d$ of the n samples. Let $(\lambda_1, \boldsymbol{v}_1), \dots, (\lambda_d, \boldsymbol{v}_d)$ be the d eigenvalue-eigenvector pairs of $\boldsymbol{R}$, with $\lambda_1 \geq \lambda_2 \geq, \dots, \lambda_d \geq 0$. Then, we can define the i-th coordinate of an observation sample $\boldsymbol{x} = (x_1, x_2, \dots, x_d)^T$ expressed with respect to the principal components as

$$y_i = \boldsymbol{v_i^T z} = v_{i1} z_1 + v_{i2} z_2 + \cdots + v_{id} z_d, \quad i = 1, \dots, d \tag{1.6}$$

where

$$\boldsymbol{v_i} = (v_{i1}, v_{i2}, \dots, v_{id})^T \quad \text{is the } i\text{-th eigenvector}$$

and

$$\boldsymbol{z} = (z_1, z_2, \dots, z_d)^T$$

is the vector of standardized observations defined as

$$z_j = \frac{x_j - \overline{x}_j}{\sqrt{s_{jj}}}, \quad j = 1, \dots, d$$

where $\overline{x}_j$ and s_{jj} are respectively the sample mean and sample variance of the variable X_j as defined previously.
The main elements of this anomaly detection model are the two summations (also called principal components scores)

$$T_1 = \sum_{i=1}^{k} \frac{y_i^2}{\lambda_i}, \quad T_2 = \sum_{i=d-r+1}^{d} \frac{y_i^2}{\lambda_i}, \tag{1.7}$$

where k indicates the number of major components and r corresponds to the minor ones. The scopes of those two summations are different. Indeed, the first summation aims to detect outliers with large values on some original features, while the second one aims to help the first one in detecting the observations that deviate from the normal behavior. The whole outlier detection scheme is based on those two summations in the following way: we compute, for a new observation $\boldsymbol{w}$ for which we want to determine if it is an anomaly or not, the two principal components scores given by the results of two summations reported in Equation 1.7, and we classify $\boldsymbol{w}$ as an *attack* if

$$T_1 > c_1 \quad \textbf{or} \quad T_2 > c_2 \tag{1.8}$$

or as *normal* if

$$T_1 <= c_1 \quad \textbf{and} \quad T_2 <= c_2 \tag{1.9}$$

where c_1 and c_2 are the outlier thresholds. To compute them, we use the equation from [45]

$$\alpha = \alpha_1 + \alpha_2 - \alpha_1\alpha_2, \tag{1.10}$$

where α is the false alarm rate (which in statistic is defined as FAR=$\frac{FP}{FP+TN}$), $\alpha_1 = P(T_1 > c_1|\boldsymbol{x}$ is normal instance), and $\alpha_2 = P(T_2 > c_2|\boldsymbol{x}$ is normal instance). We assume that $\alpha_1 = \alpha_2$, so that Equation 1.10 becomes a second degree equation. In this way, specifying the value of α, we are able to compute $\alpha_1 = \alpha_2$. For example, taking $\alpha = 2\%$, we found $\alpha_1 = \alpha_2 = 0.0101$. At this point, we set the outlier thresholds using the empirical distribution of T_1 and T_2 in the training data. Therefore, c_1 and c_2 are the $1-\alpha_1 = 0.9899$ quantile of the empirical distribution of $\sum_{i=1}^{k} \frac{y_i^2}{\lambda_i}$ and $\sum_{i=d-r+1}^{d} \frac{y_i^2}{\lambda_i}$ respectively, which means the data point that is bigger than 98.99% of all the other points. To make this more clear, we describe briefly how we implement this concept, focusing on c_1. For each training data, we compute the corresponding summation score $\sum_{i=1}^{k} \frac{y_i^2}{\lambda_i}$. In this way, we find a list of length n of scores, with n number of samples in the training set. Then, we simply sort the data in ascending order and find the 0.9899 (in this example) largest value, that will be c_1.
We report the Principal Component Classifier (PCC) procedure in Algorithm 1.1.

Algorithm 1.1 PCC.

Input: Training set X of normal samples, test set W of normal/anomalies, $0 \leq p \leq 1$, $0 \leq \nu \leq 1$.

Output: Classify each test observation either as normal or anomaly.

1: Fit PCA model with training set X, specifying the retained variance p.
2: Extract k major components such that their factor score ratios sum matches the retained variance p.
3: Extract the r minor components such that their corresponding eigenvalues are less or equal than ν.
4: Compute outlier thresholds c_1 and c_2 that refer to the major and minor components summations respectively, using the training samples X.
5: **for** each new observation $w \in W$ **do**
6: Compute $T_1 = \sum_{i=1}^{k} \frac{y_i^2}{\lambda_i}$ and $T_2 = \sum_{i=d-r+1}^{d} \frac{y_i^2}{\lambda_i}$.
7: **if** $T_1 > c_1$ **or** $T_2 > c_2$ **then**
8: Classify w as attack.
9: **else if** $T_1 \leq c_1$ **and** $T_2 \leq c_2$ **then**
10: Classify w as normal.
11: **end if**
12: **end for**

Just to point out, we can use this model in two ways: either using both major and minor summations or using only the major summations. In the latter case, we classify a new observation as attack if $\sum_{i=1}^{k} \frac{y_i^2}{\lambda_i} > c_1$ or as normal if $\sum_{i=1}^{k} \frac{y_i^2}{\lambda_i} <= c_1$.

1.2.2. Ensemble of principal components classifiers

Taking a cue from the previous model, we extend it using *cosine* similarity and *correlation* measure, as we are going to show. Our idea derives from the basic concept of the previous model which is

$$\sum_{i=1}^{k} \frac{y_i^2}{\lambda_i}$$

where

$$y_i = \boldsymbol{v_i^T z} = v_{i1} z_1 + v_{i2} z_2 + \cdots + v_{id} z_d, \quad i = 1, \ldots, d.$$

We notice that y_i is computed with the dot product between the i-th eigenvector $\boldsymbol{v}_i$ and the normalized sample $\boldsymbol{z}$. The dot product can be interpreted geometrically as projecting

a vector onto the other as if we wanted to compare them. From here we wondered: why not use *similarity measures* or the *correlation* between the two vectors instead of the dot product to compute y_i?

In particular, we consider:

- the *cosine* similarity, which is defined as $\frac{\boldsymbol{u}\cdot\boldsymbol{m}}{\|\boldsymbol{u}\|_2\|\boldsymbol{m}\|_2}$, with vectors $\boldsymbol{u}, \boldsymbol{m} \in \mathbb{R}^n$.
- the *correlation* measure defined as $R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$, with covariance matrix C (by Numpy [19]).

In this way, we have three different ways of computing the principal components scores T_1 and T_2 reported in Equation 1.7: in the first, we use the classical dot product, in the second the cosine similarity, and in the last one the correlation measure. We can think of it as having three different models and our extension to the basic model is exactly this: to merge together all those three different "models" like an *ensemble*. In general, the ensemble is a method in which we aggregate predictions made by multiple classifiers to boost the performances. The ensemble exploits the fact that there is not much chance that, if a model fails, also the others fail. We call our model *ensemble principal components classifiers*, just because it recalls the use of different models, but it is not exactly an ensemble model in the true sense of the term. Indeed, we do not fit three different models with three different training sets. We just fit a PCA model with a training set (composed of only normal samples) and then, using the major and minor components extracted by the PCA model, we compute different outlier thresholds using the dot product, the cosine similarity, and the correlation measure. Finally, all these thresholds are put together like a voter to predict the class of a new observation.

In Figure 1.4, we report a sketch of this new model, in case of using only major components to classify new observations. We mainly have three steps: in the first one, we fit PCA with normal training data, extracting all the eigenvalues and eigenvectors that retain a specified percentage of variance. In the second step, we compute all the thresholds using the three different approaches. For example, c_{1cos} is the threshold computed with the summation that involves the major components, where y_i is computed using the cosine similarity between the top-k eigenvectors, extracted by the fitted PCA, and the normalized training samples (i.e., $y_i = \frac{v_i \cdot z}{\|v_i\| \cdot \|z\|}$). The same goes for c_{2cos} but using the summation of the minor components (the same for the other thresholds, with c_{1dot} and c_{1cor} that refer to the computation of the thresholds using dot product and correlation measure respectively). Then, in the last step, we put all the conditions of prediction in OR, considering that s_{1cos}, for example, corresponds to the anomaly score of a new observation sample, computed using $\sum_{i=1}^{k} \frac{y_i^2}{\lambda_i}$, where, as before, y_i is computed using the cosine similarity. In the case

of using both major and minor components, we need to add, in the OR control, also the conditions relative to the thresholds c_2. Finally, if the condition in the green square is true, we classify the new observation as an *attack*, otherwise as *normal*. So, just to clarify, let us suppose that c_{1cos}, c_{1corr}, and c_{1dot} are the threshold computed using the summation involving the major components considering the cosine similarity, the correlation measure, and the dot product respectively in the computation. Using only the major components, we classify a sample as an attack if

$$s_{1cos} > c_{1cos} \quad \textbf{or} \quad s_{1corr} > c_{1corr} \quad \textbf{or} \quad s_{1dot} > c_{1dot}$$

or as normal if

$$s_{1cos} \leq c_{1cos} \quad \textbf{and} \quad s_{1corr} \leq c_{1corr} \quad \textbf{and} \quad s_{1dot} \leq c_{1dot}$$

where s_{1cos}, for example, is the anomaly score of the current sample computed using only the major components in the summation and considering the cosine similarity in the computation. As said before, in case we use also the minor components, we need to consider also the corresponding thresholds c_2 and anomaly scores s_2.

The strength of this model is the fact that in the case that a malicious sample escapes the dot product control, which means that we are not able to detect the bad sample with the threshold computed using the dot product, perhaps it may not get out of control where we use correlation or cosine, resulting in a more robust classifier. On the other side, there could be the problem of an increase in the false-positive rate. Indeed, using more thresholds in OR conditions to assess whether an action is anomalous or not increases the possibility to classify the action as an attack, increasing the risk of false-positive, especially in unbalanced datasets.

1.2.3. PCA with reconstruction loss

We report this PCA-based model from Verkerken et al. [49]. We always assume to fit our PCA model with a training set composed only of normal data. We project each new sample into the PCA feature space. To do this, we compute the dot product between the sample and the principal components transposed. Indeed, let us consider a new sample $\boldsymbol{w} \in \mathbb{R}^d$ and the principal components (or equivalently right singular vectors, as we have seen in Section 1.1.1) matrix $\boldsymbol{V} \in \mathbb{R}^{k \times d}$. By performing the dot product between $\boldsymbol{w}$ and $\boldsymbol{V}^T$, we map the new observation from the d-dimensional feature space to the k-dimensional one, with $k \leq d$. Then, we reconstruct the original sample. This can be done by computing the

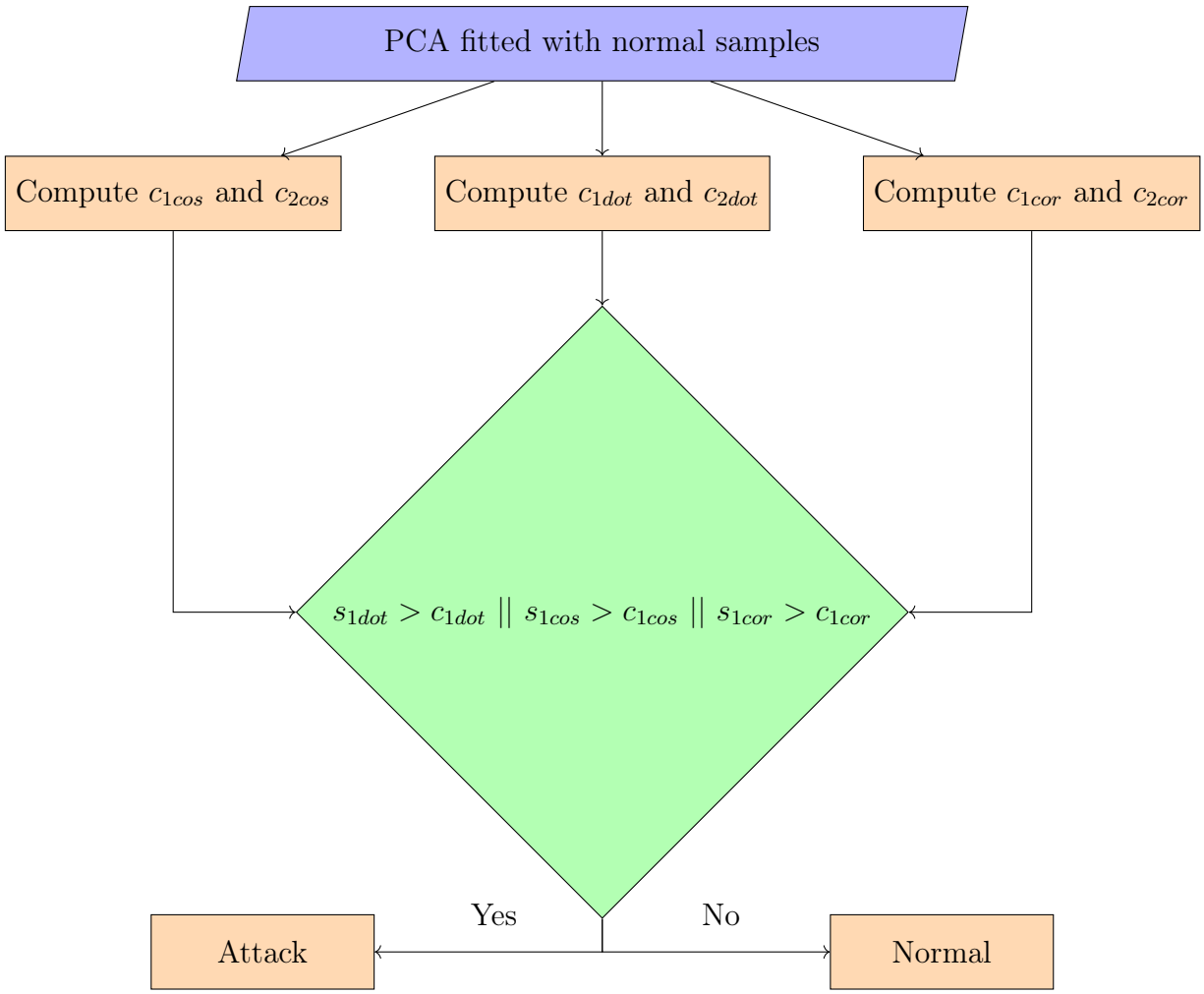


Figure 1.4: Sketch of the ensemble of principal components classifier model with "only major components".

dot product between the transformed sample and the principal components, projecting back the transformed sample from the k-dimensional feature space to the d-dimensional one. We call this reconstructed sample $\widehat{\boldsymbol{w}}$. The process of transforming the original data into a lower-dimensional feature space and then projecting them back into the original feature space inevitably brings with it a reconstruction error. Therefore, the final step, after the re-projection into the original feature space, is the computation of the *loss* between the original sample test and the reconstructed one using the residual sum of square (RSS) error

$$RSS = \sum_{i=1}^{d} (w_i - \widehat{w}_i)^2.$$

We use this loss value as anomaly score. In theory, if a sample is anomalous is more likely to have a higher reconstruction error with respect to a normal one. Therefore, using a threshold that we find with hyperparameter tuning, we can classify if a sample is anomalous or not, based on the corresponding reconstruction error.

1.2.4. PCA and K-Means

One of the biggest challenges when doing clustering is clustering evaluation. So, with this in mind, we did an experiment following the work reported by Peng et al. [35] since it is one of the most cited documents in this regard. In this paper, different approaches are described and one of them is the usage of K-Means after having reduced the dimensionality of the data with PCA. The objective is to compute the CH index of the K-Means clustering that, basically, indicates if the clustering is good or not. CH index refers to the Calinski-Harabasz index, which measures how an element conforms to its cluster (cohesion) compared to other clusters (separation). It is defined as (from the definition reported in Sklearn [34]):

$$CH = \frac{tr(B_{n_k})}{tr(W_{n_k})} \times \frac{n_E - n_k}{n_k - 1}$$

where $tr(B_{n_k})$ and $tr(W_{n_k})$ are the trace of the between group dispersion matrix and the trace of the within-cluster dispersion matrix:

$$W_k = \sum_{q=1}^{n_k} \sum_{x \in C_q} (x - c_q)(x - c_q)^T$$

$$B_k = \sum_{q=1}^{n_k} n_q (c_q - c_E)(c_q - c_E)^T$$

with n_k number of centroids, C_q set of points in cluster q, c_q center of q-th cluster, c_E center of dataset E and n_q, n_E number of points in cluster q and in dataset E respectively. So, this model is not a true anomaly detection model where recall, precision, and accuracy are computed to see the anomaly detection capabilities of the model. However, we also report it to see the K-Means algorithm applied even in its quantum counterpart.
The procedure is very simple: firstly, we apply PCA to reduce the dimensionality of the dataset. Then, on that lower-dimensional projected data, we fit K-Means with different number of clusters $n_k = [10, 20, 30, \ldots, 100]$. Finally, we extract the CH value of the resulting clustering.

2 | Quantum preliminaries

In this chapter, we describe in detail the theory relative to the world of quantum computing. We start from the notation up to the main algorithms. This is an introductory chapter that is useful if one has no experience or confidence with quantum computing. If this is the case, we still recommend consulting the book of Kaye et al. [22]. The reader might need a basic knowledge of quantum computing to understand the rest of this work.

2.1. Quantum information

We start with the basic notation in quantum computing, and then we present the main postulates of the quantum mechanics to show their application in quantum computing. Finally, we report how we can represent, load, and retrieve data in a quantum computer, introducing the important concept of QRAM and KP-trees.

2.1.1. The bra-ket notation

The *bra-ket* (or Dirac) notation is at the base of quantum information and it is used to denote quantum states. The notation $|\cdot\rangle$, which is called *ket*, denotes a vector in a complex space. Instead, $\langle\cdot|$, which is called *bra*, denotes the complex conjugate (that we indicate with the symbol *) transpose of the ket elements. Just to clarify, if

$$|x\rangle = \begin{bmatrix} 1-2i \\ 1+3i \\ 4-9i \end{bmatrix}$$

then

$$\langle x| = [1+2i, \quad 1-3i, \quad 4+9i]$$

In the case of a real vector, the conjugate transpose is just the transpose of the vector. Similarly, the complex conjugate of a real number is the number.

We can define three important and very useful linear algebra operations using the bra-ket

notation. Let us take two vector $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$, the *inner product* of the two can be written as:

$$\langle \mathbf{x}|\mathbf{y}\rangle = [x_1^* \dots x_n^*] \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} x_i^* y_i.$$

Considering now two vectors $\mathbf{x} \in \mathbb{C}^n$ and $\mathbf{y} \in \mathbb{C}^m$, the *outer product* is defined as:

$$|\mathbf{x}\rangle \langle \mathbf{y}| = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} [y_1^* \cdots y_m^*] = \begin{bmatrix} x_1 y_1^* & \cdots & x_1 y_m^* \\ \vdots & & \vdots \\ x_n y_1^* & \cdots & x_n y_m^* \end{bmatrix}.$$

Finally, we define the *tensor product* in this way:

$$|\mathbf{x}\rangle \otimes |\mathbf{y}\rangle = \begin{bmatrix} x_1 \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \\ \vdots \quad \vdots \\ x_n \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \end{bmatrix} = \begin{bmatrix} x_1 y_1 \\ x_1 y_2 \\ \vdots \\ x_{n-1} y_{m-1} \\ \vdots \\ x_n y_m \end{bmatrix}.$$

2.1.2. The qubit

In quantum information, the *qubit* is the basic unit of information, as the bit in classical information. It is important to underline the first substantial difference between the two units: if it is true that the bit can only be worth 0 or 1, this is not generally true for the qubit, which can be in the state $|0\rangle$, $|1\rangle$, or in a superposition of two states $|0\rangle$ and $|1\rangle$ at the same time, as it is a form of a linear combination of the two states. More formally, a qubit is a vector in $\mathbb{C}^2$ and can be specified as

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

where $\alpha, \beta \in \mathbb{C}$ are called *amplitudes* and are such that $|\alpha|^2 + |\beta|^2 = 1$.
A single qubit can be seen as a unit vector in the Hilbert space $\mathcal{H}_1$ (indeed $\mathcal{H}_n \simeq \mathcal{C}^{2^n}$), in which the states

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

form an orthonormal basis called *computational basis*.
Indeed,

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

$$\langle 1|0\rangle = \langle 0|1\rangle = 0.$$

These properties come from the first postulate of quantum mechanics.

First postulate of quantum mechanics [33]

Associated to any isolated physical system is a complex vector space with inner product (that is, a Hilbert space) known as the state space of the system. The system is completely described by its state vector, which is a unit vector in the system's state space.

It can be shown [22], that we can rewrite a qubit in $\mathbb{C}^2$ as

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

with $e^{i\varphi}$ that represents the phase of the state. In this way, we can visualize the qubit on the Bloch sphere, reported in Figure 2.1. The points on the surface of the Bloch sphere represent the so called *pure* states, which are the ones whose sum of the squares of the amplitudes is one. The points intern to the sphere, instead, are called *mixed* states and are the ones whose sum of squares of the amplitudes is not one. In this work, when we talk about quantum states, we always implicitly refer to pure states.

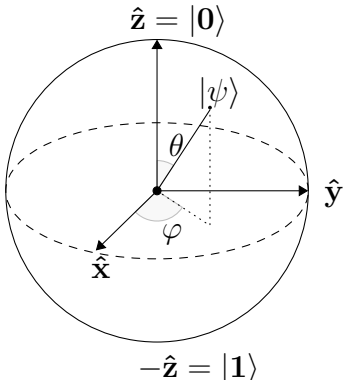


Figure 2.1: Visualization of a qubit through the Bloch sphere [51].

The computational basis is also called z-basis. Obviously, there are also other orthonormal

basis than the z one as, for example, the x-basis, which we write as

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \quad |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.3}$$

or the y-basis

$$|R\rangle = \frac{|0\rangle + i\,|1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} \end{bmatrix}, \quad |L\rangle = \frac{|0\rangle - i\,|1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{i}{\sqrt{2}} \end{bmatrix}. \tag{2.4}$$

Of course, a quantum state written with respect to a specific computational base can be written also in another basis, as for example

$$|0\rangle = \frac{|+\rangle + |-\rangle}{\sqrt{2}}, \quad |1\rangle = \frac{|+\rangle - |-\rangle}{\sqrt{2}} \tag{2.5}$$

so that the state $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ can be rewritten as

$$|\psi\rangle = \alpha\frac{|+\rangle + |-\rangle}{\sqrt{2}} + \beta\frac{|+\rangle - |-\rangle}{2} = \frac{\alpha+\beta}{\sqrt{2}}\,|+\rangle + \frac{\alpha-\beta}{\sqrt{2}}\,|-\rangle\,. \tag{2.6}$$

2.1.3. Multiple qubits and quantum systems

We can extend the discussion for a single qubit to the multiple qubits one. We can consider a quantum state with n-qubits, which can be seen as a vector in a complex Hilbert space of dimension 2^n and can be written as

$$|\mathbf{x}\rangle = \sum_{i=0}^{2^n-1} \alpha_i\,|i\rangle \tag{2.7}$$

with $\alpha_i \in \mathbb{C}$ and $\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1$. It is important to note that $|i\rangle$ is binary encoded which means that, for example, $|7\rangle = |0111\rangle$ with 4 qubits. Multiple qubits together are often called *quantum register*.

The fourth postulate of quantum mechanics is the one that describes how to define a quantum system.

Fourth postulate of quantum mechanics [33]

The state space of a composite physical system is the tensor product of the state spaces of the component physical systems. Moreover, if we have systems numbered from 1 through n and each state is described as $|\psi_i\rangle$*, the join state of the total system is* $\bigotimes_{j=1}^{n} |\psi_i\rangle = |\psi_1\rangle |\psi_2\rangle \dots |\psi_n\rangle$.

For example, we can describe a system of 2 qubits with vectors in $\mathbb{C}^4$. In this case, we can write the computational basis as

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad |01\rangle = |0\rangle \otimes |1\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$|10\rangle = |1\rangle \otimes |0\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad |11\rangle = |1\rangle \otimes |1\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Therefore, we can express every state of a 2-qubit system as

$$|\psi\rangle = \alpha_0 |00\rangle + \alpha_1 |01\rangle + \alpha_2 |10\rangle + \alpha_3 |11\rangle .$$

2.1.4. Evolution of quantum states

To process qubits and quantum registers, we use quantum gates. These are unitary operators in the Hilbert space that can be seen as unitary matrices $\boldsymbol{U}$, which are norm-preserving and map a unit-norm vector to a unit-norm vector:

$$\boldsymbol{U} |\psi\rangle \to |\psi'\rangle . \tag{2.8}$$

A unitary $\boldsymbol{U} \in \mathbb{C}^{n \times n}$ has the property $\boldsymbol{U}^\dagger \boldsymbol{U} = \boldsymbol{U}\boldsymbol{U}^\dagger = \mathbb{I}$, with $\boldsymbol{U}^\dagger$ that is the conjugate transpose of $\boldsymbol{U}$. Equation 2.8 describes the evolution of quantum states through unitary operators, which is the concept described in the second postulate of quantum mechanics.

Second postulate of quantum mechanics [33]

The evolution of a closed quantum-mechanical system is described by a unitary transformation. That is, the state $|\psi\rangle$ of the system at the time t_1 is related to the state $|\psi'\rangle$ at time t_2 by the unitary operator $\boldsymbol{U}(t_1, t_2)$ from the relation $|\psi'\rangle = \boldsymbol{U}|\psi\rangle$.

The unitary matrices preserve the inner products between vectors, as well as the norm of the vectors. Indeed, a unitary matrix applied to a vector takes the vector and rotates it, without changing the vector norm. Therefore, the application of unitary matrices to vectors corresponds to a rotation of the state representing the vector on the Bloch sphere, represented in Figure 2.1. We can see any unitary matrix as a valid reversible quantum circuit.

There are lots of quantum logical gates. For a single qubit, it is important to mention first the Pauli-X gate. It applies over a single qubit and it corresponds to the classical NOT operator. It is defined in this way:

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

So, for example, it is easy to see that by taking $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ and applying the Pauli-X operator to it, we obtain

$$\boldsymbol{X}|\psi\rangle = \alpha|1\rangle + \beta|0\rangle. \tag{2.9}$$

Indeed,

$$\boldsymbol{X}|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

$$\boldsymbol{X}|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle.$$

If we want to visualize the quantum NOT gate operator, we can think of it as a rotation of 180° for the θ angle in the Bloch sphere.

Another important single-qubit operator is the Hadamard gate:

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

which maps $|0\rangle \to \frac{|0\rangle+|1\rangle}{\sqrt{2}} = |+\rangle$ and $|1\rangle \to \frac{|0\rangle-|1\rangle}{\sqrt{2}} = |-\rangle$, thus creating an equal superpo-

sition of the two basis states. Indeed,

$$\boldsymbol{H}\left|0\right\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ 1\end{bmatrix} = \frac{\left|0\right\rangle + \left|1\right\rangle}{\sqrt{2}} = \left|+\right\rangle$$

$$\boldsymbol{H}\left|1\right\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}0\\ 1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ -1\end{bmatrix} = \frac{\left|0\right\rangle - \left|1\right\rangle}{\sqrt{2}} = \left|-\right\rangle.$$

A multiple-qubits Hadamard gate allows creating an n-qubit state in a uniform superposition of all the states in the computational basis

$$\boldsymbol{H}_n\left|0\right\rangle^{\otimes n} = \frac{1}{\sqrt{n}}\sum_{i}^{n}\left|i\right\rangle. \tag{2.10}$$

There also exist the so called *controlled* quantum gates, which are operators that act on two or more qubits, of which one or more act as control for some quantum operation. For instance, the controlled-NOT gate (or CNOT) over two qubits applies the NOT operator on the second qubit only when the first qubit is $\left|1\right\rangle$ and does nothing otherwise. The CNOT operator can be represented with the following matrix

$$CNOT = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}. \tag{2.11}$$

More generally, let $\boldsymbol{U}$ be a unitary that acts on a single qubit with matrix representation

$$\boldsymbol{U} = \begin{bmatrix}u_{00} & u_{01}\\ u_{10} & u_{11}\end{bmatrix}. \tag{2.12}$$

The controlled-$\boldsymbol{U}$ gate operates over two qubits, with the first one that acts as control. It maps the basis states $\left|00\right\rangle, \left|01\right\rangle, \left|10\right\rangle, \left|11\right\rangle$ in this way

$$\left|00\right\rangle \rightarrow \left|00\right\rangle$$

$$\left|01\right\rangle \rightarrow \left|01\right\rangle$$

$$\left|10\right\rangle \rightarrow \left|1\right\rangle \otimes \boldsymbol{U}\left|0\right\rangle = \left|1\right\rangle \otimes \left(u_{00}\left|0\right\rangle + u_{10}\left|1\right\rangle\right)$$

$$\left|11\right\rangle \rightarrow \left|1\right\rangle \otimes \boldsymbol{U}\left|1\right\rangle = \left|1\right\rangle \otimes \left(u_{01}\left|0\right\rangle + u_{11}\left|1\right\rangle\right).$$

So the CU-matrix is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix}.$$

Let us consider taking as $\boldsymbol{U}$ the Pauli gate $\boldsymbol{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. If we substitute this into the previous found CU-matrix, it results in the CNOT matrix operator reported above (indeed the CNOT operator is also called controlled Pauli-X operator[52]). The controlled operators are very useful and we refer to these many times in the various algorithms that we report, calling them *controlled rotations* since we apply an operator (that reflects in a *rotation* of the vector over which we apply the unitary) only if some conditions over the control qubits are met (*controlled*). As we are going to see, quantum algorithms can be described by a set of quantum logical gates, or unitary operators, and some measurements.

2.1.5. Measurements of quantum states

An important property to underline is that quantum states are measurable, and this derived from the third postulate of quantum mechanics.

Third postulate of quantum mechanics [33]

Measurements are described through a set $\{\boldsymbol{M}_m\}$ *of measurements operator, where index* m *identifies one of the possible outcomes of the experiment. The probability of measuring the outcome* m *from the state* $|\boldsymbol{\psi}\rangle$ *is defined as* $p(m) = \langle\boldsymbol{\psi}| \boldsymbol{M}_m^\dagger \boldsymbol{M}_m |\boldsymbol{\psi}\rangle$ *(with the completeness equation* $\sum_m \boldsymbol{M}_m^\dagger \boldsymbol{M}_m = \mathbb{I}$*, which ensures that* $\sum_m p(m) = 1$*. The state after the measurements is altered, as a matter of fact, if the measurements yields outcome* m*, the quantum states after the measurements becomes* $|\boldsymbol{\psi}'\rangle = \frac{\boldsymbol{M}_m|\boldsymbol{\psi}\rangle}{\sqrt{\langle\boldsymbol{\psi}|\boldsymbol{M}_m^\dagger \boldsymbol{M}_m|\boldsymbol{\psi}\rangle}}$.

The first thing to do when we want to measure a quantum register is to choose an orthogonal basis for the Hilbert space. Then, we can measure the quantum register in this base. Once measured, the quantum register is not any more in superposition and it collapses in one of the quantum states of the chosen basis. Each of these possible states can be measured with probability equal to its corresponding squared amplitude.
As a proof-example of this concept, we take the quantum state $|\boldsymbol{\psi}\rangle = \alpha|0\rangle + \beta|1\rangle$. Suppose we want to measure the state with respect to the orthonormal basis $\{|0\rangle, |1\rangle\}$.

Considering that

$$\boldsymbol{M}_0^\dagger \boldsymbol{M}_0 = |0\rangle \langle 0|0\rangle \langle 0| = |0\rangle \langle 0| = \boldsymbol{M}_0$$
$$\boldsymbol{M}_1^\dagger \boldsymbol{M}_1 = |1\rangle \langle 1|1\rangle \langle 1| = |1\rangle \langle 1| = \boldsymbol{M}_1,$$

we can write:

$$P(|0\rangle \,|\, |\boldsymbol{\psi}\rangle) = \langle \boldsymbol{\psi}| \, \boldsymbol{M}_0^\dagger \boldsymbol{M}_0 \, |\boldsymbol{\psi}\rangle = \langle \boldsymbol{\psi}| \, \boldsymbol{M}_0 \, |\boldsymbol{\psi}\rangle = \langle \boldsymbol{\psi}| \, (|0\rangle \langle 0|) \, |\boldsymbol{\psi}\rangle = |\langle 0|\boldsymbol{\psi}\rangle|^2 = |\alpha|^2,$$

since

$$\langle 0|\boldsymbol{\psi}\rangle = \langle 0| \, (\alpha \, |0\rangle + \beta \, |1\rangle) = \alpha \langle 0|0\rangle + \beta \langle 0|1\rangle = \alpha.$$

This means that the probability to measure $|0\rangle$, given the state $|\boldsymbol{\psi}\rangle$ is $|\alpha|^2$. Therefore, in this example, we can see that the amplitudes are linked to the probabilities of observing 0 or 1 when measuring the qubit, since

$$P(0) = |\alpha|^2, \quad P(1) = |\beta|^2.$$

More generally, if we have a quantum state $|\boldsymbol{\psi}\rangle$ and we want to perform the measurements with respect to $|\boldsymbol{\phi}\rangle$ (state of the orthogonal basis), the probability of measuring $|\boldsymbol{\phi}\rangle$ is:

$$p(\boldsymbol{\phi}) = \langle \boldsymbol{\psi}|\boldsymbol{\phi}\rangle \langle \boldsymbol{\phi}|\boldsymbol{\phi}\rangle \langle \boldsymbol{\phi}|\boldsymbol{\psi}\rangle = \langle \boldsymbol{\psi}|\boldsymbol{\phi}\rangle \langle \boldsymbol{\phi}|\boldsymbol{\psi}\rangle = |\langle \boldsymbol{\phi}|\boldsymbol{\psi}\rangle|^2$$

So for example, taking the quantum state $|\boldsymbol{\psi}\rangle = \frac{1}{\sqrt{n}} \sum_i^n |i\rangle$, we can retrieve each state $|j\rangle, j \in [n]$, with uniform probability $\frac{1}{n}$:

$$p(j) = |\langle j|\boldsymbol{\psi}\rangle|^2 = \left| \langle j| \frac{1}{\sqrt{n}} \sum_i^n |i\rangle \right|^2 = \left| \frac{1}{\sqrt{n}} \langle j|j\rangle + \frac{1}{\sqrt{n}} \sum_i^n \langle j|i\rangle \right|^2 = \left| \frac{1}{\sqrt{n}} \right|^2 = \frac{1}{n}.$$

In the classical setting, one can easily verify if a bit is either 0 or 1, but in the quantum case, since a qubit can be in a superposition of orthogonal states of the complex vector space, one cannot immediately retrieve the information about the amplitudes of the states for a particular basis.
Furthermore, there is a fundamental theorem in quantum computing, called *no-cloning* theorem, which states that if we have only a copy of a quantum state it is impossible to reconstruct it because, as said before, when we perform a measurement on a quantum state, it collapses to one of the states of the measurement basis.

Theorem 2.1. *(No Cloning [33]) There is no quantum algorithm that can create a copy of a general quantum state.*

So we need a large number of copies of the quantum state that has to be measured. As we are going to show, there are some routines, like the *pure state tomography*, that help in retrieving information from quantum states (in Section 2.2.2, we report an algorithm that helps in this function).

2.2. Quantum tools for machine learning

Now we introduce the concepts of quantum algorithms complexity, data representation, loading, and retrieving data, which are arguments more related to quantum machine learning. Then, we describe the main quantum routines that we implement and use in this work.

Complexity of quantum algorithms In the following theorems and algorithms, we use the notation $T(\boldsymbol{U})$, that indicates the time complexity needed to implement the unitary operator $\boldsymbol{U}$ measured in terms of depth of the circuit. Furthermore, in the time complexity formula we use the notation $\widetilde{O}$ to hide polylogarithmic factors in the standard O notation since those terms do not impact too much in the runtime complexity. Therefore, $\widetilde{O}(1) = polylog(n)$.

2.2.1. Quantum data representation

Previously, we have seen how we can represent a vector in a complex Hilbert space of dimension 2^n. Now, we explain how we can represent data in quantum computers. There are basically two ways of representing data: the *amplitude* and the *binary* encodings. In the first one, we store our data in the amplitudes of a quantum state, therefore we can encode n real values using $O(\log(n))$ qubits. In the binary encoding instead, we store a bit in the state of each qubit. Considering the latter, we can say that if we have a scalar $x \in \mathbb{N}$ and we consider its binary expansion, that for example can be $1010 \cdots 11$, encoded in n qubits, then we can write

$$|x\rangle = \bigotimes_{i=0}^{n} |x_i\rangle \,. \tag{2.13}$$

We can also use one more qubit for the sign. For what concern vectors and matrices, we have already seen that, taking a vector $\boldsymbol{x} \in \mathbb{R}^{2^n}$, we can write it as

$$|\boldsymbol{x}\rangle = \frac{1}{\|\boldsymbol{x}\|} \sum_{i=0}^{2^n-1} x_i \, |i\rangle = \begin{bmatrix} \frac{x_0}{\|\boldsymbol{x}\|_2} \\ \vdots \\ \frac{x_{2^n-1}}{\|\boldsymbol{x}\|_2} \end{bmatrix}$$

using the *amplitude encoding*. It is important to underline that to represent a vector of dimension $N = 2^n$, we just need $\log(N) = n$ qubits: we encode each value of the classical vector in the amplitudes of a state. To highlight this concept, consider a general vector state

$$|\psi\rangle = \frac{1}{\sqrt{4}}|0\rangle + \frac{1}{\sqrt{4}}|1\rangle + \frac{1}{\sqrt{4}}|2\rangle + \frac{1}{\sqrt{4}}|3\rangle$$

with uniform probability $\frac{1}{4}$ to measure a specific state. To represent this vector of length $N = 4$, we just need $n = 2$ qubits. Indeed,

$$|\psi\rangle = \frac{1}{\sqrt{4}} \sum_{i=0}^{2^2-1} |i\rangle .$$

The same applies for matrix representation. Let $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ be a data matrix of dimension $n \times d$. We can use amplitude encoding and write it as

$$\frac{1}{\sqrt{\sum_{i=0}^{n} \|x(i)\|^2}} \sum_{i=0}^{n} \|x(i)\| \, |i\rangle \, |x(i)\rangle$$

or equally

$$\frac{1}{\sqrt{\sum_{i,j=0}^{n,d} \|X_{ij}\|^2}} \sum_{i,j=0}^{n,d} X_{ij} \, |i\rangle \, |j\rangle .$$

In this case we just need $O(\log(nd))$ qubits to store a matrix of dimension $n \times d$.

QRAM model A concept that now it is interesting to mention is the one related to the *QRAM* model. QRAM recalls the random access memory (RAM) in the classical computer and, indeed, it addresses the data in memory using a tree structure. It is like a classical data structure that stores classical information but that is able to answer queries in quantum superposition. It is a data structure that allows to efficiently encode classical data structure in quantum states, using KP-tree structure [23]. In this way, we can store a dataset with a space complexity that scales linearly in the dimension and number of data points (indeed the KP-tree structure has a linear number of leaves). Moreover, we can perform update, insert, and delete operations in time that is poly-logarithmic in the dimension and number of data points (indeed the tree structure has logarithmic depth).

Definition 1. *(QRAM model, Definition 3.4 in [30])*

A QRAM is a device that stores indexed data (i, x_i) for $i \in [n]$ and $x_i \in \mathbb{R}$. It allows query in the form $|i\rangle |0\rangle \to |i\rangle |x_i\rangle$ and has circuit depth $O(ploylog(n))$.

Theorem 2.2 (Theorem 5.1 in [23]). *Let $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ be a matrix. Entries (i, j, X_{ij}) arrive in the system in some arbitrary order, and w denotes the number of entries that have already arrived in the system. There exists a data structure to store the matrix $\boldsymbol{X}$ with the following properties:*

- *The size of the data structure is $O(w \log^2(nd))$.*
- *The time to store a new entry (i, j, X_{ij}) is $O(\log^2(nd))$.*
- *A quantum algorithm that has quantum access to the data structure can perform the mapping $\widetilde{\boldsymbol{U}} : |i\rangle |0\rangle \to |i\rangle |\boldsymbol{X}_i\rangle$, for $i \in [n]$, corresponding to the rows of the matrix currently stored in memory and the mapping $\widetilde{\boldsymbol{V}} : |0\rangle |j\rangle \to |\widetilde{\boldsymbol{X}}\rangle |j\rangle$, for $j \in [d]$, where $\widetilde{\boldsymbol{X}} \in \mathbb{R}^n$ has entries $\widetilde{\boldsymbol{X}}_i = ||\boldsymbol{X}_i||$ in time poly $\log(nd)$*

In this work, we assume that matrices are stored in QRAM.

The concept we have not discussed yet is how to retrieve the data from a quantum computer. As mentioned in Section 2.1.5, quantum pure state tomography helps in this problem. In the next section, we describe this quantum task.

2.2.2. Quantum state tomography

Quantum pure state tomography is a useful quantum task that enables reconstructing classically the information that a quantum state brings with it through measurements. Clearly, the more measures we make, the closer we get to the true state and, also for this reason, the tomography is considered a bottleneck of quantum algorithms.
Given a vector $\boldsymbol{x} \in \mathbb{R}^{2^n}$, we have seen that we can write it as $|\boldsymbol{x}\rangle = \frac{1}{\|\boldsymbol{x}\|} \sum_{i=0}^{2^n-1} x_i |i\rangle$, using the amplitude encoding. When we measure this state, it results in one of the $|i\rangle$ registers, depending on the specific probability. Therefore, performing a large number of measurements allows us to forecast the probability distribution of the $|i\rangle$ registers and so to forecast also the amplitudes $x_0^2, \ldots, x_{2^n-1}^2$. Of course, the number of measurements cannot be infinite and so, being able to perform only a large but finite number of measures, the final estimates bring with them some error. Depending on the guarantees on the error that we want, we need to perform different number of measurements. As states in Kerenidis and Prakash [24], we can have both ℓ_2-norm and ℓ_∞-norm guarantees for the tomography. This means that, given a vector $\mathbf{x} \in \mathbb{R}^d$, we can obtain an estimate $\overline{\mathbf{x}}$ such that $\|\mathbf{x} - \overline{\mathbf{x}}\|_2 \leq \delta$ with a probability at least $1 - \frac{1}{poly(d)}$, or equivalently $\|\mathbf{x} - \overline{\mathbf{x}}\|_\infty \leq \delta$ for the ℓ_∞-norm defined as $\|\boldsymbol{x}\|_\infty = \max_i(\|\boldsymbol{x}_i\|)$.

Theorem 2.3 (ℓ_2 Vector state tomography, Theorem 4.2 in [28]). *Let $\boldsymbol{U}$ be a unitary*

operator that creates copies of the vector state $|\boldsymbol{x}\rangle$ *for a unit vector* $\boldsymbol{x} \in \mathbb{R}^d$ *in time* $T(\boldsymbol{U})$*. We also assume that we can apply the controlled version of* $\boldsymbol{U}$ *in the same time. The tomography algorithm outputs a unit vector* $\overline{\boldsymbol{x}}$ *such that* $\|\boldsymbol{x}-\overline{\boldsymbol{x}}\|_2 \leq \delta$ *in time* $O(T(\boldsymbol{U})\frac{d\log d}{\delta^2})$ *with probability at least* $1-\frac{1}{poly(d)}$*.*

We also report the statement for the ℓ_∞-tomography.

Theorem 2.4 (ℓ_∞ Vector state tomography, Theorem 4.3 in [28])**.** *Let* $\boldsymbol{U}$ *be a unitary operator that creates copies of the vector state* $|\boldsymbol{x}\rangle$ *for a unit vector* $\boldsymbol{x} \in \mathbb{R}^d$ *in time* $T(\boldsymbol{U})$*. We also assume that we can apply the controlled version of* U *in the same time. The tomography algorithm outputs a unit vector* $\hat{\boldsymbol{x}}$ *such that* $\|\boldsymbol{x}-\overline{\boldsymbol{x}}\|_\infty \leq \delta$ *in time* $O(T(\boldsymbol{U})\frac{\log d}{\delta^2})$ *with probability at least* $1-\frac{1}{poly(d)}$*.*

Algorithm 2.1 Vector state tomography [24].

Require: Access to unitary $\boldsymbol{U}$ such that $\boldsymbol{U}|0\rangle = |\boldsymbol{x}\rangle = \sum_{i\in[d]} x_i |i\rangle$ and to its controlled version.

1. **Vector values estimation**
   (a) Measure $N = \frac{36d\log(d)}{\delta^2}$ copies of $|\boldsymbol{x}\rangle$ in the standard basis and obtain estimates $p_i = \frac{n_i}{N}$ where n_i is the number of times outcome i is observed.
   (b) Store $\sqrt{p_i}, i \in [d]$ in QRAM data structure so that $|p\rangle = \sum_{i\in[d]} \sqrt{p_i}|i\rangle$ can be prepared efficiently.
2. **Sign estimation**
   (a) Create $N = \frac{36d\log(d)}{\delta^2}$ copies of the state $\frac{1}{\sqrt{2}}|0\rangle\sum_{i\in[d]} x_i|i\rangle + \frac{1}{\sqrt{2}}|1\rangle\sum_{i\in[d]}\sqrt{p_i}|i\rangle$ using a control qubit.
   (b) Apply Hadamard gate on the first qubit of each copy of the state to obtain $\frac{1}{2}\sum_{i\in[d]}[(x_i+\sqrt{p_i})|0,i\rangle + (x_i-\sqrt{p_i})|1,i\rangle]$.
   (c) Measure each copy in the standard basis and maintain counts $n(b,i)$ of the number of times outcome $|b,i\rangle$ is observed for $b \in 0,1$.
   (d) Set $\sigma_i = 1$ if $n(0,i) > 0.4p_iN$ and -1 otherwise.
3. Output the unit vector $\overline{x}$, with $\overline{x_i} = \sigma_i\sqrt{p_i}$.

In Algorithm 2.1, we report the procedure presented in Kerenidis and Prakash [24] that we use to implement vector state tomography. It describes the tomography of pure states with real amplitudes (complex amplitudes could be tackled in a similar way). Let $\boldsymbol{U}$ be a unitary operator that creates copies of the vector state $|\boldsymbol{x}\rangle$ for a unit vector $\boldsymbol{x} \in \mathbb{R}^d$ in time $T_{\boldsymbol{U}}$. The tomography algorithm outputs a unit vector $\overline{\boldsymbol{x}}$ such that $\|\overline{\boldsymbol{x}} - \boldsymbol{x}\|_2 < \sqrt{7}\delta$ with probability $1 - 1/poly(d)$, as reported in Section 4 by Kerenidis and Prakash [24]. The algorithm runs in time $O(T_{\boldsymbol{U}}N)$, with $N = \widetilde{O}\left(\frac{d}{\delta^2}\right)$. Basically, it takes N

copies of the initial quantum state $|\boldsymbol{x}\rangle = \sum_{i\in[d]} x_i |i\rangle$. Once we have measured the N copies, we are able to build the state $|p\rangle = \sum_{i\in[d]} \sqrt{p_i} |i\rangle$, with $p_i = \frac{n_i}{N}$ where n_i is the number of times outcome i is observed. Then, by applying an Hadamard gate on the first qubit of the state $\frac{1}{\sqrt{2}} |0\rangle \sum_{i\in[d]} x_i |i\rangle + \frac{1}{\sqrt{2}} |1\rangle \sum_{i\in[d]} \sqrt{p_i} |i\rangle$, we obtain the state $\frac{1}{2} \sum_{i\in[d]} [(x_i + \sqrt{p_i}) |0, i\rangle + (x_i - \sqrt{p_i}) |1, i\rangle]$. By measuring N copies of that state, we obtain the number of times outcome $|b, i\rangle$ is observed (with $b \in 0, 1$). Based on this number, we set the sign of each estimated value $\overline{x}_i$ using the logic reported in step d. Finally, we return the estimated vector $\overline{\boldsymbol{x}}$.

2.2.3. Phase estimation

Quantum phase estimation has the goal to output an estimation of the phase of the eigenvectors of a unitary operator. It is based on the Quantum Fourier Transform (QFT), which is a basic routine also used in other quantum algorithms, like *amplitude estimation* (which we explain in Section 2.2.5). Let us suppose that a unitary $\boldsymbol{U}$ has an eigenvector $|\boldsymbol{\psi}\rangle$ associated to the eigenvalue $e^{2\pi i\omega}$. With phase estimation, we want to estimate the unknown phase ω. More formally, given a unitary matrix $\boldsymbol{U}$ and a quantum state $|\boldsymbol{\psi}\rangle$ such that $\boldsymbol{U} |\boldsymbol{\psi}\rangle = e^{2\pi i\omega} |\boldsymbol{\psi}\rangle$, phase estimation estimates ω with high probability, to additive error ϵ.

In Nielsen and Chuang [33] and Cleve et al. [13], there is a long discussion which explains this algorithm and here we report the main points. Quantum phase estimation routine uses two registers: the first store n qubits initially in the state $|0\rangle$ and the second starts in the state $|\boldsymbol{\psi}\rangle$. It is important to notice that the number of qubits n depends both on the ϵ accuracy we want to have in the estimate of ω and on the $1-\gamma$ probability of success of the routine we wish. Indeed, as described by Nielsen and Chuang [33], we can compute $n = t + m$, where $t = \left\lceil log_2 \left(\frac{1}{\epsilon}\right) \right\rceil$ and $m = \left\lceil log_2 \left(2 + \frac{1}{2\gamma}\right) \right\rceil$, that results in

$$n = \left\lceil \log_2 \left(\frac{1}{\epsilon}\right) \right\rceil + \left\lceil \log_2 \left(2 + \frac{1}{2\gamma}\right) \right\rceil . \tag{2.14}$$

So we can increase or decrease the number of qubits in the algorithm, increasing or decreasing ϵ and γ parameters. More precisely, fixed a value of γ, if we want more accurate estimates we need to increment the accuracy, which means decreasing ϵ value because we want estimates with a lower error. This results in an increasing of the $\log_2 \left(\frac{1}{\epsilon}\right)$ value and, consequently, in an increasing of n. The same reasoning applies for the probability of success $1-\gamma$: fixed an accuracy value ϵ, we can increase the probability of success $1-\gamma$, which means decreasing the value of γ that, as for ϵ, results in an increasing of the number of qubits n. Going deeper in the procedure, phase estimation requires two stages. Firstly,

a n-bit Hadamard gate is applied to the first register, obtaining the state

$$\frac{1}{2^{n/2}}(|0\rangle + |1\rangle)^{\otimes n} |\boldsymbol{\psi}\rangle ,$$

followed by a controlled-$\boldsymbol{U}$ operations on the second register, with $\boldsymbol{U}$ raised to successive powers of two, that apply unitary operator $\boldsymbol{U}$ on the target register only if its corresponding control bit is $|1\rangle$. Being $|\boldsymbol{\psi}\rangle$ the eigenvector of $\boldsymbol{U}$, such that $\boldsymbol{U} |\boldsymbol{\psi}\rangle = e^{2\pi i\omega} |\boldsymbol{\psi}\rangle$, we can formalize the application of the controlled-$\boldsymbol{U}$ operations, with $\boldsymbol{U}$ raised to successive power of two, in this way:

$$\boldsymbol{U}^{2^j} |\boldsymbol{\psi}\rangle = \boldsymbol{U}^{2^j-1}\boldsymbol{U} |\boldsymbol{\psi}\rangle = \boldsymbol{U}^{2^j-1} e^{2\pi i\omega} |\boldsymbol{\psi}\rangle = \cdots = e^{2\pi i 2^j\omega} |\boldsymbol{\psi}\rangle .$$

Applying all the controlled-$\boldsymbol{U}^{2^j}$ operations, with $0 \leq j \leq n-1$, and taking into account the equality

$$|0\rangle \otimes |\boldsymbol{\psi}\rangle + |1\rangle \otimes e^{2\pi i\omega} |\boldsymbol{\psi}\rangle = (|0\rangle + e^{2\pi i\omega} |1\rangle) \otimes |\boldsymbol{\psi}\rangle$$

at the end of the first stage, the state of the first register can be seen in this way:

$$\frac{1}{2^{n/2}}(|0\rangle + e^{2\pi i 2^{n-1}\omega} |1\rangle) \otimes (|0\rangle + e^{2\pi i 2^{n-2}\omega} |1\rangle) \otimes \cdots \otimes (|0\rangle + e^{2\pi i 2^0\omega} |1\rangle) \otimes |\boldsymbol{\psi}\rangle =$$

$$= \frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{2\pi i\omega k} |k\rangle \otimes |\boldsymbol{\psi}\rangle \tag{2.15}$$

During the first stage, the second register does not change and remains always in the state $|\boldsymbol{\psi}\rangle$.

Then the inverse Quantum Fourier Transform is applied to the first register, reversing the circuit of the QFT at the cost of $\Theta(n^2)$. This allows to translate the state from the Fourier to the computational basis: $\frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{2\pi i\omega k} |k\rangle \otimes |\boldsymbol{\psi}\rangle \rightarrow |\overline{\omega}\rangle |\boldsymbol{\psi}\rangle$, where $\overline{\omega}$ is the estimate of the phase. More formally:

$$\frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{2\pi i\omega k} |k\rangle \otimes |\boldsymbol{\psi}\rangle \xrightarrow{QFT^{-1}} \frac{1}{2^n} \sum_{x=0}^{2^n-1} \sum_{k=0}^{2^n-1} e^{-\frac{2\pi ik}{2^n}(x-2^n\omega)} |x\rangle \otimes |\boldsymbol{\psi}\rangle .$$

Finally, given this new quantum state, we perform a measurement on the computational basis to read the state of the first register that contains the estimate $\overline{\omega}$ of the true phase. Indeed, QFT^{-1} returns an integer x, encoded in binary by an n-qubit state, with a certain probability and the output x corresponds to the estimate $\overline{\omega} = \frac{x}{2^n}$. How likely it is that a value x is returned as output, is explained in the following theorem.

Theorem 2.5. *(Lemma 7.1.2 in [22]) Let $\omega = \frac{x}{2^n} = 0.x_1x_2\ldots x_n$ be some fixed number.*

The phase estimation algorithm applied to the input state $|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} e^{2\pi i \omega x} |x\rangle$ *outputs the integer* x *with probability:*

$$p(x) = \frac{1}{2^{2n}} \frac{\sin^2(\pi(2^n\omega - x))}{\sin^2(\pi(\omega - \frac{x}{2^n}))} \tag{2.16}$$

Following what the previous theorem states, for each possible estimate value x, the probability to output an estimate of ω is given by Equation 2.16. Basically, this computation assigns a higher probability to x, the more $\frac{x}{2^n}$ is closer to the real value ω. We see better this concept in Section 3.3, where we show a numerical simulation of phase estimation so that one can better understand what is going on in this procedure.
In case $2^n\omega$ is an integer, measuring in the computational basis gives the phase in the auxiliary register with high probability:

$$|2^t\omega\rangle \otimes |\psi\rangle .$$

If $2^n\omega$ is not an integer, that happens when ω is not a fraction of a power of two, the phase estimation will output an estimate with a probability better than $\frac{4}{\pi^2}$. In the following two theorems we report more formally these cases.

Theorem 2.6. *(Theorem 7.1.4 in [22]) Let* $\overline{\omega} = \frac{\overline{x}}{2^n}$ *be an integer multiple of* $\frac{1}{2^n}$ *closest to* ω*. The phase estimation algorithm returns* $\overline{x}$ *with probability at least* $\frac{4}{\pi^2}$

Basically, Theorems 2.6 states that with probability at least $\frac{4}{\pi^2}$, the phase estimation algorithm outputs an estimate $\overline{x}$ such that $\left|\frac{\overline{x}}{2^n} - \omega\right| \leq \frac{1}{2^{n+1}}$. But if ω lies exactly in between $\frac{x}{2^n}$ and $\frac{x+1}{2^n}$, then phase estimation outputs one of the two closest value of ω with probability at least $\frac{8}{\pi^2}$.

Theorem 2.7. *(Theorem 7.1.5 in [22]) If* $\frac{x}{2^n} \leq \omega \leq \frac{x+1}{2^n}$*, then the phase estimation algorithm returns one of* x *or* $x+1$ *with probability at least of* $\frac{8}{\pi^2}$*.*

Therefore, with probability at least $\frac{8}{\pi^2}$, the phase estimation will return $\overline{x}$ such that $\left|\frac{\overline{x}}{2^n} - \omega\right| \leq \frac{1}{2^n}$.

We summarize phase estimation with the following theorem.

Theorem 2.8 (Phase estimation, Theorem 4.1 in [30])**.** *Let* $\boldsymbol{U}$ *be a unitary operator with eigenvectors* $|\boldsymbol{v}_j\rangle$ *and eigenvalues* $e^{i\omega_j}, \omega_j \in [-\pi, \pi]$*, i.e we have* $\boldsymbol{U}|\boldsymbol{v}_j\rangle = e^{i\omega_j}|\boldsymbol{v}_j\rangle$ *for* $j \in [n]$*. For a precision parameter* $\epsilon > 0$*, there exist a quantum algorithm that runs in* $O(\frac{T(\boldsymbol{U})log(n)}{\epsilon})$ *and with probability* $1 - \frac{1}{poly(n)}$ *maps a state* $\sum_{j\in[n]} \alpha_j |\boldsymbol{v}_j\rangle$ *to the state*

$\sum_{j\in[n]} \alpha_j |\boldsymbol{v}_j\rangle |\overline{\omega}_j\rangle$ *such that* $|\overline{\omega}_j - \omega_j| < \epsilon$ *for all* $j \in [n]$.

To illustrate and explain better how the algorithm works at high level, we make use of a circumference, by representing each phase value on this circle. In Figure 2.2, we make the assumption that $n = 3$, so we get $2^3 = 8$ possible phase values. As we will see, we are going to approximate the phase with one of these 2^n values. Here is important to remark that each possible estimate is $\frac{1}{2^n}$ distant from the others. Indeed, each point in the circle is distant $\pm\frac{1}{2^n}$ from its neighbors. The larger is n, the more numerous and closer to each other the points in the circumference are. Therefore, with a large n, also the estimates would be more accurate since it is very likely that there is an estimate value very close to ω. This is because we have seen in Theorem 2.5 that we can express the phase in binary notation as $\omega = \frac{x}{2^n} = 0.x_1x_2\ldots x_n$, that means $x1 \cdot 2^{-1} + x_2 \cdot 2^{-2} + \cdots + x_n \cdot 2^{-n}$ where $x_1, \ldots, x_n \in 0, 1$. For example, let us consider $n = 1$ qubit. In this case, w can be expressed as $0.x_1$. If we increase the number of qubit to $n = 3$ as in our example, we can express ω as $0.x_1x_2x_3$. Therefore, if we increase the number of qubit n we are able to discretize better the phase value.

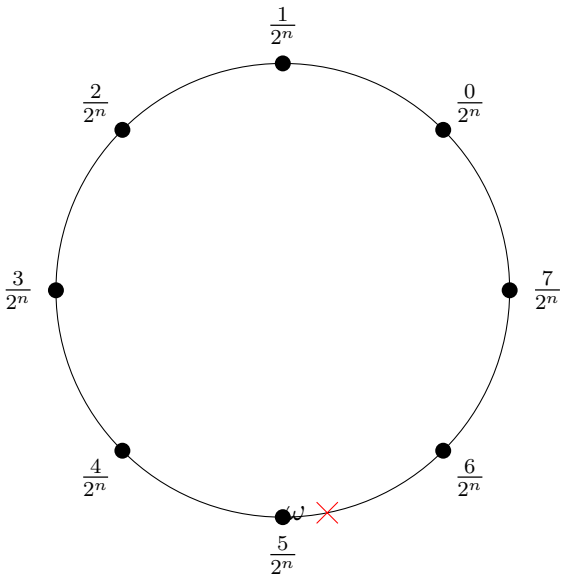


Figure 2.2: Representation of phase ω as a red cross in a circle where the black dots are the possible outputs of phase estimation procedure.

A constraint that is important to underline is that $\omega \in [0, 1)$. Indeed, also in the circumference, the points on the circle have values that go from 0 to $\frac{2^n-1}{2^n}$ (so potentially ω=1 would be exactly in the same position of $\frac{0}{2^n}$, and this is one of the main reason why 1 is not included in the interval, otherwise our estimate would be exactly 0). The fact

that ω cannot be 1 is also because by expressing ω in binary notation exactly in n bits, as $\omega = \frac{x}{2^n} = 0.x_1x_2\dots x_n$, we can never have $\omega = 1$. As we can see in Figure 2.2, the phase ω that we want to estimate is represented with a red cross and, since a real phase parameter may not be an integer of $\frac{1}{2^n}$, it can lie between two nodes representing the possible phases.
Therefore, phase estimation approximates ω, with one of those points in the circumference: the more a point is near to ω, the more likely it is to be output as best estimate. The probability that phase ω will be approximated with a point x in the circle is reported in Equation 2.16. Basically, we calculate the distance between ω and x and, the closer they are, the higher is the probability associated to that specific x. If $\omega = 0$, phase estimation returns 0 with certainty.

We summarize the procedure in Algorithm 2.2.

Algorithm 2.2 Phase estimation.

Input: $\omega \in [0, 1), \epsilon, \gamma > 0$.
Output: estimate of the phase $\overline{\omega}$.

1: Compute $n = \left\lceil \log_2(\frac{1}{\epsilon}) \right\rceil + \left\lceil \log_2(2 + \frac{1}{2\gamma}) \right\rceil$.
2: **for** x in range(2^n) **do**
3: $\quad p(x) = \frac{1}{2^{2n}} \frac{\sin^2(\pi(2^n\omega - x))}{\sin^2(\pi(\omega - \frac{x}{2^n}))}$.
4: **end for**
5: Sample $\overline{x} \sim p$.
6: Return $\overline{\omega} = \frac{\overline{x}}{2^n}$.

2.2.4. Consistent phase estimation

In our framework, we also study and implement a variant of the quantum phase estimation, called *consistent phase estimation* [46]. We have seen that phase estimation can output an estimate with a certain probability, given by the formula in Equation 2.16. This means that, across multiple executions, the output of phase estimation can change, since each possible value has a probability to be extracted as output. In this extended routine, instead, the estimates across multiple runs are *consistent* in the true sense of the world, meaning they are *almost always* the same. This means that the error between different executions of the phase estimation is almost deterministic.
But why *almost always*? Now we try to clarify this sentence, going into more theoretical details.
To explain better this concept, we introduce the probabilistic machine with deterministic outputs. This is because making these machines deterministic is equivalent to make phase

estimation consistent. Suppose that there exists a value $u = u(x) \in \mathbb{R}$, that is determined by an input $x \in \{0,1\}^n$, and there exists a probabilistic Turing machine M defined as $M(x,y)$, where y is a fixed random input that makes the machine probabilistic. M takes x and y as input and outputs an estimate, such that:

$$\forall x \in \{0,1\}^n, \mathbb{P}_y[|M(x,y) - u(x)| \leq \epsilon] \geq 1 - \gamma$$

where ϵ and γ are the accuracy of the estimate and the probability of failure respectively. It means that, given the input value x, the probability that the random input value y causes M to approximate $u(x)$ with with an accuracy at least of ϵ, is at least $1 - \gamma$. But being a probabilistic machine, changing the randomness value y, it may return different results. To ensure that the output of M depends only by the input x alone and not by the randomness value y of the probabilistic machine, a *shift and truncate* method is applied [38, 46], which is formalized in the following theorem and proof.

Theorem 2.9 (from Lemma 2.1 in [46]). *Suppose M(x,y), with $x \in \{0,1\}^n$ and random input y, approximates u(x) with with ϵ accuracy and γ error using $O(log(\frac{n}{\epsilon\gamma}))$ space. Let $\zeta > 0$ be an arbitrary constant. Then there exist another probabilistic algorithm $M'(x,y;s)$, which takes x, y as input and outputs the shift s, with $|y| = O(log(\frac{n}{\epsilon\gamma\zeta}))$ and $|s| = O(log(1/\zeta))$ and there exist a function $u'(x,s)$ such that:*

- *For all s: $|u'(x,s) - u(x)| \leq \epsilon$;*
- *$Pr_s[Pr_y(M'(x,y;s) = u'(x,s)) < 1 - \gamma] \leq \zeta$*

Proof (from Ta-Shma [46]).
Set $\epsilon' = \frac{\epsilon\zeta}{2}$ and let $L = \frac{2}{\zeta} = \frac{\epsilon}{\epsilon'}$. Pick the shift s randomly from $\{1, \ldots, L\}$ and fix it. Divide $\mathbb{R}$ to the consecutive sections $s\epsilon' \pm [k\epsilon, (k+1)\epsilon]$ for $k \in \mathbb{Z}$ (e.g. $[s\epsilon' - \epsilon, s\epsilon')$ and $[s\epsilon', s\epsilon' + \epsilon)$ are sections). Then run M with ϵ' accuracy and return the section to which $M(x,y)$ belongs. For correctness, say a shift s is *good* for u if u is ϵ' away from a boundary, i.e, there exists a section $[c,d)$ such that $u \in [c + \epsilon', d - \epsilon')$. As $L\epsilon' < \epsilon$, for every u there exist at most 2 shifts that bring it ϵ' close to a boundary, and therefore the probability a shift is not good is at most $\frac{2}{L} \leq \zeta$. For a good shift, the probability over the random coins of M that the output is the section to which u belongs is at least $1 - \gamma$ (because u is ϵ' away from a boundary), the section number depends only on u and the shift s, and any point within the section (in particular, say, its middle point) is ϵ close to u. □

Basically, the first four lines of the previous proof describe the main steps of consistent phase estimation algorithm, considering that M plays the role of phase estimation. To get a better understanding of this routine, we report in Figure 2.3 a high-level representation

of consistent phase estimation algorithm. In this way, the reader can better comprehend the link between the previous proof and this quantum routine. First of all, a little bit of context: the upper dotted line is a modified version of Figure 2.2, representing the unrolled circle of phases, where each dot is a possible estimate. Therefore, the upper line represents the phase estimation procedure. Instead, in the bottom ticked line, we show the n sections (corresponding to the sections $s\epsilon' \pm [k\epsilon, (k+1)\epsilon]$ for $k \in \mathbb{Z}$ reported in the proof), each of dimension ϵ as reported in the proof, in which we map the output of phase estimation. Suppose we want to estimate ω, represented graphically with the red cross. We execute phase estimation with accuracy $\epsilon' \ll \epsilon$, as required by the algorithm, and the values close to ω are the ones with the highest probability of being extracted as output. In this case, the cyan and the blue dots represent the most likely values (the rightmost dot is filled with a darker color and it is bigger than the left one since the red cross is nearer to this value). For how phase estimation works, we are pretty sure that both the cyan and the blue estimates are ϵ'-close to the real value ω. These two values have a good chance to be extracted as output and, if we execute phase estimation more than once, it is probable that the output of different runs differs sometimes. But since $\epsilon' \ll \epsilon$, we map all those estimates, even if different, in the same section which we assume to be s_2 (this means that, considering $s_2 = [c, d)$, $\omega \in [c + \epsilon', d - \epsilon')$). We can see that, using s_2 section to extract the final *consistent* estimate of ω and considering the section's middle point as estimate [46], we are *almost always* sure that all the estimates are the same and, not least, even ϵ-close to ω, being ϵ the accuracy required at the beginning of the algorithm.

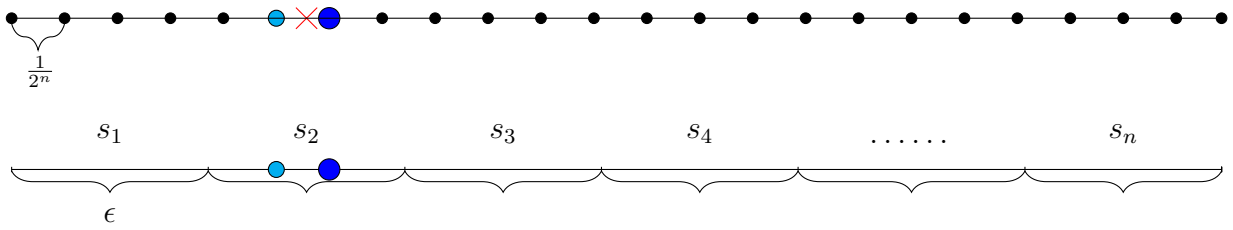


Figure 2.3: Consistent phase estimation procedure.

We said that consistent phase estimation is consistent almost always, but why *almost always*? Let us try to see what are the cases in which it failed to be consistent. Since the output depends on the input value and on the shift s (which, in the previous proof, we saw to be a random integer that determines the sections), surely one of the most important things to be consistent is to obtain a *good* shift because there might be the case in which the possible phase estimation's outputs lie ϵ'-close to a border between two consecutive sections, as the following example in Figure 2.4.
As we can see, the two more likely estimates of phase estimation (now in orange and

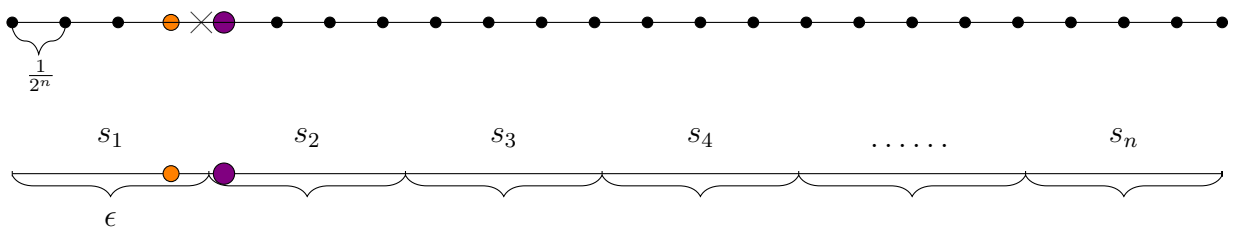


Figure 2.4: Case of "no" consistent phase estimation procedure.

violet) are always ϵ'-close to ω but, with this particular fixed shift (assumed to be the same of the one in Figure 2.3), they are mapped into two different sections s_1 and s_2, resulting in two different outputs of consistent phase estimation. Therefore, we can say that this particular shift is good for the input ω in Figure 2.3, while is not so good for the ω in Figure 2.4. In the proof of Theorem 2.9, it is established that a shift is not good with a probability of $n\zeta \leq \gamma$.

Another case in which consistent phase estimation can fail to be consistent is if the constraint $\epsilon' \ll \epsilon$ is not verified. Indeed, if this is the case, the sections are not so wide with respect to the closeness between $\overline{\omega}$ and ω, and so there are potentially more cases of estimates between two consecutive sections, as the case reported in Figure 2.4. This last case can happen in case of "failure" of phase estimation. This is because, potentially, phase estimation can output any values with a certain probability (clearly, the lower the probability, the less likely it is to be extracted, but there is still a little chance). Therefore, if phase estimation fails, so if it returns a value that is no ϵ'-close to the real one, the condition $\epsilon' \ll \epsilon$ can be violated.

We have shown at high level how consistent phase estimation works. Now, we conclude this section with a more formal explanation of this quantum routine.

The problem clearly is pretty similar to the phase estimation one. We want to map each eigenvector $\boldsymbol{v}_i$ to a *unique* vector $|\boldsymbol{v}_i\rangle \otimes |s(i)\rangle$, where $s(i)$ is the *section* relative to the eigenvalue λ_i, whose value depends on the input value and on a fixed shift (in the previous proof we define what a section is). It is important to keep in mind that consistent phase estimation is based on phase estimation. Therefore, it is as if phase estimation were a subroutine of this algorithm. So, fixed a shift s, an eigenvector $\boldsymbol{v}$, and the corresponding eigenvalue λ, after the phase estimation routine, we are in the state

$$|\boldsymbol{\psi}\rangle = |\boldsymbol{v}\rangle \otimes [\sum_j \beta_j |j\rangle \otimes |s(j)\rangle],$$

where j is the output of the phase estimation and $s(j)$ is the section to which j belongs. Considering s a good shift for the eigenvector $\boldsymbol{v}$, we can say that the corresponding

eigenvalue λ is ϵ'-close to the section boundary and so we can assert that $s(j) = s(\lambda) \forall j$, where $s(\lambda)$ is the section to which eigenvalue λ belongs [46]. We can say that, with a confidence of $1 - \gamma$, we have the ideal state

$$|\boldsymbol{\psi}'\rangle = |\boldsymbol{v}\rangle \otimes [\sum_j \beta_j |j\rangle] \otimes |s(\lambda)\rangle .$$

Finally, reversing the phase estimation process on $|\boldsymbol{\psi}'\rangle$ we can obtain the state

$$|\boldsymbol{v}\rangle \otimes |\hat{0}\rangle \otimes |s(\lambda)\rangle .$$

Summing up, consistent phase estimation maps the state

$$|\boldsymbol{v}\rangle \otimes |\hat{0}\rangle \otimes |\hat{0}\rangle \rightarrow |\boldsymbol{v}\rangle \otimes |\hat{0}\rangle \otimes |s(\lambda)\rangle ,$$

with $s(\lambda)$ which depends only on the input $\boldsymbol{v}$ and on the shift s and is the section from which we can extract the estimate $\overline{\lambda}$, that is ϵ-close to the real eigenvalue λ.

2.2.5. Amplitude estimation

Amplitude estimation relies on phase estimation. Indeed, as we are going to show, amplitude estimation implicitly executes phase estimation.
The *goal* of the amplitude estimation is to estimate the probability that a measurement over a quantum state yields a *good* state.
Let us suppose that we have an operator $\mathcal{A}$ such that

$$\mathcal{A} |0\rangle : \sqrt{1-p} |\psi_0\rangle + \sqrt{p} |\psi_1\rangle \tag{2.17}$$

where $\langle \psi_0 | \psi_1 \rangle = 0$, which means that the two states ψ_0 and ψ_1 are orthogonal. Amplitude estimation outputs an estimate for the amplitude p of the state $|\psi_1\rangle$:

$$p = \| \langle \psi_1 | \psi_1 \rangle \|^2. \tag{2.18}$$

Theorem 2.10 (Amplitude estimation, Theorem 12 in [9]). *Given a quantum algorithm*

$$\mathcal{A} : |0\rangle \rightarrow \sqrt{p} |y, 1\rangle + \sqrt{1-p} |G, 0\rangle$$

where $|G\rangle$ is some garbage state, then the amplitude estimation algorithm, using exactly P iteration of the algorithm $\mathcal{A}$, for any positive integer p, outputs $\overline{p}$ $(0 \leq \overline{p} \leq 1)$, such

that:

$$|p - \overline{p}| \leq 2\pi \frac{\sqrt{p(1-p)}}{P} + (\frac{\pi}{P})^2$$

with probability at least of $8/\pi^2$. *If p=0 then* $\overline{p} = 0$ *with certainty and if p=1, and P is even, then* $\overline{p} = 1$ *with certainty.*

We also state a simplified version from Theorem $II.2$ in Kerenidis and Prakash [25].

Theorem 2.11 (Amplitude amplification and Amplitude estimation [25])**.** *Taking an operator* $\boldsymbol{U}$ *such that* $\boldsymbol{U}|0\rangle^l = |\phi\rangle = \sin(\theta)|x,0\rangle + \cos(\theta)|G,0^\perp\rangle$, *then* $\sin^2(\theta)$ *can be estimated to multiplicative error* η *in time* $O(\frac{T(\boldsymbol{U})}{\eta \sin(\theta)})$ *and* $|x\rangle$ *can be generated in expected time* $\frac{T(\boldsymbol{U})}{\sin(\theta)}$, *where* $T(\boldsymbol{U})$ *is the time to implement* $\boldsymbol{U}$.

Algorithm 2.3, reported by Kaye et al. [22], describes the steps of quantum amplitude estimation. We can notice that the steps are almost the same of phase estimation. Indeed, here, we want to estimate $\overline{\theta}_p$ in order to estimate $\overline{p} = \sin^2(\overline{\theta}_p)$. The problem of estimating $\overline{\theta}_p = \frac{y\pi}{M}$ is the phase estimation problem, with integer $y \in [0, \ldots, M-1]$ and M number of iterations.
Let us suppose that at the beginning, we have $|\psi\rangle = \mathcal{A}|0\rangle$ (Equation 2.17), where $\mathcal{A}$ denotes the unitary operator applied to the initial zero state. After step 4 is executed, so after applying the operator $\boldsymbol{Q} = \boldsymbol{A}^{-1}\boldsymbol{U}_{0^\perp}\boldsymbol{A}\boldsymbol{U}_f$ j times to boost the probability of success p (see Equation 2.18), we get the state [9]:

$$\boldsymbol{Q}^j|\psi\rangle = \frac{1}{\sqrt{p}}\sin((2j+1)\theta_p)|\psi_1\rangle + \frac{1}{\sqrt{1-p}}\cos((2j+1)\theta_p)|\psi_0\rangle, 0 \leq \theta_p \leq \frac{\pi}{2}. \quad (2.19)$$

We see that the amplitudes of $|\psi_0\rangle$ and $|\psi_1\rangle$ are sinusoidal functions, both of period $\frac{\pi}{\theta_p}$. Recalling that $p = \sin^2(\theta_p)$, from Theorem 2.11, an estimation of θ_p gives an estimate of p.

Lemma 1. *(from Brassard et al. [9]) Let* $p = \sin^2(\theta_p)$, *and* $\overline{p} = \sin^2(\overline{\theta}_p)$, *with* $0 \leq \theta_p, \overline{\theta}_p \leq 2\pi$, *then*

$$\left|\theta_p - \overline{\theta}_p\right| \leq \epsilon \Rightarrow |p - \overline{p}| \leq 2\epsilon\sqrt{p(1-p)} + \epsilon^2$$

Therefore, the core part of amplitude estimation is taking the value p that we want to estimate, computing the value $\theta_p = \sin^{-1}(\sqrt{p})$, and applying phase estimation to estimate θ_p. In Brassard et al. [9], which is our main source of information for amplitude estimation, they state the following theorem to compute the probability to extract each of the M samples as an estimate of the true value θ_p. It is a procedure very similar to the one that we report in Theorem 2.5.

Theorem 2.12. *(Theorem 11 in [9]) Let X be a discrete random variable corresponding to the classical result of measuring $\boldsymbol{F}_M^{-1}|S_M(\omega)\rangle$ in the computational basis, where ω is the true phase, $\boldsymbol{F}_M^{-1}$ is the inverse Fourier transform and $|S_M(\omega)\rangle = \frac{1}{\sqrt{M}}\sum_{x=0}^{M-1} e^{2\pi i\omega x}|x\rangle$, with M integer ≥ 1. If $M\omega$ is an integer then Prob($X = M\omega$)=1. Otherwise, letting $\Delta = d(\omega, x/M)$,*

$$Prob(X = x) = \frac{\sin^2(M\Delta\pi)}{M^2\sin^2(\Delta\pi)}$$

The Δ parameter in the probability formula is the minimal wrap-around distance function.

Definition 2. *(Minimal wrap-around distance, Definition 9 in [9]) For any two real numbers $\omega_0, \omega_1 \in \mathbb{R}$, let $\Delta = d(\omega_0, \omega_1) = \min_{z\in\mathbb{Z}}\{|z + \omega_1 - \omega_0|\}$*

Once completed phase estimation, after step 7, amplitude estimation returns the output as $\overline{p} = \sin^2(\overline{\theta}_p)$.
An important fact to underline for amplitude estimation is that depending on the guarantee we have on the error ϵ, the running time will be different. If we have a relative error guarantee, so $|p - \overline{p}| \leq p\epsilon$, then the running time is the one specified in Theorem 2.11. Instead, if we have a guarantee such that $|p - \overline{p}| \leq \epsilon$, then the running time is $O\left(\frac{1}{\epsilon}\right)$ (consult Table 3.1 in [28] for major details).
Returning for a moment to the phase estimation algorithm, we have seen that we can increase the probability of success, which results in a higher number of qubits used, playing with the γ parameter that represents the probability of failure of the routine. Instead, if we want to boost the success and the accuracy of the estimates of the amplitude estimation (or any other quantum procedure), we can make use of the *median evaluation*, which we describe in the following section.

To better understand amplitude estimation, we can think to a problem that is a special case of this quantum algorithm application, that is the *quantum counting*.
Let us take a function $f(x)$ and a set X of n values, indexed by $\{0,\ldots, n-1\}$. Let us consider the problem of finding t of those n elements, for which $f(x) = 1$. So in this problem, we can make a clear distinction between the set of elements considered *good*, the ones that gives 1 as result, and the set of elements considered *bad*. We can define them for simplicity as X_{good} and X_{bad}, where

$$X_{good} \subseteq X, \quad X_{bad} \subseteq X, \quad X_{good} \cap X_{bad} = 0, \quad X_{good} \cup X_{bad} = X.$$

Considering Theorem 2.10, by choosing an operator $\mathcal{A}$ such that

$$\mathcal{A}\left|00\ldots0\right\rangle = \frac{1}{\sqrt{n}}\sum_{j=0}^{n-1}\left|j\right\rangle \tag{2.20}$$

the counting problem can be seen as a particular case of amplitude estimation. If $\mathcal{A}\left|0\right\rangle = \frac{1}{\sqrt{n}}\sum_{j=0}^{n-1}\left|j\right\rangle$ and $0 < t < n$, we can define the following states, representing the state associated to the good elements and the state associated to the bad ones:

$$\left|\psi_{good}\right\rangle = \sum_{j\in X_{good}}\frac{1}{\sqrt{t}}\left|j\right\rangle$$

$$\left|\psi_{bad}\right\rangle = \sum_{j\in X_{bad}}\frac{1}{\sqrt{n-t}}\left|j\right\rangle.$$

Then,

$$\mathcal{A}\left|00\ldots0\right\rangle = \sqrt{\frac{t}{n}}\left|\psi_{good}\right\rangle + \sqrt{\frac{n-t}{n}}\left|\psi_{bad}\right\rangle.$$

Referring to Theorem 2.11, we can notice that, in this case, $\sin^2(\theta) = \frac{t}{n}$ and applying amplitude estimation, we can get an estimate of the amplitude $\frac{t}{n}$, and so an estimate of the *good* elements t.

Algorithm 2.3 Amplitude estimation [22].

Input: The unitary operators $\boldsymbol{A}$,$\boldsymbol{U}_f$ and the precision parameter m.
Output: An estimate of the amplitude $p = \sin^2(\theta_p)$.

1. Set the number of iteration M.
2. Let $Q = A^{-1}U_{0^\perp}AU_f$ (where $U_{0^\perp}, U_f$ are the Grover phase shift operators).
3. Prepare an m-qubit control register, and a second register containing the state $\left|00\ldots0\right\rangle$.
4. Apply QFT to the first register.
5. Apply controlled Q^j.
6. Apply QFT^{-1} to the first register.
7. Measure the first register to obtain a string representing some integer $x \in \{0,\ldots M-1\}$.
8. Output $sin^2(\frac{\pi x}{M})$.

2.2.6. Median evaluation

Median evaluation is the quantum version of the well-known *powering-lemma* described by Jerrum et al. [21]. It is not one of the main quantum algorithms, over which we can build other quantum routines (as phase estimation is). Instead, it is a very important

technique that allows us to boost the success probability of a quantum algorithm. At high level, it consists in repeating Q times the routine we want to boost, and take the median result of the Q outputs.

More formally, we state the theorem that describes this technique.

Theorem 2.13. *(Median evaluation [28, 50]) Let $\mathcal{A}$ be a unitary operator that maps*

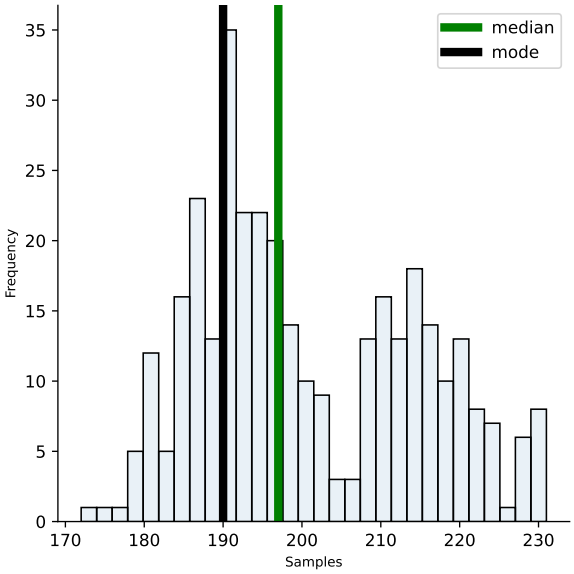
$$\mathcal{A} : |0\rangle \to \sqrt{p}\,|y, 1\rangle + \sqrt{1-p}\,|G, 0\rangle$$

for $1/2 < p \leq 1$ in time T. Then, there exists a quantum algorithm that uses Q copies of the above state, and for any $\Delta > 0$ and for any $1/2 < p_0 \leq p$, produces a state $|\psi\rangle$ such that $\left\| |\psi\rangle - |0\rangle^{\otimes Q} |y\rangle \right\|_2 \leq \sqrt{2\Delta}$ using a number of queries bounded above by

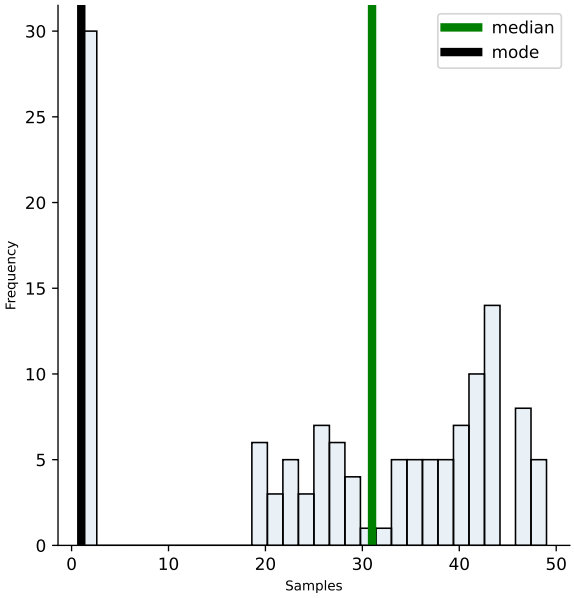
$$2T \left\lceil \frac{\log(1/\Delta)}{2(|p_0| - \frac{1}{2})^2} \right\rceil . \tag{2.21}$$

At this point, one might be asking: *why do we take the median and not the mode, for example?*

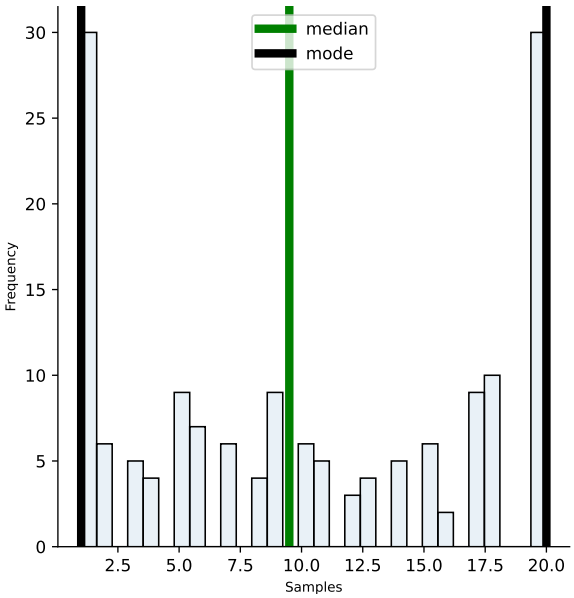
The first important consideration is that, for sure, we can use the mode instead of the median. Indeed, there are no general conditions that can lead us to conclude that one statistic is better than another. The choice to use the median is due to many reasons deriving mainly from classical computer science.
First of all, the median is a statistic which is more robust to outliers. So, with skewed distributions, the median is the one that is less influenced by the extreme values, while this is not true for the mean or the mode. Confirming this fact, in Figure 2.5a, we report a fictitious distribution with the corresponding mode and median (the black and the green vertical lines respectively). As we can see, the median is the statistic that better matches the central tendency of the distribution in contrast with the mode that, by definition, is the most frequent value. The more the data are skewed, the more this fact is evident.
Moreover, the mode does not provide us a good measure of central tendency when the most frequent item is far away from the rest of the data. Indeed, as we can see from Figure 2.5b, we have all the data distributed in the range [20, 50] and the mode at the leftmost of the picture. In this case, the mode would be misleading to describe the central tendency and it would not be representative for the data.
The last case, represented in Figure 2.5c, shows the problem of having data with not unique frequency. In this case, we would have two or more values with the same frequency while the median always matches the central tendency of the data.



(a) Fictitious distribution with mode and median reported.



(b) Fictitious distribution with all data very distant from the mode.



(c) Fictitious distribution with two values with equal frequencies.

Figure 2.5: Data distributions which show that median matches better the central tendency of the data with respect to the mode.

Summing up, the choice of the median is principally due to the fact that, with skewed distributions, it provides a more reliable measure of the central tendency of the data.

2.2.7. Inner product estimation

The *inner product estimation* routine uses internally both amplitude estimation and median evaluation. It is very useful since the inner product is a basic operation in linear algebra and it can be used in the *distance estimation* procedure, which is one of the main steps in the q-Means algorithm (see step 1 of Algorithm 3.4).
In this section, we report the corresponding theorem which describes this algorithm and the pseudocode of the algorithm that we follow from Subroutine 4 in Kerenidis et al. [2]. Basically, we assume to have a unitary operator that creates the state

Algorithm 2.4 IPE [2].

Input: $x, y \in \mathbb{R}^n, \epsilon, \gamma > 0, Q$.
Output: $s \in \mathbb{R}$.

1: Compute $a = \frac{\|x\|^2+\|y\|^2-2\langle x,y\rangle}{2(\|x\|^2+\|y\|^2)}$, $\theta_a = \sin^{-1}(\sqrt{a})$, $\epsilon_a = \frac{\epsilon \max\{1,|\langle x,y\rangle|\}}{\|x\|^2+\|y\|^2}$, $M = \lceil \frac{\pi}{2\epsilon_a}(1 + \sqrt{1+4\epsilon_a})\rceil$.
2: **Amplitude Estimation**
3: **for** j in range(M) **do**
4: $a_j = \sin^2(\frac{j\pi}{M})$.
5: $p(a_j) = \left|\frac{\sin(Md(j/M,\theta_a/\pi))}{M\sin(d(j/M,\theta_a/\pi))}\right|^2$.
6: **end for**
7: **Median Evaluation**
8: **for** q in range(Q) **do**
9: Sample $\overline{a}^{(q)} \sim p$.
10: **end for**
11: Compute $\overline{a} = \text{median}(\overline{a}^{(1)}, \ldots, \overline{a}^{(Q)})$.
12: **Compute inner product estimate**
13: Return $s = (\|x\|^2 + \|y\|^2)(1 - 2\overline{a})/2$.

$|\phi\rangle = \frac{\|x\||0\rangle|x\rangle+\|y\||1\rangle|y\rangle}{\sqrt{\|x\|^2+\|y\|^2}}$. Then, an Hadamard gate is applied to the first register producing the state

$$|\psi\rangle = \sqrt{a}\,|1\rangle\,|\psi_1\rangle + \sqrt{1-a}\,|0\rangle\,|\psi_0\rangle$$

with $a = \frac{\|x\|^2+\|y\|^2-2\langle x,y\rangle}{2(\|x\|^2+\|y\|^2)}$.

We want to get an estimate of a, such that, reversing the formula, we can get an estimate of the inner product. For this purpose, we perform amplitude estimation of $|\psi\rangle$ with $\log(M)$ qubits, so that it returns $\hat{a}$ such that $|a-\hat{a}| \leq \frac{\pi}{M} + (\frac{\pi}{M})^2$ with probability at least of $\frac{8}{\pi^2}$. Taking $M = \left\lceil \frac{\pi}{2\epsilon_a}(1+\sqrt{1+4\epsilon_a})\right\rceil$ ensures that $|a-\hat{a}| \leq \epsilon_a$.
By repeating the above procedure $Q = \left\lceil \frac{\log(1/\gamma)}{2(8/\pi^2-\frac{1}{2})^2}\right\rceil_{odd}$ times, where $\lceil z\rceil_{odd}$ denotes the smallest odd integer greater or equal to z, and taking the median (as we saw in the median

evaluation procedure) ensures that the result $\overline{a}$ satisfies $|a - \overline{a}| \leq \epsilon_a$, with probability at least $1 - \gamma$.

Summing up *IPE* algorithm computes s, such that

$$|s - \langle \boldsymbol{x}, \boldsymbol{y} \rangle| \leq \max\{\epsilon | \langle \boldsymbol{x}, \boldsymbol{y} \rangle |, \epsilon\}$$

with probability at least $1-\gamma$, for input vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$. It is important to notice that, in this procedure, we have an exact knowledge of the norms $\|\boldsymbol{x}\|$ and $\|\boldsymbol{y}\|$ of $\boldsymbol{x}$ and $\boldsymbol{y}$.

Theorem 2.14 (Distance/Inner product estimation [26]). *Assume for a data matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ and a centroid matrix $\boldsymbol{C} \in \mathbb{R}^{k \times d}$ that the following unitaries $|i\rangle|0\rangle \to |i\rangle|\boldsymbol{x}_i\rangle$, and $|j\rangle|0\rangle \to |j\rangle|\boldsymbol{c}_j\rangle$ can be performed in time T and the norms of the vectors are known. Let $d(\boldsymbol{x}_i, \boldsymbol{c}_j)$ be the Euclidean distance between vectors $\boldsymbol{x}_i$ and $\boldsymbol{c}_j$. For any $\Delta > 0$ and $\epsilon_1 > 0$, there exist a quantum algorithm that computes*

$$|i\rangle|j\rangle|0\rangle \to |i\rangle|j\rangle|\overline{d^2(\boldsymbol{x}_i, \boldsymbol{c}_j)}\rangle, \text{ where } |\overline{d^2(\boldsymbol{x}_i, \boldsymbol{c}_j)} - d^2(\boldsymbol{x}_i, \boldsymbol{c}_j)| \leq \epsilon_1, \text{ with probability} \geq 1 - 2\Delta$$

or

$$|i\rangle|j\rangle|0\rangle \to |i\rangle|j\rangle|\overline{(\boldsymbol{x}_i, \boldsymbol{c}_j)}\rangle, \text{ where } |\overline{(\boldsymbol{x}_i, \boldsymbol{c}_j)} - (\boldsymbol{x}_i, \boldsymbol{c}_j)| \leq \epsilon_1 \text{with probability} \geq 1 - 2\Delta$$

in time $\widetilde{O}(\frac{\|\boldsymbol{x}_i\|\|\boldsymbol{c}_j\| T \log(1/\Delta)}{\epsilon_1})$.

2.2.8. Singular value estimation

This routine enables estimating singular values using phase estimation. We report the theorem that describes this procedure, highlighting the main steps. The interested reader can check the detailed algorithm described by Kerenidis and Prakash [25].

Theorem 2.15. *(Singular Value Estimation [25]) Let there be quantum access to $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, with singular value decomposition $\boldsymbol{X} = \sum_i^r \sigma_i \boldsymbol{u_i} \boldsymbol{v_i^T}$ and $r = \min(n, d)$. Let $\epsilon > 0$ be a precision parameter. It is possible to perform the mapping $|b\rangle = \sum_i \alpha_i |\boldsymbol{v_i}\rangle \to \sum_i \alpha_i |\boldsymbol{v_i}\rangle|\overline{\sigma}_i\rangle$, such that $|\frac{\sigma_i}{\mu(\boldsymbol{X})} - \overline{\sigma}_i| \leq \epsilon$ with probability at least $1 - \frac{1}{poly(d)}$, in time $\widetilde{O}\left(\frac{1}{\epsilon}\right)$, where $\mu(\boldsymbol{X}) = \min_{p \in [0,1]}(\|\boldsymbol{X}\|_F, \sqrt{s_{2p}(\boldsymbol{X}) s_{2(1-p)}(\boldsymbol{X}^T)})$ and $s_p(\boldsymbol{X}) = \max_i \|\boldsymbol{x}_{i,\cdot}\|_p^p$. Similarly, we can have $|\sigma_i - \overline{\sigma}_i| \leq \epsilon$ in time $\widetilde{O}\left(\frac{\mu(\boldsymbol{X})}{\epsilon}\right)$.*

Basically, the routine described in Theorem 2.15 performs phase estimation to estimate singular values with an ϵ error. We note, in Algorithm *IV*.1 by Kerenidis and Prakash

[25], that singular values can be described in the following way

$$\sigma_i = \cos\left(\frac{\theta_i}{2}\right)\mu(\boldsymbol{X}). \tag{2.22}$$

This relationship between σ_i and θ_i comes from Lemma 5.3 in Kerenidis and Prakash [23] which we report in Lemma 2.

Lemma 2. *Let $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ be a matrix with singular value decomposition $\boldsymbol{X} = \sum_i \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T$ stored in QRAM. Then, there exist matrices $\boldsymbol{P} \in \mathbb{R}^{nd\times n}$, $\boldsymbol{Q} \in \mathbb{R}^{nd\times d}$ such that*

- *The matrices $\boldsymbol{P}, \boldsymbol{Q}$ are a factorization of $\boldsymbol{X}$, i.e. $\frac{\boldsymbol{X}}{\|\boldsymbol{X}\|_F} = \boldsymbol{P}^T\boldsymbol{Q}$. Moreover, $\boldsymbol{P}^T\boldsymbol{P} = \mathbb{I}_n, \boldsymbol{Q}^T\boldsymbol{Q} = \mathbb{I}_d$, and multiplication by $\boldsymbol{P}, \boldsymbol{Q}$, i.e. the mappings $|y\rangle \to |\boldsymbol{P}y\rangle$ and $|x\rangle \to |\boldsymbol{Q}x\rangle$ can be performed in time $O(polylog(nd))$.*
- *The unitary $\boldsymbol{W} = \boldsymbol{U}\cdot\boldsymbol{V}$, where $\boldsymbol{U}, \boldsymbol{V}$ are the reflections $\boldsymbol{U} = 2\boldsymbol{P}\boldsymbol{P}^T - \mathbb{I}_{nd}$ and $\boldsymbol{V} = 2\boldsymbol{Q}\boldsymbol{Q}^T - \mathbb{I}_{nd}$ can be implemented in time $O(polylog(nd))$.*
- *The isometry $\boldsymbol{Q} : \mathbb{R}^d \to \mathbb{R}^{nd}$ maps a row singular vector $\boldsymbol{v}_i$ of $\boldsymbol{X}$ with singular value σ_i to an eigenvector $\boldsymbol{Q}\boldsymbol{v}_i$ of $\boldsymbol{W}$ with eigenvalue $e^{i\theta_i}$ such that $\cos(\theta_i/2) = \sigma_i/\|\boldsymbol{X}\|_F$.*

Basically, the main idea of the singular value estimation algorithm is to map the input vector $\sum_i \alpha_i |\boldsymbol{v}_i\rangle \to \sum_i \alpha_i |\boldsymbol{Q}\boldsymbol{v}_i\rangle$ by applying $\boldsymbol{Q}$. Then, using phase estimation with unitary $\boldsymbol{W}$ to compute an estimate of θ_i and therefore of σ_i. Indeed, by inverting the formula reported in Equation 2.22, for each singular value σ_i, we get the corresponding θ_i. In this way, we can get the estimate of each singular value by estimating $\overline{\theta}_i$ with phase estimation and substituting this value into the previous formula to get $\overline{\sigma}_i = \cos\left(\frac{\overline{\theta}_i}{2}\right)\mu(\boldsymbol{X})$. We describe our approach to simulate SVE in the quantum machine learning algorithms that use this routine in q-PCA.

2.2.9. Spectral norm estimation

This routine simulates the estimation of the spectral norm, which corresponds to the greatest singular value of an input matrix stored in QRAM. Here below we report the algorithm that we take into account to implement it (Algorithm IV.3 in [25]).

It is like a binary search where, at the end of the $O(\log(1/\epsilon))$ iterations, we obtain a value of τ that, multiplied by $\|\boldsymbol{X}\|_F$, allows us to obtain an estimate of the spectral norm of matrix $\boldsymbol{X}$. Thanks to the QRAM structure, the Frobenious norm $\|\boldsymbol{X}\|_F$ can be retrieved in constant time. The steps are principally based on Singular Value Estimation (SVE) (described in Theorem 2.15), on conditional rotations, and on amplitude estimation, which

Algorithm 2.5 Spectral norm estimation [25].

Input: Input data matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, ϵ error for the SVE, and ϵ' error for the amplitude estimation step.
Output: Returns an estimate for $\eta := \|\boldsymbol{X}\|/\|\boldsymbol{X}\|_F$ with additive error ϵ.

1. Let $l = 0$ and $u = 1$ be the upper and lower bounds for η, the estimate $\tau = (l+u)/2$ is refined using binary search in steps 2-5 over $O(\log(1/\epsilon))$ iterations.
2. Prepare $|\phi\rangle = \frac{1}{\|\boldsymbol{X}\|_F} \sum_{i,j} x_{ij} |i,j\rangle = \frac{1}{\|\boldsymbol{X}\|_F} \sum_{i,j} \sigma_i |u_i, v_i\rangle$ and perform SVE with precision ϵ to obtain $\frac{1}{\|\boldsymbol{X}\|_F} \sum_{i,j} \sigma_i |u_i, v_i, \overline{\sigma}_i\rangle$, where $|\overline{\sigma}_i - \frac{\sigma_i}{\|\boldsymbol{X}\|_F}| \leq \epsilon$.
3. Append single qubit register $|R\rangle$ and set it to $|1\rangle$ if $\overline{\sigma}_i \geq \tau$ and $|0\rangle$ otherwise. Uncompute the SVE output from step 2.
4. Perform amplitude estimation on $\frac{1}{\|\boldsymbol{X}\|_F} \sum_{i,j} \sigma_i |u_i, v_i, R\rangle$ conditioned on $R = 1$ to estimate $\sum_{i:\overline{\sigma}_i \geq \tau} \sigma_i^2 / \|\boldsymbol{X}\|_F^2$ to relative error $(1 \pm \epsilon')$.
5. If estimate in step 4 is 0 then $u = \tau$ else $l = \tau$. Update $\tau = (u + l)/2$.

involve a total time complexity of $\widetilde{O}\left(\log(1/\epsilon)/\epsilon\eta\right)$.

3 | The Sqlearn framework

In this chapter, we present the open-source *Sqlearn* framework. We introduce and describe how we simulate the quantum algorithms presented in Chapter 2. To better describe our implementation, we report code snippets. Then, we show some experiments and numerical simulations for the main quantum tasks and routines implemented, such as *pure state tomography, amplitude estimation, phase estimation*, and *consistent phase estimation* to verify their correctness. In particular, we are going to see interesting insights into vector state tomography that we exploit in its application in real machine learning problems.
With this framework, which is an expansion of *Sklearn* API [34], we can perform in a simple way machine learning experiments both in the classical way and by simulating their quantum counterpart. In this way, we can have an efficient way to compare the performances between classical and quantum machine learning models.
All the framework is developed using the Python language.

Stucture of the framework The framework follows the same structure as Sklearn. Therefore, users already confident with this famous classical framework will find Sqlearn very familiar. Basically, the framework rests on a Python module, called *QuantumUtilities.py*, which contains the implementation of the main quantum routines such as tomography, amplitude estimation, phase estimation, consistent phase estimation, and median evaluation. Therefore, a user, by exploiting this module, is able to simulate these quantum functions separately. Then, we extend the PCA and K-Means modules, already present in Sklearn, implementing the quantum machine learning algorithms that we are going to describe in Section 3.8 that exploit the quantum routines implemented in *QuantumUtilities.py*. In this way, we provide the *q-PCA.py* and *q-Means.py* Python modules that enable simulating the corresponding machine learning algorithms both in their classical and quantum version.
Together with the framework, we provide a complete documentation, generated with Sphinx [8], that describes in detail all the implemented quantum routines to make this framework even more user-friendly.

3.1. Quantum state simulation

Since quantum routines such as state tomography are based on the implementation of a *quantum state*, we need to have a way to simulate the quantum state object (indeed, in state tomography, we perform a fixed number of measures of a specific quantum state). For this purpose, we implement a class that simulates a pure quantum state

$$|\mathbf{x}\rangle = \frac{1}{\|\mathbf{x}\|} \sum_{i}^{n} x_i |i\rangle .$$

We initialize the quantum state object, passing in the constructor of QuantumState class the list of registers values and the list of the corresponding amplitudes. In this work, we assume to deal with a generic quantum register of n-qubits, so we do not specify its precision n as it is not fundamental in the execution of the experiments. For this class implementation, we refer to the code reported by Bellante [6].
The implemented class exposes two important methods:

- *get_state*() that returns a dictionary which has as key the value of the register and as value the corresponding probability.
- *measure*() that simulates the measurements of the state using a weighted random choice according to the probabilities of the registers. It is also possible to specify the number of measurements with the parameter *n_times*.

```
1     class QuantumState(object):
2     """This class simulates a simple Quantum State
3
4     Parameters
5     ----------
6
7     registers: list, ndarray.
8         List of values that represents the superposition of the states of
    ↪ the quantum state.
9
10    amplitudes: list, ndarray.
11        List of values that represents the amplitudes of each register.
12    """
13
```

```
14      def __init__(self, registers, amplitudes):
15          super(QuantumState, self).__init__()
16          self.registers = registers
17
18          # Amplitudes must be normalized to have the right probabilities
            ↪  for each register
19          self.norm_factor = math.sqrt(sum([pow(x, 2) for x in
            ↪  amplitudes]))
20          self.amplitudes = [x / self.norm_factor for x in amplitudes]
21
22          # Each register_i appears with probability amplitude_i^2
23          self.probabilities = [pow(x, 2) for x in self.amplitudes]
24          assert (len(self.registers) == len(self.amplitudes))
25          assert (abs(sum(self.probabilities) - 1) < 0.0000000001)
26
27      def measure(self, n_times=1):
28
29          return np.random.choice(self.registers, p=self.probabilities,
            ↪  size=n_times)
30
31      def get_state(self):
32          return {self.registers[i]: self.probabilities[i] for i in
            ↪  range(len(self.registers))}
```

Source Code 3.1: Quantumstate class from Bellante [6].

Another important method implemented but not included in the QuantumState class is the *estimate_wald*() that, given a list of measurements, estimates the probabilities of each value of a quantum register. This is the usual frequentist estimator, which can be simply seen as the number of measurements for each specific register value divided by the total number of measurements done.

```
1       def estimate_wald(measurements):
2       counter = Counter(measurements)
3       estimate = {x: counter[x] / len(measurements) for x in counter}
4       return estimate
```

3.2. Vector state tomography

Algorithm 2.1 in Section 2.2.2 describes the steps of this quantum task. Now, we describe our approach to classically implement it. We remember the goal: given a quantum state $|\boldsymbol{x}\rangle = \frac{1}{\|\boldsymbol{x}\|}\sum_{i=0}^{d-1} x_i |i\rangle$, with unit vector $\boldsymbol{x} \in \mathbb{R}^d$, vector state tomography outputs an estimate $\overline{\boldsymbol{x}}$ such that $\|\overline{\boldsymbol{x}} - \boldsymbol{x}\|_2 \leq \delta$.

The first important thing to check is that the input vector $\boldsymbol{x}$ has unitary norm since it is a requirement of the algorithm. For non-unitary norm vector, we normalize it at the beginning of the routine, dividing it for its ℓ_2-norm. After that, we build the *quantum state* $|\boldsymbol{x}\rangle$ (using the implementation reported in Section 3.1), passing as registers the indexes of the vector {0,. . . , d-1} and as amplitudes the values of the vectors corresponding to that indexes $\{x_0,\ldots, x_{d-1}\}$.

The main task of tomography is to measure copies of a given quantum state to retrieve classical information about it. The number of measurements N, that need to be performed in order to guarantee that $\|\overline{\boldsymbol{x}}-\boldsymbol{x}\|_2 \leq \delta$, is computed using the formula $\frac{36d\log(d)}{\delta^2}$ [24]. Then, to obtain the estimates of the amplitudes of the state $|\boldsymbol{x}\rangle$, we make use of the *measure*() and *estimate_wald*() methods of the created quantum state object: with the first, we can simulate the measurements of N copies of $|\boldsymbol{x}\rangle$, and with the latter, we can extract the estimates of the amplitudes as $p_i = \frac{n_i}{N}$, as required in the first step of the algorithm. Clearly, the bigger is N, the more consistent are the estimates.

To complete the first part of the algorithm, we fill an empty vector with the amplitudes $\sqrt{p_i}$, based on the index i. Therefore, in this way, we store the estimates of the amplitudes, as described by the algorithm with the state $|p\rangle$.

```
1     def real_tomography(V, N=None,
      ↪  delta=None,stop_when_reached_accuracy=False, norm='L2'):
2     """ Official version of the tomography function.
3
4     Parameters
5     ----------
6     V: array-like that has to be estimated.
7
8     N: int value, default=None.
9         Number of measures to be performed for the quantum state. If
  ↪   None, it is computed in the function itself.
```

```
10
11      delta: float value, default=None.
12          It represent the error that you want to introduce to estimate the
    ↪   representation of the vector V.
13
14      stop_when_reached_accuracy: bool, default=True.
15          If True it stops the execution of the tomography when the L2-norm
    ↪   of the
16          difference between V and its estimation is less or equal then
    ↪   delta. Otherwise
17          N measures are performed (very memory intensive for large
    ↪   vectors).
18
19      norm: string, default='L2'.
20          If 'L2':
21              perform L2-norm tomography.
22          If 'inf':
23              perform L-inf tomography.
24
25      Returns
26      -------
27      dict_res: dictionary of shape {N_measure: vector_estimation}.
28
29      """
30
31      if np.isclose(np.linalg.norm(V), 1, rtol=1e-2):
32          pass
33      else:
34          V = V / np.linalg.norm(V, ord=2)
35      d = len(V)
36      index = np.arange(0, d)
37      if N is None:
38          if norm == 'L2':
39              N = int((36 * d * np.log(d)) / (delta ** 2))
40          elif norm == 'inf':
41              N = int((36 * np.log(d)) / (delta ** 2))
42
```

```
    q_state = QuantumState(amplitudes=V, registers=index)
    P = estimate_wald(q_state.measure(n_times=int(N)))
    P_i = np.zeros(d)

    P_i[list(P.keys())] = np.sqrt(list(P.values()))
```

Source Code 3.2: Implementation of the first step (vector values estimation step in Algorithm 2.1) of vector state tomography algorithm.

For what concern the implementation of sign estimation part (part 2 of Algorithm 2.1), the tricky part is to obtain the state

$$\frac{1}{2}\sum_{i\in[d]}[(x_i+\sqrt{p_i})\,|0,i\rangle+(x_i-\sqrt{p_i})\,|1,i\rangle].$$

The amplitudes of this state can be easily computed since we have both the values x_i and $\sqrt{p_i}$. To encode the values of the registers $|0,i\rangle$ and $|1,i\rangle$, we use strings with either 0 or 1 as initial bitstring. For example, suppose the vector length $d=4$, we generate as registers values the strings $\mathbf{0}0,\mathbf{0}1,\mathbf{0}2,\mathbf{0}3$ and $\mathbf{1}0,\mathbf{1}1,\mathbf{1}2,\mathbf{1}3$. Then, we create a new quantum state object with these registers and $(x_i+\sqrt{p_i})$ and $(x_i-\sqrt{p_i})$ as amplitudes. We create and measure N copies of this state using the *measure*() function, registering in a dictionary (using the *Counter()* function) the times each outcome is observed $n(b,i)$, with $b\in 0,1$. At the very end, we can estimate the sign σ_i of each value, checking if $n(0,i)>0.4p_iN$ or not, and return the estimated vector $\overline{\boldsymbol{x}}$, composed by the estimated values $\overline{x}_i=\sigma_i\sqrt{p_i}$.

```
    max_index = max(index)
    digits = len(str(max_index)) + 1
    registers = [str(j).zfill(digits) for j in index] + [re.sub('0', '1',
    ↪  str(j).zfill(digits), 1) for j in index]

    amplitudes = np.asarray([V[k] + P_i[k] for k in range(len(V))] +
    ↪  [V[k] - P_i[k] for k in range(len(V))])

    amplitudes *= 0.5

    new_quantum_state = QuantumState(registers=registers,
    ↪  amplitudes=amplitudes)
```

```
10
11      measure = new_quantum_state.measure(n_times=int(N))
12
13      str_ = [str(ind).zfill(digits) for ind in index]
14      dictionary = dict(Counter(measure))
15
16      if len(dictionary) < len(registers):
17          keys = set(list(dictionary.keys()))
18          tot_keys = set(registers)
19          missing_keys = list(tot_keys - keys)
20          dictionary.update({l: 0 for l in missing_keys})
21
22      d_ = list(map(dictionary.get, str_))
23
24      P_i = [P_i[e] if x > 0.4 * P_i[e] ** 2 * N else P_i[e] * -1 for e, x
        ↪  in enumerate(d_)]
25
26      return P_i
```

Source Code 3.3: Implementation of the second step (sign estimation step in Algorithm 2.1) of vector state tomography algorithm.

The procedure described up to now refers to ℓ_2-tomography but we also provide the possibility to simulate ℓ_∞-tomography (described in Theorem 2.4). The fundamental difference is in the number of measurements that we perform: instead of $\frac{36d\log(d)}{\delta^2}$, we perform $\frac{36\log(d)}{\delta^2}$ measures.

That said, we have verified the correctness of this algorithm with the experiments that we are going to describe. We perform vector state tomography onto 4 randomly generated vectors of length 784: the first one generated with a *uniform* distribution using *np.random.uniform()* functionality of Numpy and the second one with an *exponential* distribution, always using Numpy. Regarding the other two vectors, we generate two sparse vectors with different percentages of sparsity: the first one with 80% of sparsity and the other one with the 50%. All four vectors are of unitary norm.

3.2.1. The error-measurements trade-off

The goal of this test is to see how the error of the vector estimates decreases as the number of measures performed by tomography increases, comparing the results between the four different vectors. In this test, we fix the error δ such that, taking an initial state $|\boldsymbol{x}\rangle$, the estimated vector $\overline{\boldsymbol{x}}$ ensures that $\|\boldsymbol{x} - \overline{\boldsymbol{x}}\|_2 \leq \delta$. To be able to perform this type of experiment, instead of computing the number of measurements $N = \frac{36d\log(d)}{\delta^2}$ and performing at once these measures, we can execute the routine with different values of N and see how the estimates change. In Figure 3.1, we report the curves resulting from the simulation of the ℓ_2-tomography over the four vectors, where on the x-axis we report the number of measurements performed by tomography, and on the y-axis we report the error of the estimate as $\|\boldsymbol{x} - \overline{\boldsymbol{x}}\|_2$.

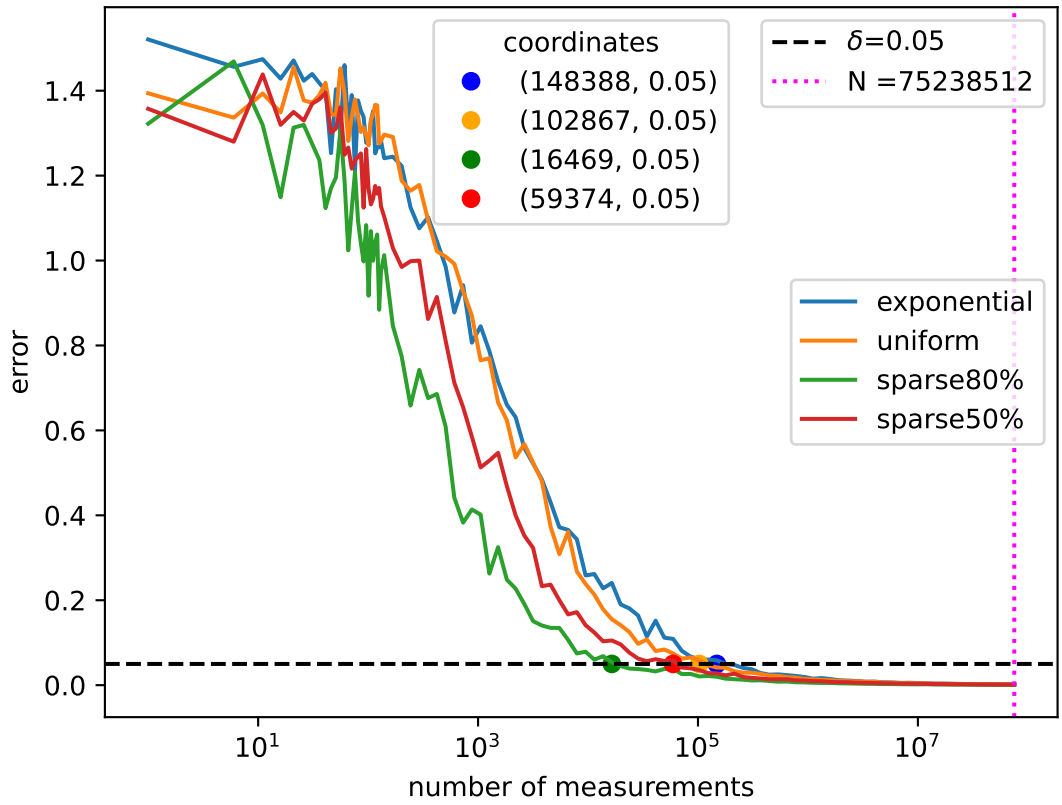


Figure 3.1: Decreasing of the error as the number of measurements increases performing tomography on four different vectors. The coordinates reported in the legends are relative to the colored dots in the plot, which represent the number of necessary measures, for each vector, to reach the desired accuracy in the estimates of the vectors.

As we can see, we compute the total number of measures N using $\delta = 0.05$ as error. We represent the desired level of accuracy with a dashed horizontal black line. The colored dots highlight, for each vector, what is the number of measures for which we obtain estimates with an error of 0.05. For example, let $\boldsymbol{x} \in \mathbb{R}^{784}$ be the uniform vector. We can see that by performing tomography with a number of measures equal to 102,867, we obtain an estimate $\overline{\boldsymbol{x}}$ such that $\|\boldsymbol{x} - \overline{\boldsymbol{x}}\|_2 \leq 0.05$. Therefore, also from this picture, we can see an insight into tomography that we analyze better in the next experiment: we see that the number of measures $N = \frac{36d\log(d)}{\delta^2}$ (represented with the vertical magenta line) is very large and we can reach the desired accuracy also with a smaller number of measurements. Indeed, if we perform the number of measures N stated by the theory, we obtain an estimate very close, practically equal to the real vector with an error almost equal to zero (as we can see at the intersection of the curves with the magenta dashed line). Another interesting thing that we can notice is that the sparser the vector the more the error curves decrease rapidly. This is an expected result since the more a vector is sparse, the more by definition it contains zeros and so we need fewer measurements to retrieve the classical values of this vector since we start always our estimation procedure with a zero-filled vector, as shown in our implementation.

3.2.2. Insights into the measurements bound

With this experiment, we confirm the insights into the number of measurements that we find in the previous experiment. Indeed, we show that, by performing $N = \frac{36d\log d}{\delta^2}$ measures (with approximation error δ), ℓ_2-tomography outputs estimates that are closer to the true values with respect to the approximation error δ.

In Figure 3.2, we report how many measures (x-axis) are necessary to get a vector estimate with a specific error (y-axis) both considering the theoretical bound (represented with the blue curve) and by actually performing the tomography (represented with the orange curve) over the uniform vector. We obtain the theoretical curve by substituting each of the δ value of the y-axis into $N = \frac{36d\log(d)}{\delta^2}$. In this way, we compute exactly how many measures we need, in theory, to get an estimate with error δ. In this experiment, the choice of considering the uniform vector has no particular motivation. The same experiment could be executed using the other vectors obtaining the same results.

We can notice that the shape of the two curves is almost equal. However, the magnitudes of the values between the curves are very different. Indeed, we see that we need a smaller number of measurements to reach the desired accuracy with respect to what the theory expects. Let us take, for example, the orange and blue dots that indicate where the two curves intersect the horizontal line corresponding to $\delta = 0.05$: the orange curve intersects

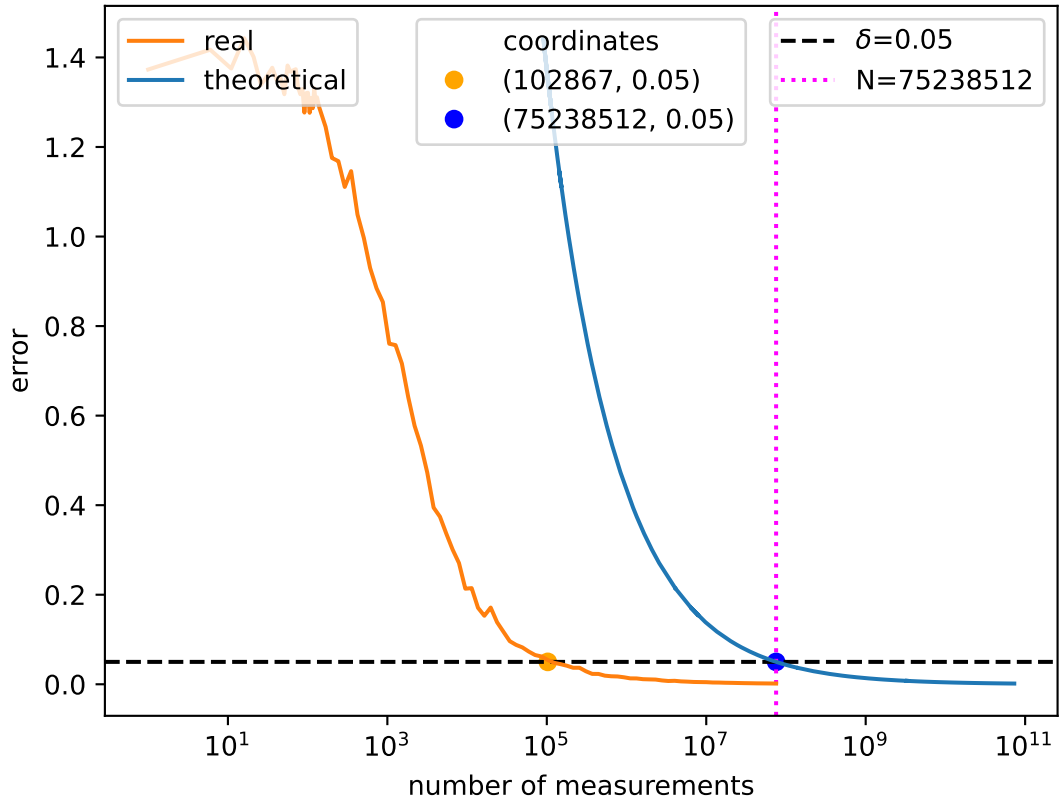
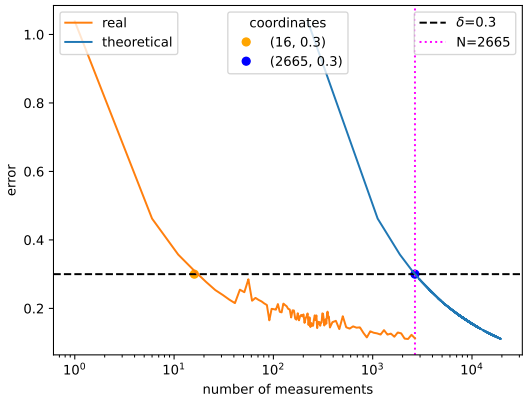


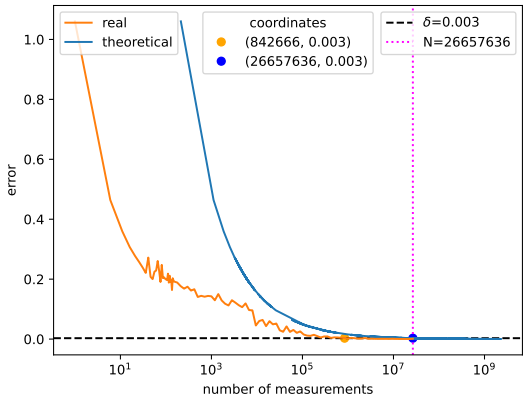
Figure 3.2: Comparison between real and theoretical decrease of the error by performing the ℓ_2-tomography over the uniform vector with $\delta = 0.05$.

it at N=102,867, while the blue one at N = 75,238,512 (that corresponds rightly to the intersection of the magenta vertical line with the horizontal black line). We see that, by effectively performing tomography with this big N number, we get a vector estimate very close to the true vector, with error δ near to 0, as shown by the orange curve at the intersection with the magenta line. To reach the same value, we would need almost 10^{11} measurements, according to the theoretical bound which, therefore, can be considered as an upper bound (also in Kerenidis and Prakash [24] it is reported as upper bound).
As we are going to see in the next chapter, we exploit this property into quantum machine learning algorithms that execute tomography internally. Basically, we relax the error bound $\|\boldsymbol{x} - \overline{\boldsymbol{x}}\|_2 \leq \delta$ by considering larger error δ. This is because we still obtain tomography estimates with lower error than δ (as we have seen in this experiment by inserting $\delta = 0.05$ and getting as output an estimate with δ almost zero) and, at the same time, we impact less on the running time since it scales as $O(\frac{d}{\delta^2})$.

For completeness, we repeat the same experiment for the ℓ_∞-tomography, as reported in Figure 3.3. In this case, we report two plots, one for $\delta = 0.3$ and the other for $\delta = 0.003$.



(a) Comparison between real and theoretical decrease of the error by performing the ℓ_∞-tomography over the uniform vector with $\delta = 0.3$.



(b) Comparison between real and theoretical decrease of the error by performing the ℓ_∞-tomography over the uniform vector with $\delta = 0.003$.

Figure 3.3: Two cases example of ℓ_∞-tomography application.

If we take a look at Figure 3.3a, we notice a strange thing. We see that we only need 16 measures to obtain an estimate with an error of 0.3 of a vector long 784. At a first sight, this is not possible because it would mean that we do not measure all the values of the vector. The reason for this result is that the values of the vector are very small, in the order of 10^{-2}, and so an error of 0.3 is very big with respect to the values of the vector. This means that if the tomography estimates a value as 0, which in our case is equivalent to saying that it does not measure this specific value since we start the tomography procedure with a zero-filled vector (as reported also in the code), we are consistent with the error bound, since $|0 - 10^{-2}| \leq 0.3$. For this reason, with the ℓ_∞-tomography, if we use an error too big with respect to the values that we need to estimate, we obtain this "strange" result. In Figure 3.3b, we show, indeed, that decreasing δ to 0.003 results in a larger number of necessary measures, more than the vector length. Also for the ℓ_∞-tomography, we can see that the theoretical bound is very large with respect to the real measures needed in our experiments.

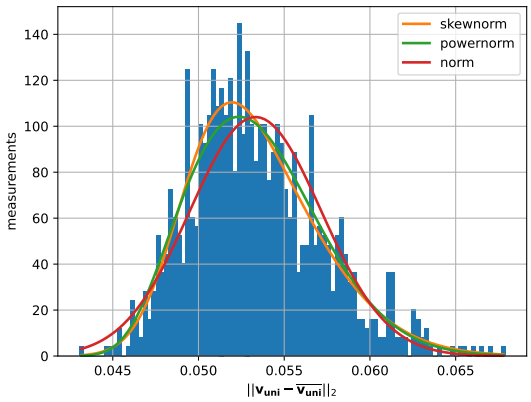
The fact that tomography returns vector estimates with an error lower than δ is probably caused by specific vectors with particular distributions for which we need to perform exactly N measurements to obtain the desired accuracy. We tried to reach the theoretical bound by changing the vector distributions but the results that we found show the same behavior. Finding the vectors for which exactly N measurements are necessary to get an

estimate with error δ could be a topic for future investigation.

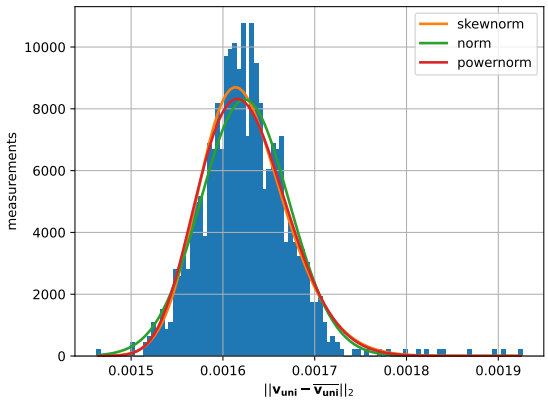
3.2.3. Distribution of the tomography error

The last type of experiment for the vector state tomography involves finding the distribution of the error in the estimates of the tomography. Once fixed the approximation error δ, we repeat $R = 1000$ times the tomography over the same vector (for simplicity and consistency we took always the uniform vector) and we plot the ℓ_2-norm of the difference between the estimated vector and the real one, for all the trials. In this way, we can see how the error of the estimates is distributed among different tomography executions. This is very useful because, by knowing that distribution, one might not even perform exactly the true tomography routine but a faster approximation of it. For instance, one might directly extract the estimates of the tomography by adding a random noise value, extracted from the specific error distribution found, to the true values that we need to estimate.

To find the distribution that fits better, we make use of the Python library *Fitter*, which is a package that provides a simple class to identify the distribution from which a data sample is generated, choosing from the 80 distributions present in *Scipy*. To choose the best distribution, we use as metric the sum of square error (SSE) between the data and the fitted distribution. We remark that we perform all the following experiments using the uniform vector $\boldsymbol{v}_{uni}$ and fixing the error to $\delta = 0.05$. At a first sight, we can observe that,



(a) Tomography error distribution after 1000 repetitions of the procedure with N=102,867 measures over a uniform vector.



(b) Tomography error distribution after 1000 repetitions of the procedure with $N = \frac{36d \log d}{\delta^2}$ measures over a uniform vector.

Figure 3.4: Tomography error distribution with different number of measures N over a uniform vector.

in general, as the number of measurements N increases, the distribution is increasingly skewed. Going into more details, we have seen that the number of real measure N, for which we get an estimate such that $\|\overline{\boldsymbol{v}}_{uni} - \boldsymbol{v}_{uni}\|_2 \leq 0.05$, is N=102,867 (see the results in Figure 3.2). Therefore, fixing the number of measurements to N=102,867, we repeat the tomography routine 1000 times in order to have enough samples to get a consistent estimate of the error distribution. What the plot in Figure 3.4a tells us is coherent with the result reported in Figure 3.2. Indeed, we can see that we have a *skewed Gaussian* shaped distribution almost centered in 0.05. This means that in 1000 trials, with this specific uniform vector and N=102,867, we have 0.05 as estimate error on average. Obviously, not being a deterministic procedure, it can happen that with that number of measures, we get an estimate $\overline{\boldsymbol{v}}_{uni}$ such that $\|\overline{\boldsymbol{v}}_{uni} - \boldsymbol{v}_{uni}\|_2 > 0.05$, but this is clearly less likely since we find the values relative to those estimates at the tails of the Gaussian distribution. For instance, it is very unlikely that with 102,867 measures we get an error of 0.065.
For what concern the results reported in Figure 3.4b, we execute tomography without specifying N, thus letting the procedure calculates it as $N = \frac{36d \log d}{\delta^2}$. We can see that the distribution is increasingly skewed towards the left and it is much narrower almost like a normal distribution. Even in this case, we are consistent with the experiments reported in the previous sections. Indeed, the error found is in the order of 0.001 which is much less than 0.05.
This last plot is the most faithful since, in reality, we cannot know how many measures we need to estimate a certain value with a specific accuracy and so we need to perform all the N measures computed by the tomography procedure in the function of δ (in the first case, the error δ does not affect N since the number of measures is fixed by us). We report these two cases both to show how the tomography error distribution changes at the increase of N and to show, with the first plot, that with a number of measures much less than N (in this case 102,867), on average, we get an estimate $\overline{\boldsymbol{v}}$ that guarantees the required error δ. In both cases, we have seen, using Fitter library, that the most promising distribution is the *skewednorm* distribution.

3.3. Phase estimation

We summarize the phase estimation procedure in Algorithm 2.2.
For what concerns our implementation, first we compute the number of qubits n as reported in Equation 2.14 using the specified error parameter ϵ and the probability of failure γ. If desired, the user can directly specify the number of qubits n to use in the routine. If this is the case, the parameters ϵ and γ have no effect in the computation of n. In the Source Code 3.4, we also show our implementation for the core part of phase estimation

that refers to steps 2-5 of Algorithm 2.2. Basically, we compute the probability for each of the 2^n possible values (computed as $\frac{x}{2^n}$, with $x \in [0, \ldots, 2^n - 1]$) using the formula reported in Equation 2.16. Practically, we end up with two lists: in the first (named *omega_x* in the code), we have all the possible 2^n values, while in the second (named p in the code), we have all the probabilities associated with those values. Then, with a weighted random choice using the list of probabilities as weights, we extract the estimate $\overline{\omega} = \frac{\overline{x}}{2^n}$ (*omega_tilde* in the code).

```
def phase_estimation(omega, epsilon, gamma=0.1, n=None):
""" Official version of the phase estimation function.

        Parameters
        ----------
        omega: float or int value.
             Value that has to be estimated by the phase estimation. It must be in the range of [0,1).
        epsilon: float value.
            Precision that you want to have in the phase estimation procedure.
        gamma: float value, default=0.1.
            It represent the probability of failure of the routine.

        n: int value, default=None.
            The number of qubits you want to use for the computation.

        Returns
        -------
        omega_tilde: float value.
            Estimate of the true omega value.

    """
    if n == None:
        n = int(np.ceil(np.log2(1 / epsilon)) + np.ceil(np.log2(2 + 1 / (2 * gamma))))

    M=2**n
    omega_x=[]
```

```
    p=[]
    for x in range(M):
        omega_est = x / M
        omega_x.append(omega_est)
        try:
            p.append(np.abs((math.sin((M * omega - x) * np.pi)) / (M *
            ↪  (math.sin((omega - x / M) * np.pi)))) ** 2)
        except:
            # if the division is 0/0 -> case when M*omega is an integer.
            p.append(1)
    omega_tilde = random.choices(omega_x, weights=p, k=1)[0]
    return omega_tilde
```

Source Code 3.4: Implementation of the phase estimation algorithm.

In our implementation, we can increase or decrease the probability of success of the routine playing with the γ parameter.

Once implemented this routine in the framework, we simulate it to verify its correct functioning. We report two numerical examples: the first one aims to show the probability distribution of the outputs of phase estimation, showing that the value closer to the real one is the one that is most likely to be extracted. In the second, we report a numerical simulation of phase estimation since we think it is useful to understand better how the implemented phase estimation algorithm works.

Probability distribution of phase estimation outputs With this example, we show that given the true phase ω that we want to estimate, phase estimation associates the higher probability to the value closest to ω so that this will be the most likely estimate. We take $\omega = 0.3$ (it is important to take a value between $[0, 1)$ as required by the algorithm) and we perform phase estimation routine with error $\epsilon = 0.001$ and probability of failure $\gamma = 0.1$. This results in a number of qubits $n = 13$ (by computing it with Equation 2.14 reported in Section 2.2.3) from which we get $2^{13} = 8192$ possible estimate values. For each of the possible 8192 values, we compute the corresponding probability as specified in Theorem 2.5 and in the Source Code 3.4 that we reported previously. Therefore, we get two lists: the first with all the possible values $\frac{x}{2^n}$ (with $x \in [0, \dots, 2^n - 1]$) that we can extract as estimate by a weighted random choice and the second with all the probabilities of each value to be extracted as estimate. We report the distribution probability in Figure 3.5 where on the x-axis is represented each possible estimate $\overline{\omega} = \frac{x}{2^n}$ and on the y-axis is

represented the probability associated to each sample. In the plot, we also report the true value ω that we are estimating, represented with a vertical yellow dashed line, and the relative error with red vertical dashed lines, given by $\omega \pm \epsilon\omega$. It is interesting to notice that the distribution is almost *Gaussian* shaped, with the maximum value corresponding to the value which can be extracted with a higher probability that is the value closer to ω. Moreover, the further we move away from ω, the more the probabilities decrease. We compute also the actual percentage of sampled points that lay within the desired range, represented by the two red vertical lines, and we call it $\mathcal{F}$. In this case, we have that $\mathcal{F} = 0.061\%$, so a very small percentage of the total number of samples. This indicates that our estimates are very precise since there are few values in this *good* range and those values are very close to the true one.

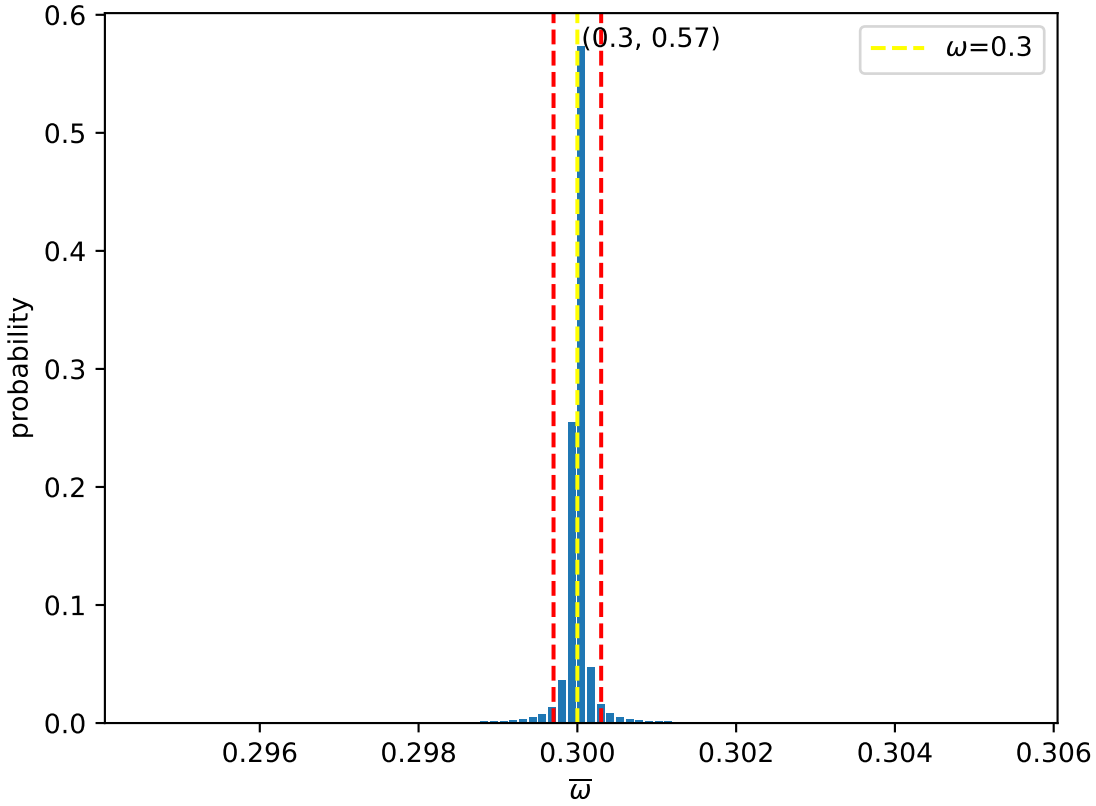


Figure 3.5: Probability distribution of phase estimation outputs with $\omega = 0.3$, $\epsilon = 0.001$, and $\gamma = 0.1$.

We also notice that our estimate will be more likely 0.3 with a probability of 0.57, as

depicted in the plot. This is a very good result considering that we are estimating $\omega = 0.3$.

Numerical simulation The phase estimation algorithm has been explained in the previous sections, but we think that a numerical simulation of the procedure is very useful to better understand it.
We report a simple example, where we choose $\omega = 0.035$ and $\epsilon = \gamma = 0.1$, with ϵ accuracy and γ probability of failure. We report the possible estimate values and the associated probabilities in Table 3.1. Each column represents the integer x, such that $\overline{\omega} = \frac{x}{2^n}$ while, in the first row, we report all the possible estimate values and, in the last row, the associated probabilities. Depending on the magnitude of each probability, each cell is filled with a more or less dark red.

	x=0	**x=1**	**x=2**	**x=3**	**x=4**	**x=5**	**x=6**	**x=7**	$\cdots$	**x=127**
$\overline{\omega}$	0.0	0.0078	0.0156	0.0234	0.0312	0.0390	0.0468	0.0546	$\cdots$	0.9921
p	0.0050	0.0083	0.0164	0.0460	0.4380	0.3732	0.0437	0.0159	$\cdots$	0.0033

Table 3.1: Snapshot of the phase estimation procedure, where $\overline{\omega}$ are the possible estimates and p are the associated probabilities to be extracted as output of the phase estimation routine.

We can see that the most highlighted values are the ones corresponding to $\overline{\omega} = 0.0312$ and $\overline{\omega} = 0.0390$, with the first as the most promising estimate value. This makes sense since our value ω is equal to 0.035. Thinking of the circle of phases represented in Figure 2.2 in Section 2.2.3, it is as if ω were between two dots, that in this case correspond to $x = 4$ and $x = 5$, but closer to the first one since $|0.035 - 0.0312| = 0.0038 < |0.035 - 0.0390| = 0.0039$. The distance between ω and the two values is very small and, indeed, also the corresponding probabilities are very similar, with the one associated to $x = 4$ higher with respect to the other, precisely because 0.0312 is closer to 0.035 with respect to 0.0390. Therefore, the most likely value for $\overline{\omega}$ is 0.0312. From these reported values, we can notice also another important thing relative to the number of qubits. Indeed, we have specified the accuracy $\epsilon = 0.1$ and the probability of failure $\gamma = 0.1$, and this, considering Equation 2.14, results in a number of qubits $n = 7$ which gives $2^7 = 128$ possible values. Given that, we can notice that each $\overline{\omega}$ value is distant $\frac{1}{2^7} = 0.0078$ from its neighbors. This means that, if we boost the probability of success or increase the accuracy ϵ, this results in a bigger n and consequently in a smaller value of $\frac{1}{2^n}$, allowing us to be even more precise in the estimates.
Moreover, with this experiment, we confirm what we report in the previous experiment.

Indeed, the red color scale in the table can be seen as the Gaussian shaped distribution shown in Figure 3.5. We can notice that, from the value associated with the highest probability (the strong red one), we go towards the tails of the distribution with probability values smaller and smaller (represented by the fading of the red color going on the left and on the right of the brighter one).

As a last note, it is important to remark that if $\omega = 0$, phase estimation returns $\overline{\omega} = 0$ with certainty since the probability value associated to $x = 0$ would be exactly 1, as reported in the theory section.

3.4. Consistent phase estimation

To implement this algorithm, we follow the logic described in Section 2.2.4, where we theoretically describe the consistent phase estimation algorithm. As we can notice in Source Code 3.5, if we do not provide the number of qubits n as a parameter, we compute it using the error ϵ and the probability of failure γ specified at the beginning, as in phase estimation. Then, we compute the constant $\zeta = \frac{\gamma}{n}$ (C in the code), $\epsilon' = \frac{\epsilon\zeta}{2}$, and $L = \left\lfloor \frac{2}{\zeta} \right\rfloor$. Now, we have to extract the shift s randomly from $\{1, \ldots, L\}$ and fix it to compute the sections of length ϵ as $[-1 - s\epsilon', 1 + \epsilon - s\epsilon']$ [46]. In these sections, we map the output of phase estimation that we execute internally with error ϵ'. In the code, if the user does not provide the shift value, we fix it to $\frac{L}{2} + 1$ (this specific value has no particular meaning, it is just to show that the shift value must be fixed during the whole execution). We compute each section using the *arange()* function of Numpy which returns evenly spaced values within the given interval. We set the step to ϵ since each section must be ϵ in length. After executing phase estimation with accuracy ϵ', we map the output of phase estimation (*pe_estimate* in the code) into the section to which it belongs using the *bisect()* functionality of Python that returns the position of a specific value in a sorted list (named *index* in the code). Using that index, we can retrieve the two extreme values of the section to which the value belongs and return the output as the middle value of the section found using *np.mean*() functionality.

```
1 def consistent_phase_estimation(omega, epsilon, gamma, n=None,
  ↪ shift=None):
2     """Official version of the Consistent Phase Estimation routine.
3
4     Parameters
5     ----------
```

```
    omega: float value.
        Value that you want to estimate. It must be in the range of
↪   [0,1).

    epsilon: float value.
        The error that you want to have in the estimates.

    gamma: float value.
        Probability of failure of the routine.

    n: int value, default=None.
        Number of qubits that you want to use in the routine. If None,
↪   this value is computed using the error and the probability of
↪   failure.

    shift: int value, default=None.
        If not None, the shift is fixed to this value, otherwise it is
↪   computed and fixed inside the function.

    Returns
    -------
    estimate: float value.
        Estimate of the omega value.
    """
    if n == None:
        n = int(np.ceil(np.log2(1 / epsilon)) + np.ceil(np.log2(2 + 1 /
        ↪  (2 * gamma))))
    C = gamma / n
    epsilon_prime = (epsilon * C) / 2
    L = np.floor(2 / C)
    # shift = random.randint(1, L)
    if shift == None:
        shift = int(L / 2) + 1
    intervals = np.arange(-1 - shift * epsilon_prime, 1 + epsilon - shift
    ↪  * epsilon_prime, epsilon)
    intervals = np.append(intervals, 1 + epsilon - shift * epsilon_prime)
```

```
36      pe_estimate = phase_estimation(omega=omega, epsilon=epsilon_prime,
        ↪  gamma=gamma)
37      index = bisect(intervals, pe_estimate)
38      section = (intervals[(index - 1)], intervals[index])
39      estimate = np.mean(section)
40
41      return estimate
```

Source Code 3.5: Implementation of the consistent phase estimation algorithm.

Also for this routine, we report a numerical simulation example mainly to highlight the difference with *phase estimation*.

Numerical simulation We execute many times consistent phase estimation to estimate the ω value and we check how the outputs, that the internally executed phase estimation returns, differ from the outputs of the consistent algorithm. We report the results in Figure 3.6 for better visualization and then we discuss it.

First a bit of context: the columns indexed by i are the different executions of the routine (in this case we repeat for 5 times the experiment). In the rows, we report the estimates of phase estimation (P.E), all indexed by the ω value that we are estimating. The ticked line below the table, instead, represents all the sections computed in the consistent phase estimation using the interval $[-1-s\epsilon', 1+\epsilon-s\epsilon']$, after fixing the shift s and the accuracy ϵ and ϵ'. For this experiment, we choose $\epsilon = \gamma = 0.1$ which make $\epsilon' = 0.00071$.
Let us focus on $\omega = 0.1$. As we can see, if we repeat the consistent phase estimation procedure 5 times with accuracy ϵ, the internal phase estimation, executed with accuracy $\epsilon' << \epsilon$ (as this is one of the constraints of the consistent phase estimation algorithm showed in Section 2.2.4), reports 3 out of 5 different results. Indeed, if we execute phase estimation multiple times, since all the possible outputs have a probability different from zero to be extracted as estimates, we can get different results. To highlight better this fact, for each row, we fill cells with equal values with the same color.
Then, all the phase estimation outputs are mapped to the section they belong to. As we can notice, they are all mapped in the section $[0.0492, 0.1492]$ (notice also that the section length is equal to $\epsilon = 0.1$). In this way, the result of consistent phase estimation is always the same. Indeed, as we have also seen from the theory, we extract the intermediate value of the section as the final estimate that, in the case of $\omega = 0.1$, is $\overline{\omega}_{0.1}$=0.09928, which is a good estimate being ϵ-close to ω. We want to highlight that not necessarily the estimate of consistent phase estimation is better than a possible output of the phase

estimation. Indeed, if we look at the possible phase estimation outputs, we see, for example, that 0.10003 is closer to $\omega = 0.1$ with respect to 0.09928. This is because we perform phase estimation with higher accuracy ϵ', or equivalently lower error, with respect to the accuracy ϵ of the consistent version. We repeat: what we want to show is the fact of being *consistent*. So, over the 5 different executions, we obtain the same output, which is a good output being ϵ-close to the true value. The fact to be consistent also depends on the shift s, that we fix at the beginning of the experiment and, in this case, seems to be a good shift. For $\omega = 0.35$ the same reasoning applies.

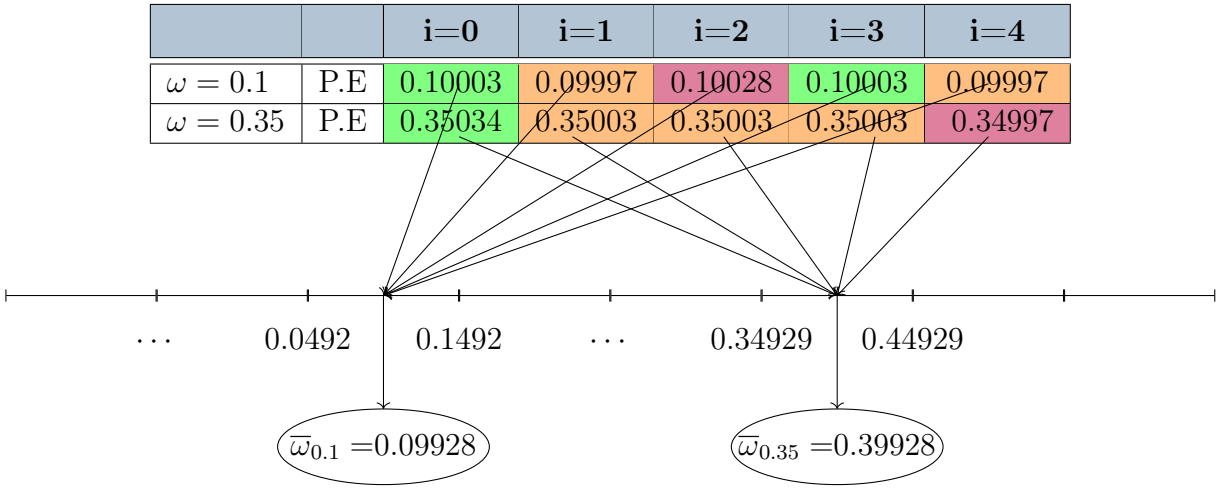


Figure 3.6: Numerical example of consistent phase estimation.

3.5. Amplitude estimation

From Section 2.2.5, we know that amplitude estimation is a direct descendant of phase estimation. Therefore, also their implementations are very similar. The first difference we can notice looking at the code below is that the number of iterations M can be a generic integer specified by the user while, in the phase estimation, we can specify the number of qubits n and the number of iterations is calculated as 2^n. In amplitude estimation, we can see that there is the parameter *nqubit* which, if set to True, causes the routine to return the number of qubits used. Since we want to be able to express the number of iteration M in the function of the error ϵ and since in Brassard et al. [9] (which is our main source of information for this routine) is only specified that M can be a generic integer ≥ 1, we use the definition of M reported by Allcock et al. [2] in their implementation of amplitude estimation, that is

$$M = \left\lceil \frac{\pi}{2\epsilon}(1 + \sqrt{1 + 4\epsilon}) \right\rceil . \tag{3.1}$$

In this way, we can specify the error that we expect in the estimate and the procedure itself computes the number of iterations needed to achieve that error in its estimate. As we already said in Section 2.2.5, we can use median evaluation (whose implementation is reported in the next section) to boost the probability of success of this routine. Indeed, by specifying the probability of failure γ (*gamma* parameter in the code), we implicitly perform median evaluation which executes recursively amplitude estimation. We decide to include this γ parameter in amplitude estimation since we think that it is very practical for a user who wants to boost the probability of failure of amplitude estimation. Indeed, in this way, the user does not have to explicitly execute median evaluation but just specifies the γ parameter on which depends the number of iterations Q of median evaluation as we will see in the next section.
Looking at the code, we can notice that the core part is almost equal to the phase estimation routine. The only difference here is that, following the algorithm reported by Brassard et al. [9], we use the minimal wrap around distance (made explicit in Definition 2) that in the code is defined as *amplitude_ est_ dist()*. The function takes in input the true value that we want to estimate (*theta_ a*, which is computed as $\sin^{-1}(\sqrt{a})$ where a is the input value) and the possible estimate value (defined as j/M as in phase estimation). Then, if the computed distance is different from 0, we compute the probability for each value to be extracted as an estimate using the formula reported in Equation 2.18 otherwise we set the probability to 1 since it means that we can exactly approximate the value with certainty. Finally, once extracted with a weighted random choice the estimate of θ_a, we return the output of amplitude estimation as $\overline{a} = \sin^2(\overline{\theta}_a)$.

For amplitude estimation, we show that, as a proof-example of the correctness of our routine, if we want to estimate $a = 0$, we obtain with certainty $\overline{a} = 0$. If we want to estimate $a = 1$, we obtain $\overline{a} = 1$ only if the number of iterations M is even, as reported in Theorem 2.10.

```
1     def amplitude_estimation(a, epsilon=0.01, gamma=None, M=None,
      ↪  nqubit=False):
2     """ Official version of the amplitude estimation function.
3
4     Parameters
5     ----------
6     a: float or int value.
7          Value that has to be estimated by the amplitude estimation. It
  ↪   must be in the range of [0,1].
```

```
8
9       epsilon: float value, default=0.01
10          Error that you want to tolerate in the estimate of amplitude
    ↪   estimation.
11
12      gamma: float value, default=None.
13          It represent the probability of failure of amplitude estimation.
    ↪   If specified, median evaluation is performed
14          to boost the probability of success of this routine.
15      M: int value, default=None.
16          The number of iteration executes in the routine.
17
18      nqubit: bool value, default=False.
19          If True, the routine returns also the number of qubits used.
20
21
22      Returns
23      -------
24      a_tilde: float value.
25          Estimate of the probability of measure a "good" state.
26      """
27      if gamma:
28          return median_evaluation(amplitude_estimation, gamma=gamma,
            ↪   Q=None, a=a, epsilon=epsilon, M=M, nqubit=nqubit)
29      if M == None:
30          M = (math.ceil((np.pi / (2 * epsilon)) * (1 + np.sqrt(1 + 4 *
            ↪   epsilon))))
31          n_qubits = np.ceil(np.log2(M))
32      else:
33          n_qubits = np.ceil(np.log2(M))
34      theta_a = math.asin(np.sqrt(a))
35      p = []
36      theta_j = []
37      for j in range(M):
38          theta_est = np.pi * j / M
39          theta_j.append(theta_est)
40          distance = amplitude_est_dist(theta_est / np.pi, theta_a / np.pi)
```

```
41          if distance != 0:
42              p_aj = np.abs(math.sin(M * distance * np.pi) / (M *
                ↪  math.sin(distance * np.pi))) ** 2
43          else:
44              p_aj = 1
45          p.append(p_aj)
46      theta_tilde = random.choices(theta_j, weights=p, k=1)[0]
47      if nqubit:
48          return theta_tilde, n_qubits, M
49      a_tilde = np.sin(theta_tilde) ** 2
50      return a_tilde
```

Source Code 3.6: Implementation of the amplitude estimation algorithm.

Proof-example Inside amplitude estimation, we firstly compute $\theta_a = sin^{-1}(\sqrt{a})$ and then we estimate $\overline{\theta}_a$ which gives us the final estimate $\overline{a} = \sin^2(\overline{\theta}_a)$. For $a = 0$ and $a = 1$, we obtain respectively $\theta_a = 0$ and $\theta_a = 1.5707$ that need to be estimated with phase estimation procedure. Now, one might argue that phase estimation over $\theta_a = 1.5707$ cannot be executed since it is a value greater than 1 and we have the constraint that $\theta_a \in [0, 1)$. But we have to remember that the shape of the argument to be estimated is $\theta_a = \frac{y\pi}{M}$ (we can check it also from the last step in Algorithm 2.3 and from line 40 in Source Code 3.6). Therefore, we divide it by π obtaining a value less than 1. Indeed, considering first $\theta_a = 0$, dividing it by π does not change anything and we know that phase estimation estimates $\overline{\theta}_a = 0$ with certainty. Instead, by dividing $\theta_a = 1.5707$ by π, we obtain 0.50 which is the value that needs to be estimated by phase estimation. In this case, to be sure that phase estimation returns as estimate 0.50 with certainty and, consequently, that amplitude estimation returns 1 with certainty, we need to consider what Theorem 2.10 enunciates about amplitude estimation. It states that, if we want to estimate $a = 1$, only if the number of iterations M is an *even* number, we can obtain the estimate $\overline{a} = 1$ with certainty. This is very simple to show: if we take any even number, there is always an integer $y \in [0, \ldots, M-1]$, which makes $\frac{y}{M} = 0.5$. Therefore, the distance between that specific y value divided by M and our value that needs to be estimated is zero, resulting in an exact estimate. Just to clarify better, if we take $M = 32$ (remembering that in amplitude estimation we can set M to be any integer ≥ 1), we see that, for $y = 16$, we obtain $\frac{y}{M} = 0.5$. We approximate our initial $\theta_a = 0.5$ with $\frac{16}{32} = 0.5$ with certainty since the distance between those values is zero and so the probability to approximate θ_a with $\frac{16}{32}$ is equal to 1. This is not true if M is an *odd* value. Indeed, there

is no integer $y \in [0, \ldots, M-1]$ which makes the fraction $\frac{y}{M} = 0.5$ exactly. Consequently, phase estimation does not estimate $\overline{\theta}_a = 0.5$ with certainty. Once phase estimation is concluded, we reconstruct the final estimate by multiplying the phase estimation output by π. For how the routine is built, we obtain the two values to $\overline{\theta}_a = 0$ and $\overline{\theta}_a = 1.5707$ with probability 1. This guarantees us that, when we compute the output of amplitude estimation, as $\overline{a} = sin^2(\overline{\theta}_a)$, we get back 0 and 1, with probability 1.

3.6. Median evaluation

For what concerns the implementation choices for the median evaluation algorithm [2] (see Section 2.2.6 for more details), we basically think about which is the main feature of this routine. To recap in short, median evaluation has the goal of boosting the probability of success of quantum algorithms.
In our framework, we generalize this function such that every quantum routine can use median evaluation as a bracket tool to improve the probability of success and therefore the precision of the estimates. The logic behind our implementation is very simple as we can see in the source code reported below. We just take a quantum routine (called *func* in the code) and repeat its procedure Q times passing the necessary parameters *args* and *kwargs* (for instance, for amplitude estimation the parameters *args* would be the value to estimate a and the error ϵ). Q is a parameter that can be specified by the user or it can be computed by specifying the failure probability γ in this way

$$Q = \left\lceil \frac{\log(1/\gamma)}{2(8/\pi^2 - \frac{1}{2})^2} \right\rceil_{odd} \tag{3.2}$$

which is basically the formula in Theorem 2.13 in Section 2.2.6, where $\lceil z \rceil_{odd}$ denotes the smallest odd integer greater or equal to z (for major details check Subroutine 4 in [2]). In this way, we obtain Q outputs from which we extract and report the median value as the output of the median evaluation using the median Numpy functionality. Basically, the key concept is that repeating a probabilistic procedure several times increases the probability of success and the reliability of the experiment.

```
1  def median_evaluation(func, gamma=0.1, Q=None, *args, **kwargs):
2      """Median evaluation.
3
4      Parameters
5      ----------
```

```
    func: Callable.
        The quantum routine that you want to execute Q times.

    gamma: float value, default=0.1.
        The probability of failure necessary to compute the number of
    ↪ iteration Q if the user does not provide it.

    Q: int value, default=None.
        Number of iterations to execute func.

    Returns
    -------
    final_estimate : float value.
        Median estimation of the quantum algorithm passed as argument.
    """
    if Q == None:
        z = np.log(1 / gamma) / (2 * (8 / np.pi ** 2 - 0.5) ** 2)
        Q = np.ceil(z)
        if Q % 2 == 0:
            Q += 1
    estimates = [func(*args, **kwargs) for _ in range(int(Q))]
    final_estimate = np.median(estimates)
    return final_estimate
```

Source Code 3.7: Implementation of median evaluation algorithm.

3.7. Inner product estimation

In the inner product estimation algorithm, as we have seen in Section 2.2.7, we use median evaluation to boost the probability of success of amplitude estimation. To implement this routine, we refer to the Algorithm 2.4 that we report from Allcock et al. [2]. Given two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, we just compute

$$a = \frac{\|\boldsymbol{x}\|^2 + \|\boldsymbol{y}\|^2 - 2\langle \boldsymbol{x}, \boldsymbol{y} \rangle}{2(\|\boldsymbol{x}\|^2 + \|\boldsymbol{y}\|^2)} \tag{3.3}$$

and

$$\epsilon_a = \frac{\epsilon \max\{1, |\langle \boldsymbol{x}, \boldsymbol{y} \rangle|\}}{\|\boldsymbol{x}\|^2 + \|\boldsymbol{y}\|^2} \tag{3.4}$$

as specified in Algorithm 2.4. The value reported in Equation 3.3 is the one that we want to estimate with amplitude estimation with an error ϵ_a. Then, since the IPE algorithm expects to perform median evaluation, we just perform amplitude estimation with the probability of failure γ which is a parameter of IPE function. Indeed, we have seen that, in our framework, specifying the parameter γ in amplitude estimation corresponds to boosting the probability of success of amplitude estimation by executing median evaluation. Once performed median evaluation, we obtain an estimate $\overline{a}$ that we use to compute the estimation of the inner product as $\overline{\langle \boldsymbol{x}|\boldsymbol{y} \rangle} = (\|\boldsymbol{x}\|^2 + \|\boldsymbol{y}\|^2) * (1 - 2 * \overline{a})$.

```
1   def ipe(x, y, epsilon, gamma=0.1):
2     """Official version of the Robust Inner Product Estimation Routine
      ↪  ((R)IPE).
3     Parameters
4     ----------
5     x: ndarray of shape (n,).
6         One of the two vector needed to compute the inner product.
7
8     y: ndarray of shape (n,).
9         The second vector needed to compute the inner product.
10
11    epsilon: float value, default=None.
12        Error that you want to have in the estimate of the inner product.
13
14    gamma: float value, default=0.1.
15        Probability of failure to insert in median evaluation to compute
    ↪  the number of iteration Q that amplitude estimation must be
    ↪  performed.
16
17    Returns
18    -------
19    s: float value.
20        Estimate of the inner product between x and y vectors.
21    """
```

```
22      a = (np.linalg.norm(x) ** 2 + np.linalg.norm(y) ** 2 - 2 *
        ↪  np.inner(x, y)) / (2 * (
23              np.linalg.norm(x) ** 2 + np.linalg.norm(y) ** 2))
24      epsilon_a = epsilon * max(1, np.abs(np.inner(x, y))) /
        ↪  (np.linalg.norm(x) ** 2 + np.linalg.norm(y) ** 2)
25      a_tilde = amplitude_estimation(a=a, gamma=gamma, epsilon=epsilon_a)
26      s = (np.linalg.norm(x) ** 2 + np.linalg.norm(y) ** 2) * (1 - 2 *
        ↪  a_tilde) / 2
27      return s
```

Source Code 3.8: Implementation of the inner product estimation algorithm.

3.8. q-PCA

Following the structure of *Sklearn* API [34], we extend the PCA class by implementing state-of-the-art quantum procedures [7] that allow to classically estimate the top-k singular values, factor scores, and factor score ratios of a $n \times d$ matrix in time poly-logarithmic in nd and the most relevant singular vectors sub-linearly in nd. These procedures can be used to solve eigenproblems and problems that are classically solved with Singular Value Decomposition (SVD). As we have seen in Section 1.1.1, PCA is strictly related to SVD, therefore we can apply these novel quantum procedures to classically simulate this machine learning algorithm. Using these quantum routines we have theoretical guarantees about their running time, their failure probability, and the error bound of the quantum algorithms over which the routines are built. Now we describe the quantum procedures that allow simulating q-PCA in our framework.
The following algorithms are all implemented inside the *fit()* function of q-PCA, indeed they are related to the extraction of the model. The other basic methods that we have "quantized" are the *transform()* and *inverse_transform()* ones.
With our implementation, the q-PCA class can be used both to execute classical and quantum machine learning experiments.
It is important to note that in the quantum machine learning algorithms that require phase estimation, we always use the consistent version of phase estimation.

Quantum check on factor score ratios' sum

The first algorithm that we introduce is the one that checks the factor score ratios' sum (Algorithm 2 reported in Bellante et al. [7]). This algorithm estimates the percentage of

variance given by the sum of the factor score ratio relative to singular values that are greater or equal than a threshold parameter θ. With this quantum algorithm, given an

Algorithm 3.1 Quantum check on factor score ratios' sum [7].

Input: Quantum access to matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ with singular value decomposition $\boldsymbol{X} = \sum_i^r \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i$ and rank r. A threshold parameter θ. Two precision parameter $\epsilon, \eta \in \mathbb{R}_{>0}$ such that the greatest singular value smaller than θ is not more than ϵ distant to it.
Output: An estimate $\overline{p}$ of the factor score ratios' sum $p = \sum_{i:\overline{\sigma}_i \geq 0} \lambda_i^{(i)}$.

1. Prepare the state $\frac{1}{\|\boldsymbol{X}\|_F} \sum_i^n \sum_j^d x_{ij} |i\rangle |j\rangle$.
2. Apply SVE with precision ϵ to get $\frac{1}{\sqrt{\sum_j^r \sigma_j^2}} \sum_i^r \sigma_i |\boldsymbol{u_i}\rangle |\boldsymbol{v_i}\rangle |\overline{\sigma}_i\rangle$.
3. Append a quantum register $|0\rangle$ to the state and set it to $|1\rangle$ if $\overline{\sigma}_i < \theta$
4. Uncompute the SVE

$$\frac{1}{\sqrt{\sum_j^r \sigma_j^2}} \sum_{i:\overline{\sigma}_i \geq \theta} \sigma_i |\boldsymbol{u_i}\rangle |\boldsymbol{v_i}\rangle |0\rangle + \frac{1}{\sqrt{\sum_j^r \sigma_j^2}} \sum_{i:\overline{\sigma}_i < \theta} \sigma_i |\boldsymbol{u_i}\rangle |\boldsymbol{v_i}\rangle |1\rangle$$

.

5. Perform amplitude estimation with precision η on the last register being $|0\rangle$, to estimate $p = \frac{\sum_{i:\overline{\sigma}_i \geq \theta} \sigma_i^2}{\sum_j^r \sigma_j^2} = \sum_{i:\overline{\sigma}_i \geq \theta} \lambda^{(i)}$.

input matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, we can estimate $p = \sum_{i:\overline{\sigma}_i \geq \theta} \lambda^{(i)}$, where $|\sigma_i - \overline{\sigma}_i| \leq \epsilon$, to relative error η, in time $\widetilde{O}\left(\frac{\mu(\boldsymbol{X})}{\epsilon\eta\sqrt{p}}\right)$ (we remember that $\mu(\boldsymbol{X}) = \min_{p \in [0,1]}(\|\boldsymbol{X}\|_F, \sqrt{s_{2p}(\boldsymbol{X}) s_{2(1-p)}(\boldsymbol{X}^T)})$ and $s_p(\boldsymbol{X}) = \max_i \|\boldsymbol{x}_{i,\cdot}\|_p^p$). Going more into the details of the algorithm, as we can see, it is principally based on Singular Value Estimation (SVE), conditional rotations, and amplitude estimation.
To implement it, first we classically decompose the input data matrix $\boldsymbol{X}$ using *linalg.svd()* function of Numpy. Then, we apply SVE: for each singular value σ_i, we get

$$\theta_i = 2 \arccos\left(\frac{\sigma_i}{\mu(\boldsymbol{X})}\right) \tag{3.5}$$

from Equation 2.22. We put all these θ_i values in input to the consistent phase estimation procedure to estimate the corresponding $\overline{\theta}_i$ with error ϵ specified at the beginning of the procedure. By reversing the formula reported in Equation 3.5, we get the corresponding estimates of singular values

$$\overline{\sigma}_i = \cos\left(\frac{\overline{\theta}_i}{2}\right) \mu(\boldsymbol{X}). \tag{3.6}$$

Then, we select the singular values σ_i using the indices i such that $\overline{\sigma}_i \geq \theta$ (with θ defined at the beginning). Finally, we can compute $p = \frac{\sum_{i:\overline{\sigma}_i \geq \theta} \sigma_i^2}{\sum_j^r \sigma_j^2}$ that we input to the amplitude estimation to estimate it with error η.

Quantum binary search for the singular value threshold

In the extraction of q-PCA, this algorithm is very useful and we use it many times. In PCA, generally, we specify a certain percentage of variance that we want the principal components to retain. This quantum algorithm, given that percentage of variance p, allows estimating a threshold θ such that the estimated singular values greater than θ are the ones corresponding to the components that retain p variance. Therefore, as we can show later, using this algorithm together with the top-k singular vector extraction (described in Algorithm 3.3) is the core part for the extraction of the principal components and therefore of q-PCA model.

So, given a quantum access to a matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ and considering η, ϵ as precision parameters and θ as the smallest singular value to consider, the following quantum algorithm runs in time $\widetilde{O}\left(\frac{\mu(\boldsymbol{X}) \log(\mu(\boldsymbol{X})/\epsilon)}{\epsilon\eta}\right)$ and outputs an estimate θ such that $|p - \sum_{i:\overline{\sigma}_i \geq \theta} \lambda^{(i)}| \leq \eta$, where $|\overline{\sigma}_i - \sigma_i| \leq \epsilon$, or detects whether such θ does not exists.

We report this procedure in Algorithm 3.2. We can notice that it works exactly like a classical binary search, where the value that we are searching for is τ which at the end, by multiplying it for $\mu(\boldsymbol{A})$, gives the right magnitude of θ. Starting from $\tau = 0.5$ (in step 4), we decrease τ if amplitude estimation outputs $\overline{p}_\tau < p$, otherwise we increase it, until we found the right τ which corresponds to the condition in step 10 (it can also happen not to find it if the initial parameters are not compliant).

For what concerns our implementation choices, we can notice that steps 6,7,8, and 9 are those performed also in the Algorithm 3.1. So, once done the initial checks in step 2 and 3, we repeat for $\log\left(\frac{\mu(\boldsymbol{X})}{\epsilon}\right)$ times the procedure reported in Algorithm 3.1, with precision parameters $\epsilon = \epsilon/\mu(\boldsymbol{X})$ and $\eta = \eta/2$. Pay attention: what in Algorithm 3.1 is called θ, now corresponds to τ. So, by performing the procedure described before, we get an estimate $\overline{p}_\tau$. Finally, we check if $|\overline{p}_\tau - p| \leq \eta/2$: if it is, we return $\theta = \tau\mu(\boldsymbol{X})$, otherwise we update τ according to whether $\overline{p}_\tau \leq p$ or not, as in steps 10 and 11, and we perform another iteration of the procedure.

We think that can be useful to report a simulation example to show better how this algorithm works because it is used many times in our experiments.

Simulation example Let us suppose to have a training set of samples over which we perform PCA, so implicitly we perform SVD extracting all the singular values, right/left singular vectors, and factor scores. Now, let us suppose we want to estimate θ using Algorithm 3.2, considering $\epsilon = 8, \eta = 0.2$, and $p = 0.30$, which means that we want to retain the 30% of the total variance. Let us also consider that $\mu(\boldsymbol{X}) = 362.11$. What happens is the following: starting from $\tau = 0.5$ (that corresponds to θ of Algorithm

Algorithm 3.2 Quantum binary search [7].

Input: Quantum access to matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ with singular value decomposition $\boldsymbol{X} = \sum_i^r \sigma_i \boldsymbol{u_i} \boldsymbol{v_i}$ and rank r. The desired amount of factor score ratio sum $p \in [0, 1]$. Two precision parameters $\epsilon, \eta \in \mathbb{R}_{>0}$.
Output: A threshold θ such that $|p - \sum_{i:\overline{\sigma}_i \geq \theta} \lambda_i^{(i)}| \leq \eta$ where $|\overline{\sigma}_i - \sigma_i| \leq \epsilon$ or -1 if no such θ exists.

1: Let $l = 0$ and $u = 1$ be upper and lower bounds for the binary search.
2: If $|1 - p| \leq \eta$, then return $\theta = 0$.
3: If $|0 - p| \leq \eta$, then return $\theta = \mu(\boldsymbol{X})$.
4: Initialize $\tau = (l + u)/2$.
5: **for** $O\left(\log(\frac{\mu(\boldsymbol{X})}{\epsilon})\right)$ *times* **do**
6: Prepare the state $|\boldsymbol{X}\rangle$ and apply SVE to get $\frac{1}{\sqrt{\sum_j^r \sigma_i^2}} \sum_i^r \sigma_i |\boldsymbol{u_i}\rangle |\boldsymbol{v_i}\rangle |\overline{\sigma}_i\rangle$ so that $|\overline{\sigma}_i - \frac{\sigma_i}{\mu(\boldsymbol{X})}| < \frac{\epsilon}{\mu(\boldsymbol{X})}$.
7: Append a quantum register $|0\rangle$ to the state and set it to $|1\rangle$ if $\overline{\sigma}_i < \tau$.
8: Uncompute the SVE

$$\frac{1}{\sqrt{\sum_j^r \sigma_j^2}} \sum_{i:\overline{\sigma}_i \geq \tau} \sigma_i |\boldsymbol{u_i}\rangle |\boldsymbol{v_i}\rangle |0\rangle + \frac{1}{\sqrt{\sum_j^r \sigma_j^2}} \sum_{i:\overline{\sigma}_i < \tau} \sigma_i |\boldsymbol{u_i}\rangle |\boldsymbol{v_i}\rangle |1\rangle$$

.
9: Perform amplitude estimation with precision η on the last register being $|0\rangle$, to estimate $p_\tau = \frac{\sum_{i:\overline{\sigma}_i \geq \tau} \sigma_i^2}{\sum_j^r \sigma_j^2} = \sum_{i:\overline{\sigma}_i \geq \tau} \lambda^{(i)}$, such that $|\overline{p}_\tau - p| \leq \eta/2$.
10: If $|\overline{p}_\tau - p| \leq \eta/2$, then return $\theta = \tau\mu(\boldsymbol{X})$.
11: If $\overline{p}_\tau < p$, then set $u = \tau$ and set $l = \tau$ otehrwise.
12: Update $\tau = (u + l)/2$.
13: **end for**
14: Return -1.

3.1), we execute Algorithm 3.1 that returns an estimate of p_τ, using $\epsilon = \epsilon/\mu(\boldsymbol{X})$ and $\eta = \eta/2$. We get $\overline{p}_\tau = 0.0$ and since $0.0 < 0.30$, we set $u = \tau = 0.5$. Therefore, we update $\tau = (0.5 + 0)/2 = 0.25$. We repeat the same procedure but, instead of $\tau = 0.5$, we estimate $p_{\tau=0.25}$. We obtain $\overline{p}_\tau = 0.52$ that is greater than $p = 0.30$. Setting $l = 0.25$, for the next iteration we have $\tau = 0.375$, which leads to an estimate $\overline{p}_\tau = 0.34$. Therefore, the check $|\overline{p}_\tau - p| \leq \eta/2$ is successful and we return $\theta = 0.375 * 362.11 = 135.7$.

As we will see, we use many times this algorithm, principally in combination with the following algorithm that we are going to describe.

Quantum top-k singular vectors extraction

Up to now, the algorithms reported refer to procedures that have to do with the most relevant singular values, factor scores, and factor score ratios of the input matrix $\boldsymbol{X}$. This new algorithm, instead, allows us to extract the corresponding right/left singular vectors. The initial conditions are almost always the same. So, let us consider to have a quantum access to the matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, with singular value decomposition $\boldsymbol{X} = \sum_i^r \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T$ with r rank of the matrix $\boldsymbol{X}$. Fixed the precision parameter $\delta > 0$ for the singular vectors, $\epsilon > 0$ for the singular values and considering $\theta > 0$ a threshold such that there exist k singular values greater than θ, we define also $p = \frac{\sum_{i:\overline{\sigma}_i \geq \theta} \sigma_i^2}{\sum_i^r \sigma_j^2}$. The algorithm that we are going to show estimates:

- the top-k left singular vectors $\boldsymbol{u}_i$ of $\boldsymbol{X}$ with unit vectors $\overline{\boldsymbol{u}}_{\boldsymbol{i}}$ such that $\|\boldsymbol{u}_i - \overline{\boldsymbol{u}}_i\|_2 \leq \delta$ with probability at least $1 - 1/poly(n)$, in time $\widetilde{O}\left(\frac{\|\boldsymbol{X}\|\mu(\boldsymbol{X})kn}{\theta\sqrt{p}\epsilon\delta^2}\right)$.
- the top-k right singular vectors $\boldsymbol{v}_i$ of $\boldsymbol{X}$ with unit vectors $\overline{\boldsymbol{v}}_{\boldsymbol{i}}$ such that $\|\boldsymbol{v}_i - \overline{\boldsymbol{v}}_i\|_2 \leq \delta$ with probability at least $1 - 1/poly(d)$, in time $\widetilde{O}\left(\frac{\|\boldsymbol{X}\|\mu(\boldsymbol{X})kd}{\theta\sqrt{p}\epsilon\delta^2}\right)$.
- the top-k singular values σ_i, factor score λ_i, and factor score ratios $\lambda^{(i)}$ of $\boldsymbol{X}$ to precision $\epsilon, 2\epsilon\sqrt{\lambda_i}$, and $\epsilon\frac{\sigma_i}{\|\boldsymbol{X}\|_F^2}$ respectively, with probability at least $1 - 1/poly(d)$, in time $\widetilde{O}\left(\frac{\|\boldsymbol{X}\|\mu(\boldsymbol{X})k}{\theta\sqrt{p}\epsilon}\right)$.

We implement this algorithm in the following way: once classically decomposed the input data matrix $\boldsymbol{X}$, we estimate all singular values using SVE such that $|\overline{\sigma}_i - \sigma_i| \leq \epsilon$. To do this, as seen in the quantum factor score ratios' sum check algorithm, we use Equation 3.5 to get, for each singular value σ_i, its corresponding θ_i. Then, we apply consistent phase estimation to get the estimate $\overline{\theta}_i$ with error ϵ. We use Equation 3.6 to retrieve the corresponding estimates of the singular values $\overline{\sigma}_i$. Then, we select the top-k right/left singular vectors by determining the corresponding estimated singular values greater than θ. By applying vector state tomography over these singular vectors, we estimate the top-k right/left singular vectors with error δ. Finally, we return all the estimates. As previously said, in the extraction of q-PCA model, this algorithm is the one of the most used together with the quantum binary search one. Indeed, given the percentage of variance p that we want the principal components to retain, we are able, using these two algorithms, to extract the corresponding principal components that retain p variance. We report a possible use case of this algorithm, combined with Algorithm 3.2.

Simulation example First, we can use quantum binary search to estimate θ, given a specified percentage of variance p. Then, the estimated θ, which corresponds to the smallest singular value that needs to be considered to retain that percentage of variance,

Algorithm 3.3 Quantum top-k singular vectors extraction [7].

Input: Quantum access to matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$. Two precision parameters $\epsilon, \delta \in \mathbb{R}_{>0}$ and a threshold $\theta > 0$.

Output:

- the top-k left singular vectors $\boldsymbol{u}_i$ of $\boldsymbol{X}$ with unit vectors $\overline{\boldsymbol{u}}_{\boldsymbol{i}}$ such that $||\boldsymbol{u}_i - \overline{\boldsymbol{u}}_i||_2 \leq \delta$.
- the top-k right singular vectors $\boldsymbol{v}_i$ of $\boldsymbol{X}$ with unit vectors $\overline{\boldsymbol{v}}_{\boldsymbol{i}}$ such that $||\boldsymbol{v}_i - \overline{\boldsymbol{v}}_i||_2 \leq \delta$.
- the top-k singular values σ_i, factor score λ_i, and factor score ratios $\lambda^{(i)}$ of $\boldsymbol{X}$ to precision $\epsilon, 2\epsilon\sqrt{\lambda_i}$, and $\epsilon \frac{\sigma_i}{\|\boldsymbol{X}\|_F^2}$.

1: Prepare the state $\frac{1}{||\boldsymbol{X}||_F} \sum_i^n \sum_j^d x_{ij} |i\rangle |j\rangle$.
2: Apply SVE to get $\frac{1}{\sqrt{\sum_j^r \sigma_j^2}} \sum_i^r \sigma_i |\boldsymbol{u_i}\rangle |\boldsymbol{v_i}\rangle |\overline{\sigma}_i\rangle$, where $|\sigma_i - \overline{\sigma}_i| \leq \epsilon$.
3: Append a quantum register $|0\rangle$ to the state and set it to $|1\rangle$ if $|\overline{\sigma}_i\rangle < \theta$.
4: Perform amplitude amplification for $|0\rangle$, to get the state $\frac{1}{\sqrt{\sum_j^k \sigma_j^2}} \sum_i^k \sigma_i |\boldsymbol{u}_i\rangle |\boldsymbol{v}_i\rangle |\overline{\sigma}_i\rangle$.
5: Append a second ancillary register $|0\rangle$ and perform the controlled rotation $\frac{C}{||\boldsymbol{X}^{(k)}||_F} \sum_i^k \frac{\sigma_i}{\overline{\sigma}_i} |\boldsymbol{u}_i\rangle |\boldsymbol{v}_i\rangle |\overline{\sigma}_i\rangle |0\rangle + \frac{1}{||\boldsymbol{X}^{(k)}||_F} \sum_i^k \sqrt{1 - \frac{C^2}{\overline{\sigma}_i^2}} |\boldsymbol{u}_i\rangle |\boldsymbol{v}_i\rangle |\overline{\sigma}_i\rangle |1\rangle$, where C is a normalization constant.
6: Perform again amplitude amplification for $|0\rangle$ to get the uniform superposition $\frac{1}{\sqrt{k}} \sum_i^k |\boldsymbol{u}_i\rangle |\boldsymbol{v}_i\rangle |\overline{\sigma}_i\rangle$.
7: Measure the last register and, according to the measured $|\overline{\sigma}_i\rangle$, apply state-vector tomography on $|\boldsymbol{u}_i\rangle$ for the i^{th} left singular vector or on $|\boldsymbol{v}_i\rangle$ for the right one.
8: Repeat 3-9 until the tomography requirements are met.
9: Output the k singular vectors $\boldsymbol{u}_i$ or $\boldsymbol{v}_i$ and, optionally, the singular values $\overline{\sigma}_i$.

is used as input parameter in top-k singular vector extraction algorithm. In this way, we estimate the singular values and the ones greater than θ correspond to the top-k singular values needed to retain the initial percentage of variance p. Clearly, in these algorithms, the error parameters are fundamental and influence the final estimates. For instance, a too large ϵ error could cause the quantum binary search to extract a wrong θ which reflects in selecting the wrong top-k singular vectors in top-k singular vectors extraction. As done for the previous algorithm, we report a numerical example to show the combined procedure between the quantum binary search and the top-k singular vector extraction algorithm. We want to estimate the top-k right singular vectors that explain the 30% of the total variance. Therefore, we can estimate θ using the quantum binary search algorithm. Using the example reported before, we found $\theta = 135.7$. Then, we use this value in the quantum top-k singular vectors extraction algorithm. We note that the estimated singular values, retrieved with SVE, are 2. This means that the top-2 components corresponding to those singular values are enough to retain the 30% of the variance. Indeed, we see that the classical explained variance ratios of those two components are $\approx 0.17\%$ and $\approx 0.14\%$, resulting in $\approx 30\%$ of total explained variance for the two first principal

components. Therefore, we verify that by extracting the PCA model using quantum algorithms, we are consistent with the classical case since in both cases we retain the first two principal components to explain 30% of the variance.

Quantum least-k singular vectors extraction

The procedure described in Algorithm 3.3 can be slightly modified to extract the least-k singular values and the corresponding singular vectors. This can be done by performing amplitude amplification, in step 5, for the state $|1\rangle$. This means that we consider all the estimated singular values, computed with SVE, that are less than θ. We report only the theorem that describes the procedure since the pseudocode of the algorithm is almost equal to the one of Algorithm 3.3 as well as the implementation.

Theorem 3.1 (Least-k singular vectors extraction)**.** *Let there be efficient quantum access to a matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, with singular value decomposition $\boldsymbol{X} = \sum_i^r \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T$. Let $\delta > 0$ be a precision parameter for the singular vectors, $\epsilon > 0$ a precision parameter for the singular values, and $\theta > 0$ be a threshold such that $\boldsymbol{X}$ has k singular values less than θ. Define $p = \frac{\sum_{i:\overline{\sigma}_i < \theta} \sigma_i^2}{\sum_j^r \sigma_j^2}$. There exist quantum algorithms that estimate:*

- *The least k left singular vectors $\boldsymbol{u}_i$ of $\boldsymbol{X}$ with unit vectors $\overline{\boldsymbol{u}}_i$ such that $\|\boldsymbol{u}_i - \overline{\boldsymbol{u}}_i\|_2 \leq \delta$ with probability at least $1 - 1/poly(n)$, in time $\widetilde{O}\left(\frac{\theta}{\sigma_{min}} \frac{1}{\sqrt{p}} \frac{\mu(\boldsymbol{X})}{\epsilon} \frac{kn}{\delta^2}\right)$.*
- *The least k right singular vectors $\boldsymbol{v}_i$ of $\boldsymbol{X}$ with unit vectors $\overline{\boldsymbol{v}}_i$ such that $\|\boldsymbol{v}_i - \overline{\boldsymbol{v}}_i\|_2 \leq \delta$ with probability at least $1 - 1/poly(d)$, in time $\widetilde{O}\left(\frac{\theta}{\sigma_{min}} \frac{1}{\sqrt{p}} \frac{\mu(\boldsymbol{X})}{\epsilon} \frac{kd}{\delta^2}\right)$.*
- *The least k singular values σ_i, factor scores λ_i, and factor score ratios $\lambda^{(i)}$ of $\boldsymbol{X}$ to precision ϵ, $2\epsilon\sqrt{\lambda_i}$, and $\epsilon \frac{\sigma_i}{\|\boldsymbol{X}\|_F^2}$ respectively, with probability at least $1 - 1/poly(d)$, in time $\widetilde{O}\left(\frac{\theta}{\sigma_{min}} \frac{1}{\sqrt{p}} \frac{\mu(\boldsymbol{X})k}{\epsilon}\right)$ or during any of the two procedures above.*

We can notice that the algorithm is almost equal to the top-k one. There are little differences, such as the definition of p which obviously changes since now we consider the estimated singular values less than θ. In this case, p refers to the variance retained by the least-k components. This fact has an impact on the running time since now the term $\frac{1}{\sqrt{p}}$ is bigger than the one in the Algorithm 3.3. Also the other term in the running time changes from $\frac{\|\boldsymbol{X}\|}{\theta}$ to $\frac{\theta}{\sigma_{min}}$, where σ_{min} can be obtained from the condition number $\kappa(\boldsymbol{X}) = \frac{\sigma_{max}}{\sigma_{min}}$, which can be estimated using a variant of the spectral norm estimation procedure described in Algorithm 2.5 [25].

Quantum mapping in PCA space

The quantum procedures described up to this point refer to the fitting of q-PCA. Indeed, they are principally used to classically describe the principal components and the associated eigenvalues. Now, we describe the quantized method that allows performing dimensionality reduction, projecting the original data into the new feature space described by the extracted principal components of q-PCA.
Basically, in a quantum computer, what happens is that we fit a q-PCA model with some noise obtaining vectors that are more or less close to the ones of the classical PCA. Then, we use those vectors to transform the input data matrix and projecting it into the new q-PCA feature space. As seen in Section 1.1.1, the way of doing this is the product $\boldsymbol{X'} = \boldsymbol{X}\overline{\boldsymbol{V}}^{(k)}$, where $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ is the input data matrix and $\overline{\boldsymbol{V}}^{(k)} \in \mathbb{R}^{d \times k}$ is the matrix representing the estimated top-k principal components (or right singular vectors) of the fitted quantum model, with $k \leq d$. In this way, we can map $\boldsymbol{X}$ from the d-dimensional feature space to the k-dimensional one. Therefore, our *quantum_ transform()* implemented method only deals with multiplying the original data matrix by the estimated principal components retrieved in the q-PCA model's fitting.

Quantum inverse mapping

The last procedure that we quantize is the mapping of a transformed vector into the original feature space that in Sklearn is called *inverse_ transform()*. We perform the following operation: $\widehat{\boldsymbol{X}} = \boldsymbol{X'}\overline{\boldsymbol{V}}^{(k)^T} + \mu$, where $\boldsymbol{X'} \in \mathbb{R}^{n \times k}$, $\overline{\boldsymbol{V}}^{(k)} \in \mathbb{R}^{d \times k}$, and μ is the computed sample mean that is initially removed from the data to have a centered zero mean data matrix (we explained this concept in Section 1.1.1). We can easily see that, by performing this dot product, we map back the data matrix from the k-dimensional feature space to the original d-dimensional one. Even in this case, the *quantum_ inverse_ trasform()* method is just about multiplying the transformed data matrix $\boldsymbol{X'}$ by the estimated principal components $\overline{\boldsymbol{V}}^{(k)}$.

3.9. q-Means

q-Means is the quantum version of the K-Means clustering algorithm (see Section 1.1.2 for more theoretical details about it). At high level, q-Means algorithm follows the same steps of the classical algorithm. The difference is that here, as for q-PCA, we use quantum routines for distance estimation, matrix multiplication, and tomography. Firstly, we pick k initial random centroids, or we use the efficient quantum version of k-means++ (see Theorem 2.5 in [26]). Then, all the data points are assigned to clusters in superposition

and not one after the other. In the last steps, we update the centroids of the cluster. All these procedures are repeated until convergence (in Section 1.1.2 we report a possible stopping condition for K-Means algorithm). q-Means can also be seen as the quantum equivalent of the δ-k-means algorithm, which is a version of k-means with noise, as well described by Kerenidis et al. [26].
For what concern the time complexity, q-Means depends poly-logarithmically on the number of elements in the dataset, differently from classical K-Means that runs in $O(tndk)$, with n number of elements in the dataset, d number of features, t number of iterations, and k number of classes. Therefore, q-Means gives us an exponential speed-up over the classical algorithm.

This is a brief explanation of q-Means, now we present its principal steps, as previously done for the q-PCA, following Algorithm 3.4.

Centroid distance estimation

In the first step of the algorithm, we compute the square distances between all the data points and the centroids using the quantum inner product procedure (that we described in Section 2.2.7). The following theorem describes this procedure.

Theorem 3.2. *(Theorem 2.1 in [26]) Let a data matrix $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ and a centroid matrix $\boldsymbol{C} \in \mathbb{R}^{k\times d}$ be stored in QRAM, such that the following unitaries $|i\rangle|0\rangle \to |i\rangle|\boldsymbol{x}_i\rangle$ and $|j\rangle|0\rangle \to |j\rangle|\boldsymbol{c}_j\rangle$ can be performed in time $O(\log(nd))$ and the norms of the vectors are known. For any $\Delta > 0$ and $\epsilon_1 > 0$ there exists a quantum algorithm that, given the state $\frac{1}{\sqrt{n}}\sum_{i=1}^{n}|i\rangle \otimes_{j\in[k]}(|j\rangle|0\rangle)$, performs the mapping to*

$$\frac{1}{\sqrt{n}}\sum_{i=1}^{n}|i\rangle \otimes_{j\in[k]}(|j\rangle|\overline{d(\boldsymbol{x}_i,\boldsymbol{c}_j)^2}\rangle),$$

where $\left|\overline{d(\boldsymbol{x}_i,\boldsymbol{c}_j)^2} - d(\boldsymbol{x}_i,\boldsymbol{c}_j)^2\right| \leq \epsilon_1$ with probability at least $1-2\Delta$ in time $\widetilde{O}\left(\frac{k\eta\log(\Delta^{-1})}{\epsilon_1}\right)$ where $\eta = \max_i(||x_i||^2)$.

Basically, what we have done to implement this step is to use the quantum inner product estimation (IPE algorithm, reported in Algorithm 2.4). This is because, by taking two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$, we can define the squared Euclidean distance between the two as

$$d(\boldsymbol{x},\boldsymbol{y})^2 = \|\boldsymbol{x}\|^2 + \|\boldsymbol{y}\|^2 - 2\langle\boldsymbol{x},\boldsymbol{y}\rangle\,. \tag{3.7}$$

As we can see, if we get an estimate of the inner product $\langle\boldsymbol{x},\boldsymbol{y}\rangle$, we also get an estimate

of the squared distance $d(\boldsymbol{x}, \boldsymbol{y})^2$. Therefore, for each different pair of vectors $(\boldsymbol{x}_i, \boldsymbol{c}_j)$, with $\boldsymbol{x}_i$ i-th sample of matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ and $\boldsymbol{c}_j$ j-th centroid vector, we compute the inner product $\langle \boldsymbol{x}_i, \boldsymbol{c}_j \rangle$. Then, we use the IPE procedure to estimate $\overline{\langle \boldsymbol{x}_i, \boldsymbol{c}_j \rangle}$ and, by substituting it in Equation 3.7, we obtain the estimate

$$\overline{d(\boldsymbol{x}_i, \boldsymbol{c}_j)^2} = \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{c}_j\|^2 - 2\overline{\langle \boldsymbol{x}_i, \boldsymbol{c}_j \rangle}.$$

Cluster assignment

Once finished the first step, we obtain the squared distance between each point in the dataset and the k centroids. Therefore, we assign the index j, corresponding to the closest centroid to the data point under consideration, as label of that data point

$$\ell(\boldsymbol{x}_i) = \arg\min_{j \in [k]}(d(\boldsymbol{x}_i, \boldsymbol{c}_j)). \tag{3.8}$$

In our framework, we implement this step by taking, for each sample i, the minimum of the distances, and then checking which is its corresponding index (i.e., if we have 2 centroids, suppose that the i-th sample is closer to the first centroid, then the index that we return as label assignment is 0). In the case we found more than one index, which means that the corresponding sample has more than one centroid at the same minimum distance, we choose at random the label.
Generally, we can easily extract the minimum thanks to the following Lemma.

Lemma 3. *(Circuit for finding the minimum [26].) Given k different $\log p$ bit registers $\otimes_{j \in [k]} |x_j\rangle$ there is a quantum circuit $\boldsymbol{U}_{min}$ that maps $(\otimes_{j \in [k]} |x_j\rangle) |0\rangle \to (\otimes_{j \in [k]} |x_j\rangle) |\arg\min(x_j)\rangle$ in time $O(k \log p)$.*

After applying the minimum lemma, we obtain the state

$$|\psi^t\rangle := \frac{1}{\sqrt{n}} \sum_{i=1}^{n} |i\rangle |\ell^t(\boldsymbol{x}_i)\rangle \tag{3.9}$$

that contains all the necessary information to extract the new centroids in the following step. Indeed, it is a uniform superposition, in which the two registers represent the index of the data point and the corresponding assigned label respectively.

Centroid state creation

In the previous step, we obtain the state reported in Equation 3.9, where in the first register we have the index of the n data points while in the second register we have the estimated labels for each data points at the current iteration. At this point, we need to update the centroids $|\boldsymbol{c}_j^{t+1}\rangle$, which basically are the median positions of the data points having the same label.

Lemma 4. *(Claim 2.3 [26]) Let $\boldsymbol{\chi}_j^t \in \mathbb{R}^n$ be the scaled characteristic vector for $\boldsymbol{C}_j$ at iteration t and $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ be the data matrix, then $\boldsymbol{c}_j^{t+1} = \boldsymbol{X}^T\boldsymbol{\chi}_j^t$.*

The above lemma allows us to extract the new centroids $\boldsymbol{c}_j^{t+1}$ using quantum linear algebra routines. Indeed, we can write the state $|\psi^t\rangle$ as [26]

$$|\psi^t\rangle = \sum_{j=1}^{k} \sqrt{\frac{|\boldsymbol{C}_j|}{n}} \left(\frac{1}{\sqrt{|\boldsymbol{C}_j|}} \sum_{i\in\boldsymbol{C}_j} |i\rangle \right) |j\rangle = \sum_{j=1}^{k} \sqrt{\frac{|\boldsymbol{C}_j|}{n}} |\boldsymbol{\chi}_j^t\rangle\, |j\rangle \tag{3.10}$$

Then we can measure the label register $|j\rangle$.

Centroid Update

In order to update the centroids, we need to retrieve classical information of the centroids from the corresponding quantum states. Therefore, we apply vector state tomography (refer to Algorithm 2.1 for more details) on the state $|\boldsymbol{c}_j^{t+1}\rangle$ created in the previous step using Lemma 4. By applying tomography, we get an estimate of the centroids within error ϵ_4, so that $\||\boldsymbol{c}_j\rangle - |\overline{\boldsymbol{c}}_j\rangle\| < \epsilon_4$.

Lemma 5. *(Claim 2.4 in [26]) Let ϵ_4 be the error we commit in estimating $|\boldsymbol{c}_j\rangle$ such that $\||\boldsymbol{c}_j\rangle - |\overline{\boldsymbol{c}}_j\rangle\| < \epsilon_4$ and ϵ_3 the error we commit in estimating the norms, $|\|\boldsymbol{c}_j\| - \|\overline{\boldsymbol{c}}_j\|| \leq \epsilon_3\|\boldsymbol{c}_j\|$. Then $\|\overline{\boldsymbol{c}}_j - \boldsymbol{c}_j\| \leq \eta(\epsilon_3 + \epsilon_4) = \epsilon_{centroid}$.*

Basically, we implement this step by computing the centroids as barycenters of all the samples assigned with the same labels, and then applying tomography with error $\eta(\epsilon_3+\epsilon_4)$.

Error analysis

The last important things to investigate is the determination of the values of the error parameters. We express all the errors $\epsilon_1, \epsilon_2, \epsilon_3$, and ϵ_4 in terms of δ. In the square distances estimation, we have seen that we get an estimate $\overline{d(\boldsymbol{x}_i, \boldsymbol{c}_j)^2}$ such that $|\overline{d(\boldsymbol{x}_i, \boldsymbol{c}_j)^2} - d(\boldsymbol{x}_i, \boldsymbol{c}_j)^2| \leq \epsilon_1$ and in the second step, where we find the minimum of these distances,

we do not add any additional error. To be consistent with the δ-k-means algorithm, we ensure that:

$$\forall j \in [k], |\overline{d(\boldsymbol{x}_i, \boldsymbol{c}_j)^2} - d(\boldsymbol{x}_i, \boldsymbol{c}_j)^2| \leq \frac{\delta}{2} \tag{3.11}$$

which implies that no centroid with distance more than δ above the minimum distance can be chosen as label by the q-Means algorithm. So, we have to take $\epsilon_1 < \delta/2$.
Then, the other step in which we consider an approximation error is in the centroid updates when we apply the vector state tomography to get a classical description of the centroids. As show in Lemma 5, we get $\epsilon_{centroid} = \sqrt{\eta}(\epsilon_3+\epsilon_4)$, with $\epsilon_3 < \frac{\delta}{4\sqrt{\eta}}$ and $\epsilon_4 < \frac{\delta}{4\sqrt{\eta}}$.

Running time analysis

The following theorem summarizes the running time of q-Means.

Theorem 3.3. *(Q-Means Theorem 5.1 in [26]) For a data matrix $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ and parameter $\delta > 0$, the q-Means algorithm with high probability outputs centroids consistent with the classical δ-k-Means algorithm in time $\widetilde{O}\left(kd\frac{\eta}{\delta^2}\kappa(\boldsymbol{X})(\mu(\boldsymbol{X}) + k\frac{\eta}{\delta}) + k^2\frac{\eta^{1.5}}{\delta^2}\kappa(\boldsymbol{X})\mu(\boldsymbol{X})\right)$ per iteration, where $1 \leq \|x_i\|^2 \leq \eta, \kappa(\boldsymbol{X})$ is the condition number, and*

$$\mu(\boldsymbol{X}) = \min_{p\in\mathcal{P}}(\|\boldsymbol{X}\|_F, \sqrt{s_{2p}(\boldsymbol{X})s_{2(1-p)}(\boldsymbol{X}^T)})$$

, where $\mathcal{P} \subset [0, 1]$ such that $|\mathcal{P}| = O(1)$ and $s_p(\boldsymbol{X}) := \max_{i\in[n]}\|\boldsymbol{X}_i\|_p^p$

It is interesting also to report the case in which the data are well-clusterable.

Definition 3. *(Well-clusterable dataset, [26]) A data matrix $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ with rows $\boldsymbol{x}_i \in \mathbb{R}^d, i \in [n]$ is said to be well-clusterable if there exist constants $\xi, \beta > 0, \lambda \in [0, 1], \eta \leq 1$, and cluster centroids $\boldsymbol{c}_i$ for $i \in [k]$ such that:*

- *(separation of cluster centroids): $d(\boldsymbol{c}_i, \boldsymbol{c}_j) \geq \xi, \forall i, j \in [k]$*
- *(proximity to cluster centroid): At least λn points $\boldsymbol{x}_i$, in the dataset satisfy $d(\boldsymbol{x}_i, \boldsymbol{c}_{l(\boldsymbol{x}_i)}) \leq \beta$ where $\boldsymbol{c}_{l(\boldsymbol{x}_i)}$ is the centroid nearest to $\boldsymbol{x}_i$.*
- *(Intra-cluster smaller than inter-cluster square distances): The following inequality is satisfied $4\sqrt{\eta}\sqrt{\lambda\beta^2 + (1-\lambda)4\eta} \leq \xi^2 - 2\sqrt{\eta}\beta$.*

Basically, this definition states that most of the data can be easily assigned to one of the k clusters since these points are close to the centroids and the centroids are sufficiently far from each other. The runtime of q-Means over a well-clusterable dataset is described in Theorem 3.4.

Algorithm 3.4 q-Means [26].

Input: Data matrix $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ stored in QRAM data structure. Precision parameters δ, error parameters ϵ_1 for distance estimation, ϵ_2 and ϵ_3 for matrix multiplication and ϵ_4 for tomography.
Output: Vectors $c_1, c_2, \ldots, c_k \in \mathbb{R}^d$ that correspond to the centroids at the final step of the δ-k-means algorithm.

1: Select k initial centroids $c_1^0, \ldots, c_k^0$ and store them in QRAM data structure and set counter $t = 0$.
2: **while** convergence condition is not satisfied **do**
3: **Step 1: Centroid Distance Estimation** Perform the mapping

$$\frac{1}{\sqrt{n}}\sum_{i=1}^{n}|i\rangle \otimes_{j\in[k]} |j\rangle\,|0\rangle \rightarrow \frac{1}{\sqrt{n}}\sum_{i=1}^{n}|i\rangle \otimes_{j\in[k]} |j\rangle\,|\overline{d(x_i, c_j^t)^2}\rangle$$

where $|\overline{d(x_i, c_j^t)^2} - d(x_i, c_j^t)^2| \leq \epsilon_1$.
4: **Step 2: Cluster Assignment** Find the minimum distance among $\{d(x_i, c_j^t)^2\}_{j\in[k]}$, then uncompute Step 1 to create the superposition of all points and their labels

$$\frac{1}{\sqrt{n}}\sum_{i=1}^{n}|i\rangle \otimes_{j\in[k]} |j\rangle\,|\overline{d(x_i, c_j^t)^2}\rangle \rightarrow \frac{1}{\sqrt{n}}\sum_{i=1}^{n}|i\rangle\,|\ell^t(x_i)\rangle$$

.
5: **Step 3: Centroid state creation**
- Measure the label register to obtain a state $|\chi_j^t\rangle = \frac{1}{\sqrt{|C_j^t|}}\sum_{i\in C_j^t}|i\rangle$, with probability $\frac{|C_j^t|}{n}$.
- Perform matrix multiplication with matrix $\boldsymbol{X}^T$ and vector $|\chi_j^t\rangle$ to obtain the state $|c_j^{t+1}\rangle$ with error ϵ_2 along with an estimation of $\|c_j^{t+1}\|$ with relative error ϵ_3.

6: **Step 4: Centroid Update**
- Perform tomography for the states $|c_j^{t+1}\rangle$ with precision ϵ_4 using the operation from Steps 1-3 and get a classical estimate $\overline{c}_j^{t+1}$ for the new centroids such that $|c_j^{t+1} - \overline{c}_j^{t+1}| \leq \eta(\epsilon_3 + \epsilon_4) = \epsilon_{centroids}$.
- Update the QRAM data structure for the centroids with the new vectors $\overline{c}_0^{t+1}, \ldots, \overline{c}_k^{t+1}$.

7: Increment the counter $t = t + 1$.
8: **end while**

Theorem 3.4. *(Q-Means on well-clusterable data, Theorem 3.2 in [26]) For a well-clusterable dataset $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ stored in QRAM, the q-Means algorithm returns with high probability the k centroids consistently with the classical δ-k-Means algorithm for a constant δ in time $\widetilde{O}\left(k^2 d\frac{\eta^{2.5}}{\delta^3} + k^{2.5}\frac{\eta^2}{\delta^3}\right)$ per iteration, for $1 \leq \|\boldsymbol{x}_i\|^2 \leq \eta$.*

4 | Applications to cybersecurity

In this chapter, we describe the cybersecurity experiments done and the results found, making a comparison between classical and quantum results. In the result tables, we highlight the best F1-score results such that the reader can better check the best performances. The models that we use to detect intrusions are classically described in Section 1.2.

Goals

We remark that the objective of the following experiments is not to compare the performances of our anomaly detection models with the state-of-the-art ones. Our goal is to show how the performances change, both in terms of capability to detect anomalies and in terms of running time, between quantum and classical machine learning models. In particular, with these experiments, we want to study how the error that we need to consider in quantum machine learning algorithms affects the anomaly detection capability and the running time of our models. We want to find cases of error that allow us to match classical performances and, at the same time, obtain a reasonable number of samples and features for which quantum machine learning is advantageous over classical machine learning in terms of running time.
In the evaluation of the results, we need to take into account also the balancing of the datasets. For this purpose, we present experiments over different combinations of datasets, sometimes unbalanced towards the normal samples, sometimes towards the anomalies, to see how the balancing affects the detection. We also remember that the objective of our network intrusion detection models is to detect whether or not an action is an attack. Therefore, we do not distinguish between different types of attacks, but the only distinction that we make is between normal and anomaly actions.

Metrics

We compare the performances in terms of precision $p = \frac{TP}{TP+FP}$, recall $r = \frac{TP}{TP+FN}$, F1-score $f1 = \frac{2}{(\frac{1}{p})+(\frac{1}{r})}$, and accuracy $a = \frac{TP+TN}{TP+FP+TN+FN}$

where

- TP are the samples that we classify as attacks and they are effectively attacks;
- TN are the samples that we classify as normal and they are effectively normal;
- FP are the samples that we classify as attacks but they are normal samples;
- FN are the samples classified as normal but in reality, they are attacks.

Datasets description

We use three publicly available network intrusion datasets: *KDDCUP 99* [47], *CICIDS 2017* [43], and *Darknet* [17]. We report a brief description for each one of them to give an overview and make the reader more comfortable with these datasets.

KDDCUP 99 This dataset is used for network-based anomaly detection systems. It was originally intended for a competition where the goal was to build a network intrusion detector, a system capable to distinguish between normal and anomaly connections. This dataset includes different types of attacks simulated in a military network environment. We use the *kddcup.data_10_percent* data to run our experiments, which is a 10% subset of the total dataset. This is to make the experiments less burdensome from a computational point of view. This subset of the KDDCUP dataset contains 494,021 samples described by 41 features. Out of 494,021 samples, 97,278 are labeled as *normal* and 396,743 as *attack*. In the attack category are included different types of simulated network attacks:

- *Denial of Service* attack (DoS): the attacker exploits seemingly-legitimate requests, making some computing or memory resources too busy to handle legitimate requests by other users, not allowing the users to access the machine.
- *User To Root* attack (U2R): the attacker exploits some vulnerabilities of a system to gain root after logging into a normal system user account (perhaps gained by sniffing passwords, a dictionary attack, or social engineering).
- *Remote to Local* attack (R2L): the attacker exploits some vulnerability to gain local access as a user of a machine on which he/she does not have an account by sending packets to that machine over a network.
- *Probing* attack: the attacker exploits some vulnerabilities to gather information about a network of computers to jeopardize its security controls.

CICIDS 2017 The CICIDS 2017 dataset is intended for network security and intrusion detection purposes. It contains benign samples and some of the most common network attacks. The data capturing period started at 9 a.m. Monday, July 3, 2017 and ended at 5 p.m on Friday, July 7, 2017, for a total of five days. On Monday only benign samples are recorded. The attacks comprehend Brute Force FTP, Brute Force SSH, DoS, HeartBleed, Web attack, BotNet, and DDoS. They have been executed both in the morning and afternoon on Tuesday, Wednesday, Thursday, and Friday.
As we see better in the experiments, we use two different "versions" of this dataset: in some experiments, we use samples composed only of benign and *DDoS* attacks (we call it *CICIDS_DDoS* dataset), while, in other experiments, we consider all the possible types of attacks (we call it *CICIDS_ALL* dataset).

Darknet A darknet is an overlay network within the internet that often use a customized communication protocol. We can only access this network using specific software. This dataset captures regular, VPN, and Tor traffic for seven diverse categories under respective applications: Browsing, Chat, Email, File Transfer, FTP over SSL, Streaming, Voip, and P2P. It consists of 158,659 total samples with 134,348 ($\approx 85\%$ of the total samples) *benign* which refers to normal samples, and 24,311 ($\approx 15\%$ of the total samples) *darknet* samples which are the anomalous ones (more precisely the samples labeled as TOR and VPN).

4.1. Principal components classifier over KDDCUP

This experiment is a reproduction of the experiment done by Shyu et al. [45]. We try to reproduce the results of the paper to be able to compare them with our quantum version. We described the principal components classifier model in Section 1.2.1.

Preprocessing As we have seen, in KDDCUP 99 there are 41 features, 34 of which are numerical and 7 categorical. We consider only numerical features. Since the current model requires that the training set is composed only by normal samples, we divide the training and test set in this way: 5000 *normal* samples for the training set and a test set of the remaining 92,278 *normal*s and 39,674 *attack* samples (the proportions are equal to the paper ones). To extract the training set, we use *trimming* (that is a procedure described in the Section 1.2.1 to remove outliers from the training set) and *random systematic sampling*. The latter is a probability sampling method in which a random sample, with a fixed periodical interval, is selected from a larger population without replacement so that a sample cannot be in the train and test sets at the same time. The fixed periodic interval is calculated by dividing the population size by the

desired sample size and extracting randomly a value in the range of this number[1]. The last step of the preprocessing is the normalization since there are features of different magnitudes. We use the *StandardScaler()* Python method to remove the mean and scale to unit variance. Pay attention: before normalizing the training set, we compute the sample mean and standard deviation, which we also use to normalize the respective test samples since it is required that also the new observations, which are evaluated by the model, are normalized with the sample mean and variance.

Classical model building We want to compare 5 different PCA-based models that retain the 30%,40%,50%,60%, and 70% of the total variance respectively. Therefore, from the preprocessing, we get 5 different normalized train/test sets pairs that we are going to use for the 5 different models. From the train sets, we drop the class labels as we use unsupervised machine learning algorithms. We see that fitting models with these percentages of variance corresponds to retain the first 2,4,5,7, and 10 principal components respectively. Then, we compute the quantiles that we use to extract the outlier thresholds (we describe this procedure in Section 1.2.1) using six different values of α, which correspond to the false alarm rates: $0.01, 0.02, 0.04, 0.06, 0.08$, and 0.10. For instance, taking $\alpha = 0.01$ and considering Equation 1.10 with $\alpha_1 = \alpha_2$, we have that

$$\alpha_1^2 - 2\alpha_1 + 0.01 = 0$$

which gives, as a possible solution, $\alpha_1 = 0.005$. We can now compute the empirical distributions of $\sum_{i=1}^{k} \frac{y_i^2}{\lambda_i}$ and $\sum_{i=d-r+1}^{d} \frac{y_i^2}{\lambda_i}$ in the training set, where k is the number of the principal components that retain the specified percentage of variance in the model, r is the number of the minor components whose corresponding eigenvalues are less than 0.20 [45], and d is the number of features of the input data. To compute y_i, in this case, we make the dot product between the extracted eigenvectors and the training samples. Finally, we extract the $1 - \alpha_1 = 1 - 0.005$ quantile of these two distributions to find the outlier thresholds. Once computed the outlier thresholds, we are ready to verify the performances on the test sets.

Quantum model building We know that we want to retain $30\%, 40\%, 50\%, 60\%$, and 70% of the total variance in the five different q-PCA models. Therefore, we use quantum binary search (reported in Algorithm 3.2) specifying the percentage of variance that we

[1]To make this more clear, let us consider the normal population size of our experiment that is 97,278. Since we want each training set composed of 5000 samples, we compute the periodic interval as $97,278/5000 \simeq 19$. Therefore, we randomly select a number in the range of [0,19], let us say 4. Then, to extract the training set, starting from the 4-th sample in the population and with a step of 19, we select 5000 samples. The remaining will be part of the test set.

```
[ ]: import pandas as pd
     from sklearn.decomposition import qPCA
     from sklearn.QuantumUtility.Utility import *
     import warnings
```

```
[ ]: qpca70 = qPCA(svd_solver='full', name='q-PCA70').fit(training_set,␣
      ↪theta_estimate=True,eps_theta=1,p=0.70,estimate_all=True,delta=0.1, eps=1,␣
      ↪true_tomography=True, eta=0.1,theta_minor=np.sqrt(0.20 * training_set.
      ↪shape[0]),spectral_norm_est=True, estimate_least_k=True)
```

Figure 4.1: Example of q-PCA model building from a Jupiter notebook.

want to keep in the major components, the error parameters ϵ_θ for the SVE procedure and η for the amplitude estimation internally executed to estimate θ, which corresponds to the smallest singular value to consider to retain the specified percentage of variance. Then, we use θ in quantum top-k singular vector extraction (Algorithm 3.3) to extract top-k singular values and singular vectors, with errors ϵ and δ respectively such that $\|\boldsymbol{v}_i - \overline{\boldsymbol{v}}_i\| \leq \delta$ and $\|\sigma_i - \overline{\sigma}_i\| \leq \epsilon$. In this way, we can compute both $\overline{y}_i = \overline{\boldsymbol{v}}_i^T \boldsymbol{z}$, with $\overline{\boldsymbol{v}}_i$ which is the i-th estimated eigenvector corresponding to the i-th estimated eigenvalue $\overline{\lambda}_i$ that we can retrieve by squaring the corresponding estimated singular value $\overline{\sigma}_i$. Therefore, we can compute the summation $\sum_{i=1}^{k} \frac{\overline{y}_i^2}{\overline{\lambda}_i}$ both to extract the outlier threshold and to compute anomaly scores. For the minor components, we use the quantum least-r singular vectors extraction algorithm, described in Theorem 3.1, to get a classical estimate of least-r singular vectors. Basically, what changes from quantum top-k singular vectors extraction is that, in this case, we retain all the estimated singular values that are less than θ. However, we cannot use the same θ found in quantum binary search for top-k singular vectors since it is related to the percentage of variance that top-k major components retain. Moreover, in this case, we cannot apply quantum binary search to estimate θ since, by definition of minor components, we do not know what variance the minor components retain. We only know that the minor components are the ones whose corresponding eigenvalues are less than a specific value ν. Since in PCA, eigenvalues are related to singular values by the formula $\lambda_i = \frac{\sigma_i^2}{n-1}$, where n is the number of samples of the input data matrix, and since $\lambda_i = \frac{\sigma_i^2}{n-1}$ is an increasing monotone function, we can use $\theta = \sqrt{\nu(n-1)}$ in the least-r components extractor to select the estimated singular values that are less than that θ. This corresponds to estimate the least-r components whose corresponding estimated eigenvalues are less than ν, as we do in the classical procedure. In this way, we are also able to compute $\sum_{i=d-r+1}^{d} \frac{\overline{y}_i^2}{\overline{\lambda}_i}$.

In Figure 4.1, we report two cells of a Jupiter notebook to show how easy it is to use our framework to build quantum models. Indeed, just import the qPCA class from the

decomposition.qPCA.py module and use the *fit()* method passing the necessary parameters whose description is reported in the official Sqlearn documentation. Inside the *fit()* method, we implement all the logic of the quantum procedures presented in Section 3.8 used to extract the q-PCA model.

Results comparison with only major components In this case, we classify a sample as attack if $\sum_{i=1}^{k} \frac{y_i^2}{\lambda_i} > c1$, as normal otherwise. For the quantum model, we use the following parameters, remembering that p is the percentage of variance that we want the principal components to explain, ϵ_θ is the error ϵ inserted in the SVE in quantum binary search to estimate θ such that $|\sigma_i - \overline{\sigma}_i| \leq \epsilon_\theta$, η is the error in the amplitude estimation internally executed in the quantum binary search such that $|p - \overline{p}_\tau| \leq \eta/2$, ϵ is the error inserted to estimate singular values such that $|\sigma - \overline{\sigma}_i| \leq \epsilon$, and δ is the error we tolerate in estimating singular vectors such that $|\boldsymbol{v}_i - \overline{\boldsymbol{v}}_i| \leq \delta$:

- q-PCA30: $\epsilon_\theta = 8$, $p = 0.30$, $\delta = 0.1$, $\epsilon = 8$ and $\eta = 0.2$.
- q-PCA40: $\epsilon_\theta = 1$, $p = 0.40$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.16$.
- q-PCA50: $\epsilon_\theta = 1$, $p = 0.50$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.
- q-PCA60: $\epsilon_\theta = 1$, $p = 0.60$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.
- q-PCA70: $\epsilon_\theta = 1$, $p = 0.70$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.

Pay attention: we use the notation ϵ_θ to underline the ϵ error used in the quantum binary search to estimate θ. However, since both ϵ and ϵ_θ refer to the error used in the SVE to estimate singular values, it makes sense to use the same values for both. With those parameters, we are able to retain respectively the top 2,4,5,6, and 10 components, which are quite consistent with the numbers in the classical experiments, which are 2,4,5,7, and 10.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.6713	0.6629	0.9577	0.9557	0.9328	0.9296	0.9354	0.9482	0.9314	0.9284
2%	0.6881	0.6820	0.9853	0.9815	0.9721	0.9753	0.9618	0.9799	0.9319	0.9299
4%	0.7117	0.7076	0.9857	0.9825	0.9846	0.9829	0.9861	0.9823	0.9604	0.9575
6%	0.7136	0.7104	0.9868	0.9828	0.9859	0.9838	0.9873	0.9830	0.9851	0.9812
8%	0.7138	0.7107	0.9872	0.9838	0.9872	0.9842	0.9880	0.9832	0.9867	0.9836
10%	0.7141	0.7110	0.9874	0.9845	0.9874	0.9843	0.9890	0.9844	0.9944	0.9912

Table 4.1: Recall comparison for classical (c) and quantum (q) principal components classifier with major components over KDDCUP.

We can notice that, with these error parameters, we are able to match classical per-

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.9849	0.9852	0.9846	0.9849	0.9930	0.9937	0.9901	0.9885	0.9863	0.9868
2%	0.9715	0.9696	0.9743	0.9738	0.9810	0.9829	0.9822	0.9796	0.9818	0.9823
4%	0.9428	0.9408	0.9573	0.9507	0.9587	0.9598	0.9595	0.9582	0.9651	0.9657
6%	0.9121	0.9134	0.9358	0.9311	0.9329	0.9400	0.9400	0.9384	0.9420	0.9421
8%	0.8889	0.8893	0.9199	0.9156	0.9114	0.9187	0.9289	0.9245	0.9201	0.9207
10%	0.8649	0.8644	0.8969	0.8969	0.8944	0.8963	0.9166	0.9082	0.9005	0.9010

Table 4.2: Precision comparison for classical (c) and quantum (q) principal components classifier with major components over KDDCUP.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.7984	0.7926	0.9709	0.9701	0.9620	0.9606	0.9620	0.9679	0.9581	0.9567
2%	0.8056	0.8007	**0.9798**	**0.9776**	**0.9765**	**0.9791**	0.9719	**0.9798**	0.9562	0.9548
4%	**0.8111**	**0.8077**	0.9713	0.9663	0.9715	0.9712	**0.9726**	0.9701	0.9628	**0.9615**
6%	0.8007	0.7992	0.9606	0.9562	0.9587	0.9614	0.9631	0.9602	**0.9630**	0.9612
8%	0.7918	0.7900	0.9524	0.9485	0.9478	0.9503	0.9576	0.9529	0.9522	0.9511
10%	0.7823	0.7802	0.9400	0.9386	0.9386	0.9382	0.9514	0.9448	0.9451	0.9440

Table 4.3: F1_score comparison for classical (c) and quantum (q) principal components classifier with major components over KDDCUP.

formances. In general, we can notice that as α increases, recall increases and precision decreases. This is an expected result since increasing α means increasing the false alarm rate, which reflects in lowering the outlier threshold and therefore in classifying more observations as attacks. In this way, the false-positive rate increases and the false negative one decreases, leading to an increase in recall and a decrease in precision. Moreover, the increase of false positives naturally reflects in a decrease in the true negative rate which decreases accuracy, as we can see in Tables A.1 in Appendix A. In Table 4.3, we also report the corresponding classical and quantum F1-score results highlighting the best results found for each model. In this way, the reader can better visualize the best performances.

To see how the errors affect performances, we also try to change the values of the error parameters incrementing δ and ϵ and fixing the other ones. The error ϵ is related to the estimates of singular values, so a big ϵ error leads to totally wrong estimates of singular values. This is a problem in the quantum top-k singular vectors extraction algorithm since we extract the components whose corresponding estimated singular values are greater than θ. Therefore, if the estimates of singular values are totally wrong, we extract the wrong principal components with error δ, resulting in subsequent wrong model computations. For example, let us consider the following parameters:

- q-PCA30: $\epsilon_\theta = 8$, $p = 0.30$, $\delta = 0.9$, $\epsilon = 8$, and $\eta = 0.2$.

- q-PCA40: $\epsilon_\theta = 20$, $p = 0.40$, $\delta = 0.9$, $\epsilon = 20$, and $\eta = 0.16$.
- q-PCA50: $\epsilon_\theta = 20$, $p = 0.50$, $\delta = 0.9$, $\epsilon = 20$, and $\eta = 0.1$.
- q-PCA60: $\epsilon_\theta = 20$, $p = 0.60$, $\delta = 0.9$, $\epsilon = 20$, and $\eta = 0.1$.
- q-PCA70: $\epsilon_\theta = 20$, $p = 0.70$, $\delta = 0.9$, $\epsilon = 20$, and $\eta = 0.1$.

So, we increment ϵ (and ϵ_θ) to 20 (singular values are in the range [0, 150]) for each model except PCA30 (since it already retains few components and, by increasing too much ϵ, we risk reducing the number of principal components to 0), and $\delta = 0.9$. We report the results in the following tables where we compare them always with the classical results.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.6713	0.6881	0.9577	0.6699	0.9328	0.0001	0.9354	0.9471	0.9314	0.9281
2%	0.6881	0.6835	0.9853	0.6814	0.9721	0.0001	0.9618	0.9814	0.9319	0.9648
4%	0.7117	0.7076	0.9857	0.7062	0.9846	0.0001	0.9861	0.9828	0.9604	0.9823
6%	0.7136	0.7102	0.9868	0.7097	0.9859	0.0001	0.9873	0.9828	0.9851	0.9829
8%	0.7138	0.7106	0.9872	0.7104	0.9872	0.0001	0.9880	0.9838	0.9867	0.9847
10%	0.7141	0.7109	0.9874	0.7106	0.9874	0.0001	0.9890	0.9841	0.9944	0.9849

Table 4.4: Recall comparison for classical (c) and quantum (q) principal components classifier with major components over KDDCUP increasing ϵ and δ.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.9849	0.9848	0.9846	0.9867	0.9930	0.017	0.9901	0.9899	0.9863	0.9924
2%	0.9715	0.9703	0.9743	0.9759	0.9810	0.0079	0.9822	0.9791	0.9818	0.9837
4%	0.9428	0.9425	0.9573	0.9511	0.9587	0.0035	0.9595	0.9592	0.9651	0.9570
6%	0.9121	0.9155	0.9358	0.9167	0.9329	0.0026	0.9400	0.9430	0.9420	0.9402
8%	0.8889	0.8848	0.9199	0.8890	0.9114	0.0019	0.9289	0.9224	0.9201	0.9167
10%	0.8649	0.8649	0.8969	0.8603	0.8944	0.0015	0.9166	0.9033	0.9005	0.9019

Table 4.5: Precision comparison for classical (c) and quantum (q) principal components classifier with major components over KDDCUP increasing ϵ and δ.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.7984	0.7961	0.9709	0.7980	0.9620	0.0003	0.9620	0.9680	0.9581	0.9592
2%	0.8056	0.8020	**0.9798**	0.8025	**0.9765**	**0.0003**	0.9719	**0.9803**	0.9562	**0.9741**
4%	**0.8111**	**0.8083**	0.9713	**0.8106**	0.9715	0.0003	**0.9726**	0.9709	0.9628	0.9695
6%	0.8007	0.7999	0.9606	0.8000	0.9587	0.0003	0.9631	0.9625	**0.9630**	0.9611
8%	0.7918	0.7882	0.9524	0.7897	0.9478	0.0003	0.9576	0.9521	0.9522	0.9494
10%	0.7823	0.7803	0.9400	0.7783	0.9386	0.0003	0.9514	0.9420	0.9451	0.9416

Table 4.6: F1_score comparison for classical (c) and quantum (q) principal components classifier with major components over KDDCUP increasing ϵ and δ.

As we can see, now the results for q-PCA40 and q-PCA50 are worse than the ones of the previous experiment. The reason for this behavior is to be found in the number of components extracted by quantum models considering the new errors. Indeed, the five models retain 2,2,1,4, and 6 principal components respectively. So, all the models retain less number of components with respect to the previous case (except for q-PCA30 that we do not change) and this fact can only worsen the performances. We see that q-PCA50 has very low recall and precision near to zero. This is probably due to the fact that q-PCA50 retains only the first principal component (with respect to the five principal components retained by the corresponding classical model). For q-PCA40, things are a little bit different. Indeed, we see that the performances decrease but not as in the q-PCA50 model. This is because, in that model, we retain 40% of the variance, which is a closer percentage to the model which retains 30%. Therefore, by keeping only the first two principal components, we are very close to q-PCA30 model. As a matter of fact, the results between the two are very similar. For what concerns q-PCA60 and q-PCA70, the results are very similar to the previous ones since the number of principal components retained by the two models is close to the previous case.
Also changing the η error parameter leads to a different number of components retained and therefore to a worsening of the performances. Anyway, these experiments would need further investigation to better understand the behavior of the quantum models at the increase of ϵ and δ.

Results comparison with both major and minor components In this experiment, we classify a sample as an attack if

$$\sum_{i=1}^{k} \frac{y_i^2}{\lambda_i} > c_1 \quad \textbf{or} \quad \sum_{i=d-r+1}^{d} \frac{y_i^2}{\lambda_i} > c_2$$

and as normal if

$$\sum_{i=1}^{k} \frac{y_i^2}{\lambda_i} \leq c_1 \quad \textbf{and} \quad \sum_{i=d-r+1}^{d} \frac{y_i^2}{\lambda_i} \leq c_2.$$

We maintain the same error parameters reported in the previous paragraph (the ones that give better results) to estimate both major and minor components. To extract the minor components, we also add, as previously said, $\theta = \sqrt{0.20(n-1)}$ as a parameter in the quantum least-r singular vectors extraction, where n is the number of training samples (in this case $n = 5000$). With these error parameters, we obtain the following models that both in terms of number of components retained and in terms of performances are very consistent with the classical ones:

- q-PCA30 with 2 major and 7 minor components (with respect to 2 major and 7 minor of the classical case).
- q-PCA40 with 5 major and 6 minor components (with respect to 4 major and 6 minor of the classical case).
- q-PCA50 with 5 major and 7 minor components (with respect to 5 major and 6 minor of the classical case).
- q-PCA60 with 6 major and 7 minor components (with respect to 7 major and 7 minor of the classical case).
- q-PCA70 with 10 major and 7 minor components (with respect to 10 major and 6 minor of the classical case).

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.9427	0.9490	0.9857	0.9826	0.9853	0.9861	0.9864	0.9862	0.9862	0.9868
2%	0.9610	0.9835	0.9870	0.9835	0.9865	0.9872	0.9875	0.9874	0.9866	0.9873
4%	0.9891	0.9847	0.9880	0.9847	0.9873	0.9881	0.9889	0.9889	0.9880	0.9890
6%	0.9901	0.9852	0.9894	0.9852	0.9877	0.9894	0.9895	0.9894	0.9884	0.9899
8%	0.9907	0.9867	0.9903	0.9867	0.9897	0.9918	0.9898	0.9899	0.9893	0.9927
10%	0.9915	0.9887	0.9917	0.9887	0.9909	0.9941	0.9919	0.9924	0.9959	0.9987

Table 4.7: Recall comparison for classical (c) and quantum (q) principal components classifier with major and minor components over KDDCUP.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.9758	0.9762	0.9712	0.9718	0.9786	0.9747	0.9764	0.9756	0.9726	0.9699
2%	0.9572	0.9570	0.9509	0.9505	0.9580	0.9578	0.9573	0.9534	0.9596	0.9558
4%	0.9160	0.9159	0.9187	0.9147	0.9120	0.9208	0.9207	0.9157	0.9161	0.9142
6%	0.8785	0.8782	0.8874	0.8915	0.8870	0.8885	0.8916	0.8841	0.8888	0.8825
8%	0.8483	0.8520	0.8599	0.8707	0.8580	0.8606	0.8788	0.8695	0.8611	0.8584
10%	0.8183	0.8213	0.8313	0.8460	0.8314	0.8374	0.8577	0.8440	0.8325	0.8314

Table 4.8: Precision comparison for classical (c) and quantum (q) principal components classifier with major and minor components over KDDCUP.

Even in this case, we show that we are able to get the same performances in both cases. It is evident that now, also for the PCA model that retains 30% of the variance, the results are very similar to the others. We rightly notice a decrease in precision at the increase of α, while recall increases. We can also notice that recall, in the case of using both major and minor components, is higher with respect to the model in which we use only the major ones (whose results are reported in Table 4.1), but precision is lower. We expect this behavior since using two control summations in OR condition instead of one reflects

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.9590	0.9624	**0.9784**	**0.9772**	**0.9819**	**0.9803**	**0.9814**	**0.9809**	**0.9794**	**0.9783**
2%	**0.9591**	**0.9699**	0.9686	0.9667	0.9720	0.9723	0.9722	0.9701	0.9729	0.9713
4%	0.9512	0.9495	0.9521	0.9484	0.9482	0.9533	0.9536	0.9509	0.9507	0.9501
6%	0.9310	0.9295	0.9356	0.9360	0.9347	0.9363	0.9380	0.9338	0.9360	0.9331
8%	0.9140	0.9149	0.9205	0.9251	0.9192	0.9216	0.9310	0.9258	0.9208	0.9207
10%	0.8966	0.8972	0.9045	0.9118	0.9042	0.9090	0.9199	0.9122	0.9069	0.9074

Table 4.9: F1_score comparison for classical (c) and quantum (q) principal components classifier with major and minor components over KDDCUP.

in higher chance to classify an observation as an anomaly. This leads to an increase in the false-positive rate and a decrease in false-negative cases (which results in an increase in recall) since, as previously said, the test sets are unbalanced towards the normal samples. Therefore, in this type of dataset, increasing the predictions of attacks increases the risk of false positives.
As done for the previous experiment, by increasing the error parameters for the quantum models, we found almost the same behavior that we discuss in the previous experiments. We saw that the performances decrease when we increase both δ and ϵ error since not only the principal components are more distant from the real ones due to the increase of δ, but also their number differs considerably from the classical case.
The interested reader can also check Table A.3 to verify the accuracy results.

4.2. Principal components classifier over CICIDS 2017

Preprocessing Here we take into account the CICIDS 2017 dataset, considering only DDoS as attacks (we called it CICIDS_DDoS). More precisely, we measure our performances over a slightly unbalanced test set towards *benign* cases. We consider the training set composed of 5000 *benign* samples and the test set composed respectively of 87,300 *benign* and 70,000 *DDoS* samples. First of all, we extract from the *DateTime* feature the *hours, minutes, month*, and *day*, which become four new features. Then, we encode the categorical features *Flow ID, Source IP, Destination IP, and Timestamp* using the *LabelEncoder()* Python function, converting them into numerical features. With the *VarianceThreshold()* function, we find the constant features, which are the ones that bring no information with them or, equivalently, have 0 variance. After having normalized the dataset, we remove these features together with the categorical ones as the *Source IP* or the *Source Port* since they do not bring any useful or additional information with respect to the others. The normalization process is equal to the one performed for KDDCUP 99 in the previous experiments: we standardize the features by removing the mean and scal-

ing to unit variance. We have to consider also another important thing. In the CICIDS dataset, there is the feature *FLOW ID* which is a unique index that corresponds to a specific packet flow. Before removing it, we use it to be sure to not include observations with the same Flow ID into the training and test sets. This is because, otherwise, it could cause the model to make misleading predictions on the base of what it has learned in the training. If we had two samples with the same Flow ID, one in the training and the other in the test set, it is as if we had two equal samples in the two sets, leading to overfitting problems.

Classical and quantum model building The fitting of classical and quantum models is equal to the one reported in the previous experiment's section. We always compare five different PCA models that retain $30\%, 40\%, 50\%, 60\%$, and 70% of the variance in the principal components. In this experiment, we use only the principal components classifier model (and q-PCC) with both minor and major components. We refer the reader to the previous section for a more detailed explanation.

Results comparison First of all, we report the error parameters used in quantum models:

- q-PCA30: $\epsilon_\theta = 1$, $p = 0.30$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.1$.
- q-PCA40: $\epsilon_\theta = 1$, $p = 0.40$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.2$.
- q-PCA50: $\epsilon_\theta = 1$, $p = 0.50$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.1$.
- q-PCA60: $\epsilon_\theta = 1$, $p = 0.60$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.1$.
- q-PCA70: $\epsilon_\theta = 1$, $p = 0.70$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.1$.

We also report the components retained for each model that, as we can see, are very consistent with the classical ones:

- q-PCA30 with 1 major and 32 minor components (with respect to 2 major and 31 minor of the classical case).
- q-PCA40 with 2 major and 32 minor components (with respect to 2 major and 31 minor of the classical case).
- q-PCA50 with 3 major and 32 minor components (with respect to 3 major and 31 minor of the classical case).
- q-PCA60 with 4 major and 32 minor components (with respect to 5 major and 31 minor of the classical case).

- q-PCA70 with 6 major and 32 minor components (with respect to 7 major and 31 minor of the classical case).

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.4086	0.4076	0.3506	0.3515	0.3487	0.3390	0.4107	0.4093	0.4084	0.3605
2%	0.4948	0.4371	0.4166	0.4167	0.4189	0.4113	0.5017	0.4861	0.4596	0.5897
4%	0.5919	0.5899	0.5613	0.6012	0.5588	0.5554	0.5945	0.5981	0.6326	0.6330
6%	0.6290	0.6271	0.6292	0.6587	0.6265	0.6227	0.6315	0.6320	0.6334	0.6337
8%	0.6371	0.6358	0.6931	0.7118	0.6320	0.6310	0.6993	0.6387	0.6384	0.6443
10%	0.6504	0.6529	0.7476	0.7621	0.6477	0.6449	0.6523	0.6548	0.6465	0.6590

Table 4.10: Recall comparison for classical (c) and quantum (q) principal components classifier with major and minor components over CICIDS.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.9759	0.9718	0.9700	0.9709	0.9689	0.9705	0.9746	0.9738	0.9695	0.9694
2%	0.9586	0.9429	0.9496	0.9492	0.9481	0.9492	0.9605	0.9564	0.9483	0.9654
4%	0.9277	0.9187	0.9300	0.9300	0.9312	0.9312	0.9375	0.9339	0.9396	0.9436
6%	0.9013	0.8937	0.9045	0.9052	0.9077	0.9059	0.9134	0.9093	0.9135	0.9161
8%	0.8777	0.8684	0.8880	0.8917	0.8846	0.8817	0.8883	0.8859	0.8854	0.8899
10%	0.8551	0.8400	0.8727	0.8751	0.8590	0.8593	0.8649	0.8623	0.8655	0.8692

Table 4.11: Precision comparison for classical (c) and quantum (q) principal components classifier with major and minor components over CICIDS.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.5760	0.5743	0.5150	0.5161	0.5129	0.5025	0.5779	0.5763	0.5747	0.5257
2%	0.6527	0.5973	0.5791	0.5792	0.5811	0.5739	0.6592	0.6446	0.6191	0.7322
4%	0.7227	0.7185	0.7001	0.7303	0.6985	0.6958	0.7276	0.7290	**0.7561**	**0.7577**
6%	**0.7410**	**0.7370**	0.7421	0.7625	**0.7414**	**0.7381**	**0.7470**	**0.7457**	0.7481	0.7492
8%	0.7383	0.7341	0.7785	0.7917	0.7373	0.7356	0.7435	0.7423	0.7419	0.7475
10%	0.7388	0.7347	**0.8054**	**0.8147**	0.7385	0.7368	0.7437	0.7444	0.7401	0.7497

Table 4.12: F1_score comparison for classical (c) and quantum (q) principal components classifier with major and minor components over CICIDS.

Taking a look at Tables 4.10 and 4.11, that report the comparison between quantum and classical recall and precision respectively, we can see that, by tolerating the previously reported error, we are able to match classical performances. However, it is evident that the performances have worsened compared to the results on KDDCUP reported in Tables 4.7 and 4.8. This is not surprising given that there is no machine learning model that always performs well, but it depends on the specific problem and the data on which we are evaluating it. This is a consequence of the famous *No free lunch* theorem, stated by Wolpert and Macready [53] in which it is demonstrated that, in absence of any assumptions

on the data, there is no reason to prefer one model over another.
In Table A.4, we report the corresponding results of the accuracy comparison.

4.3. Ensemble of principal components classifiers over CICIDS 2017

We described this model in Section 1.2.2. We just remember briefly that this model is an extension to the principal components classifier model by adding the cosine similarity measure and the correlation measure to compute y_i which is used to calculate the anomaly thresholds and scores.

Preprocessing In this experiment, we use the same training and test sets of the previous experiment, so that we can compare performance between the two models over the same data. Therefore, the preprocessing is the same as the one reported in the previous experiment. We remember the proportions of the dataset: 5000 benign samples in the training set and 87,300 benign plus 70,000 DDoS samples in the test set.

Classical model building Also the fitting of the model is almost identical to the one of the principal components classifier model with both major and minor components. The difference is that here, we have to extract also the outlier thresholds computed with the cosine similarity and with the correlation between eigenvectors and training samples. As before, we fit always five different models with $30\%, 40\%, 50\%, 60\%$, and 70% of variance retained in the principal components and we extract the minor components whose corresponding eigenvalues are less than 0.20.

Quantum model building We briefly report this description since it is equal to the case of the fitting of quantum principal components classifier model with major and minor components described before. Indeed, to extract the major components we execute the quantum binary search, specifying ϵ_θ, η, and p to estimate θ which is used, in the quantum top-k singular vector extraction, to retrieve a classical description of top-k singular values and singular vectors, with error ϵ and δ respectively. To extract the minor components, we use the least-r singular vectors extraction, always using $\theta = \sqrt{0.20 * (n-1)}$. Once estimated the principal components, we need to compute the estimate $\overline{y}_i$ using the dot product, the cosine similarity, and the correlation measure. Then, as before, since after fitting q-PCA we are also able to get an estimate of the eigenvalue $\overline{\lambda}_i$, we can compute all the thresholds and all the anomaly scores, as in the classical case.

Results comparison We execute all the five q-PCA models with the following error parameters:

- q-PCA30: $\epsilon_\theta = 1$, $p = 0.30$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.
- q-PCA40: $\epsilon_\theta = 1$, $p = 0.40$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.2$.
- q-PCA50: $\epsilon_\theta = 1$, $p = 0.50$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.
- q-PCA60: $\epsilon_\theta = 1$, $p = 0.60$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.
- q-PCA70: $\epsilon_\theta = 1$, $p = 0.70$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.

We obtain the following models:

- q-PCA30 with 1 major and 32 minor components (with respect to 2 major and 31 minor of the classical case).
- q-PCA40 with 2 major and 32 minor components (with respect to 2 major and 31 minor of the classical case).
- q-PCA50 with 3 major and 32 minor components (with respect to 3 major and 31 minor of the classical case).
- q-PCA60 with 4 major and 32 minor components (with respect to 5 major and 31 minor of the classical case).
- q-PCA70 with 7 major and 32 minor components (with respect to 7 major and 31 minor of the classical case).

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.4962	0.4758	0.5302	0.5862	0.3727	0.3570	0.4976	0.4846	0.4410	0.3980
2%	0.6278	0.5714	0.6388	0.6854	0.5375	0.5104	0.6341	0.6263	0.5805	0.7384
4%	0.8468	0.8493	0.8523	0.8822	0.7775	0.7368	0.8488	0.8688	0.8525	0.8961
6%	0.9510	0.9541	0.9112	0.9127	0.9458	0.9206	0.9518	0.9629	0.9499	0.9696
8%	0.9660	0.9673	0.9415	0.9419	0.9646	0.9570	0.9666	0.9682	0.9729	0.9756
10%	0.9749	0.9770	0.9780	0.9766	0.9740	0.9717	0.9779	0.9779	0.9752	0.9778

Table 4.13: Recall comparison for classical (c) and quantum (q) ensemble principal components classifiers over CICIDS.

It is very interesting to notice that by exploiting the ensemble model, we have a boost in performance, particularly in recall. Indeed, as we can note, we have an increase of $\approx 30\%$ over the principal components classifier model with major and minor components (except for $\alpha = 1\%$), reported in Table 4.10. We can also see that the precision results, in Table 4.14, are lower than the ones reported in Table 4.11. But, in comparison, the

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.9580	0.9431	0.9565	0.9611	0.9244	0.9194	0.9545	0.9446	0.9459	0.9530
2%	0.9228	0.9075	0.9241	0.9279	0.8982	0.8935	0.9295	0.9180	0.9196	0.9407
4%	0.8921	0.8868	0.8919	0.8947	0.8847	0.8783	0.8960	0.8915	0.8972	0.9079
6%	0.8718	0.8669	0.8607	0.8607	0.8619	0.8623	0.8702	0.8687	0.8648	0.8725
8%	0.8429	0.8383	0.8368	0.8365	0.8402	0.8392	0.8394	0.8432	0.8296	0.8345
10%	0.8167	0.8074	0.8169	0.8143	0.8111	0.8126	0.8144	0.8156	0.8014	0.8044

Table 4.14: Precision comparison for classical (c) and quantum (q) ensemble principal components classifiers over CICIDS.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.6538	0.6325	0.6823	0.7283	0.5312	0.5143	0.6542	0.6406	0.6015	0.5615
2%	0.7473	0.7013	0.7554	0.7884	0.6726	0.6497	0.7539	0.7446	0.7117	0.8274
4%	0.8689	0.8676	0.8717	0.8884	0.8276	0.8013	0.8718	0.8800	0.8743	0.9019
6%	**0.9097**	**0.9084**	0.8852	0.8859	**0.9019**	0.8905	**0.9091**	**0.9134**	**0.9054**	**0.9185**
8%	0.9003	0.8982	0.8860	0.8861	0.8981	**0.8942**	0.8985	0.9014	0.8955	0.8995
10%	0.8888	0.8842	**0.8902**	**0.8881**	0.8851	0.8850	0.8876	0.8894	0.8798	0.8827

Table 4.15: F1_score comparison for classical (c) and quantum (q) ensemble principal components classifiers over CICIDS.

increase in recall is much higher than the decrease in precision and this makes the F1-score increases with respect to the previous case, as reported in Table 4.15. The fact that the precision is lower depends always on the fact that using more conditions in OR in the classification procedure (since now we have to consider also the conditions relative to the cosine and correlation measures) increases the probability of classifying a sample as an attack increasing the risk of false-positive cases. But we can see that, with this type of data, the gain in recall is much higher than the risk of increasing the false positive. In Table A.5, we also see that the accuracy benefits from this model. Indeed, it increases, such as for PCA30 with $\alpha = 6\%$, more than 10% with respect to the accuracy corresponding to the PCC model reported in Table A.4.

4.4. PCA with reconstruction loss over CICIDS 2017 and Darknet

In this section, we report the experiments done with the PCA with reconstruction loss model over CICIDS and Darknet datasets. For what concerns the CICIDS dataset, we use two different versions: the CICIDS_ALL and the CICIDS_DDoS datasets. In the first one, we use a test set very unbalanced towards the anomaly samples with the same proportions used in Verkerken et al. [49]. In the second, we use an unbalanced test set but

towards the benign samples, considering as anomalies only the DDoS attacks, as reported by Yulianto et al. [54]. It is important to underline that, for this model, we need also a validation set since we need to tune some hyperparameters to choose the best model.
We briefly remember how this model works: we project the original data into the PCA feature space that we obtain by fitting PCA over normal samples. Then, we map back the transformed data into the original feature space. We compute the loss (or anomaly score) ℓ between the original and reconstructed sample and we classify a sample as anomalous if the corresponding anomaly score $\ell > t$, where t is the outlier threshold that we find in the hyperparameter tuning.

Preprocessing We follow the preprocessing procedure reported by Verkerken et al. [49]. Therefore, we encode the categorical features, we find the constant features and we remove them, paying attention to consider training and test sets with no observations with the same Flow ID. For what concern the normalization, we can use many normalizers such as *StandardScaler()*, *QuantileTransformer()*, and many others. The choice of which to use is made through hyperparameter tuning, looking at which one results in better performance on the validation set.

Validation process We use the Optuna library [1] of Python to perform hyperparameter optimization. It works by executing many times the function that we want to optimize, searching for the global minimum of that function. Therefore, we specify all the parameters that we want to optimize, which in our case are: the normalization methods, the classification threshold, the number of components that the PCA model retains, and the number of quantiles, in the case we choose the QuantileTransformer as normalization process. Then, Optuna chooses a normalization method (for example, QuantileTransformer with the relative number of quantiles) and normalizes the dataset. After that, we fit the PCA model with n_c number of components over the normalized training set. Finally, we apply the procedure of the PCA with reconstruction error model on the validation set. So, we map each observation of the validation set in the PCA feature space and we map back the projected data in the original feature space using the fitted PCA. Then, we compute the loss using the residual sum of square between all the original observations and the reconstructed ones. Based on a threshold t, we classify the specific observation as an anomaly or not. This process is repeated many times by Optuna (we can specify the number of iterations) that, as said, tries to minimize the objective value, which in our case is the F1-score (we choose this metric since, being the harmonic mean of the precision and recall, it is like if we minimize together those two metrics). Obviously, we want to maximize the F1-score, so we minimize the negative F1-score, which is the same

as maximizing the aforementioned metric.

Classical model building. Once finished with the validation step, we fit a PCA model with the training set composed only of normal instances and normalized with the normalizer found in the validation phase, specifying the best number of components found in the optimization step. This is the final model we use to transform and reconstruct each sample.

Quantum model building We fit the q-PCA model with the normalized training set composed of only normal instances specifying the number of components to retain. Here there is a little difference with respect to the usual procedure. Indeed, the quantum binary search requires the percentage of variance p that we want to retain (together with the error parameters ϵ_θ and η), and it provides an estimate of θ. Now, since we have the number of principal components that we want to keep from the classical hyperparameter tuning, we can classically compute what percentage of total variance it corresponds to, using the factor score ratio of each component. More precisely, after classically decomposing the input data matrix with SVD, we get a description of singular vectors, singular values, factor scores, and factor score ratios. Therefore, if we want to retain n_c principal components, the sum of the first n_c factor score ratios corresponds to the percentage of variance retained by the first n_c components. Then, we use that percentage as input parameter p into the quantum binary search. In this way, we retrieve θ that we use in quantum top-k singular vectors extraction to extract an estimate of top-k principal components with error δ. Once the principal components have been extracted, we are able to map and re-project back the data using quantum mapping and inverse quantum mapping methods respectively to compute the reconstruction loss and perform the detection (we described these two methods in the previous chapter).

Results comparison over CICIDS_ALL dataset with a test set unbalanced towards anomalies In this experiment, we consider a training set of 50,000 normal samples, a validation of 60,000 normal and 166,966 attacks, and a test set of 140,000 normal and 389,590 attacks [49]. In the hyperparameters tuning, we found that the best PCA model has 12 principal components (which retain 94.88% of the variance) and it is fitted with a training set normalized with QuantileTransformer with 751 quantiles. The outlier threshold is $t = 0.4259394035517815$. Therefore, each sample whose anomaly score is higher than t is classified as anomalous.
The error parameters used in the quantum model are $\epsilon_\theta = \epsilon = 3$, $p = 94.88\%$, and $\eta = 0.04$. With these error parameters, we are able to extract the top-11 principal components

in the quantum model. For this type of experiments, it is even more important to be sure to be consistent, with the classical case, in the number of principal components retained since all the other parameters such as the number of quantiles and the outlier threshold are optimized together with the number of principal components. Therefore, since in the quantum models we reuse the parameters found in the classical hyperparameters tuning, it is important that also the number of principal components is consistent with the classical case.

Recall	Precision	F1-Score	Accuracy
0.9585	0.9203	0.9390	0.9085

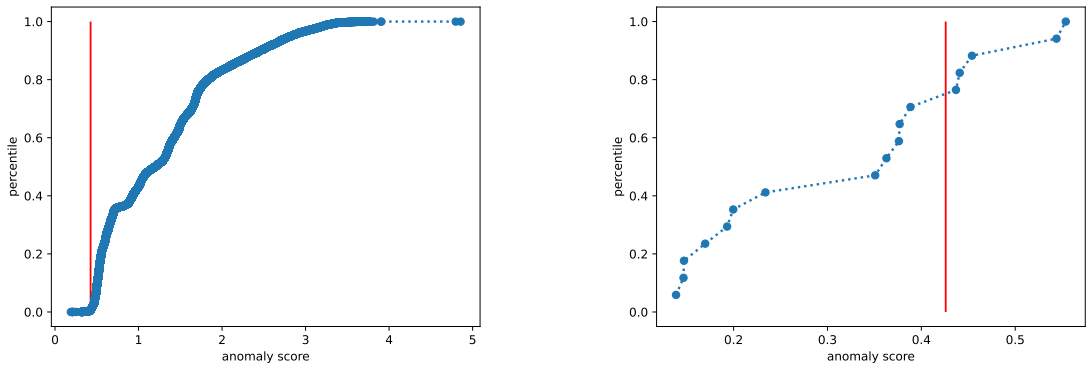
Table 4.16: Results of PCA with reconstruction loss over unbalanced CICIDS 2017 towards anomaly samples.

δ	Recall	Precision	F1-Score	Accuracy
0.01	0.9643	0.9072	0.9349	0.9012
0.1	0.9644	0.9071	0.9349	0.9012
0.9	0.9649	0.9015	0.9321	0.8966
2	0.9807	0.8743	0.9244	0.8821

Table 4.17: Results of q-PCAs with reconstruction loss over CICIDS 2017 unbalanced towards anomalies with different values of δ error.

As we can see, by increasing the error δ, precision and accuracy decrease while recall increases. In this experiment, we can notice the impact of the balancing of the dataset in the performance. Indeed, even inserting $\delta = 2$ as error, the performance do not decrease so much compared to the classical one. There are two main reasons. The first is related to the fact that tomography returns estimates with an error lower than the error δ inserted, as shown in Section 3.2.2. This allows us to increase the error and still get good performances. The second reason is related to the balancing of the dataset. Indeed, in this case, anomalies are much more than normal samples in the test set. Therefore, the unbalancing towards anomalies still helps to achieve good performance even with larger errors.

In this experiment, we also report the cumulative anomaly score distribution for different types of attacks. In this way, we can see which are the type of attacks well recognized by this model and which are not. Here we only report the plot of the distribution of two types of attacks: DDoS, in Figure 4.2a, and SQL Injections, in Figure 4.2b, since they well describe two extreme cases. In Section A.2 in Appendix A, we report the plots for all the other types of attacks present in CICIDS dataset. On the x-axis, we report the



(a) Cumulative anomaly score distribution for DDoS attack.

(b) Cumulative anomaly score distribution for SQL Injection attack.

Figure 4.2: Cumulative anomaly score distribution for DDoS and SQL-Injection attacks. The red vertical line indicates the outlier threshold. The blue points at the right of the threshold are the ones correctly classified as attacks by the model. On the x-axis, we report the anomaly scores for each observation, while the y-axis represents the corresponding data percentile.

anomaly score for each observation, while the y-axis represents the corresponding data percentile. Therefore, taking a specific anomaly score value x, we can see, by checking its corresponding percentile value y, how many data ($y\%$) have an anomaly score less than x. Moreover, these plots indicate that if a blue dot is at the right of the threshold is correctly classified as an attack from the model (indeed, the model classifies a sample as an attack if its anomaly score is greater than the outlier threshold). If not, it is a false-negative observation, which means that it is classified as a normal sample but, in fact, it is an anomaly. We can notice that this model is able to detect the majority of the DDoS attacks, while it is more difficult to detect attacks like SQL-Injections. For this reason, in the next experiment, we test this model over a test set where we consider only DDoS as attacks.

Results comparison over CICIDS_DDoS dataset with the test set unbalanced towards normal samples Considering a training set of 158,000 normal samples, a test set of 50,000 normal samples and 12,000 anomalies (as said, we use the same train/test split reported in [54]) and a validation set of 50,000 normal and 10,000 anomaly samples where, in this case, we consider only the DDoS attack as anomaly. With the hyperparameter tuning, we found that the best PCA model has 32 principal components (that retain 99.75% of the variance) and is fitted on a training set normalized with Quantile-

Transformer with 24 quantiles. The outlier threshold is $t = 0.06632108379654125$. For the quantum case, we are able to extracts exactly 32 principal components, as in the classical case, with the following error parameters in quantum binary search: $\epsilon_\theta = \epsilon = 0.3$, $p = 99.75\%$, and $\eta = 0.00075$.

Recall	Precision	F1-Score	Accuracy
0.9912	0.9128	0.9504	0.9808

Table 4.18: Results of PCA with reconstruction loss over unbalanced CICIDS 2017 towards normal samples.

δ	Recall	Precision	F1-Score	Accuracy
0.01	0.9912	0.9128	0.9504	0.9808
0.1	0.9917	0.9123	0.9503	0.9808
0.9	0.9979	0.7131	0.8318	0.9252
2	1.0	0.2730	0.4289	0.5066

Table 4.19: Results of q-PCAs with reconstruction loss over CICIDS 2017 with different values of δ error.

Here the interesting thing to notice is that comparing quantum results with the ones in Table 4.18, we see that, also with an error $\delta = 0.1$, we can achieve almost the same performances as the classical version of the model. This is mainly due to the fact that tomography returns an estimate with an error that is smaller than δ, as seen in the previous experiment. However, in this case, by having more normal samples than anomalies, the balancing does not help in maintaining good performances as δ increases. Indeed, as expected, at the increase of δ, precision and accuracy decrease much more than that reported in Table 4.17. With $\delta = 2$, we get 1.0 of recall and about 0.27 of precision. This means that the model almost always predicts attack and it is also confirmed by the accuracy which goes to 50%. We remember that, when we estimate a vector, an error of $\delta = 2$ means that the estimated vector goes in the opposite direction with respect to the real one and so it is the maximum error that we can have when estimating a vector.
Comparing also with the results reported by Yulianto et al. [54], where they use the AdaBoost model fitted over a training set and evaluated over a test set with the same proportions as ours, we found that our model outperforms the best AdaBoost. We can also see that, with a test set composed of much more normal samples than anomalies (like in a real scenario) and considering only the DDoS attacks, we obtain very good performances. A very high accuracy, which in this case means that 98 out of 100 observations are correctly classified, together with a 95% of F1-score indicate very good performances. These results confirm the goodness of this model in detecting DDoS attacks, as shown in Figure 4.2a.

Results comparison over Darknet dataset We show the results obtained with PCA with reconstruction loss model over the Darknet dataset. More precisely, we use a test set of 21,000 normal plus 14,000 anomalies samples (remember that in this dataset we consider *Tor* and *VPN* as the same anomalies), using a PCA model fitted over a training set of 50,000 normal samples tuned over a validation set of 20,000 normal plus 10,000 anomalies. In this case, the best hyperparameters found are $n_c = 35$ principal components (which retain 99.65% of the variance) and outlier threshold $t = 0.4438816547139376$. We do not report the number of quantiles since the best normalizer found is the StandardScaler which takes no input parameters. For the quantum model, we use the following error parameters: $\epsilon_\theta = \epsilon = 0.35, \eta = 0.0011$, and $p = 99.65\%$. With these error parameters, we extract 34 principal components which is very consistent with the classical case.

Recall	Precision	F1-Score	Accuracy
0.8417	0.7371	0.7859	0.8135

Table 4.20: Results of PCA with reconstruction loss over Darknet dataset.

δ	Recall	Precision	F1-Score	Accuracy
0.1	0.8440	0.7318	0.7839	0.8107
0.9	0.8708	0.6812	0.7644	0.7817
2	0.9147	0.5997	0.7244	0.7169

Table 4.21: Results of q-PCA with reconstruction loss over Darknet dataset.

Looking at the classical results reported in Table 4.20, we can see that they are slightly better than the one of the 1D-CNN model reported in Table 6 by Habibi Lashkari et al. [17] (only the precision is slightly worst), where deep learning models over the Darknet dataset are used. Of course, we do not know how they preprocess data and how they split training and test set. So, this is not a true faithful comparison, but it is just to have an idea of the goodness of this model. For what concerns the comparison between classical and quantum results, we can notice the same behavior of the previous experiments with the CICIDS 2017 dataset. Indeed, we can see that with $\delta = 0.1$, we achieve almost the same classical performances and as δ increases, recall increases while precision and accuracy decrease.

4.5. PCA and K-Means over KDDCUP

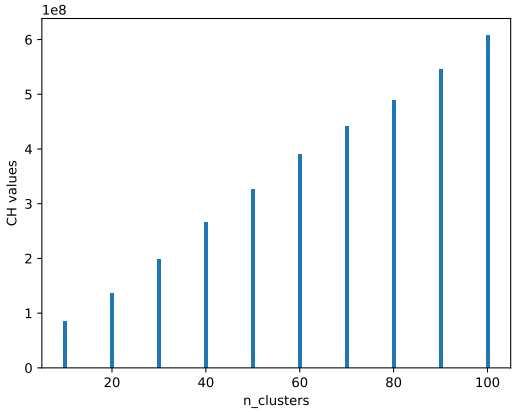
The last shown experiment is the one in which we use both PCA and K-Means. We replicate the experiments reported by Peng et al. [35]. As said in the corresponding paragraph in Section 1.2, this is not a true anomaly detection model since the goal is to evaluate the goodness of the clustering by extracting the CH index: higher values of CH indicate that clusters are dense and well-separable. The score is defined as the ratio between the within-cluster dispersion and the between-cluster dispersion. So it is not a classification model but the objective is to find the best clustering between the data.

Preprocessing As depicted before, for this experiment, we use the KDDCUP 99 dataset. We follow the preprocessing reported by Peng et al. [35]: we encode the categorical features with random numbers and then normalize the dataset using the MinMaxScaler to get all the values between [0,1].

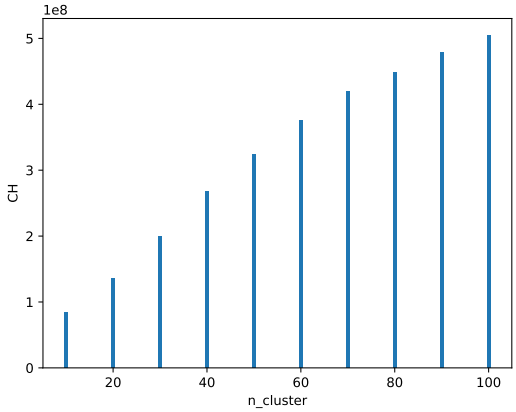
Classical model building By trial and error, we find that to reproduce the paper's results, we need to set the number of principal components $n\_components = 1$. Once fitted the PCA model over all the preprocessed data, we project all the data into the new 1-dimension PCA space using the *transform()* method. Over these transformed data, we apply the K-Means clustering with different number of clusters $n_k = [10, 20, 30, \ldots, 100]$. Using the already implemented *calinski_harabasz_score()* method of *Sklearn*, we compute the specific CH index and we plot the results.

Quantum model building In this case, we know that the PCA model has to retain only the first principal component. Therefore, we classically compute the percentage of variance to which the first principal component corresponds to by looking at the corresponding factor score ratio that we found to be ≈ 0.6. We use this value as input parameter $p = 0.6$ into the quantum binary search, with $\epsilon_\theta = \epsilon = 5$ and $\eta = 0.1$. Then, the procedure is always the same: once retrieved θ, we use it to extract top-k components with error $\delta = 0.1$. After retrieving a classical description of the first principal component, we use it to project the data into the q-PCA feature space, using the *quantum_transform()* method. Now, we apply the q-Means algorithm over these 1-dimensional data with error $\delta = 0.0005$, as required in Algorithm 3.4, specifying the number of clusters n_k. Finally, once obtained a classical description of the clustering, we compute the CH score as in the classical case.

Comparison results If we take a look at Figure 4.3a, we can see that we are able to classically reproduce the CH results reported by Peng et al. [35] in Figure 14. Moreover, also with quantum models with the error parameters previously specified, we are able to get almost the same CH results with respect to the classical case.



(a) CH results with different number of clusters n_k.



(b) Quantum CH results with different number of clusters n_k.

Figure 4.3: CH comparison between PCA and K-Means and q-PCA and q-Means.

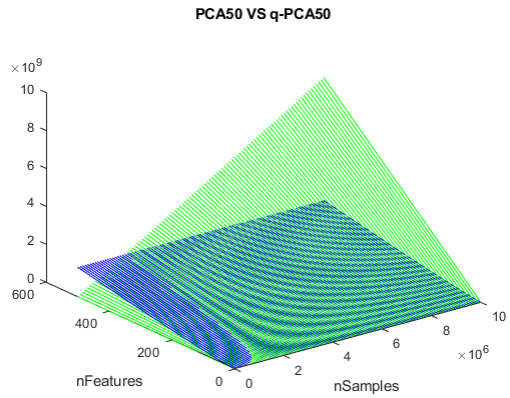
4.6. Runtime comparison and analysis

We conclude this experiment chapter with the comparison of the running time between the classical and quantum extraction of anomaly detection models. This topic is very important because it is the conclusion of our work. Indeed, in the previous sections, we report our study about the error that we have to consider in quantum algorithms and how it affects the detection capabilities of our models. Therefore, after demonstrating that we can obtain the same performances as the classical model, the last part of our research concerns the study of the running time to see when quantum models offer advantages in terms of time with respect to the classical ones. We already know from the theory that with high-dimensional data quantum machine learning shows a running time advantage over classical machine learning. In our experiment, we start to quantify the number of samples and features needed to observe such a quantum speed-up. We compare the running times by plotting the classical and quantum time complexity as the number of samples and features increases, keeping fixed all the error parameters previously found and fixing the probability of failure to $\gamma = \frac{1}{d}$ (therefore probability of success $1 - \frac{1}{d}$), with d number of features of the input data matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, as reported in the description of Algorithm 3.3 for the extraction of top-k singular vectors.

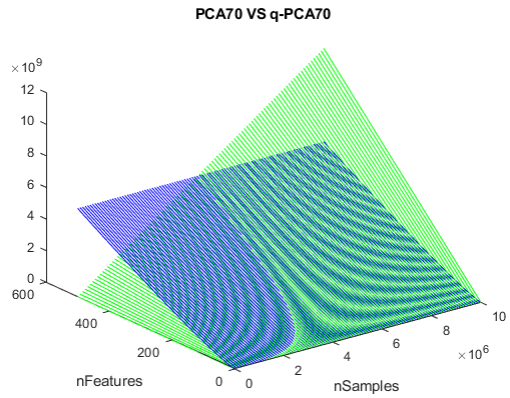
We report three case studies: one for the principal components classifier model with only major components, one for the principal components classifier model with major and minor components, and the other for PCA with reconstruction loss since they cover all the main case studies. For each one of these cases, we analyze the results found with a practical perspective.
An important thing to remark is that we make a comparison between the time complexity of the model's extraction. For example, the time required to store the normalized input data matrix $\boldsymbol{X}$ in the QRAM, in the quantum case, is not considered. Instead, we consider the cost of the necessary steps for the fitting of the model, as the cost of the quantum binary search and the top-k singular vectors extraction procedures. In the same way, we do not consider the classical way to load the data matrix $\boldsymbol{X}$ on the hard disk. Therefore, the quantum and classical running times are comparable.

Principal component classifier with major components over KDDCUP In this first case, we report the running time comparison between the fitting of the principal components classifier model and its quantum version with only major components over the KDDCUP dataset. Just to clarify, we are referring to the experiments whose results are reported in Tables 4.1 and 4.2. We consider PCA models that retain 50% and 70% of the total variance, called PCA50 and PCA70 (or q-PCA50 and q-PCA70 in quantum case).
For what concerns the time complexity for the classical case, since in the model's fitting we search for the principal components that retain 50% and 70% of the variance respectively, there is no need to perform the full SVD. Indeed, we compare with a randomized version of SVD, also present in the Sklearn implementation, which has a cost of $O\left(nd\log(k)\right)$ [18], where n, d are respectively the number of samples and features of the dataset and k is the number of principal components. In this way, we compare the running time of the quantum model with a faster classical algorithm with respect to the classical full SVD.
For the quantum running time, we consider the time complexity of quantum routines used in the fitting of the model, which are the quantum binary search and the quantum top-k singular vector extraction. The first has a cost of $\widetilde{O}\left(\frac{\mu(\boldsymbol{X})\log(\mu(\boldsymbol{X})/\epsilon)}{\epsilon\eta}\right)$ while the latter requires $O\left(\frac{\|\boldsymbol{X}\|\mu(\boldsymbol{X})k\log(k)}{\theta\sqrt{p}\epsilon}\right)$ steps to estimate the top-k singular values, factor score, and factor score ratios, and $O\left(\frac{\|\boldsymbol{X}\|\mu(\boldsymbol{X})k\log(k)d\log(d)}{\theta\sqrt{p}\epsilon\delta^2}\right)$ to estimate the top-k right singular vectors. We remember that ϵ and δ are the error that we tolerate in estimating singular values and singular vectors respectively, θ is the value estimated in the quantum binary search, p is the variance that we want to retain in the principal components, $\|\boldsymbol{X}\|$ is the spectral norm of the input matrix $\boldsymbol{X}$ that we can estimate using the procedure described in Algorithm 2.5, and $\mu(\boldsymbol{X}) = \min_{p\in[0,1]}(\|\boldsymbol{X}\|_F, \sqrt{s_{2p}(\boldsymbol{X})s_{2(1-p)}(\boldsymbol{X}^T)})$ where

$s_p(\boldsymbol{X}) = \max_i \|\boldsymbol{x}_{i,\cdot}\|_p^p$. Therefore, by fixing these parameters we compare the running times at the increasing of n and d. In Figure 4.4, we can see the comparison between the running time of the quantum fitting model (blue) and the randomized classical one (green). We report two plots: one for the model that retains 50% of the total variance (in Figure 4.4a), and the other for the model that retains 70% of the variance (in Figure 4.4b). We remember the quantum parameters that we use in these experiments: $\delta = 0.1, \epsilon = 1, p = 0.70, \epsilon_\theta = 1$, and $\eta = 0.1$ for the q-PCA70 model and the same, but with $p = 0.50$, for q-PCA50. As said, we fix the probability of failure to $\gamma = \frac{1}{d}$. In general,



(a) Running time comparison between classical (green) and quantum (blue) principal components classifier model with only major components over KDDCUP with 50% of the retained variance.



(b) Running time comparison between classical (green) and quantum (blue) principal components classifier model with only major components over KDDCUP with 70% of the retained variance.

Figure 4.4: Running time comparison between classical and quantum principal components classifier model with only major components with different percentages of retained variance over KDDCUP.

these results confirm the already known fact that, with high dimensional data, quantum machine learning is advantageous over the classical one in terms of running time. Indeed, the blue plane at high dimension is below the green one.
But let us see more precisely when we start to have a quantum advantage. Let us focus on Figure 4.4b. We see that, even with a dataset of $\approx 4 * 10^6$ samples and ≈ 50 features (or even less), we note an advantage in using quantum instead of classic machine learning. These number of samples and features are reasonable in intrusion detection tasks since they are in line with samples and features of publicly available datasets. For example, KDDCUP, which was one of the main dataset used in IDS literature before becoming obsolete, contains more than 4 million samples with 41 features. Another frequently used

dataset of that dimension is NSL-KDD, which derives from KDDCUP. With a dataset of that dimension, we find that the classical model's fitting requires $\approx 4.6 * 10^8$ steps against $\approx 3.5 * 10^8$ of the quantum one. We can see that the difference is minimal and, considering the fact that we do not have considered in our computations things like storing the input data matrix in the QRAM, we can say that in real case scenarios, with a dataset of this dimension, probably, is always convenient to use classical machine learning to extract the model. Fixing the number of features to 50 and incrementing the number of samples to 100,000,000, we find that the quantum model requires $\approx 2.3 * 10^7$ steps against $\approx 1.1 * 10^{10}$ of the classical model. Now, with this number of samples, the quantum advantage becomes clearer. Even in this case, 100,000,000 samples are reasonable in intrusion detection tasks. Indeed, we can think of big companies such as Netflix or Amazon which count more than 200 million subscribers. Considering that even non-subscribers can perform intrusions or attacks on the servers of these companies, we can say that, for these big companies, 100,000,000 samples are realistic. Let us suppose that most users are provided with optical fiber with a connection of 10Gbps in upload. Considering the maximum size of a TCP/IP packet which is about 65000 bytes. We can say that, on average, users send 19,230 packets every second. This means that to reach 100,000,000 packets, almost 5000 users have to send packets around the same time, which, as said, is a reasonable number (indeed, 5000 users is almost 0.0025% of the total Amazon or Netflix subscribers). Furthermore, we have to take into account that, nowadays and even more so in the future, with the optic fiber there will be the possibility to reach more than 10Gbps transmission rate and also that network packets rarely reach the dimension of 65K. Therefore, some users could send more than 19,230 packets per second.
In Figure 4.5, we report statistics about the total visit in November 2021 for the most popular websites worldwide. As we can see, companies like Google, YouTube, Facebook, or Amazon deal with billions of visits in just a month. This is to highlight even more that the amount of data and network traffic that these companies have to manage is very high and, in the future, it is destined to grow more and more. For instance, in the case of Google, if we assume to fit the model whose running time is reported in Figure 4.4b "only" with the data relative to the visits in November 2021 (therefore 45,000,000,000 samples, assuming always data described by 50 features), we find that quantum model requires $\approx 3.5 * 10^8$ steps against $\approx 5.1 * 10^{12}$ of the approximated classical one.
Returning to the plots in Figure 4.4, by fixing the number of features to 500 and incrementing the number of samples, we can see that the advantage of using quantum increases even more. However, these dimensions do not conform to those of the publicly available intrusion detection dataset. In this case, a more complex task such as malware detection could fit better. For instance, Shabtai et al. [42] in their analysis for classifying Android

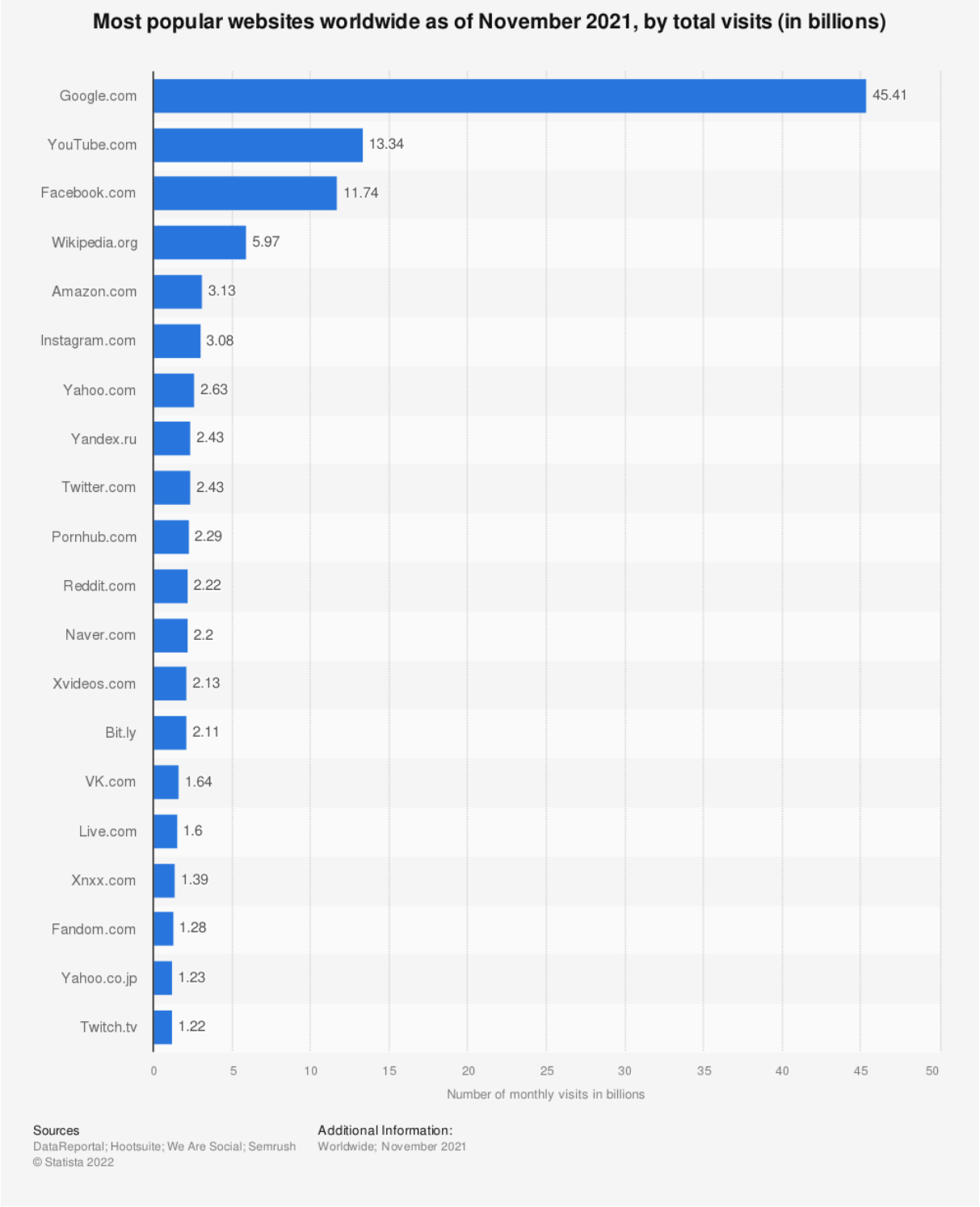


Figure 4.5: Ranking of the most accessed websites in November 2021 by Clement [12].

applications with machine learning used 100, 200, 300, 500, and 800 features showing that with 800 features they achieve the best performances. Furthermore, if we consider the Drebin dataset [3], which is one of the most famous malware detection datasets, we see

that it is composed of 545,356 features which is one of the main obstacle for building a classical machine learning classifier to detect Android malware applications. In a survey of machine learning for malware detection, reported by Gibert et al. [15], we can see that there are some malware detection models that use 4000 or even 50,000 features to detect malware. These data confirm that, due to the increasing complexity of malware, we might need even more features to describe them, resulting in high-dimensional data. This could be the subject of future research.
A final comment on the comparison between running times reported in Figures 4.4a and 4.4b. As expected, we notice that the running time of PCA/q-PCA50 is lower than the one of PCA/q-PCA70. We expect this result since PCA50, retaining only 50% of the variance instead of 70%, is a lighter model than PCA70. Therefore, with the same performance (as shown in Table 4.1), in a real scenario, one might consider using the lighter model since it requires less time to fit.

Principal component classifier with both major and minor components over CICIDS 2017 The second comparison is between the principal components classifier model with both major and minor components and its quantum counterpart, retaining 70% of the variance. We refer to the models whose results are reported in Table 4.12. In this case, to measure the classical running time, we consider the classical full SVD time complexity $O\left(nd^2\right)$, with n and d number of samples and features respectively. This is because we need all the components that we will then divide in major and minor components. Indeed, the model initially retains all the components and then, based on the retained variance that we specify and on the magnitudes of the specific eigenvalues, we distinguish the major components from the minor ones. For the quantum computation, instead, we consider the running time of quantum binary search and of the quantum top-k singular vectors extractor, to estimate the top-k components, plus the running time of the quantum least-r singular vectors extractor to estimate the minor components which is $\widetilde{O}\left(\frac{\theta_{minor}}{\sigma_{min}} \frac{\mu(\boldsymbol{X})}{\epsilon} \frac{rd}{\sqrt{p_{min}}}\right)$, where θ_{minor} is the custom θ passed to the function to extract the least singular values and p_{min} is the variance retained from the least components.
In this case, the quantum parameters are: $\delta = 0.1, \epsilon = \epsilon_\theta = 1, p = 0.70$, $\eta = 0.1$, $\gamma = \frac{1}{d}$, and $\theta_{minor} = \sqrt{0.20(n-1)}$, with $n = 5000$.

The first thing to notice is that, with respect to Figures 4.4a and 4.4b, we reach higher values of the running time. We expect this result since we compare quantum running time with the classical full SVD complexity instead of the randomized one. Moreover, for the quantum case, we not only use the binary search and the top-k right singular vectors extraction but also the least-r one, which increases the quantum running time. We can note, as expected, an advantage in using quantum model extraction with a dataset

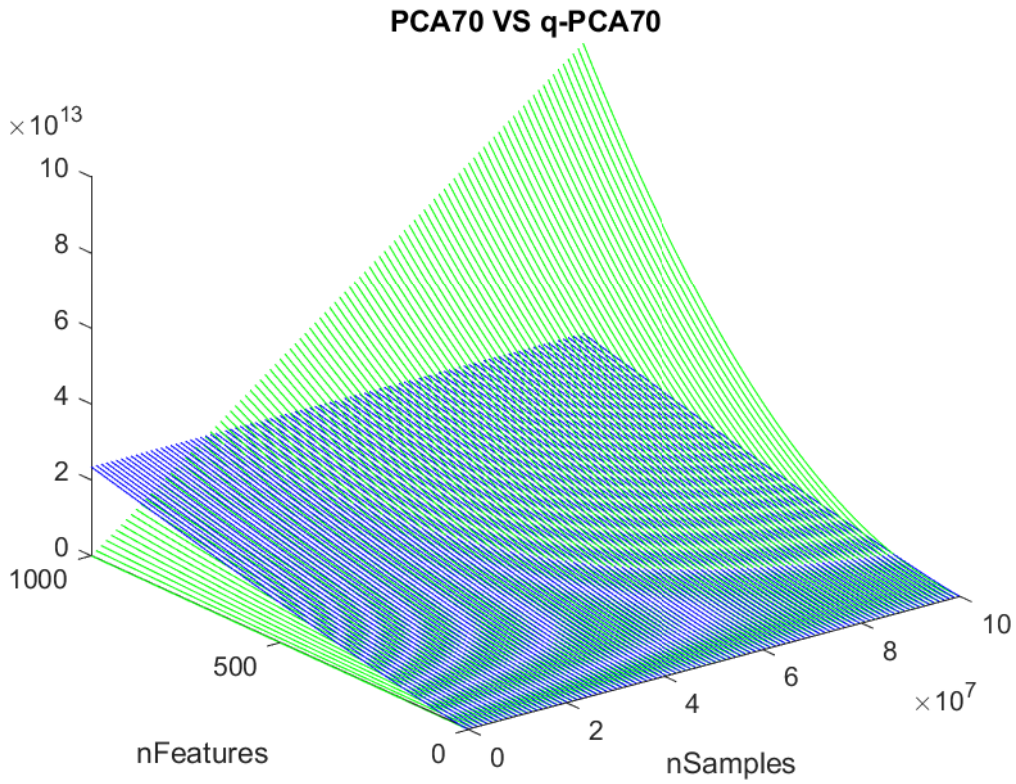


Figure 4.6: Running time comparison between classical (green) and quantum (blue) principal components classifier with both major and minor components over CICIDS 2017 data.

composed of a large number of samples at very high dimensions. But we need to take into account that we compare quantum time complexity with the classical full SVD one. Indeed, the fact that we do not have an advantage in using quantum with respect to the classical full SVD with a dataset of $\approx 3 * 10^7$ samples of ≈ 500 features shows us that, for this type of problem where we need to extract also the minor components, quantum machine learning is not so advantageous over the classical machine learning. Moreover, it is very likely that if we use a more effective classical algorithm to extract the least components (such as Minor Component Analysis (MCA) [29]) as we do for the principal ones with the randomized SVD, the quantum advantage we have would be reduced even more, making its practical application in intrusion detection very difficult in this specific case. It would be interesting to consider this type of model in complex tasks that expect higher dimensional data, such as malware detection.

Finally, we remember that the study of the running time for the ensemble model is equal

to the current one since, as described in Section 4.3, the extraction of the ensemble model is equal to the principal components classifier model one.

PCA with reconstruction loss over CICIDS 2017 This is the last case of running time comparison that we report. For the classical computation, since we want to compare our quantum results with a faster classical top-k principal components extraction method than the full SVD one, we use the randomized version of SVD which has a time complexity of $O\left(nd\log(k)\right)$.
For the quantum counterpart, we use the same computation of the first case shown: $\widetilde{O}\left(\frac{\mu(\boldsymbol{X})\log(\mu(\boldsymbol{X})/\epsilon)}{\epsilon\eta}\right)$ for the quantum binary search and $\widetilde{O}\left(\frac{\|\boldsymbol{X}\|\mu(\boldsymbol{X})k}{\theta\sqrt{p}\epsilon}\right)$ to estimate the top-k singular values, factor score, and factor score ratios, plus $\widetilde{O}\left(\frac{\|\boldsymbol{X}\|\mu(\boldsymbol{X})kd}{\theta\sqrt{p}\epsilon\delta^2}\right)$ to estimate the top-k right singular vectors. The quantum parameters that we use are $\delta = 0.1, \epsilon =, \epsilon_\theta = 0.3$, $\gamma = \frac{1}{d}$, and $\eta = 0.00075$, which are the errors we tolerate to best match classical performances, as reported in Table 4.19. We do not consider $\delta = 0.01$ since we have seen that, with $\delta = 0.1$, we obtain the same performances and, at the same time, we impact less on the running time. As we can see in Figure 4.7, the trend of the running time is very similar to the one reported in the first case example. Here, we can notice how the error parameters influence the quantum running time. Indeed, the order of magnitude in the z-axis is much greater with respect to the one reported in Figures 4.4a and 4.4b. For instance, just by taking look at η, we can see that from 0.1 it goes to 0.00075, with a difference of almost three orders of magnitude, which reflects in a higher running time with respect to the ones reported in Figures 4.4a and 4.4b. We have to underline that even the greater number of principal components retained by this model with respect to the one reported in the first running time comparison case makes the running time increase. Indeed, we go from 10 components retained in the model which retains 70% of the variance in the first case example to 32 components (that retain $p = 99.75\%$ of the variance) in the current model. As we can see, the increase in the running time reflects in observing a quantum speed-up at higher number of samples and features with respect to the cases reported in Figure 4.4a and 4.4b.

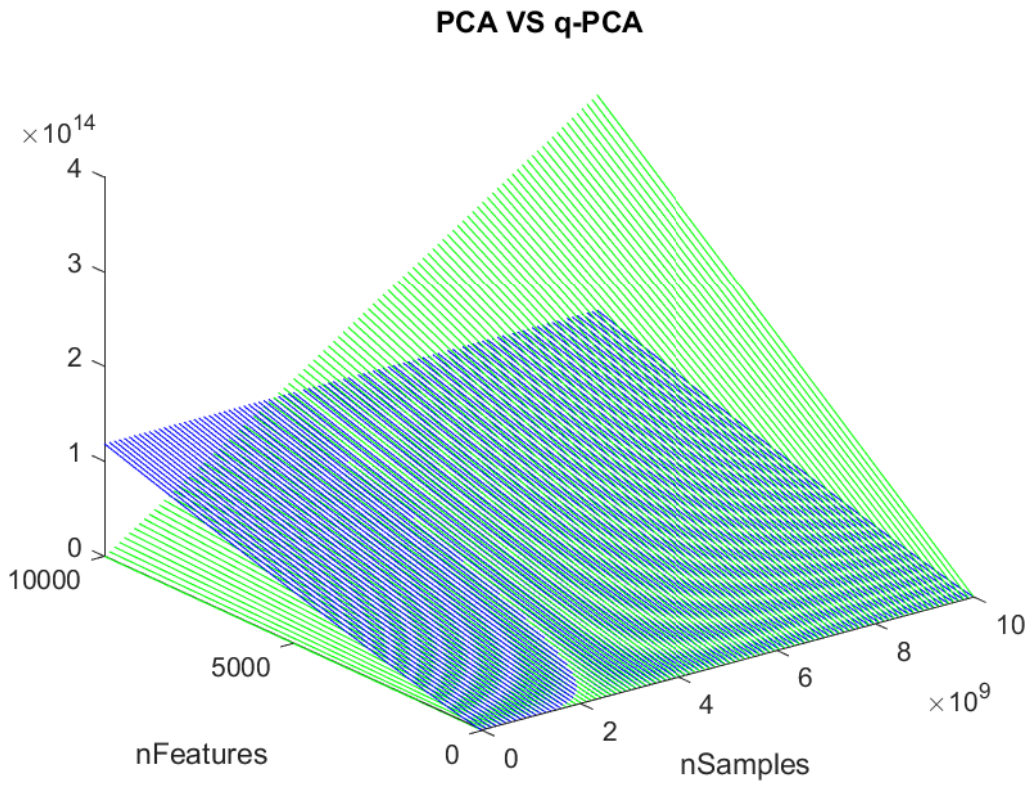


Figure 4.7: Running time comparison between PCA (green) and q-PCA (blue) with reconstruction loss model fitting over CICIDS 2017 data.

5 | Open research directions

It is fair to say that this work cannot be taken as an absolute demonstration that quantum machine learning will surely pay off cybersecurity in the future, even for the simple fact that we have not compared ourselves with the state-of-the-art anomaly detection models such as autoencoders and neural networks. However, with this work, we pave the way towards a more in-depth study and research of the use of quantum machine learning in cybersecurity, starting to quantify the advantages and disadvantages of quantum over classical machine learning on real data.
In this chapter, we report a brief discussion for each of the main contributions of this work, trying to clarify the limitations and the possible future developments.

Sqlearn framework We help the future research in this field, by providing an open-source framework to simulate quantum algorithms. As we have seen, we can simulate quantum machine learning algorithms by simulating the approximation error to see how it affects the performance with respect to classical machine learning. Moreover, with this framework, one can also study the probability of failure of quantum algorithms by playing with the γ parameter as we have done with the parameters ϵ, δ, and η to study the approximation error. We have focused only on the study of the error since to study the probability of failure we should have domain knowledge of the problem that we are dealing with. For instance, in intrusion detection, we can accept a specific probability of failure of the models that, maybe, we could not accept in fraud detection in banking transactions, as it may require a much more stringent failure probability. An in-depth study of the failure probability would allow an even more detailed assessment of the running time of quantum machine learning algorithms.
Another development concerning the framework would be to integrate it with new quantum machine learning algorithms and simulate them to solve intrusion detection problems.

Tomography insights We verify the correctness of the Sqlearn framework by simulating the main quantum routines such as amplitude estimation, phase estimation, and state tomography. Precisely for the latter, we find interesting insights into the number of measurements that are needed to get estimates with error δ: we find that by performing

tomography with $N = \frac{36d \log d}{\delta^2}$ measures, as specified by the theoretical bound, we get estimates with much lower error than δ. We show that even performing tomography with far fewer measurements than N, we get estimates that are δ-close to the true values. This does not mean that the theoretical bound is not correct also because it is reported by Kerenidis and Prakash [24] which is one of the most cited papers on this subject. Possible future work in this direction could be exactly to understand if there are cases of vectors for which tomography requires exactly N measures to estimate vectors with error δ.

Application to cybersecurity We exploit the Sqlearn framework to simulate quantum machine learning algorithms to solve intrusion detection problems. We propose many anomaly detection models by comparing their classical performances with their quantum counterparts. We show that we can find cases of approximation errors to best match classical performances. We have also seen that the performances decrease as the error increases. We also show the importance of balancing the dataset: increasing the error over a very high unbalanced dataset does not affect the performance as in a more balanced dataset.
Surely, future work will be to simulate other quantum machine learning algorithms, such as quantum Gaussian mixture models or SVM, to see the behavior of the error in the application of these algorithms to solve intrusion detection problems. Indeed, we have focused on PCA and K-Means algorithms but there are much more efficient algorithms to solve anomaly detection problems. The point of arrival will be to simulate the quantum counterpart of the state-of-the-art anomaly detection models.
One of the main limitations in doing these experiments is the scarcity of intrusion detection data publicly available. Therefore, simulating quantum machine learning algorithms also on other datasets than KDDCUP, CICIDS or Darknet would be appropriate.

Ensemble extension In simulating quantum machine learning models, we also extend the model initially proposed by Shyu et al. [45] by adding cosine similarity and correlation measures in the computation of the model. We have seen that this has great benefits as performances improve on the CICIDS dataset. Trying other similarity measures instead of cosine or correlation would be interesting to see how they affect the results.

Running time results For what concerns the running time, we have seen with our experiments that it is not a utopia to think of companies that build their network defense systems on quantum machine learning in the near future. Just think, for example, of companies such as Netflix or Amazon Prime which today have more than 200 million subscribers. Considering that even non-subscribers can send packets, connect, or perform

intrusions and attacks on the servers of these companies, we can understand how the network traffic that these companies must and will have to manage will be increasingly high. Confirming this, recently, big companies such as Microsoft [48] and Amazon [5], or services such as GitHub [27] and WordPress [41] reported large-scale cyberattacks, in the order of millions of packets per second. Moreover, in general, these cyberattacks not only create temporary inconvenience to both companies and users but also huge losses of money as reported by Bandwith, which estimated a loss of 9-12 million dollars due to a cyberattack [40]. With the advent of new technologies that provide a very high transfer rate (such as 5G technology) and with a world that is digitizing more and more thanks to IoT, the network connections and the cyberattacks will become more frequent and even larger. As a consequence, we will be dealing with millions and millions of network data that, as we have seen in our running time results, make quantum machine learning very promising in this type of field.

6 | Conclusion

Contributions In this thesis, we try to combine quantum machine learning with cybersecurity by studying if and how quantum machine learning algorithms can be useful to solve cybersecurity tasks. To simplify the simulation of quantum machine learning algorithms, we provide an open-source framework. The framework rests on the implementation of the main quantum routines such as phase estimation, consistent phase estimation, amplitude estimation, and state tomography. We perform experiments and numerical simulations for these implemented quantum routines to verify their correct functioning. We find interesting insights about vector state tomography which we also exploit in its applications in real machine learning cases. We use these quantum routines in quantum machine learning algorithms that we simulate to solve intrusion detection tasks. We derive possible advantages and disadvantages of using quantum machine learning algorithms in this type of field by studying their running time. We evaluate the running time by fixing the failure probability and by studying the approximation error of quantum machine learning algorithms since they are both running time parameters. We study how the error affects the accuracy in detecting intrusions showing that, with quantum machine learning models, we can achieve the same performances as classical machine learning.
In reproducing anomaly detection experiments already reported in the literature, we also extend an anomaly detection model, showing an improvement in performance over the CICIDS dataset.

Experiments We have performed two types of experiments: in the first one, we simulate quantum algorithms implemented in the Sqlearn framework, such as tomography and phase estimation, to verify their correct functioning. The other experiments regard the previous quantum routines applied to machine learning models used to solve anomaly detection problems. We use three different datasets (KDDCUP, CICIDS2017, and Darknet) to test the performances and the time complexity of our quantum models over those data. In particular, we study the trade-off between approximation error and running time of quantum machine learning algorithms applied in intrusion detection tasks. We discuss the running times with a practical perspective: we find the number of samples

and features for which we get a quantum speed-up over classical machine learning trying to understand in which cybersecurity tasks we deal with datasets of that size.

Future works Surely future work will be to complete the Sqlearn framework with other quantum machine learning algorithms and simulate them to solve intrusion detection problems. We think it can also be useful to concentrate on other cybersecurity problems such as malware detection. This is because nowadays malware have an ever-increasing complexity that could make them suitable for quantum machine learning study. An accurate study of the failure probability of quantum machine learning, as we have done for the error, would be interesting to study running time even more in detail.
Finally, an in-depth study about quantum state tomography to search for cases of vector for which tomography requires exactly $N = \frac{36d \log d}{\delta^2}$ measures to get estimates with an error of δ would be interesting to complete the experiments for vector state tomography.

Conclusions With this thesis, we pave the way toward new research and discoveries in the quantum machine learning area applied in cybersecurity. We think that the framework that we have made available will help to simplify future research in this field. Furthermore, for what concerns our experiments, we found reasonable results that lead us to conclude that quantum machine learning seems to be promising in intrusion detection tasks and, even more so, in more complex tasks such as malware detection. For these reasons, we encourage even more research in these directions.

Bibliography

[1] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[2] J. Allcock, K. Hsieh, I. Kerenidis, and S. Zhang. Quantum algorithms for feedforward neural networks. *ACM Transactions on Quantum Computing*, 1:1–24, 10 2020. doi: 10.1145/3411466.

[3] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 02 2014. doi: 10.14722/ndss.2014.23247.

[4] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *SODA '07*, 2007.

[5] BBC. Amazon 'thwarts largest ever ddos cyber-attack', 2020. URL `https://www.bbc.com/news/technology-53093611`.

[6] A. Bellante. Quantum singular value estimation techniques for data representation, 2020. URL `http://hdl.handle.net/10589/166445`.

[7] A. Bellante, A. Luongo, and S. Zanero. Quantum algorithms for data representation and analysis, 2021. URL `https://arxiv.org/abs/2104.08987`.

[8] G. Brandl. Sphinx documentation. *URL http://sphinx-doc. org/sphinx. pdf*, 2021.

[9] G. Brassard, P. Høyer, M. Mosca, and A. Tapp. Quantum amplitude amplification and estimation. *Quantum Computation and Information*, page 53–74, 2002. ISSN 0271-4132. doi: 10.1090/conm/305/05215. URL `http://dx.doi.org/10.1090/conm/305/05215`.

[10] S. Chakraborty, A. Gilyén, and S. Jeffery. The power of block-encoded matrix powers: Improved regression techniques via faster hamiltonian simulation. *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP*

2019), pp. 33:1-33:14, 2019. doi: 10.4230/LIPICS.ICALP.2019.33. URL http://drops.dagstuhl.de/opus/volltexte/2019/10609/.

[11] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), jul 2009. ISSN 0360-0300. doi: 10.1145/1541880.1541882. URL https://doi.org/10.1145/1541880.1541882.

[12] J. Clement. Leading websites worldwide 2021, by monthly visits, 2022. URL https://www.statista.com/statistics/1201880/most-visited-websites-worldwide/.

[13] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca. Quantum algorithms revisited. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1969):339–354, Jan 1998. ISSN 1471-2946. doi: 10.1098/rspa.1998.0164. URL http://dx.doi.org/10.1098/rspa.1998.0164.

[14] J. Ding and B.-Y. Yang. *Multivariate Public Key Cryptography*, pages 193–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-540-88702-7. doi: 10.1007/978-3-540-88702-7_6. URL https://doi.org/10.1007/978-3-540-88702-7_6.

[15] D. Gibert, C. Mateu, and J. Planes. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 03 2020. doi: 10.1016/j.jnca.2019.102526.

[16] L. K. Grover. A fast quantum mechanical algorithm for database search, 1996. URL https://arxiv.org/abs/quant-ph/9605043.

[17] A. Habibi Lashkari, G. Kaur, and A. Rahali. Didarknet: A contemporary approach to detect and characterize the darknet traffic using deep image learning. In *2020 the 10th International Conference on Communication and Network Security*, ICCNS 2020, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450389037. doi: 10.1145/3442520.3442521. URL https://doi.org/10.1145/3442520.3442521.

[18] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev., Survey and Review section, Vol. 53, num. 2, pp. 217-288, June 2011*, 2009. doi: 10.48550/ARXIV.0909.4061. URL https://arxiv.org/abs/0909.4061.

[19] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus,

S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

[20] A. W. Harrow, A. Hassidim, and S. Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, 103(15), Oct 2009. ISSN 1079-7114. doi: 10.1103/physrevlett.103.150502. URL `http://dx.doi.org/10.1103/PhysRevLett.103.150502`.

[21] M. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, 43:169–188, 1986.

[22] P. Kaye, R. Laflamme, and M. Mosca. *An Introduction to Quantum Computing*. Oxford University Press, Inc., USA, 2007. ISBN 0198570007.

[23] I. Kerenidis and A. Prakash. Quantum Recommendation Systems. In C. H. Papadimitriou, editor, *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*, volume 67 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49:1–49:21, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-029-3. doi: 10.4230/LIPIcs.ITCS.2017.49. URL `http://drops.dagstuhl.de/opus/volltexte/2017/8154`.

[24] I. Kerenidis and A. Prakash. A quantum interior point method for lps and sdps, 2018. URL `https://arxiv.org/abs/1808.09266`.

[25] I. Kerenidis and A. Prakash. Quantum gradient descent for linear systems and least squares. *Phys. Rev. A*, 101:022316, Feb 2020. doi: 10.1103/PhysRevA.101.022316. URL `https://link.aps.org/doi/10.1103/PhysRevA.101.022316`.

[26] I. Kerenidis, J. Landman, A. Luongo, and A. Prakash. q-means: A quantum algorithm for unsupervised machine learning. *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*, 2018. doi: 10.48550/ARXIV.1812.03584. URL `https://arxiv.org/abs/1812.03584`.

[27] S. Kottler. Github ddos attack in 2018, 2018. URL `https://github.blog/2018-03-01-ddos-incident-report/`.

[28] J. Landman. Quantum algorithms for unsupervised machine learning and neural networks, 2021. URL `https://arxiv.org/abs/2111.03598`.

[29] F.-L. Luo, R. Unbehauen, and A. Cichocki. A minor component analysis algo-

rithm. *Neural Networks*, 10(2):291–297, 1997. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(96)00063-9. URL https://www.sciencedirect.com/science/article/pii/S0893608096000639.

[30] A. Luongo. Quantum algorithms for data analysis, 2020. URL https://quantumalgorithms.org.

[31] V. Mavroeidis, K. Vishi, M. D., and A. Jøsang. The impact of quantum computing on present cryptography. *International Journal of Advanced Computer Science and Applications*, 9(3), 2018. doi: 10.14569/ijacsa.2018.090354. URL https://doi.org/10.14569%2Fijacsa.2018.090354.

[32] D. Micciancio and O. Regev. *Lattice-based Cryptography*, pages 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-540-88702-7. doi: 10.1007/978-3-540-88702-7_5. URL https://doi.org/10.1007/978-3-540-88702-7_5.

[33] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge university press, 2010. ISBN ISBN 978-1-107-00217-3.

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

[35] K. Peng, V. C. M. Leung, and Q. Huang. Clustering approach based on mini batch kmeans for intrusion detection system over big data. *IEEE Access*, 6:11897–11906, 2018. doi: 10.1109/ACCESS.2018.2810267.

[36] J. Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, Aug. 2018. ISSN 2521-327X. doi: 10.22331/q-2018-08-06-79. URL https://doi.org/10.22331/q-2018-08-06-79.

[37] P. Rebentrost, M. Mohseni, and S. Lloyd. Quantum support vector machine for big data classification. *Physical Review Letters*, 113(13), Sep 2014. ISSN 1079-7114. doi: 10.1103/physrevlett.113.130503. URL http://dx.doi.org/10.1103/PhysRevLett.113.130503.

[38] M. Saks and S. Zhou. Bphspace(s) in dspace(s3/2). *Journal of Computer and System Sciences*, 58(2):376–403, 1999. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.1998.1616. URL https://www.sciencedirect.com/science/article/pii/S0022000098916166.

[39] V. Scarani, H. Bechmann-Pasquinucci, N. J. Cerf, M. Dušek, N. Lütkenhaus, and

M. Peev. The security of practical quantum key distribution. *Reviews of Modern Physics*, 81(3):1301–1350, Sep 2009. ISSN 1539-0756. doi: 10.1103/revmodphys.81.1301. URL http://dx.doi.org/10.1103/RevModPhys.81.1301.

[40] SEC.gov. Bandwidth announces preliminary third quarter 2021 revenue results exceeding guidance and estimated full year revenue impact of ddos attack, 2021. URL https://www.sec.gov/Archives/edgar/data/1514416/000151441621000280/q32021exh991-preliminaryth.htm.

[41] L. Segall. Wordpress hammered by massive ddos attack, 2011. URL https://money.cnn.com/2011/03/03/technology/wordpress_attack/index.htm.

[42] A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *2010 International Conference on Computational Intelligence and Security*, pages 329–333, 2010. doi: 10.1109/CIS.2010.77.

[43] I. Sharafaldin, A. Habibi Lashkari, and A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *ICISSP*, pages 108–116, 01 2018. doi: 10.5220/0006639801080116.

[44] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, Oct 1997. ISSN 1095-7111. doi: 10.1137/s0097539795293172. URL http://dx.doi.org/10.1137/S0097539795293172.

[45] M.-L. Shyu, S.-C. Chen, K. Sarinnapakorn, and L. Chang. A novel anomaly detection scheme based on principal component classifier. In *in Proceedings of the IEEE Foundations and New Directions of Data Mining Workshop, in conjunction with the Third IEEE International Conference on Data Mining (ICDM'03)*, 01 2003.

[46] A. Ta-Shma. Inverting well conditioned matrices in quantum logspace. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, page 881–890, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320290. doi: 10.1145/2488608.2488720. URL https://doi.org/10.1145/2488608.2488720.

[47] M. Tavallaee, E. Bagheri, W. Lu, and A. Ghorbani. A detailed analysis of the kdd cup 99 data set. *IEEE Symposium. Computational Intelligence for Security and Defense Applications, CISDA*, 2, 07 2009. doi: 10.1109/CISDA.2009.5356528.

[48] A. Toh. Microsoft ddos attack in 2021, 2022. URL https://azure.microsoft.com/en-us/blog/azure-ddos-protection-2021-q3-and-q4-ddos-attack-trends.

[49] M. Verkerken, L. D'hooge, T. Wauters, B. Volckaert, and F. De Turck. Unsupervised machine learning techniques for network intrusion detection on modern data. In *2020 4th Cyber Security in Networking Conference (CSNet)*, pages 1–8, 2020. doi: 10.1109/CSNet50428.2020.9265461.

[50] N. Wiebe, A. Kapoor, and K. Svore. Quantum algorithms for nearest-neighbor methods for supervised and unsupervised learning. *Quantum Information and Computation*, 15:318–358, 03 2015.

[51] Wikipedia contributors. Bloch sphere — Wikipedia, the free encyclopedia, 2022. URL https://en.wikipedia.org/w/index.php?title=Bloch_sphere&oldid=1084268363. [Online; accessed 4-May-2022].

[52] Wikipedia contributors. Quantum logic gate — Wikipedia, the free encyclopedia, 2022. URL https://en.wikipedia.org/w/index.php?title=Quantum_logic_gate&oldid=1070883867. [Online; accessed 4-May-2022].

[53] D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. doi: 10.1109/4235.585893.

[54] A. Yulianto, P. Sukarno, and N. Suwastika. Improving adaboost-based intrusion detection system (ids) performance on cic ids 2017 dataset. *Journal of Physics: Conference Series*, 1192:012018, 03 2019. doi: 10.1088/1742-6596/1192/1/012018.

A | Appendix A

A.1. Supplementary results

In this supplementary section, we complete the results reported in Chapter 4, showing the accuracy results for each experiment.

Principal components classifier with only major components over KDDCUP

We report the error parameters used in quantum models.

- q-PCA30: $\epsilon_\theta = 8$, $p = 0.30$, $\delta = 0.1$, $\epsilon = 8$, and $\eta = 0.2$.
- q-PCA40: $\epsilon_\theta = 1$, $p = 0.40$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.16$.
- q-PCA50: $\epsilon_\theta = 1$, $p = 0.50$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.1$.
- q-PCA60: $\epsilon_\theta = 1$, $p = 0.60$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.1$.
- q-PCA70: $\epsilon_\theta = 1$, $p = 0.70$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.1$.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.8980	0.8956	0.9827	0.9823	0.9778	0.9770	0.9777	0.9811	0.9755	0.9747
2%	0.9001	0.8979	0.9877	0.9856	0.9859	0.9875	0.9833	0.9878	0.9743	0.9735
4%	0.9003	0.8987	0.9825	0.9794	0.9826	0.9825	0.9833	0.9818	0.9776	0.9769
6%	0.8932	0.8927	0.9756	0.9729	0.9744	0.9762	0.9772	0.9755	0.9772	0.9772
8%	0.8871	0.8864	0.9703	0.9967	0.9673	0.9690	0.9737	0.9708	0.9702	0.9696
10%	0.8805	0.8795	0.9621	0.9613	0.9696	0.9610	0.9890	0.9654	0.9653	0.9646

Table A.1: Accuracy comparison for classical (c) and quantum (q) principal components classifier with major components over KDDCUP.

By increasing ϵ and δ in the following way

- q-PCA30: $\epsilon_\theta = 8$, $p = 0.30$, $\delta = 0.9$, $\epsilon = 8$, and $\eta = 0.2$.

- q-PCA40: $\epsilon_\theta = 20$, $p = 0.40$, $\delta = 0.9$, $\epsilon = 20$, and $\eta = 0.16$.
- q-PCA50: $\epsilon_\theta = 20$, $p = 0.50$, $\delta = 0.9$, $\epsilon = 20$, and $\eta = 0.1$.
- q-PCA60: $\epsilon_\theta = 20$, $p = 0.60$, $\delta = 0.9$, $\epsilon = 20$, and $\eta = 0.1$.
- q-PCA70: $\epsilon_\theta = 20$, $p = 0.70$, $\delta = 0.9$, $\epsilon = 20$, and $\eta = 0.1$.

we obtain

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.8980	0.8971	0.9827	0.8980	0.9778	0.6963	0.9777	0.9912	0.9755	0.9762
2%	0.9001	0.8985	0.9877	0.8991	0.9859	0.6927	0.9833	0.9881	0.9743	0.9846
4%	0.9003	0.8991	0.9825	0.9007	0.9826	0.6846	0.9833	0.9822	0.9776	0.9814
6%	0.8932	0.8931	0.9756	0.8933	0.9744	0.6793	0.9772	0.9770	0.9772	0.9760
8%	0.8871	0.8852	0.9703	0.8862	0.9673	0.6716	0.9737	0.9702	0.9702	0.9685
10%	0.8805	0.8796	0.9621	0.8783	0.9696	0.6645	0.9890	0.9635	0.9653	0.9632

Table A.2: Accuracy comparison for classical (c) and quantum (q) PCA with major components over KDDCUP increasing ϵ and δ.

Principal components classifier with both major and minor components over KDDCUP

We use the same error parameters of the previous experiment:

- q-PCA30: $\epsilon_\theta = 8$, $p = 0.30$, $\delta = 0.1$, $\epsilon = 8$, and $\eta = 0.2$.
- q-PCA40: $\epsilon_\theta = 1$, $p = 0.40$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.16$.
- q-PCA50: $\epsilon_\theta = 1$, $p = 0.50$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.1$.
- q-PCA60: $\epsilon_\theta = 1$, $p = 0.60$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.1$.
- q-PCA70: $\epsilon_\theta = 1$, $p = 0.70$, $\delta = 0.1$, $\epsilon = 1$, and $\eta = 0.1$.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.9757	0.9777	0.9869	0.9862	0.9891	0.9881	0.9887	0.9884	0.9875	0.9868
2%	0.9753	0.9816	0.9807	0.9796	0.9829	0.9831	0.9830	0.9817	0.9835	0.9824
4%	0.9694	0.9685	0.9701	0.9678	0.9675	0.9708	0.9710	0.9692	0.9691	0.9688
6%	0.9558	0.9549	0.9591	0.9595	0.9585	0.9595	0.9607	0.9578	0.9578	0.9573
8%	0.9439	0.9447	0.9485	0.9519	0.9476	0.9492	0.9559	0.9523	0.9488	0.9485
10%	0.9313	0.9319	0.9370	0.9425	0.9369	0.9402	0.9480	0.9426	0.9385	0.9387

Table A.3: Accuracy comparison for classical (c) and quantum (q) principal components classifier with major and minor components over KDDCUP.

Principal components classifier with both major and minor components over CICIDS 2017

Here below the error parameters for quantum models:

- q-PCA30: $\epsilon_\theta = 1$, $p = 0.30$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.
- q-PCA40: $\epsilon_\theta = 1$, $p = 0.40$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.2$.
- q-PCA50: $\epsilon_\theta = 1$, $p = 0.50$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.
- q-PCA60: $\epsilon_\theta = 1$, $p = 0.60$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.
- q-PCA70: $\epsilon_\theta = 1$, $p = 0.70$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.

α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.7260	0.7248	0.6993	0.6999	0.6983	0.6943	0.7267	0.7260	0.7247	0.7036
2%	0.7602	0.7316	0.7242	0.7242	0.7249	0.7219	0.7637	0.7559	0.7425	0.8035
4%	0.7931	0.7895	0.7810	0.7978	0.7803	0.7788	0.7973	0.7975	0.8142	0.8156
6%	0.7997	0.7962	0.8009	0.8132	0.8009	0.7987	0.8051	0.8037	0.8057	0.8068
8%	0.7943	0.7903	0.8204	0.8294	0.7949	0.7934	0.7991	0.7980	0.7977	0.8017
10%	0.7906	0.7853	0.8354	0.8421	0.7911	0.7902	0.7952	0.7952	0.7933	0.7996

Table A.4: Accuracy comparison for classical (c) and quantum (q) principal components classifier with major and minor components over CICIDS 2017.

Ensemble principal components classifier over CICIDS2017

As always, we report the error parameters for quantum models:

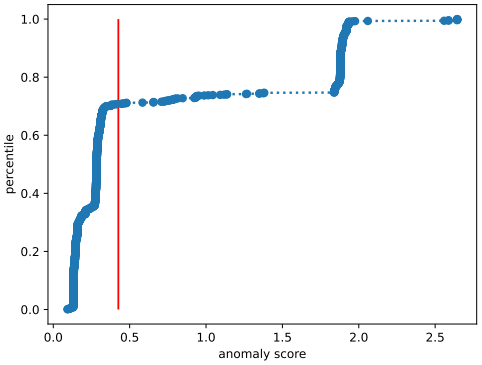
- q-PCA30: $\epsilon_\theta = 1$, $p = 0.30$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.

- q-PCA40: $\epsilon_\theta = 1$, $p = 0.40$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.2$.
- q-PCA50: $\epsilon_\theta = 1$, $p = 0.50$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.
- q-PCA60: $\epsilon_\theta = 1$, $p = 0.60$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.
- q-PCA70: $\epsilon_\theta = 1$, $p = 0.70$, $\delta = 0.1$, $\epsilon = 1$ and $\eta = 0.1$.

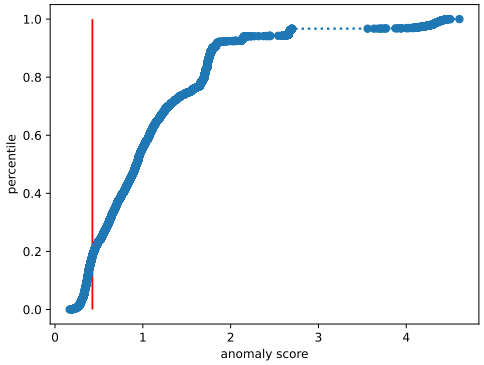
α	PCA30		PCA40		PCA50		PCA60		PCA70	
	c	q	c	q	c	q	c	q	c	q
1%	0.7606	0.7482	0.7751	0.8008	0.7004	0.6929	0.7604	0.7523	0.7339	0.7169
2%	0.8066	0.7783	0.8116	0.8325	0.7616	0.7493	0.8115	0.8043	0.7858	0.8597
4%	0.8836	0.8820	0.8857	0.8991	0.8525	0.8336	0.8863	0.8921	0.8883	0.9113
6%	0.9140	0.9124	0.8924	0.8930	0.9063	0.8969	0.9134	0.9169	0.9096	0.9216
8%	0.9025	0.9002	0.8897	0.8897	0.9003	0.8969	0.9006	0.9036	0.8966	0.9008
10%	0.8889	0.8834	0.8902	0.8879	0.8849	0.8850	0.8875	0.8893	0.8786	0.8816

Table A.5: Accuracy comparison for classical (c) and quantum (q) ensemble of principal components classifier over CICIDS 2017.

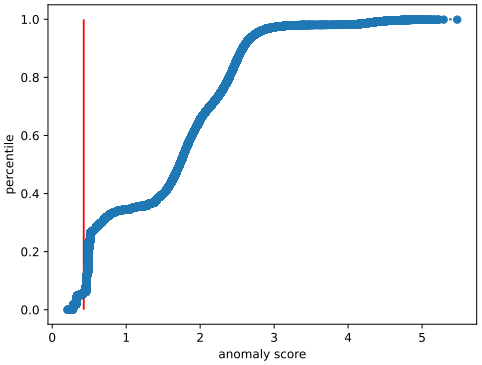
A.2. Anomaly scores distributions of CICIDS 2017 attacks



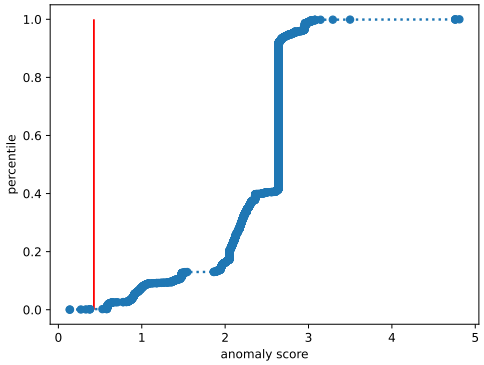
(a) Cumulative anomaly score distribution for Bot attack.



(b) Cumulative anomaly score distribution for Dos-GoldenEye attack.

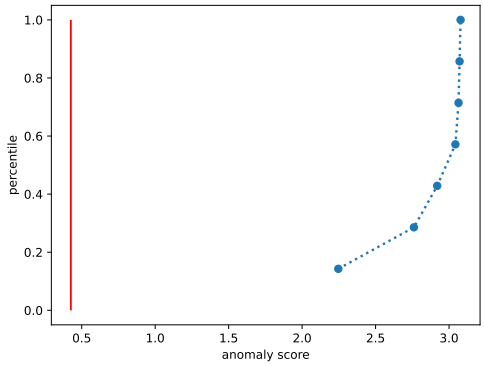


(c) Cumulative anomaly score distribution for Dos-hulk attack.

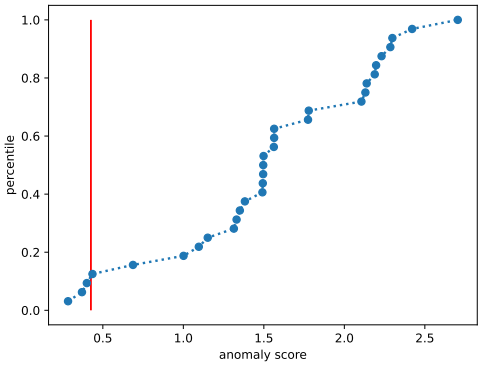


(d) Cumulative anomaly score distribution for Dos-slowhttptest attack.

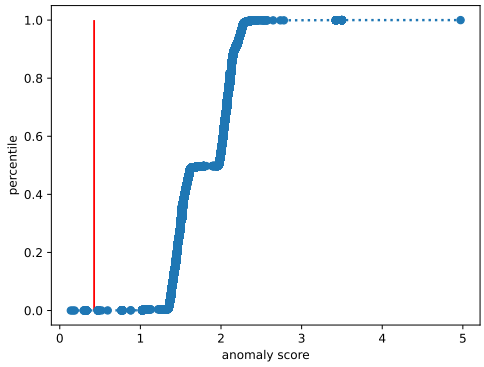
Figure A.1: Cumulative anomaly score distribution for different type of attacks. The red vertical line indicates the outlier threshold. The blue points at the right of the threshold are the one correctly classified as attack by the model.



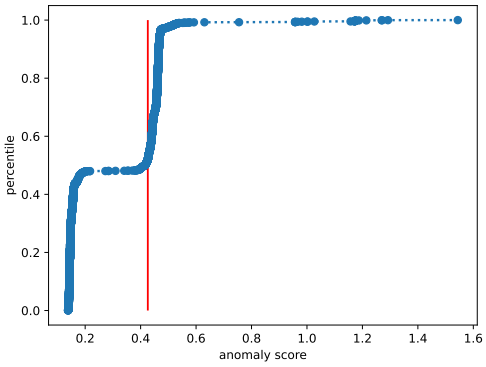
(a) Cumulative anomaly score distribution for Heartbleed attack.



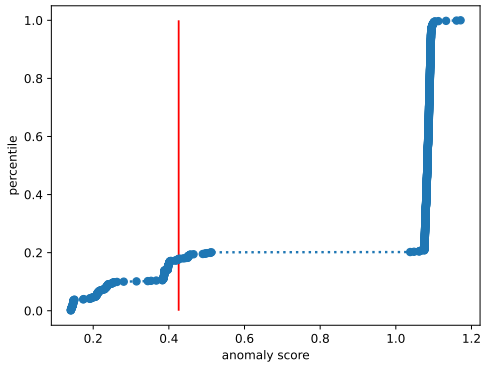
(b) Cumulative anomaly score distribution for Infiltration attack.



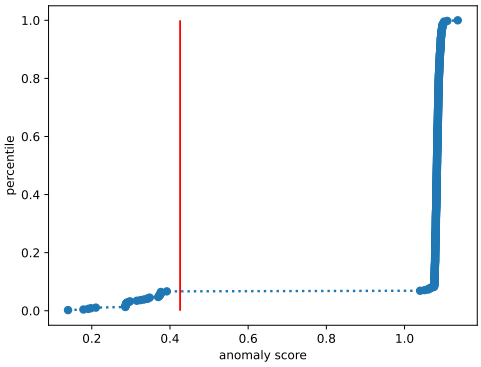
(c) Cumulative anomaly score distribution for PortScan attack.



(d) Cumulative anomaly score distribution for SSH-Patator attack.

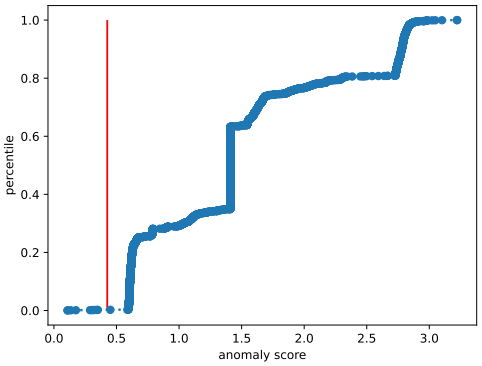


(e) Cumulative anomaly score distribution for BruteForce attack.

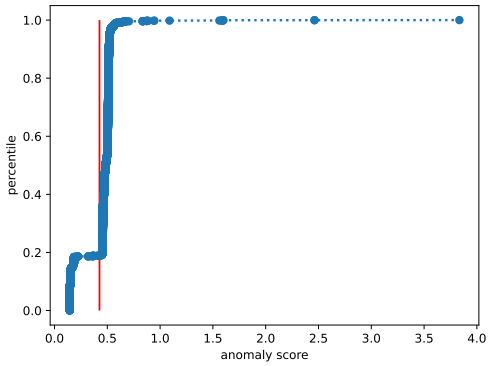


(f) Cumulative anomaly score distribution for XSS attack.

Figure A.2: Cumulative anomaly score distribution for different type of attacks. The red vertical line indicates the outlier threshold. The blue points at the right of the threshold are the one correctly classified as attack by the model.



(a) Cumulative anomaly score distribution for Dos-slowloris attack.



(b) Cumulative anomaly score distribution for FTP-Patator attack.

Figure A.3: Cumulative anomaly score distribution for different type of attacks. The red vertical line indicates the outlier threshold. The blue points at the right of the threshold are the one correctly classified as attack by the model.

List of Figures

List of Tables