Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

QUAMET: Structural and Semantic Metric Analysis for Quantum Code Smell Detection

Ruchika Malhotra
Delhi Technological University
Bhawna Jain
bhawnajain@igdtuw.ac.in

Indira Gandhi Delhi Technical University for Women
Marouane Kessentini
Grand Valley State University

Research Article

Keywords: Code Smell detection, Machine Learning, Quantum Computing, Quantum-Specific Code Smell

Posted Date: July 6th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9417538/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

QUAMET: Structural and Semantic Metric Analysis for Quantum Code Smell Detection

Ruchika Malhotra[1], Bhawna Jain[2*], Marouane Kessentini[3]

[1]Delhi Technological University, New Delhi, India.
[2*]Indira Gandhi Delhi Technical University for Women, New Delhi, India.
[3]Grand Valley State University, Michigan, USA.

*Corresponding author(s). E-mail(s): bhawnajain@igdtuw.ac.in;
Contributing authors: ruchikamalhotra@dtu.ac.in;
marouane@umich.edu;

Abstract

The field of Quantum Computing (QC) is advancing rapidly, introducing new programming paradigms and frameworks. This rapid growth also increases the vulnerability of quantum programs (QPs) to bugs and code smells. Despite considerable research, detecting code smells in QPs remains underexplored. To bridge this gap, this paper presents *QUAMET*, a novel structural and semantic analysis approach designed to establish software metrics for QPs and detect quantum code smells. Structural analysis examines code structure and program architecture, focusing on rules and patterns essential for identifying code smells. Semantic analysis, utilising quantum circuit analysis and parse trees, explores the meaning of the code. The study used a publicly available LintQ dataset, which contains real-world QPs implemented with Qiskit. After thorough preprocessing, the dataset was refined to extract structural and semantic metrics, resulting in a labelled dataset of 4,102 projects. This dataset is instrumental in identifying six distinct types of code smells, reviewed by industry and academic experts. The dataset and metrics are validated using reviews and classifications, demonstrating significant advancements in detecting quantum code smells. The results highlight the importance of combining structural and semantic analyses to enhance code smell detection accuracy, providing insights into their potential impact on QPs' performance and maintainability.

Keywords: Code Smell detection, Machine Learning, Quantum Computing, Quantum-Specific Code Smell

1 Introduction

With notable improvements in speed and efficiency over classical computing methods, quantum computing (QC) is an interdisciplinary field that applies the principles of quantum physics to complex real-world challenges in a variety of domains, such as machine learning (ML) [1], computational physics [2], and cryptography [3]. This technological approach capitalises on quantum phenomena, such as superposition and entanglement, yielding exponential accelerations in activities including factorisation, searching, and solving complex problems [4]. As a result, substantial investments from a global consortium of private enterprises, government research institutions, and educational establishments have propelled QC into the forefront of pioneering research [5]. This collective endeavour has concurrently paved the way for the development of innovative programming languages and frameworks explicitly tailored for QC [6]. The advent of dedicated quantum programming languages and frameworks, such as Google's Cirq, Microsoft's Q#, and IBM's Qiskit, has played a pivotal role in expediting the creation of increasingly intricate QPs [7]. Such frameworks provide an abstraction to ease the development of quantum algorithms, enabling researchers to focus on the business logic of their applications. The next evolution in this field, however, was Quantum-as-a-Service (QaaS) platforms, which have democratized quantum programming by allowing a wider community of researchers and developers to access universal quantum devices [8].

The continuous availability and upkeep of QPs are vital to ensure the smooth operation of quantum applications. However, the swift progression of QC technology has introduced formidable challenges in the maintenance of these programs, owing to various factors [9, 10]. Analogous to classical software development, where suboptimal design and implementation choices can impede effective maintenance [11, 12], the quantum programming field is undergoing rapid evolution, with libraries and frameworks undergoing frequent revisions [6, 13, 14]. Consequently, QPs often comprise a patchwork of code pieces to adapt to these changes, potentially introducing complications and issues as QPs grow in scale and intricacy [15].

Recognising these challenges, it is important for the software engineering research community to proactively explore techniques to support QP maintenance before widespread issues emerge [14, 16]. In the realm of software engineering, prior research has extensively examined maintenance issues, primarily focusing on indicators commonly known as "code smells", and methodologies to assess their impact on classical software [11, 12, 17–20]. Detection of code smells in the context of QC constitutes a burgeoning and relatively unexplored research domain. A noteworthy contribution in this field is the work by Chen et al. [21], wherein they introduced a specialised tool for detecting code smells in QPs. Furthermore, Openja et al. [22] have employed ML algorithms for a similar purpose. Nonetheless, only a few studies have focused on identifying code smells in QPs. Moreover, to detect code smells properly, more emphasis must be placed on code metrics.

Code metrics are quantitative measures for evaluating and assessing various elements of software code. These metrics convey information about the code's quality, complexity, and performance to make it easier for developers to understand its structure and behaviour. Lines of code, cyclomatic complexity, and code coverage are

typical code metrics used in classical programming [23, 24]. These traditional code metrics used for software quality should be modified to reflect the unique features of quantum algorithms and circuits. QC works on quantum bits, or qubits, that can be in superposition states and represent multiple states anywhere between 0 and 1. This dynamic nature differs from classical computation, necessitating a more thorough investigation and analysis of code metrics designed for quantum systems. Quantum code metrics are intended to analyse QP complexity, resource utilisation, and performance. These metrics help to understand the behaviour and behaviour of quantum circuits, which make it possible to analyse, optimise, and build quantum algorithms.

Keeping in view these research gaps, this study presents a novel analysis of structural and semantic features for code smell detection in QPs. The study aims to highlight the need for a large dataset of real-world QPs and their specific quantum metrics for classification. This method extracts nine structural and three semantic metrics, which serve as the foundation for detecting six different code smells in QC. Industry experts and university researchers meticulously examined the detected quantum code smells to validate their authenticity. Once identified, these code smells can be combined with the metrics data to create a labelled dataset for their respective code smells. This dataset is then input to the ML method and augmented with 14 ML classifiers to improve the accuracy and reliability of classification. The use of structural and semantic metrics is a breakthrough in QP research.

This study answers the following research questions (RQs):

RQ1: ***What are the researchers' views on quantum-specific code smells?***
This RQ examines how industry experts and academic experts perceive proposed quantum-specific code smells. It also examines the feedback and validity assessments provided by these reviewers.

RQ2: ***How does metrics analysis contribute to a more comprehensive analysis of code smells in quantum programming?***
This RQ investigates in what way are structural and semantic metrics helpful in identifying quantum code smells. It gives an overview of these metrics along with value, outliers, and correlation analysis.

RQ3: ***Does a model trained on a dataset with both structural and semantic features perform better than one trained on only structural or semantic metrics?***
This RQ examines the performance of models trained on datasets containing both structural and semantic features compared to those trained with only one type of feature. It assesses whether the combined methodology results in improved performance.

RQ4: ***How do various ML classifiers contribute to the classification of quantum-specific code smells?***
This RQ evaluates the performance of 14 different ML classifiers in detecting quantum code smells. It analyses how well each classifier identifies six specific quantum code smells.

RQ5: ***How do statistical analyses validate the dataset's efficacy for accurately detecting code smells?***
This RQ validates the accuracy of ML classifiers using the Wilcoxon signed-rank test.

It summarises the findings and establishes the reliability of the dataset for quantum code smell detection.

RQ6: ***How does the proposed methodology compare to existing state-of-the-art approaches in detecting quantum code smells?***
This RQ aims to validate this study's efficiency against existing state-of-the-art approaches on their programs and compare the results to check which method performs better.

Thus, this paper offers the following major contributions:

- First, this paper presents a metric-based labelled dataset comprising approximately 4,102 real-world quantum projects, providing a substantial resource for further research and analysis.
- Secondly, it conducts a comprehensive and detailed structural and semantic metric analysis of QPs to enhance the detection of quantum code smells, marking a pioneering effort in this domain.
- Lastly, the study identifies and enumerates six code smells predominantly found in QPs, contributing valuable insights into common issues in quantum software and advancing the field of quantum programming analysis.

The remainder of this paper is organised as follows: Section 2 provides a discussion on the background, emphasising the foundational knowledge of QC essential for this study. Section 3 reviews the relevant literature and situates this research within the broader context of existing studies. Section 4 outlines the methodology employed for metric analysis and the detection of quantum code smells, including data collection, processing techniques, quantum-specific code smells, and analytical procedures. Section 5 provides a real-world QP as an example to extract structural and semantic metrics, followed by the detection of quantum-specific code smells. Section 6 presents the results of applying the proposed approach to various open-source projects, demonstrating its effectiveness and practicality. Section 7 discusses potential threats to the validity of the findings and the steps taken to mitigate these risks. Finally, Section 8 concludes the paper by summarising the key contributions of the paper, discussing the implications of the findings, and suggesting directions for future research.

2 Background

QC is a field of computer science that leverages the principles of quantum mechanics to perform computations. Unlike classical computation, which is based on bits that can be either 0 or 1, quantum computation uses qubits as the basic unit of information. A qubit can exist in a superposition state $|\phi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$, where α_0 and α_1 are complex numbers representing the probabilities of the qubit being in states $|0\rangle$ and $|1\rangle$, respectively. This means a qubit can be in both states simultaneously, although this superposition cannot be directly observed. These qubits, along with classical bits, are the building blocks of quantum circuits.

Quantum computation is typically performed using quantum circuits, which generalise classical circuits. The primary advantage of representing QPs with quantum

Table 1: Overview of Common Quantum Gates

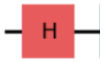
Gate Name	Gate Figure	Description
	Unary Gates	
H-Gate		Creates a superposition state from a basis state, represented by the matrix $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$.
Pauli-X Gate		Flips the state of a qubit, represented by the matrix $\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$.
Pauli-Y Gate		Applies a bit and phase flip, represented by the matrix $\begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}$.
Pauli-Z Gate		Applies a phase flip, leaving $\|0\rangle$ unchanged and $\|1\rangle$ multiplied by -1, represented by $\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$.
T-Gate		Applies a $\pi/4$ phase shift, represented by the matrix $\begin{pmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{pmatrix}$.
	Binary Gates	
CNOT Gate		Flips the target qubit if and only if the control qubit is in state $\|1\rangle$.
CZ Gate		Applies a Z gate to the target qubit if the control qubit is in state $\|1\rangle$, represented by the matrix $\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & -1\end{pmatrix}$.
	Other Gates	
Measure Gate		The measurement gate collapses a qubit's state to either $\|0\rangle$ or $\|1\rangle$ based on the probabilities determined by its quantum state.
Barrier		The barrier gate in QC is used to prevent optimisation across it, ensuring the preservation of the intended sequence of operations.

Table 2: Comprehensive Review of Literature in Quantum Code Smell Detection

Work	Authors	Description	Ref
Empirical Study	Openja et al.	Analyzed 118 open-source quantum projects, identified prevalent issues	[22]
QSmell	Chen et al.	Defined 8 quantum-specific code smells, developed detection tool	[21]
QChecker	Zhao et al.	Introduced static analysis tool for bug detection in Qiskit	[25]
QPAC	Nayak et al.	Developed automated framework for detecting bug-fix patterns in IBM Qiskit	[26]
Size and Structure Metrics	Jianjun Zhao	Proposed basic metrics for quantum software, validated on various projects	[27]
Metrics for Quantum Circuits	Cruz-Lemus et al.	Developed metrics for assessing quantum circuit understandability	[28]
This Study	Malhotra et al.	Structural and semantic metrics for quantum code smell detection	-

circuits is the abstraction it provides, eliminating the need to understand the programming language details used to implement them. A quantum system is modelled by a series of quantum gates in a specific order. They work similarly to classical logic gates in circuits—for these quantum gates, all processing is done by acting on qubits to change their state (i.e., making unitary transformations on the qubits). While basic operations can be done on an individual qubit, most manipulations/transformations will involve multiple qubits using multiqubit gates. These are the gates shaping up a quantum circuit fully. All quantum gates used in this study to comprehend quantum circuits are mentioned in Table 1. Real quantum machines or simulators can also be used to construct quantum circuits. The procedure requires programming the initial states, the specific set of quantum gates to be applied, and the measurement of these states relative to the classical bits used to store the results. This ensures that QPs may be uniformly represented and used, irrespective of the programming language in which they are written.

3 Related Work

With the rapid rise in the implementation of QPs, concerns about the emergence of bugs in these programs have grown. In response, several research initiatives in QC have been taken to focus on detecting bugs and code smells, aiming to improve the performance and reliability of quantum software. Table 2 provides an overview of the studies conducted in the field of QC for detecting bugs and code smells.

One of the significant studies in this domain was done by Openja et al. [22]. The study analysed 118 open-source quantum projects from GitHub, categorising them into 10 groups. It identified prevalent issues in quantum software, such as code convention violations, error handling, and suboptimal code design. Additionally, a

significant correlation was found between poor code practices and the occurrence of faults in quantum software. These findings underscore the importance of addressing these issues to improve the quality and reliability of QC systems.

Further, Chen et al. [21] investigated the prevalence of code smells in QPs as an indicator of maintenance issues, filling a research gap concerning maintenance in quantum software development. Eight quantum-specific smells were defined and validated through a survey involving 35 quantum developers. To facilitate the detection of these smells, a tool named *QSmell* was developed. An empirical investigation was conducted on 15 open-source QPs to analyse the occurrence of these quantum-specific code smells. The results revealed that 11 programs (73.33%) contained at least one smell, with an average of three smells per program. Additionally, the "Long Circuit" was identified as the most prevalent smell, appearing in 53.33% of the analysed programs.

Recently, Zhao et al. [25] introduced QChecker, a static analysis tool designed to detect bugs in QPs written in Qiskit. Unlike conventional static analysis tools tailored for classical software, QChecker addresses the unique challenges posed by QPs. The tool comprises two primary modules: one for extracting program information based on the abstract syntax tree (AST) and another for bug detection using specific patterns. QChecker was tested on the publicly available Bugs4Q benchmarking dataset to verify its potential in detecting different types of bugs in QPs and thus Improving the maintainability of quantum software.

Similarly, Nayak et al. [26] studied the current lack of automatic bug-fix pattern detection for quantum software, introducing a research agenda called Q-Repair. As part of this agenda, they developed an automated framework called Q-PAC, tailored for detecting bug-fix patterns in IBM Qiskit quantum code. This framework incorporated seven bug-fix pattern detectors utilising techniques such as abstract syntax trees, syntactic filters, and semantic checks. To validate the effectiveness of Q-PAC, the researchers executed it on various quantum bug-fix patterns, using both real-world and handcrafted examples. The results demonstrated that Q-PAC was successful in identifying bug-fix patterns in IBM Qiskit, highlighting its potential utility in improving the quality of quantum programming.

Recently, some studies have started exploring the metrics that can be calculated in QPs. In [27], researchers recognised the critical role of quantum software metrics in maximising the potential of QC systems, which had recently garnered significant attention. The study proposed basic metrics focusing on the size and structure of quantum software, defined at various abstraction levels to represent key attributes like the number of quantum gates and circuit complexity. These metrics were validated by applying them to various quantum software projects, effectively capturing essential attributes and providing a quantitative basis for comparison. The results highlighted the importance of standardised metrics for assessing and optimising quantum software.

In another study, Cruz-Lemus et al. [28] developed a set of metrics to assess the understandability of quantum circuits by adapting principles from classical software engineering. They detailed these metrics and demonstrated their calculation through examples, highlighting aspects like complexity, modularity, and readability. Initial

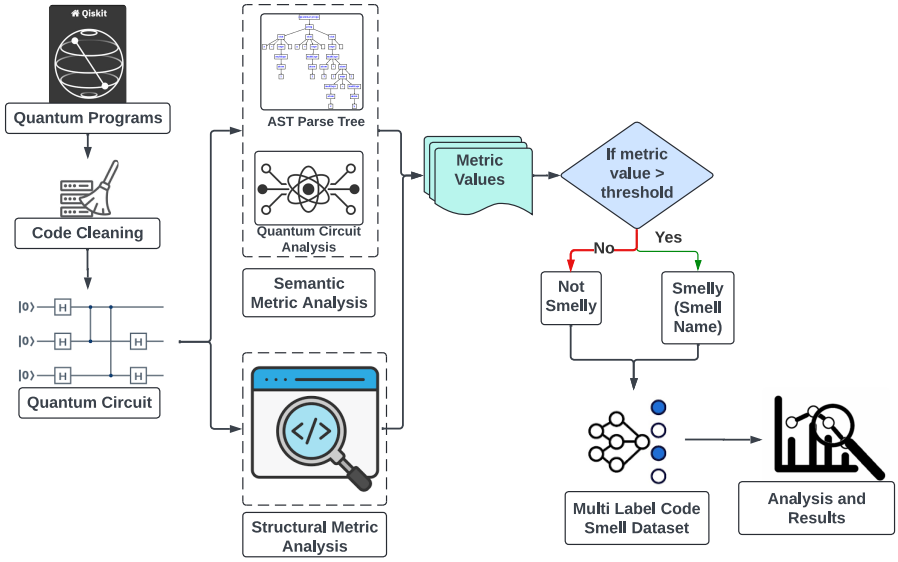


Fig. 1: An Overview of the Methodology Employed in this Study

results suggested that metrics such as circuit depth and gate count effectively captured key understandability factors. The authors concluded that these metrics showed promise but required further empirical validation through studies and experiments to ensure their reliability and effectiveness in practical applications.

These research papers showcase progress in detecting bugs in QC and introducing effective bug detection tools. However, there has been little focus on detecting code smells in QPs. Additionally, there's a lack of research on specific metrics for code smell detection in QC, particularly in semantic analysis. This paper aims to fill this gap by proposing clear and precise metrics for identifying quantum code smells.

4 Methodology

The methodology employed by QUAMET is illustrated in Figure 1, where a substantial dataset of QPs from LintQ, comprising snippets from 7,568 real-world projects, is utilised. Initially, this dataset undergoes pre-processing for code cleaning and extracting quantum circuits. Further, the processed dataset is used to extract structural and semantic metrics values from the quantum circuits present in each file within the dataset. These metrics aid in the identification of six specific code smells. Subsequently, the refined dataset is employed for classification using ML techniques to determine whether the code exhibits any of the identified code smells.

4.1 Dataset

The literature review concluded that the evaluation of QPs for bugs has been challenging due to the limited availability of a large dataset, as real-world QPs are scarce.

Table 3: Specific Quantum Functions and Formulas for Calculating Quantum Metrics

Code Metrics	Function
Structural Metrics	
Depth	qc.depth()
Number of Quantum Bits	qc.num_qubits
Number of Classical Bits	qc.num_clbits
Gate Count	$\sum_{i=1}^{N} \text{len}(qc.data[i])$
Number of Measurements	f_{measure} in qc.data
Total Gate Operations	qc.count_ops()
Circuit Width	qc.width()
Average Gate Depth per Qubit	$\frac{\text{Total Gate Depth}}{\text{Total Number of Qubits}}$
Entangling Gates	$\sum_{\text{entangling gates}}$ in qc.data
Semantic Metrics	
Code Comprehension	$\frac{\text{Total AST Nodes}}{\text{Total Lines of Code+1}}$
Circuit Connectivity	qc.num_connected_components()
Quantum Volume	$2^{\text{min(num_gates,depth)}}$

To address this issue of insufficient quantum datasets, Paltenghi et al. [29] developed a dataset named LintQ, which contains raw code snippets of QPs. This dataset successfully accommodated 7,568 real-world projects comprising code snippets containing quantum circuits. These snippets encompass various sections of projects and different commits within the same file. The dataset includes files in various formats such as Python files, Jupyter files, executable files, and text files. Additionally, the codes incorporate various quantum libraries, including Cirq, Qiskit, and TensorFlow Quantum etc. Therefore, the dataset required pre-processing and cleaning to standardise the format of each file for subsequent structural and semantic metric analysis.

For further pre-processing, only Python and Jupyter files were selected in thedataset[1] and Jupyter files were converted into python files, and the Qiskit library was utilised to execute each quantum circuit. Modifiable files were easily converted and stored accordingly. Subsequently, each file was examined for the presence of quantum circuits since the analysis could only be performed on the circuits and gate operations. This step helped eliminate many redundant files that only contained import statements or lacked any quantum circuits.

Finally, all the examined files underwent cleaning and were checked for importation errors. Given that the field of QC is continuously evolving with frequent updates and modifications to existing quantum libraries, the dataset needed to be updated to incorporate the latest libraries and functions for effective analysis. The whole pre-processing step reduced the dataset to 4102 project files. The pre-processed dataset is now ready for structural and semantic analysis.

[1]Dataset: https://zenodo.org/records/11392973

4.2 Structural Metric Analysis

Structural metric analysis refers to evaluating and measuring the physical and logical structure of quantum circuits and programs. These metrics calculate the count of various components that make up a quantum circuit. For this study, Table 3 describes the multiple metrics that have been selected to identify various types of quantum code smells, given that each metric serves a distinct purpose and carries a different weight. Quantum circuits have several components, including quantum bits, classical bits, gates, and measurement operations. In the structural analysis, the architecture of the circuits was examined to determine the quantity of each component present. These metrics are instrumental in understanding the structure of the circuit and will aid in detecting code smells within the QPs. Below is the list of each of the structural metrics that have been analysed in this study:

4.2.1 Depth

The circuit's depth is determined by determining the path's maximum length from the input to the output, which indicates the circuit's sequential depth and potential execution time.

4.2.2 Number of Quantum Bits

It represents the number of fundamental units of information a quantum computer can process. Quantum bits, commonly known as "qubits," are the fundamental units of quantum information in QC. Unlike classical bits, which can only exist in one of two states (0 or 1), qubits can exist in superposition states, representing multiple states simultaneously.

4.2.3 Number of Classical Bits

Classical bits, often simply referred to as "bits," are the basic units of classical information in classical computing. A classical bit can exist in one of two states: 0 or 1. It stores the final state of the qubits after the measurement operation is applied.

4.2.4 Number of Measurement Operations

The measurement operations metric refers to the count of measurement operations performed in a QC circuit or algorithm. Measurement operations collapse the quantum states of qubits, extracting classical information from the quantum system. Unlike quantum gates, which manipulate and evolve quantum states, measurement operations interact with the quantum system to obtain classical outcomes. Monitoring the number of measurement operations provides insights into the circuit's reliance on classical data extraction and the frequency of quantum state collapses during the computation. Its value is computed as the frequency of the measure operation in qc.data.

4.2.5 Total Gate Operations

Quantum gates are the fundamental blocks that manipulate the quantum states of qubits to perform specific quantum operations. Every gate represents a unitary transformation that progresses the quantum states of the qubits. Keeping track of the number of gates offers a valuable understanding of the complexity, resource demands, and computational depth of the quantum circuit. A greater count of gates could imply more circuit complexity, whereas a lesser count might indicate a simpler computation.

4.2.6 Gate Count

This metric quantifies the number of different quantum gates used as well as their frequency in the quantum circuit. It aids in comprehending the complexity of rotation operations at a per-qubit level. This is the sum of the total number of instructions in qc.data.

4.2.7 Circuit Width

The circuit width in QC is the sum of all bits that exist in a quantum circuit, quantum and classical bits. This measure gives a complete view of the computational power and resource usage of the circuit, counting all the quantum and classical parts.

4.2.8 Average Gate Depth per Qubit

The average gate depth per qubit metric assists in determining how often each qubit is used within a quantum circuit. This metric reveals the average number of quantum gates applied to each qubit within the circuit runtime and therefore the workload and activity level of individual qubits.

4.2.9 Entangling Gates

Entanglement is a unique quantum phenomenon whereby two or more qubits have correlated quantum states, such that the state of one cannot be described independently of the other (even when isolated from one another), showing the non-classical character of quantum mechanics. Entangling gates are necessary components in various quantum algorithms and protocols required to create complex quantum states as well as to provide quantum parallelism and quantum algorithm speed-up. They are critical for implementing quantum logic operations, quantum error correction, and quantum communication protocols, hence being core parts of quantum circuit design and optimisation. Its value is the sum of all entangling gates in qc.data.

4.3 Semantic Metric Analysis

Semantic metric analysis involves evaluating the meaning or semantics of quantum algorithms and circuits rather than just their structural properties. Unlike structural metrics, which focus on the physical and logical structure of quantum circuits (e.g., gate count, circuit depth, etc.), semantic metrics delve deeper into QPs' functional aspects, computational behaviour, and algorithmic characteristics.

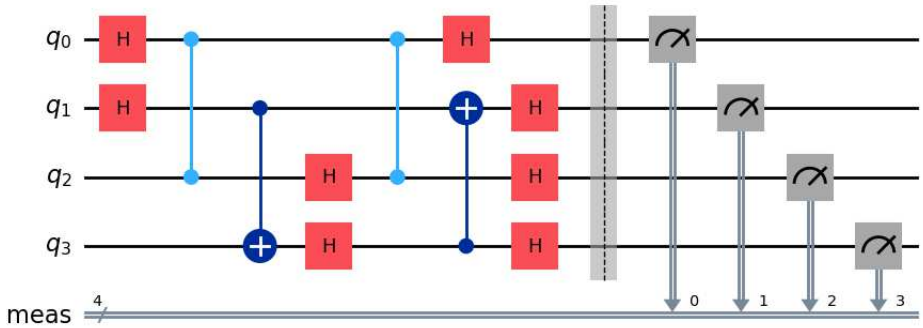


Fig. 2: An Example depicting Long Circuit Code Smell

Unlike previous studies [27, 28], which focused solely on evaluating the name and count of gates, this study employed quantum circuit analysis (QCA) and parse tree techniques to extract semantic metrics from the codes. An Abstract Syntax Tree (AST) was generated to locate each method for feature extraction. The AST aids in identifying similarities, checking for code complexities, and loops, ensuring uninterrupted code execution, and detecting any unused methods. This provides insights into the code's functionality and flow. Furthermore, an analysis of quantum circuits was carried out to gain a deeper understanding of the characteristics of each circuit. This analysis was followed by deriving comprehension of metric codes from the parsed data to identify the methods employed within the code. Through the examination of parameters such as entanglement gates, code connectivity, and quantum circuit volume, a comprehensive analysis was facilitated. Table 3 outlines the semantic metrics derived from this study, and their details are described below:

4.3.1 Code Comprehension

Code comprehension is the ability to understand and interpret the software code structure, functioning, and purpose. This understanding is measured with the code comprehension metric, which is the ratio of the total number of AST nodes to the total number of lines of code in a given code file. Qualitative parameters include code comp lexity and readability, which help test the quality of the code, aiding developers in making high-quality code reviews, speeding up in development process, and providing project visibility.

4.3.2 Circuit Connectivity

This metric characterizes the number of different non-overlapping connected components in a quantum circuit. A connected component is a set of connected qubits and gates that can be scheduled to operate together in isolation.

4.3.3 Quantum Volume

Quantum Volume is the first performance metric for quantum computers to include both the number of qubits and connectivity, in addition to gate and measurement error rates. Together, this computation measures how fast and how many errors are expected in practice from a quantum device, providing an estimation of its performance, reliability, and scalability.

4.4 Quantum Specific Code Smells

This section provides an overview of the six quantum-specific code smells identified in this research. The authors defined these code smells, and reviewers were asked to evaluate using online survey forms, considering how well the selected projects possess these smells. The thresholds for each code smell were based on the dataset evaluation and were examined and rated similarly.

4.4.1 Long Circuit (LC)

Quantum circuits that are excessively complicated because of their high depth (number of gates applied to a single qubit) or high width (gates applied simultaneously to many qubits) are referred to as *"Long Circuit"* code smells. These circuits have a high computational resource requirement, which results in longer execution times and higher error rates. When a quantum circuit becomes too lengthy, the backend simulator's computation power may become strained, resulting in degraded performance and reduced accuracy. This degradation is particularly evident when qubits become unstable, amplifying the impact of errors.

Metric Requirements : The "width-to-depth ratio" serves as a valuable metric for gauging the complexity of a quantum circuit. This metric is computed by dividing the circuit's width, which refers to the number of qubits involved, by its depth, representing the total gate operations applied as shown in Equation 1. For instance, consider the circuit illustrated in Figure 2. This circuit has a width of 4 qubits and a depth of 12 gates. Consequently, the calculated ratio $\frac{4}{12}$ equates to approximately 0.33. A ratio below 1, like this one, indicates a circuit of considerable length. Such extensive circuits can impose substantial computational demands on quantum simulators. This increased computational load may lead to heightened error rates, potentially compromising the accuracy of the results obtained from the simulation.

$$\text{Width-to-Depth Ratio} = \frac{\text{Circuit Width}}{\text{Circuit Depth}} \tag{1}$$

4.4.2 Excessive Measurement (EM)

The "Excessive Measurement" code smell pertains to circuits with an imbalanced usage of measurement operations relative to the total number of bits, encompassing both quantum and classical. Measurements are pivotal in QC as they collapse the superposition of a qubit's state into a classical outcome, facilitating the extraction of classical information and subsequent classical computation. However, excessive

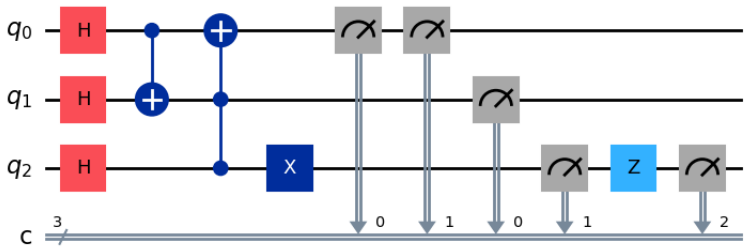


Fig. 3: An Example depicting Excessive Measurement Code Smell

measurement operations on a single qubit introduce challenges and risks. Each measurement collapses the quantum state, diminishing the system's capacity to maintain intricate quantum states vital for quantum algorithms. Moreover, multiple measurements on a qubit escalate the likelihood of error accumulation, particularly problematic in quantum systems where errors can propagate and amplify due to the fragile nature of quantum states. This over-reliance on measurements can also lead to resource inefficiencies, consuming more computational resources and slowing down the quantum computation.

Metric Requirements : EM code smell is quantified as shown in Equation 2.

$$\text{Excessive Measurements Ratio} = \frac{\text{num_qbits} + \text{num_clbits}}{2 * \text{num_measure}} \tag{2}$$

The provided quantum circuit, as shown in Figure 3 is designed with 3 qubits and 3 classical bits. Upon analysis of its structure, it contains a total of 5 measurement operations distributed across the qubits. Using the formula in equation 2, the ratio is calculated as:

$$\text{Excessive Measurements Ratio} = \frac{3+3}{2 \times 5} = \frac{6}{10} = 0.6$$

The ratio is less than 1, which indicates that half of the total operations in the circuit are measurement operations. This ratio suggests that a significant portion of the circuit is devoted to measurements, which could lead to decreased quantum coherence and computational inefficiencies.

4.4.3 Unconnected Circuit (UC)

The "Unconnected Circuit" code smell refers to circuits characterized by low connectivity and a lack of entangling gates, which suggests that there is inadequate interaction between qubits and an absence of quantum entanglement. This restricts how qubits can interact and hence reduces the circuit's ability to exploit quantum parallelism and entanglement (there are no entangling gates in a poorly connected circuit). However, this low connection and entanglement can lead to low efficiency and computing power of the circuit.

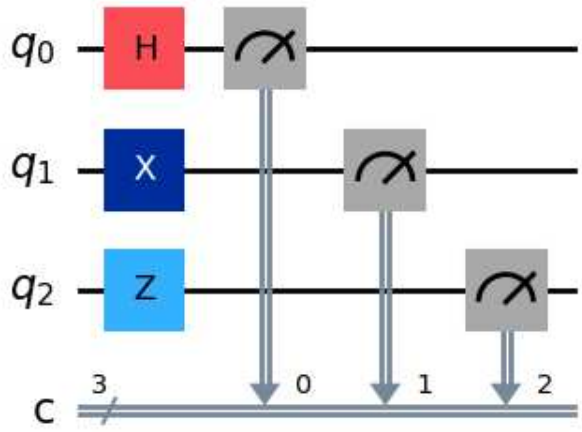


Fig. 4: An Example depicting Unconnected Circuit code smell

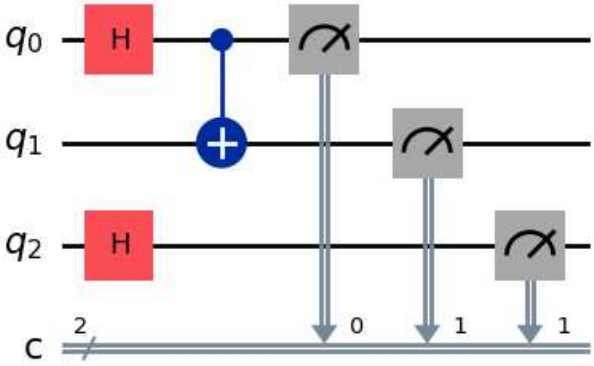


Fig. 5: An Example depicting Inconsistent Bits code smell

Metric Requirements: UC code smell is detected by a combination of two threshold values, both indicating a reduced number of qubit connections and a lack of quantum entanglement, as shown in Equations 3 and 4. The circuit, due to this, is less efficient and has lower computational performance. For instance, in Figure 4, none of the gates are entangling gates, and therefore the qubits are never in a position to connect and

change their state and fidelity. Therefore, this circuit is an example of low circuit connectivity, indicating UC smell.

$$Circuit\ Connectivity \leq 1 \tag{3}$$

$$Entanglement\ (Entangling\ Gates) = 0 \tag{4}$$

4.4.4 Inconsistent Bits (IB)

"Inconsistent Bits" code smell is related to the potential discrepancy in the number of qubits and classical bits in a quantum circuit. Qubits and classical bits each have distinct but interconnected roles in a quantum circuit. Maintaining a balanced ratio between them is essential for the circuit's effective operation. If the number of qubits greatly exceeds the number of classical bits, it may signal inefficiencies in the quantum algorithm, potentially causing underutilization of resources or computational delays. Conversely, if classical bits outnumber qubits, it could indicate that the classical processing and control mechanisms are unnecessarily complex, complicating the overall design and management of the circuit.

Metric Requirements : IB code smell is identified through a straightforward comparison of the number of quantum bits (*num_ qubits*) to the number of classical bits (*num_ clbits*) within the circuit as shown in Equation 5.

$$num_\ qubits \neq num_\ clbits \tag{5}$$

This comparison offers insights into the equilibrium or disparity between these two-bit types. As illustrated in Figure 5, the circuit contains 3 quantum bits and only 2 classical bits. This configuration implies that two quantum bits are mapped to a single classical bit for measurement. Consequently, this setup risks overwriting the prior measurement values, potentially leading to errors in the computation.

4.4.5 Low Quantum Volume (LQV)

"Low quantum volume" code smell acts as an indicator of circuits and code bases that show a decrease in Quantum Volume and a decline in code comprehension. Quantum Volume assesses a quantum system's capability to run intricate algorithms, taking into account factors such as the number of qubits, gate error rates, and circuit depth.

A lower value than the ideal low Quantum Volume indicates lower computational power and poor performance of quantum gates. Conversely, a low Code Comprehension value suggests complex, unsystematic, or badly documented code, which could hinder progress from both a collaboration and a future development point of view. Together, these factors obstruct the efficient application and comprehension of quantum circuits.

Metric Requirements : The LQV serves as a quantitative measure directly reflecting the overall performance capability of the quantum device, with its value computed using Equations 6 and 7.

$$\text{Quantum Volume} < \text{Threshold} \tag{6}$$

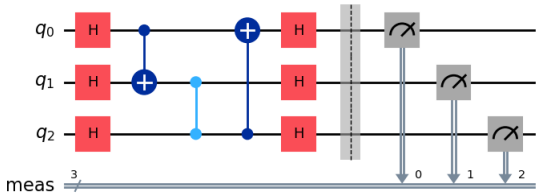


Fig. 6: An Example depicting Low Quantum Volume code smell

$$\text{Code Comprehension} < 1 \tag{7}$$

As an example, for the given circuit in Figure 6:

$$\begin{aligned} \text{num_gates} &= 3 \quad \text{(2 CNOT gates and 1 CZ gate)} \\ \text{depth} &= 4 \quad \text{(number of layers of the circuit)} \end{aligned}$$

Substituting these values into the formula:

$$\text{Quantum Volume} = 2^{\min(3,4)}$$
$$\text{Quantum Volume} = 2^3 = 8$$

The corresponding Quantum Volume for the circuit represented in Figure 6 is 8. The given quantum circuit code is devoid of loops, conditionals, or any other complicated structures that make the code long or deep. When evaluated, the ratio of nodes to the number of lines of code in the abstract syntax tree indicates a lower comprehension value. This value indicates that the circuit has a limited capacity for quantum operations. With a quantum volume threshold of 10 as an indication of lacking inefficiency, this circuit has a value less than the threshold defined. Therefore, this circuit suffers from LQV code smell and can be optimised for effective quantum resource utilisation.

4.4.6 Insufficient Depth per Qubit (IDQ)

This code smell refers to a circuit where the average gate depth per qubit is low in the circuit. The most essential metric used to estimate the complexity and depth of a quantum circuit by calculating the total number of gates is called Gate Depth. Suppose the gate depth per qubit is below the threshold. In that case, it shows that the quantum operation in that circuit is not using the quantum gates effectively and can lead to underutilization of quantum components. Several consequences arise when you decrease the depth of a quantum circuit. This constrains how much compute power the circuit has available to it, and hence ultimately limits the complexity of quantum states and algorithms that can be trained on it, leading to poor performance. This also increases error rates, however, which makes the circuit vulnerable to quantum errors and noise, reducing the accuracy and data integrity. Often, there aren't enough qubits or nondeterministic operations per second so the circuit can properly benefit

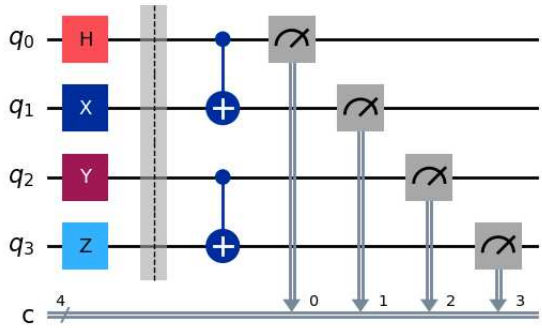


Fig. 7: Insufficient Depth per Qubit

from quantum parallelism, which results in a longer run time and reduces the effective performance.

Metric Requirements: The IDQ code smell is quantified using Equation 8.

$$\text{Gate Depth per Qubit} = \frac{\text{Total Gate Depth}}{\text{Total Number of Qubits}} \tag{8}$$

In the example shown in Figure 7, the total gate depth is 3 layers and the number of qubits is 4, leading to an average gate depth of:

$$\text{Average Gate Depth} = \frac{3}{4} = 0.75$$

A value less than 1 for this metric indicates insufficient depth per qubit, highlighting potential inefficiencies and areas for improvement. This IDQ metric suggests that the circuit might not be utilising the full computational potential of each qubit, potentially leading to under-optimised quantum operations.

4.5 Machine Learning Classifiers

Once the metric values were validated, they assisted in structuring the dataset into smelly and non-smelly categories. Table 4 describes the multi-label dataset constructed, with each file labelled to indicate the presence of quantum-specific code smells. During this process, some degree of class imbalance was observed across different smell categories, which could potentially influence the learning process. To address this, appropriate strategies were considered to ensure that the models did not become biased toward the majority classes, thereby maintaining fairness and generalisation in smell detection. This labeled dataset was used to train 14 ML classifiers listed in Table 5, including Decision Trees, Naive Bayes and Support Vector Machines variants, Logistic Regression, Multilayer Perceptron, Stochastic Gradient Descent, Random Forest, K-Nearest Neighbor, and Linear Discriminant Analysis.

The models were trained, and their performance was evaluated using accuracy, precision, recall, and F1-score. Accuracy measured the overall correctness, precision

Table 4: Description of Instances for Each Code Smell in Quantum Dataset

Code smell	Independent Variables	No. of Instances		
		Smelly	Non Smelly	Total
LC	Circuit Width, Depth	583	3519	4102
IB	q_bits, cl_bits	1871	2231	4102
UC	Entanglement, circuit connectivity	1292	2810	4102
EM	measurement count, q_bits, cl_bits	434	3668	4102
LQV	Quantum volume	3242	860	4102
IDQ	Average gate depth per qubit	1944	2158	4102

Table 5: Base ML Classifiers Selected in this Study

Base Classifier	Acronym	Ref.
Decision Tree (CART Algo)	DT	[30]
Linear SVM	LinSVM	[31]
Polynomial SVM	PolySVM	[32]
Sigmoid SVM	SigSVM	[33]
Radial Basis Function SVM	RBFSVM	[34]
Bernoulli Naïve Bayes	BNB	[35]
Gaussian Naïve Bayes	GNB	[36]
Multinomial Naïve Bayes	MNB	[37]
Logistic Regression	LR	[38]
Multilayer Perceptron	MLP	[39]
Stochastic Gradient Descent	SGD	[40]
Random Forest	RF	[41]
K-Nearest Neighbor	KNN	[42]
Linear Discriminant Analysis	LDA	[43]

indicated the proportion of true positives among predicted positives, recall assessed the model's ability to identify relevant instances, and F1-score provided a balance between precision and recall. This study identifies the models that perform well for the detection of code smells in quantum circuits, which is essential to constructing tools to assist in preserving and enhancing the quality of quantum software.

4.6 Statistical Tests

In this study, the Wilcoxon signed-rank test is used to compare the performance differences of 14 selected classifiers to perform the prediction on a curated quantum code smell dataset with six types of smell. This non-parametric test is suitable because the data does not follow a normal distribution, as shown by the Shapiro–Wilk [44, 45] test.

It is most useful as a tool for comparing samples of paired data or at least those generated by the same measurement procedure, which allows accuracies to be assigned to measured classifier performance, e.g., as a cross-validated area under the ROC curve

(AUC). According to Demsar [46], the magnitudes of the differences (not only ranks) offer more detailed comparisons, and the Wilcoxon signed-rank test is recommended for comparing several classifiers among multiple datasets since it is robust and makes fewer assumptions regarding the distribution of data.

The Wilcoxon Signed-Rank Test requires two hypotheses to be formulated: the Null Hypothesis (H_0) and the Alternative Hypothesis (H_1). The null hypothesis is that the two ML models being compared have equivalent accuracy distributions. In terms of a symbolic representation, this can be written as:

$$H_0: \quad \text{There is no difference in the accuracy distributions of the two classifiers.}$$

The alternative hypothesis (H_1) is that there is a significant difference in the accuracy of the two conditions. It can be written as:

$$H_1: \quad \text{There is a significant difference in the accuracy distribution of the two classifiers.}$$

From there, the test calculates a test statistic based on the sum of the ranks of the differences between observations paired with each other. This statistic is then compared to a critical value or evaluated to obtain a p-value. If the p-value obtained from the test is less than a predetermined significance level ($\alpha = 0.05$), the null hypothesis is rejected. This rejection indicates that there is a statistically significant difference in accuracy between the classifiers.

Mathematically, the test statistic for the Wilcoxon signed-rank test can be represented as:

$$W = \sum_{i=1}^{N} R_i$$

Where W is the test statistic, N is the number of paired observations, and R_i is the rank assigned to the i^{th} absolute difference between paired observations.

Conducting hypothesis tests with multiple comparisons over the same dataset increases the likelihood of Type I error (i.e., rejecting the null hypothesis when it should be accepted). To mitigate this, the Bonferroni correction procedure was employed. This correction aims to reduce Type I error by adjusting the p-value using the formula:

$$\text{Adjusted } p\text{-value} = \frac{\alpha}{T}$$

In this context, α denotes the initial p-value, while T represents the total number of comparisons conducted.

5 Running Example: Bernstein-Vazirani Circuit

To grasp these metrics, consider a quantum circuit featuring *Bernstein-Vazirani*, as depicted in Figure 8. The figure shows the Python code and the generated circuit from

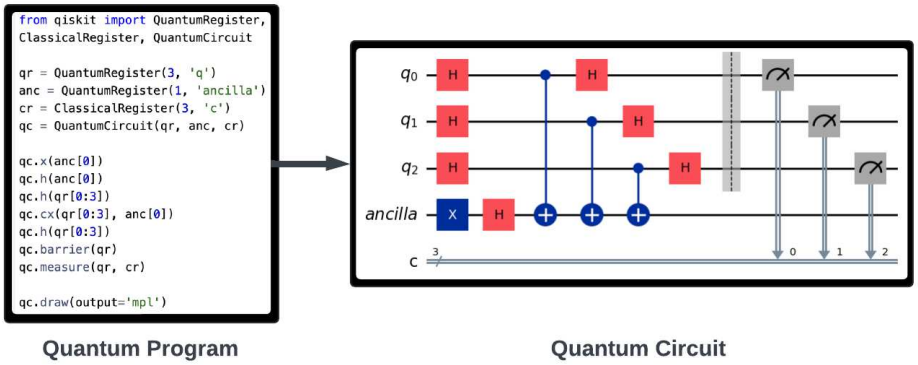


Fig. 8: Example Bernstein Vazirani Quantum Circuit to show metric values

this code. The Bernstein-Vazirani algorithm is a quantum algorithm that efficiently solves the problem of determining a hidden bit string encoded within a black-box function. Specifically, given a function

$$f(x) = a \cdot x$$

(where $\cdot$ denotes the bitwise dot product), the algorithm aims to find the hidden bit string a. Utilising quantum parallelism, the algorithm prepares a superposition of all possible inputs, applies the oracle function, which encodes the hidden string into the phase of the quantum state, and then uses a second set of Hadamard gates to transform this phase information into measurable output. The final measurement of the qubits reveals the hidden string a in a single query, showcasing a significant improvement over the classical approach, which requires multiple queries. This makes the Bernstein-Vazirani algorithm a powerful example of how QC can solve specific problems more efficiently than classical computing. The original Bernstein-Vazirani circuit from the Qiskit documentation is used. This circuit undergoes analysis to derive both structural and semantic metric values. The extracted values are detailed in Table 6 specifying the values of structural and semantic metrics for the given example circuit.

The circuit is evaluated against the conditions for each code smell listed in Table 7. The table outlines the metric conditions necessary for identifying the presence of a code smell. Alongside these conditions, the table presents the corresponding values for the Bernstein-Vazirani circuit. These values indicate whether the specific code smell is present in the example circuit or not. Based on the conditions evaluated, the Bernstein-Vazirani circuit contains the **IB** code smell.

6 Evaluation

This section will address and evaluate the study based on the defined research questions.

Table 6: Calculation of Quantum Metrics for Bernstein-Vazirani Circuit

Metric	Value
Depth	7
Number of Quantum Bits	4
Number of Classical Bits	3
Number of Measurements	3
Total Gate Operations	12
Gate Counts	{'h': 7, 'cx': 3, 'x': 1}
Circuit Width	7
Average Gate Depth per Qubit	1.75
Entangling Gates	3
Circuit Connectivity	1
Quantum Volume	128
Code Comprehension	7.526

Table 7: Metric Conditions and Computed Values for Code Smell Detection in Bernstein Vazirani Circuit

Code Smell	Condition	Values
Long Circuit (LC)	$\frac{\text{circuit_width}}{\text{depth}} < 1$	1
Excessive Measurements (EM)	$\frac{\text{num_quantum_bits+num_classical_bits}}{2\times\text{num_measurements}} < 1$	≈ 1.167
Insufficient Depth per Qubit (IDQ)	$\text{avg_gate_depth_per_qubit} < 1$	1.75
Inconsistent Bits (IB)	$\text{num_quantum_bits} \neq \text{num_classical_bits}$	$4 \neq 3$
Unconnected Circuit (UC)	Circuit Connectivity ≤ 1 and Entangling Gates $= 0$	False
Low Quantum Volume (LQV)	Quantum Volume < 10 and $\text{code_comprehension} < 1$	128

6.1 Setup Requirements

All experiments are conducted on a workstation with a 2.6 GHz Intel Core i7 13th gen CPU and 16 GB of main memory running 64-bit Windows 11. The Python version is 3.9, and the TensorFlow version is 2.16.1.

6.2 RQ1: What are the researchers' views on quantum-specific code smells?

To validate the defined code smells, an online survey was distributed to 400 reviewers. Out of which 173 filled responses were received. This group comprised industry specialists and academic authors. The industry specialists were further divided into two categories based on their experience: those with less than 10 years of experience and those with more than 10 years of experience. The academic authors were classified into three categories: assistant professors, associate professors, and professors.

Table 8: Code Smells Validation Questionnaire

Q. No.	Category	Question	Scale	Description
1	**Validity**	Do you find this code smell relevant and applicable to real-world scenarios?	1: Highly Relevant 5: Not Relevant	Evaluates whether the code smell is commonly encountered and applicable to real-world projects.
2		Is this code smell clearly defined and distinct from other code quality issues?	1: Very Clear 5: Not Clear	Assesses the clarity of the code smell definition and whether it's easy to distinguish from other issues.
3		Can you verify that this code smell exists based on your experience in software development or maintenance?	1: Strongly Agree 5: Strongly Disagree	Checks if the reviewer has personally experienced or observed the code smell in practice.
4		Do you agree that this code smell represents an actual flaw or inefficiency in the codebase?	1: Strongly Agree 5: Strongly Disagree	Confirms if the reviewer agrees that the code smell negatively impacts code quality or efficiency.
5	**Effectiveness**	How effective do you think detecting and addressing this code smell will be in improving code quality?	1: Very Effective 5: Not Effective	Measures the perceived impact of detecting and fixing the code smell on improving the overall quality of the code.
6		Do you believe that removing this code smell will have a noticeable impact on performance/maintainability?	1: Very Impactful 5: Not Impactful	Evaluates how much the removal of the code smell will affect software performance or long-term maintainability.
7		Are the suggested mitigation strategies for this code smell practical and feasible to implement?	1: Very Feasible 5: Not Feasible	Assesses the practicality of the proposed solutions for addressing the code smell.
8		Do you think current tools and techniques are effective in identifying and resolving this code smell?	1: Highly Effective 5: Not Effective	Checks if existing tools or techniques can reliably detect and help address the code smell.
9	**Severity**	How would you rank the severity of this code smell compared to other common code smells?	High / Low	Evaluates how serious this code smell is compared to others in terms of impact on the project.
10		Based on your experience, how frequently do you encounter this code smell, and how impactful is it?	High / Low	Assesses how frequent and impactful the code smell is in practice based on the reviewer's experience.

Table 9: Individual Reviewer Ratings for Validity and Effectiveness of Code Smells

Reviewer	**Aspect**	**Code Smell**					
		LC	**UC**	**IB**	**EM**	**LQV**	**IDQ**
Category 1	Validity	2.0	3.7	4.5	4.0	4.2	5.0
(Developers>=10 yrs experience)	Effectiveness	2.6	3.8	4.7	3.2	4.8	4.5
	Severity	High	Low	High	High	Low	High
Category 2	Validity	2.5	4.0	4.3	3.0	3.9	4.6
(Developers<10 yrs experience)	Effectiveness	2.8	4.1	4.6	3.1	3.7	4.7
	Severity	High	High	High	High	High	High
Category 3	Validity	3.0	3.5	4.6	3.5	4.0	4.4
(Professor)	Effectiveness	2.4	4.3	4.5	3.4	4.0	4.6
	Severity	High	High	Low	High	Low	High
Category 4	Validity	2.8	4.2	4.8	2.8	3.6	4.8
(Associate Professor)	Effectiveness	2.9	4.0	4.8	2.9	3.5	4.8
	Severity	High	Low	High	High	Low	High
Category 5	Validity	3.5	3.6	4.1	2.0	3.9	4.6
(Assistant Professor)	Effectiveness	2.7	4.2	3.7	3.8	4.2	4.5
	Severity	High	Low	Low	Low	Low	Low
	Average	2.70	3.75	4.25	3.1	3.90	4.50
	Median	2.8	4.0	4.5	3.1	3.9	4.6

Table 10: Survey results per smell

	LC	**UC**	**LQV**	**EM**	**IDQ**	**IB**	**Average**
Agree	78.00%	73.00%	70.00%	66.00%	64.00%	88.00%	73.17%
Disagree	12.00%	17.00%	20.00%	27.00%	21.00%	7.00%	17.33%
Don't know	10.00%	10.00%	10.00%	7.00%	15.00%	5.00%	9.50%

Each reviewer was given a detailed briefing about the quantum code smells developed in this paper, along with the dataset. They were asked various questions related to the validity and effectiveness of the code smells and to rank these smells based on their severity. Table 8 lists and describes each question posed by the reviewers, along with its corresponding scale range. The average ratings from reviewers in each category are detailed in Table 9. The reviewers were asked to rank the code smells based on validity and effectiveness using a 5-point scale, where a lower rank indicates a higher severity of the code smell (i.e., 1 denotes the most severe). According to the table, LC is identified as the most severe code smell, with an average rank of 2.7. This is followed by EM, which is regarded as a highly disturbing code smell, with a rank of 3.1. Additionally, they were asked if they agreed that such code smells are present in QPs. The results of their responses are shown in Table 10. On average, 73.17% of the reviewers agreed that the proposed code smells affect the functioning of QPs.

Most of the reviewers with experience in quantum programming reviewed and tested the presence of these code smells in a real-world dataset. They provided comments on both the presence and impact of these code smells. Some reviewers, particularly industry specialists, acknowledged the presence of these smells but felt they are an inherent part of programming. Additionally, many reviewers highlighted their potential to hinder code maintainability and readability, noting that while these smells are challenging, addressing them could lead to significant improvements in code quality.

One interpretation is that participants perceived the LC smell as particularly harmful due to the significant overhead introduced by long circuits in quantum processing. As one participant remarked, "A long circuit leads to higher error rates and increased decoherence, making it unsustainable for complex operations." In contrast, regarding the EM smell, one participant observed, "While excessive measurements are sometimes necessary, overusing them adds to execution time and affects the reliability of the results—better to limit them when possible." As for the IB smell, some participants mentioned, "Branching isn't inherently problematic, but unnecessary branching in quantum circuits can lead to inefficiency, especially if better alternatives are available." Interestingly, 40% of participants found the LQV smell harder to identify, possibly due to the difficulty in recognising poorly defined variables within quantum circuits. Meanwhile, for UC and IDQ, the consensus was clear: "Both unclear qubit assignments and improperly defined quantum gates contribute to circuit errors, but they're secondary concerns compared to long circuits."

While addressing these code smells presents challenges, especially with long circuits, doing so could significantly enhance the performance and maintainability of quantum programs.

Table 11: Descriptive Statistics for Quantum Metrics

Quantum Metrics	mean	std	min	25%	50%	75%	max
depth	3.852	6.749	0.0	1.0	2.0	4.0	110.0
num_quantum_bits	7.748	113.833	1.0	2.0	3.0	6.0	4200.0
num_classical_bits	6.204	113.842	0.0	0.0	2.0	4.0	4200.0
num_measurements	1.677	8.759	0.0	0.0	0.0	2.0	367.0
num_gates	5.995	17.961	0.0	1.0	2.0	8.0	736.0
circuit_width	13.952	227.641	1.0	3.0	4.0	10.0	8400.0
avg_gate_depth_per_qubit	1.157	1.596	0.0	0.253	1.0	1.5	47.0
code_comprehension	2.145	3.158	0.0	0.0	1.0	2.672	62.0
Entanglement (Entangling Gates)	4.651	131.382	0.0	0.0	1.0	3.0	8400.0
Quantum Volume	3.170	2.030	0.048	0.979	2.0	8.0	1.300

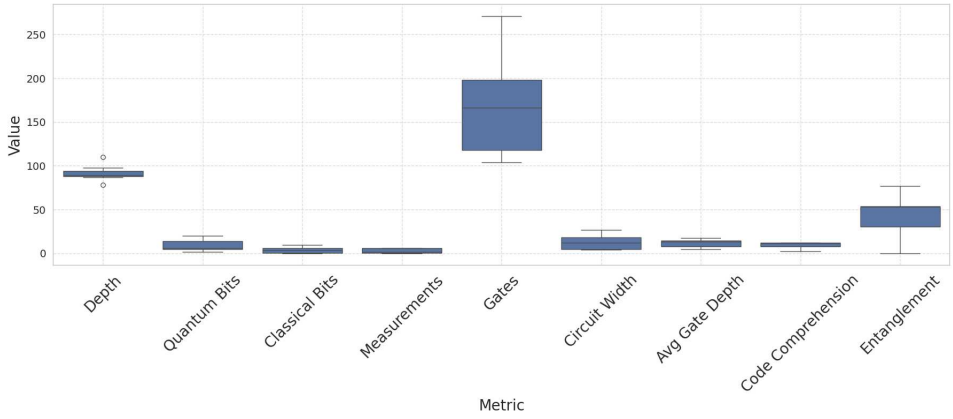


Fig. 9: Outliers in Quantum Metrics

Summary of RQ1: The study conducted a comprehensive review of code smells in QPs, involving 173 industry specialists and academic authors. Reviewers briefed on the identified code smells, ranked them based on severity, and assessed their validity and effectiveness. On average, 73.17% of reviewers agreed that these code smells impact QP functionality. Notably, LC was identified as the most severe, followed by EM. While some acknowledged the presence of these code smells, others viewed them as intrinsic to programming.

Table 12: Correlation Analysis Results of Metrics

	depth	q_bits	cl_bits	measurements	gates	circuit_width	avg_gate_depth	code_comprehension	Entanglement	Quantum Volume
depth	1.0									
q_bits	-0.0055	1.0								
cl_bits	-0.0083	0.9994	1.0							
measurements	0.0753	0.0680	0.0700	1.0						
gates	0.6595	0.0372	0.0351	0.5452	1.0					
circuit_width	-0.0069	0.9999	0.9999	0.0690	0.0362	1.0				
avg_gate_depth	0.6607	-0.0281	-0.0243	-0.0150	0.3202	-0.0262	1.0			
code_comprehension	0.5736	-0.0038	-0.0032	0.0845	0.4086	-0.0035	0.2303	1.0		
Entanglement	0.0095	0.5767	0.5766	0.0270	0.0106	0.5767	-0.0036	-0.0079	1.0	
Quantum Volume	0.2457	0.0013	0.0005	-0.0012	0.2305	0.0009	0.0520	0.0396	-0.0006	1.0

6.3 RQ2: How does metrics analysis contribute to a more comprehensive analysis of code smells in quantum programming?

This study incorporated the use of both structural and semantic metrics. When proposing new metrics, it is essential to provide detailed information about them. Table 11 describes the descriptive statistics of all the metrics used in the study. This table presents various statistical measures for each metric, including the mean, median, standard deviation, minimum, and maximum values. By providing these statistical ranges and measures, the table helps to understand the distribution, central tendency, and variability of the metrics, thereby giving a comprehensive overview of their characteristics and behavior within the dataset.

Another important aspect of these metrics is the presence of outliers in the dataset, as shown in Figure 9. This figure shows whether the metrics have any values outside their usual range. For example, the 'depth' metric has 2 outliers, whereas the 'gates' metric falls between 100 and 250. This visualization helps understand the distribution and differences in the metrics dataset.

Additionally, the correlation analysis provides more intricate information on the inter-relationship of different metrics as shown in Table 12. The correlation coefficients allow researchers to understand when one metric changes and how the other metrics would change. If two metrics are strongly positively correlated, then they tend to grow together in proportion. For QP, when the 'Total Gate Operations' is highly positively correlated to the depth of quantum circuits, it indicates the case that as the depth gets larger, using more gates. This helps in understanding the quantum circuits and components.

A negative correlation, however, means that as one metric grows, the other is likely to decrease or stay constant. For example, an inverse correlation between 'depth' and 'q_bits' would show that increasing the depth might lower or sustain the number of qubits consumed. These insights can power a streamlining of quantum programming, thereby realising more effective lines of code for developers.

Additionally, great effort in predicting weak or null relationships between metrics allows for changes in one metric to be made without driving the other to a significant extent. This insight is necessary to better inform where optimisations should be done and in what parts of QP have the most impact on performance.

Table 13: Performance of Different Feature Sets for Code Smell Detection

Features	LC				UC				LQV			
	Acc	F1	Recall	Precision	Acc	F1	Recall	Precision	Acc	F1	Recall	Precision
With semantic features	75.00%	73.10%	74.00%	76.00%	70.00%	85.62%	71.00%	72.00%	98.00%	97.00%	97.00%	96.00%
With structural features	80.00%	79.00%	81.00%	79.00%	75.00%	84.69%	76.00%	83.07%	98.00%	97.00%	97.87%	97.00%
Both	**99.88%**	**99.88%**	**99.63%**	**96.00%**	**99.63%**	**96.95%**	**97.19%**	**97.28%**	**99.88%**	**98.00%**	**99.00%**	**97.00%**
Features	**EM**				**IDQ**				**IB**			
	Acc	**F1**	**Recall**	**Precision**	**Acc**	**F1**	**Recall**	**Precision**	**Acc**	**F1**	**Recall**	**Precision**
With semantic features	85.62%	68.00%	71.00%	72.00%	75.90%	76.54%	66.00%	64.00%	75.00%	73.00%	76.00%	74.00%
With structural features	85.62%	90.50%	76.00%	74.00%	70.00%	90.50%	71.00%	69.00%	80.00%	78.00%	81.00%	79.00%
Both	**99.63%**	**97.00%**	**99.00%**	**98.79%**	**98.27%**	**96.95%**	**97.28%**	**97.28%**	**99.63%**	**98.05%**	**97.19%**	**98.79%**

Summary of RQ2: From this analysis, it became evident that there were little or no outliers in the dataset, implying that overall, the data was consistent. The statistical description provided detailed insights into the range of values for each metric, enhancing the understanding of their distribution and variability. The correlation analysis revealed both positive and negative relationships between metrics, demonstrating intricate patterns within the dataset. Positive correlations represented co-dependencies between structural elements, and negative correlations indicated potential strategies to optimise.

Table 14: Performance of Various Machine Learning Models on Different Code Smells

Model	LC				UC				IB			
	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1
LR	0.855	0.7311	0.855	0.7882	0.5274	0.7508	0.5274	0.3642	0.5274	0.7508	0.5274	0.3642
MLP	0.9829	0.9827	0.9829	0.9827	0.9695	0.9696	0.9695	0.9695	0.9695	0.9696	0.9695	0.9695
SGD	0.1449	0.876	0.1449	0.0366	0.4726	0.7508	0.4726	0.3033	0.4726	0.7508	0.4726	0.3033
RF	1.0000	1.0000	1.0000	1.0000	0.9866	0.9868	0.9866	0.9866	0.9903	0.9903	0.9903	0.9903
KNN	0.9866	0.9867	0.9866	0.9866	0.9415	0.9423	0.9415	0.9414	0.9415	0.9423	0.9415	0.9414
LDA	0.8855	0.8759	0.8855	0.8789	0.8307	0.8472	0.8307	0.8298	0.8307	0.8472	0.8307	0.8298
DT	0.9805	0.9813	0.9823	0.9292	0.9476	0.9778	0.9098	0.9426	0.9409	0.9661	0.9048	0.9344
LinSVM	0.7941	0.3214	0.3781	0.3474	0.6322	0.6229	0.5619	0.5908	0.6322	0.6229	0.5619	0.5908
PolySVM	0.8562	1.0000	0.0084	0.0167	0.5286	1.0000	0.0026	0.0051	0.5286	1.0000	0.0026	0.0051
SigSVM	0.8562	1.0000	0.0084	0.0167	0.5274	1.0000	0.0000	0.0000	0.5274	1.0000	0.0000	0.0000
RBFSVM	0.8562	1.0000	0.0084	0.0167	0.5274	0.0000	0.0000	0.0000	0.5274	0.0000	0.0000	0.0000
GNB	0.8562	0.8769	0.8562	0.7911	0.5286	0.7511	0.5286	0.3669	0.5286	0.7511	0.5286	0.3669
MNB	0.8611	0.8805	0.8611	0.8025	0.5274	0.2782	0.5274	0.3642	0.5274	0.2782	0.5274	0.3642
BNB	0.9719	0.9728	0.9719	0.9707	0.9635	0.9635	0.9635	0.9635	0.9635	0.9635	0.9635	0.9635
Model	**LQV**				**IDQ**				**EM**			
	Acc	**Prec**	**Rec**	**F1**	**Acc**	**Prec**	**Rec**	**F1**	**Acc**	**Prec**	**Rec**	**F1**
LR	0.5274	0.7508	0.5274	0.3642	0.5262	0.7507	0.5262	0.3628	0.9050	0.9140	0.9050	0.8599
MLP	0.9695	0.9696	0.9695	0.9695	0.9963	0.9963	0.9963	0.9963	0.9781	0.9791	0.9781	0.9784
SGD	0.4726	0.7508	0.4726	0.3033	0.5262	0.7507	0.5262	0.3628	0.0950	0.9140	0.0950	0.0165
RF	0.9903	0.9903	0.9903	0.9903	0.9988	0.9988	0.9988	0.9988	0.9915	0.9916	0.9915	0.9913
KNN	0.9415	0.9423	0.9415	0.9414	0.9878	0.9879	0.9878	0.9878	0.9440	0.9428	0.9440	0.9433
LDA	0.8307	0.8472	0.8307	0.8298	0.9622	0.9641	0.9622	0.9623	0.9281	0.9339	0.9281	0.9306
DT	0.9409	0.9661	0.9048	0.9344	1.0000	1.0000	1.0000	1.0000	0.9439	1.0000	0.4685	0.6381
LinSVM	0.6322	0.6229	0.5619	0.5908	0.6565	0.6482	0.6015	0.6240	0.8380	0.0339	0.0256	0.0292
PolySVM	0.5286	1.0000	0.0026	0.0051	0.5262	0.0000	0.0000	0.0000	0.9062	1.0000	0.0128	0.0253
SigSVM	0.5274	1.0000	0.0000	0.0000	0.5262	1.0000	0.0000	0.0000	0.9050	1.0000	0.0000	0.0000
RBFSVM	0.5274	0.0000	0.0000	0.0000	0.5262	0.0000	0.0000	0.0000	0.9050	0.0000	0.0000	0.0000
GNB	0.5286	0.7511	0.5286	0.3669	0.4750	0.7510	0.4750	0.3073	0.9062	0.9150	0.9062	0.8628
MNB	0.5274	0.2782	0.5274	0.3642	0.4799	0.7521	0.4799	0.3180	0.9050	0.8190	0.9050	0.8599
BNB	0.9635	0.9635	0.9635	0.9635	0.9805	0.9805	0.9805	0.9805	0.9488	0.9516	0.9488	0.9401

6.4 RQ3: Does a model trained on a dataset with both structural and semantic features perform better than one trained on only structural or semantic metrics?

This study investigated the impact of utilising different feature sets on model performance for quantum code smell detection. Specifically, three feature sets were evaluated: one with only semantic features, one with only structural features, and one combining both types of features. The semantic features capture the meaning and context of the code, while structural features represent the code's architecture and organisation.

Table 13 presents the experimental results of the performance of different feature sets. For LC code smell, using both features results in a 20% increase in precision and a 26.78% increase in F1-score compared to using only semantic features. Similarly, for the structural features, the dataset shows a 17% and 20.88% reduction in precision and F1-score, respectively, compared to the combined dataset.

For UC code smell detection, precision and F1-score drop by 25.28% and 11.33%, respectively, when using only semantic features, and by 14.21% and 12.26%, respectively, when using only structural features, compared to the combined feature set.

In the case of LQV code smell detection, using both semantic and structural features results in increased F1-score and recall by 1.00% and 2.00%, respectively, compared to using only one type of feature, where using only structural features, the dataset showed 1% and 1.13% reduction in the metrics.

Similar trends are observed for EM, IDQ, and IB code smells. For example, for EM code smell detection, using only semantic features results in a 29.00% reduction in F1-score and a 26.79% reduction in precision, while the structural features-only dataset shows a decrease of 6.50% and 24.79%, respectively.

These findings indicate that relying solely on either semantic or structural features significantly compromises model performance for quantum code smell detection. The datasets combining both feature types consistently yield higher accuracy across all metrics, highlighting the importance of integrating semantic and structural information for effective code smell detection.

Summary of RQ3: This analysis concluded that leveraging both structural and semantic metric analyses for code smell detection yields superior results compared to using either type of metric alone. Among the different code smells, the LC code smell showed the most significant performance improvement when using both structural and semantic features, with an increase of approximately 24.88% in accuracy, 26.78% in F1-score, 25.63% in recall, and 20% in precision compared to using only semantic features. On the other hand, the EM code smell exhibited the least improvement, with a comparatively lower increase in performance metrics, ranging from 8.13% to 29.00%, when utilising both feature types compared to using only semantic features.

6.5 RQ4: How do various ML classifiers contribute to the classification of code smells?

This subsection investigates the detection performance of the 14 individual classifiers in detecting quantum code smells. Table 14 presents the result of these ML classifiers based on four evaluation metrics: accuracy, precision, recall, and F1 score.

The table compares the performance of various ML models across different code smells: LC, UC, IB, LQV, IDQ, and EM. Overall, the RF classifier consistently outperforms other models with near-perfect scores of all metrics across almost all code smells. MLP and KNN also show strong performance, particularly for LC and EM, with high metrics close to those of RF. Conversely, models like SGD and various SVM kernels (PolySVM, SigSVM, RBFSVM) perform poorly on several code smells, often with low recall and F1 scores, indicating they are not well-suited for these tasks. LDA and DT classifiers perform relatively well but are less consistent compared to RF, MLP, and KNN. LR struggles with UC and IB but performs adequately on LC and EM. Models like GNB and BNB have variable performance, with BNB being notably better. SVM models with different kernels (Poly, Sigmoid, RBF) often achieve high precision but suffer from extremely low recall, making them unreliable for detecting code smells. In summary, the RF classifier is the best performer for detecting code smells, while models like MLP and KNN are also highly effective. Conversely, SVM models and SGD show significant limitations, particularly in recall and F1 scores.

In addition to the performance metrics, Figure 10 presents boxplots of the accuracy of each ML classifier across different cycles of testing. The figure illustrates that, apart from SGD, many classifiers have performed well, as indicated by their higher accuracy boxes. Specifically, EM and LC display the highest whiskers in the boxplots, suggesting they achieved the highest accuracy levels, while other code smells exhibit more variation in performance. The results indicate that the RF classifier consistently outperforms other models across various code smells, achieving near-perfect scores in accuracy, precision, recall, and F1. MLP and KNN also demonstrate strong performance, particularly for LC and EM code smells. This highlights the capability of machine learning models to learn complex, nonlinear patterns from the metric-based features and generalise well across multiple smell types. In particular, RF's ensemble structure makes it well-suited for capturing threshold-based relationships, while MLP leverages its deep learning capacity to model intricate decision boundaries. Conversely, models like SGD and various SVM kernels (Poly, Sigmoid, RBF) perform poorly, especially in recall and F1 scores. LR and DT classifiers perform adequately but less consistently. Naive Bayes classifiers (GNB and BNB) have variable performance. Overall, these results emphasise the utility of machine learning techniques in automating code smell detection and reinforce RF and MLP as the most reliable and effective classifiers in this context.

For each feature selection technique, machine learning (ML) classifiers are ranked based on the area under the ROC curve (AUC value). Figure 11 highlights the best ML classifiers when the eight classifiers that provide ROC results are compared. The figure illustrates the ranking from best to worst for each code smell, allowing a clear comparison of classifier performance. Across all code smells analysed, the Random Forest (RF) classifier consistently emerges as the best performer, achieving an AUC of

1, which underscores itsustness and reliability in detecting quantum code smells compared to other classifiers. The high AUC values achieved by RF suggest it effectively captures the underlying patterns and relationships in the data, crucial for accurate detection.

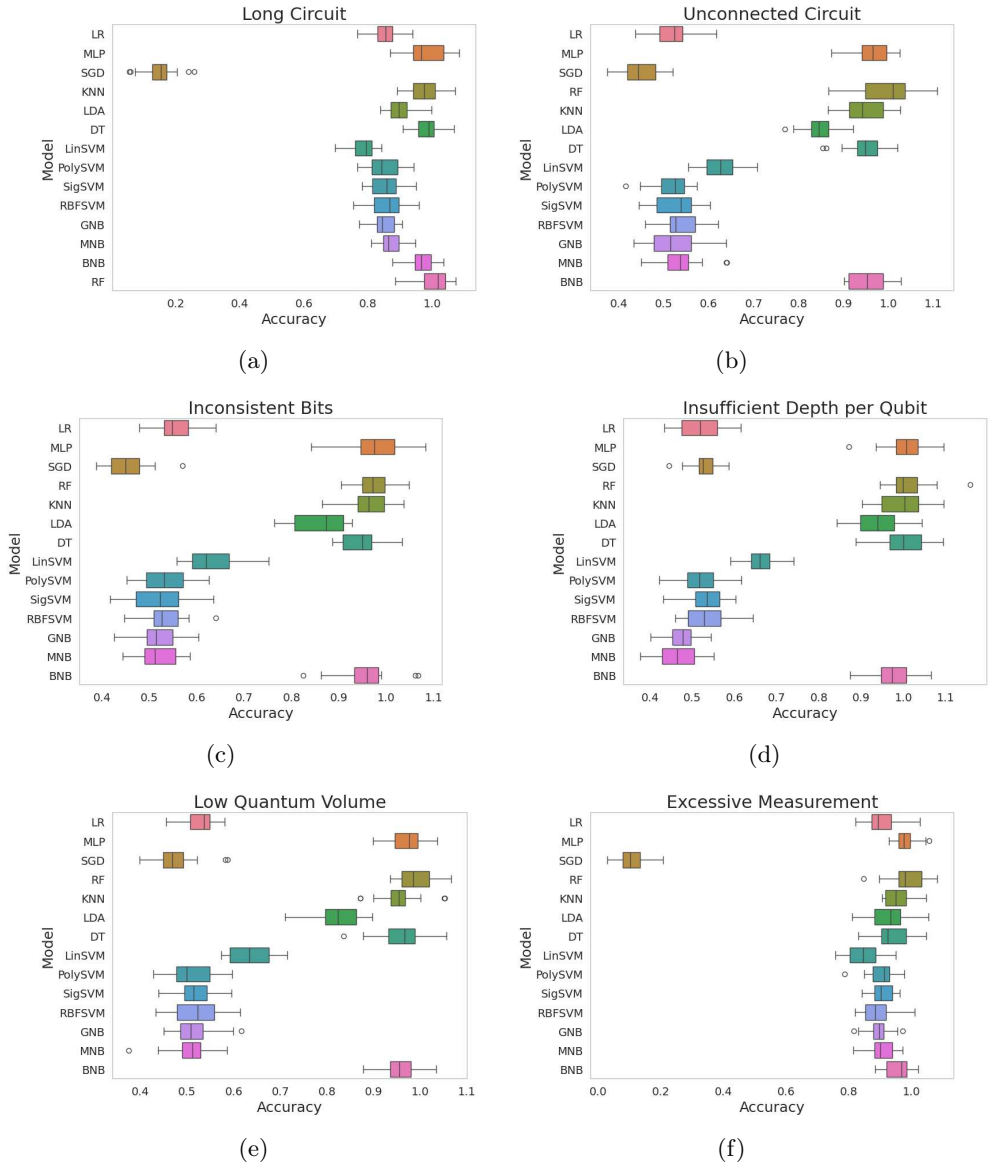


Fig. 10: Box plots of the performance (accuracy) exhibited by the considered machine learning models for all the six smells.

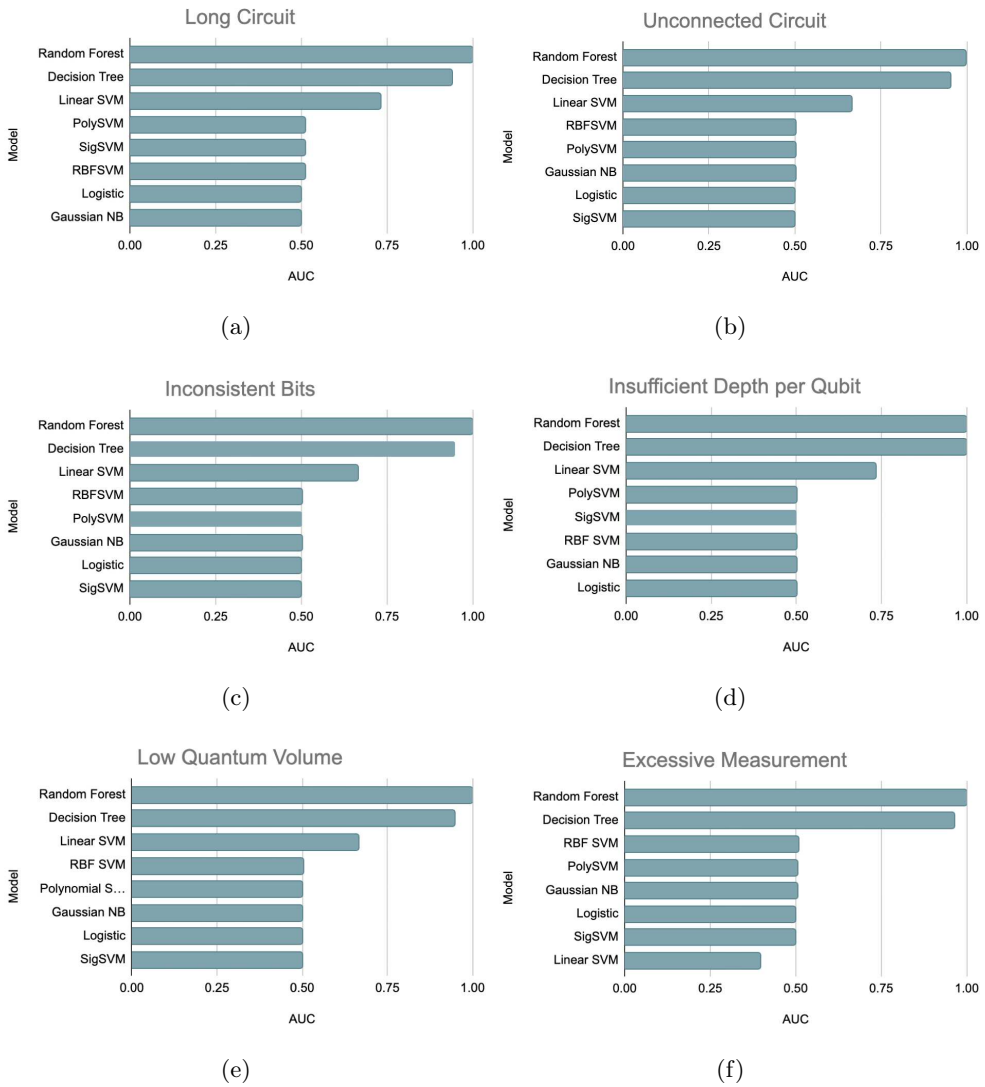


Fig. 11: ROC AUC of Classifiers from best to worst for different code smells detection

Summary of RQ4: The results indicate that the RF classifier consistently outperforms other models across various code smells, achieving near-perfect scores in accuracy, precision, recall, and F1. MLP and KNN also demonstrate strong performance, particularly for LC and EM code smells. Conversely, models like SGD and various SVM kernels (Poly, Sigmoid, RBF) perform poorly, especially in recall and F1 scores. LR and DT classifiers perform adequately but less consistently. Naive Bayes classifiers (GNB and BNB) have variable performance. RF and MLP are consistently the most reliable and effective classifiers.

Table 15: Classifiers statistical performance results Across Different Smell Types

Classifier	LC		UC		IB		LQV		IDQ		EM		Total	
	Wins	Losses	Wins	Losses	Wins	Losses	Wins	Losses	Wins	Losses	Wins	Losses	Wins	Losses
RF	11		13		13		12		13		13		75	0
MLP	11	2	13		13		13		13		13		76	2
SGD	13		13		13		9		6	7	9	4	63	11
BNB	13			13	5	8	8		13		1	12	40	33
KNN	11			13		13	13			13		10	24	49
DT	13		5	8	5	8	5	6	5	8		13	33	43
MNB		13		13		13		13		13		13	0	78
LR	2	11	9	4	4			13	6	7		12	21	47
RBFSVM	1	12	9		13		2	11	6	7	1	12	32	42
SigSVM		13		9		13		13	6	7		13	6	68
LinSVM	9		13		13		13		13		2	11	63	11
LDA		13	13		2	11	13		10	1	3	10	41	35
PolySVM	1	12		13		13		13			13		14	51
GNB	1	6	10		10	2		13		7	1	12	22	40

Table 16: Classifiers paired comparisons results

Classifier	RF	MLP	SGD	BNB	KNN	DT	MNB	LR	RBFSVM	SigSVM	LinSVM	LDA	PolySVM	GNB	Win %
RF		2	5	5	8	4	5	8	6	5	4	10	10	3	96.15
MLP				3	6	4	22	5	3	17	2	6	8		97.44
SGD					5	3	20	2	4	19		5	5		80.77
BNB					7	2	2	7	8	5	1		8		51.28
KNN				2		5	2	7	2		1	2	3		30.77
DT				2	2		3	7	5			3	1	10	42.31
MNB															0.00
LR			2	5	1	3	3		2	1	1	2	1		26.92
RBFSVM				2		8	3	2		5	1	3	1	7	41.03
SigSVM			2	1	1			1				1			7.69
LinSVM				4	9	7	8	2	5	9			10	9	80.77
LDA					5	4	7	5	2	4	1		2	11	52.56
PolySVM			2	2	3	2	1	1		2		1			17.95
GNB				7	2	1	2		5	1		2	2		28.21
Loss %	0.00	2.56	14.10	42.31	62.82	55.13	99	60.26	53.85	87.18	14.10	44.87	65.38	51.28	

6.6 RQ5: How does statistical analysis validate the dataset's efficacy for accurately detecting code smells?

This study utilised the Wilcoxon statistical test to analyse whether there is a significant difference in the performance between the two classifiers. The null hypothesis for the Wilcoxon test defines that there is no difference in performance between the

two classifiers. If the null hypothesis is rejected, it indicates that the performance of classifier A is significantly higher than that of classifier B. In such cases, classifier A is referred to as the "winner" and classifier B as the "loser." Conversely, if the null hypothesis is accepted (i.e., the difference in performance between A and B is insignificant), neither classifier is designated as a winner or a loser.

The total number of wins and losses for each classifier across various code smell datasets is displayed in Table 15. The statistical test results align well with the ML classifiers' outcomes. Although MLP was the most successful classifier, with 76 wins, RF stood out as the least losing classifier, with 0 losses. In contrast, MNB was the poorest performer, with 0 wins and 78 losses. This table effectively highlights which models perform better for each code smell.

Expanding on the results in this table, the wins and losses for each pairwise classifier comparison are shown in Table 16. For a given row classifier, the row indicates the total number of wins against the column classifier. Similarly, for a given column classifier, the column shows the total number of losses against the row classifier. The last row and column calculate the percentage of total wins and losses, respectively, for each classifier out of all possible paired comparisons. Each classifier has a total of 78 paired comparisons (6 smells multiplied by 13 classifiers). Note that the cumulative sum of wins and losses for a classifier may not equal 78, as the acceptance of the null hypothesis indicates no significant difference in performance, leading to neither classifier being reported as a winner or loser. Notably, MLP has the highest win percentage at 97.44%, while RF has the lowest loss percentage at 0%.

Summary of RQ5: The Wilcoxon statistical test results reveal that the MLP classifier emerged as the top performer with 76 wins, achieving the highest win percentage of 97.44%. Conversely, the RF classifier demonstrated remarkable robustness with the lowest loss percentage at 0%, making it the least-losing classifier. When compared to other classifiers, the MNB classifier performed the worst, with 0 wins and 78 losses, demonstrating its general inefficiency. These results demonstrate how well MLP and RF perform on a variety of code smells, and they also show that MNB is by far the least successful classifier.

6.7 RQ6: How does the proposed methodology compare to existing state -of-the-art approaches in detecting quantum code smells?

In recent years, various studies have proposed different methodologies for detecting quantum code smells, each focusing on its perspective and selecting a unique set of code smells. This has led to the proliferation of numerous quantum code smells and associated metrics, some of which are repetitive or overlapping. Moreover, certain code smells identified in previous research are either irrelevant or rarely encountered in real-world applications, making the analysis of such metrics inefficient and a potential drain on programming resources.

Table 17: Comparison of state-of-the-art Approaches on Common Code Smells

Study	Ref	Long Circuit (LC)	Excessive Measurements (EM)	Inconsistent Bits (IB)	Method
The Smelly Eight	([21])	99.15%	99%	-	QSmell Tool
Towards Metrics for Quantum Circuits Understandability	([28])	-	-	**99.93%**	Metric analysis
Quantum Program Linting with LLMs: LintQ-LLM	([47])	98.50%	98.10%	97.90%	LLM-based linting (Qiskit)
LLM-Powered Quantum Code Transpilation	([48])	95.20%	94.00%	93.75%	LLM-driven transpilation effects
This Study	-	**99.88%**	**99.63%**	99.63%	Structural and semantic metric analysis

To address this issue, this study focuses on identifying the most relevant and frequently occurring quantum code smells from previous research while proposing new, meaningful code smells based on insights from industry experts and reviewers. By narrowing down the scope to only the most impactful and common code smells, this research ensures that efforts are concentrated on detecting those that genuinely affect code quality.

This RQ investigates how the proposed methodology compares with existing state-of-the-art approaches in detecting the most relevant and common quantum code smells. The comparison includes not only earlier metric-based approaches such as The Smelly Eight [21] and Towards Metrics for Quantum Circuits Understandability [28], but also recent LLM-based approaches such as Quantum Program Linting with LLMs (LintQ-LLM) [47] and LLM-Powered Quantum Code Transpilation [48]. These LLM-driven works leverage large language models for static code analysis and transpilation, achieving strong detection performance but often limited by hallucination effects and over-generalization.

Table 17 presents a comparison with state-of-the-art approaches. As shown, this study validated the results for three existing code smells, while proposing three additional ones. The results indicate that for two out of the three existing code smells, the proposed methodology outperforms both metric-based and LLM-based approaches. These findings demonstrate the effectiveness of the structural and semantic-based approach in detecting quantum code smells.

Summary of RQ6: The proposed methodology outperformed existing state-of-the-art approaches, achieving an accuracy increase of up to 15% for two out of three validated code smells. By integrating structural and semantic analysis, it effectively identified the most relevant quantum code smells, validating its superiority in accuracy and detection efficiency compared to both traditional metric-based tools and recent LLM-based methods.

7 Threats to Validity

This section highlights about potential dimensions of bias affecting the reliability and generalization of the study. A different significant concern pertains to the quality and representativeness of the LintQ dataset: if LintQ introduces biases/errors (e.g., in the form of mislabeled code smells), the model could output inaccurate predictions. To mitigate this limitation, this study includes careful examination during the data collection and validation steps and only considers a comprehensive dataset of QP, filtering out inappropriate data through strict data cleaning. The dataset is further made robust by the verification of the labels by experts.

Overfitting can occur during model training when models learn from noise instead of correct labels and perform poorly on new data. The study applies several validation techniques (cross-validation, regularisation) to alleviate this problem and to ensure that the models can generalise to other kinds of datasets. The feature selection and extraction also suffer from this problem as these affect the model's ability to faithfully represent the code smells. This is addressed by the study during the feature engineering step, by obtaining the assistance of domain experts to help in separating the real predictors of code smells from the random noise.

The study also explores threats to the generalizability of the QUAMET methodology to alternative quantum programming frameworks and quantum algorithms, and applications. The dataset can cover a wide spectrum of quantum codes from different application domains to ensure that the models can serve various perspectives of quantum programming tasks. This issue is only made worse by the evolving side of quantum programming practices, as well as the problem of the definition and detection of code smells. The study thus repeatedly updates the dataset and models to reflect new developments in quantum programming and invites experts to collaborate on further enhancing and validating code smell definitions and metrics.

Ultimately, it is necessary to confirm the structural and semantic analysis to appropriately identify quantum code smells. Most of these analyses are subjected

to validation to verify that they work and are cross-referenced with accepted standards. This research aims to make trustworthy, practical, and genuine contributions to the field of Quantum Programming, in a scalable way, by addressing these threats systematically.

8 Conclusion and Future Work

This paper concludes by implementing structural and semantic metric analysis for code smell detection using a publicly available quantum dataset, LintQ, which consists of 7,568 QPs. This study demonstrates the significance of combining structural and semantic analysis to improve how code snippets are reviewed for metric generation. To reach this goal, this study presents a new proposal that combines semantic analysis, i.e., the analysis of the meaning of the model and structural metrics, with the detection of quantum code smells. This holistic approach combines both structural and semantic metrics to give an overall measurement of the quality of the code. These numbers are used as thresholds to detect six common quantum code smells in QPs, which were validated during the work.

As a result of this thorough analysis, a labelled dataset comprising 12 structural and semantic metrics, as well as labels for various code smells, was developed. In total, the dataset consists of 4,102 real-world QPs. After analysing the structural and semantic metrics, the results show that combining both structural and semantic metrics for code smell detection helps to increase the detection accuracy compared to the case of using only structural or semantic metrics separately.

Industry and academic experts reviewed the six code smells (LC, UC, IDQ, EM, IB, and LQV), evaluating their significance and implications. These reviews not only validated this analysis but also provided valuable insights into the practical relevance of these code smells for developers.

The study plans to use the methodology of structural and semantic analysis in other datasets for future work. Furthermore, the reviewers made the point that other code smells should be considered responsible, and this opens up a big opportunity for future research. Such future exploration could have the potential to provide a deeper insight into the quality of quantum code and possibly motivate the development of more sophisticated code smell detection tools.

Declarations

- **Funding:** No funding was received to assist with the preparation of this manuscript.
- **Conflict of Interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.
- **Ethics approval and consent to participate:** This article does not contain any studies with human participants or animals performed by any of the authors. The research involves the analysis of software codebases and publicly available datasets (MLCQ and quantum repositories).
- **Consent for publication:** Not applicable.

- **Data availability:** The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.
- **Materials availability:** Not applicable.
- **Code availability:** The custom scripts and code that support the findings of this study are available from the corresponding author [Initials] upon reasonable request.
- **Author Contribution: Ruchika Malhotra** and **Marouane Kessentini**: Proposing the idea; experimental design and analysis; Writing: review, editing and revision of the manuscript. **Bhawna Jain:** Conducting experimentation, Data curation, Design and analysis; Writing -original draft and revision of the manuscript.

References

[1] Benedetti, M., Realpe-Gómez, J., Biswas, R., Perdomo-Ortiz, A.: Estimation of effective temperatures in quantum annealers for sampling applications: A case study with possible applications in deep learning. Physical Review A **94**(2), 022308 (2016)

[2] Barends, R., Kelly, J., Megrant, A., Veitia, A., Sank, D., Jeffrey, E., White, T.C., Mutus, J., Fowler, A.G., Campbell, B., *et al.*: Superconducting quantum circuits at the surface code threshold for fault tolerance. Nature **508**(7497), 500–503 (2014)

[3] Bennett, C.H., Brassard, G.: Quantum cryptography: Public key distribution and coin tossing. arXiv preprint arXiv:2003.06557 (2020)

[4] Serrano, M.A., Cruz-Lemus, J.A., Perez-Castillo, R., Piattini, M.: Quantum software components and platforms: Overview and quality assessment. ACM Comput. Surv. **55**(8) (2022) https://doi.org/10.1145/3548679

[5] Zhao, J.: Quantum software engineering: Landscapes and horizons. arXiv preprint arXiv:2007.07047 (2020)

[6] Shaydulin, R., Thomas, C., Rodeghero, P.: Making quantum computing open: Lessons from open source projects. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pp. 451–455 (2020)

[7] Ball, H., Biercuk, M.J., Carvalho, A.R.R., Chen, J., Hush, M., Castro, L.A.D., Li, L., Liebermann, P.J., Slatyer, H.J., Edmunds, C., Frey, V., Hempel, C., Milne, A.: Software tools for quantum control: improving quantum computer performance through noise and error suppression. Quantum Science and Technology **6**(4), 044011 (2021) https://doi.org/10.1088/2058-9565/abdca6

[8] Leymann, F., Barzen, J., Falkenthal, M., Vietz, D., Weder, B., Wild, K.: Quantum in the Cloud: Application Potentials and Research Opportunities (2020)

[9] Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., Xie, Y.: Projection-based runtime

assertions for testing and debugging quantum programs. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–29 (2020)

[10] Huang, Y., Martonosi, M.: Statistical assertions for validating patterns and finding bugs in quantum programs. In: Proceedings of the 46th International Symposium on Computer Architecture, pp. 541–553 (2019)

[11] Gupta, R., Kumar Singh, S.: A novel metric based detection of temporary field code smell and its empirical analysis. Journal of King Saud University - Computer and Information Sciences **34**(10, Part B), 9478–9500 (2022) https://doi.org/10.1016/j.jksuci.2021.11.005

[12] Barbez, A., Khomh, F., Guéhéneuc, Y.-G.: Deep learning anti-patterns from code metrics history. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 114–124 (2019). https://doi.org/10.1109/ICSME.2019.00021

[13] Ali, S., Arcaini, P., Wang, X., Yue, T.: Assessing the effectiveness of input and output coverage criteria for testing quantum programs. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 13–23 (2021). IEEE

[14] Ahmed, I., Brindescu, C., Mannan, U.A., Jensen, C., Sarma, A.: An empirical examination of the relationship between code smells and merge conflicts. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 58–67 (2017). IEEE

[15] Zhao, P., Zhao, J., Ma, L.: Identifying bug patterns in quantum programs. In: 2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE), pp. 16–21 (2021). IEEE

[16] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, ??? (2018)

[17] Jebnoun, H., Ben Braiek, H., Rahman, M.M., Khomh, F.: The scent of deep learning code: An empirical study. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 420–430 (2020)

[18] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering **43**(11), 1063–1088 (2017)

[19] Yamashita, A., Moonen, L.: Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 682–691 (2013). IEEE

[20] Malhotra, R., Jain, B., Kessentini, M.: Examining deep learning's capability to spot code smells: a systematic literature review. Cluster Computing **26**(6), 3473–3501 (2023)

[21] Chen, Q., Câmara, R., Campos, J., Souto, A., Ahmed, I.: The smelly eight: An empirical study on the prevalence of code smells in quantum computing. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 358–370 (2023). IEEE

[22] Openja, M., Morovati, M.M., An, L., Khomh, F., Abidi, M.: Technical debts and faults in open-source quantum software systems: An empirical study. Journal of Systems and Software **193**, 111458 (2022) https://doi.org/10.1016/j.jss.2022.111458

[23] Agnihotri, M., Chug, A.: A systematic literature survey of software metrics, code smells and refactoring techniques. Journal of Information Processing Systems **16**(4), 915–934 (2020)

[24] Mhawish, M.Y., Gupta, M.: Predicting code smells and analysis of predictions: using machine learning techniques and software metrics. Journal of Computer Science and Technology **35**(6), 1428–1445 (2020)

[25] Zhao, P., Wu, X., Li, Z., Zhao, J.: QChecker: Detecting Bugs in Quantum Programs via Static Analysis (2023)

[26] Nayak, P.K., Kher, K.V., Chandra, M.B., Rao, M.V.P., Zhang, L.: Q-PAC: Automated Detection of Quantum Bug-Fix Patterns (2023)

[27] Zhao, J.: Some size and structure metrics for quantum software. In: 2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE), pp. 22–27. IEEE Computer Society, Los Alamitos, CA, USA (2021). https://doi.org/10.1109/Q-SE52541.2021.00012 . https://doi.ieeecomputersociety.org/10.1109/Q-SE52541.2021.00012

[28] Cruz-Lemus, J.A., Marcelo, L.A., Piattini, M.: Towards a set of metrics for quantum circuits understandability. In: International Conference on the Quality of Information and Communications Technology, pp. 239–249 (2021). Springer

[29] Paltenghi, M., Pradel, M.: Lintq: A static analysis framework for qiskit quantum programs. arXiv preprint arXiv:2310.00718 (2023)

[30] Safavian, S.R., Landgrebe, D.: A survey of decision tree classifier methodology. IEEE transactions on systems, man, and cybernetics **21**(3), 660–674 (1991)

[31] Cortes, C., Vapnik, V.: Support-vector networks. Machine learning **20**, 273–297 (1995)

[32] Santos, E.M., Gomes, H.M.: A comparative study of polynomial kernel svm applied to appearance-based object recognition. In: International Workshop on Support Vector Machines, pp. 408–418 (2002). Springer

[33] Lin, H.-T., Lin, C.-J.: A study on sigmoid kernels for svm and the training of non-psd kernels by smo-type methods. Neural Computation (2003)

[34] Ding, X., Liu, J., Yang, F., Cao, J.: Random radial basis function kernel-based support vector machine. Journal of the Franklin Institute **358** (2021) https://doi.org/10.1016/j.jfranklin.2021.10.005

[35] Domingos, P., Pazzani, M.: Beyond independence: Conditions for the optimality of the simple bayesian classi er. In: Proc. 13th Intl. Conf. Machine Learning, pp. 105–112 (1996)

[36] Anand, V., KiranBala, B., Srividhya, S., C., K., Younus, M., Rahman, M.H.: Gaussian naïve bayes algorithm: A reliable technique involved in the assortment of the segregation in cancer. Mobile Information Systems **2022**, 1–7 (2022) https://doi.org/10.1155/2022/2436946

[37] Abbas, M., Memon, K.A., Jamali, A.A., Memon, S., Ahmed, A.: Multinomial naive bayes classification model for sentiment analysis. IJCSNS Int. J. Comput. Sci. Netw. Secur **19**(3), 62 (2019)

[38] Bishop, C.M., Nasrabadi, N.M.: Pattern Recognition and Machine Learning vol. 4. Springer, ??? (2006)

[39] Dawson, C., Wilby, R.: Hydrological modelling using artificial neural networks. Progress in physical Geography **25**(1), 80–108 (2001)

[40] Bottou, L.: Large-scale machine learning with stochastic gradient descent. In: Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers, pp. 177–186 (2010). Springer

[41] Loef, B., Wong, A., Janssen, N., Strak, M., Hoekstra, J., Picavet, S., Boshuizen, H., Verschuren, W., Herber, G.-C.: Using random forest to identify longitudinal predictors of health in a 30-year cohort study. Scientific Reports **12**, 10372 (2022) https://doi.org/10.1038/s41598-022-14632-w

[42] Cover, T., Hart, P.: Nearest neighbor pattern classification. IEEE transactions on information theory **13**(1), 21–27 (1967)

[43] Fukunaga, K.: Introduction to Statistical Pattern Recognition. Elsevier, ??? (2013)

[44] Hanusz, Z., Tarasinska, J., Zieliński, W.: Shapiro–wilk test with known mean.

Revstat Statistical Journal **14**, 89–100 (2016) https://doi.org/10.57805/revstat.v14i1.180

[45] Hollander, M., Wolfe, D.A., Chicken, E.: Nonparametric Statistical Methods. John Wiley & Sons, ??? (2013)

[46] Demšar, J.: Statistical comparisons of classifiers over multiple data sets. J. Mach. Learn. Res. **7**, 1–30 (2006)

[47] Shin, S.Y., Pastore, F., Bianculli, D.: Quantum program linting with llms: Emerging results from a comparative study. arXiv preprint arXiv:2504.05204 (2025)

[48] Siavash, N., Moin, A.: Llm-powered quantum code transpilation. arXiv preprint arXiv:2507.12480 (2025)



Dr. Ruchika Malhotra is a Professor and Head of Department at the Department of Software Engineering, Delhi Technological University, Delhi, India. She was awarded the prestigious Raman Fellowship for pursuing Postdoctoral research at Indiana University Purdue University Indianapolis, USA. She received the IBM Faculty Award in 2013. Her h-index is 36, as reported by Google Scholar. She is the author of a book titled "Empirical Research in Software Engineering" published by CRC Press and co-author of a book on "Object-Oriented Software Engineering" published by PHI Learning. Her research interests are software testing, improving software quality, statistical and adaptive prediction models, software metrics, and the definition and validation of software metrics. She has published more than 220 research papers in international journals and conferences and is amongst the world's top 2% scientists as per reports from Stanford University.



Ms. Bhawna Jain is pursuing a Ph.D. in Computer Science under joint supervision from Delhi Technological University (DTU)- University of Michigan Flint, USA. She is an Assistant Professor in the Department of Computer Science and Engineering at Indira Gandhi Delhi Technical University for Women (IGDTUW). She has over three years of experience working as a Software Developer in the industry. She has won awards like All-Rounder Performer and ACE Excellence award for being Growth Catalyst and received appreciation for her remarkable efforts in the R&D department during her industry experience. She also contributed to the Guinness Book of World Records for the "Largest online video chain of people handing over an object." She has research publications with IEEE and Springer. Her areas of interest are Artificial Intelligence, Deep Learning, Software Maintenance, and Generative AI.



Dr. Marouane Kessentini is working as Dean of the College of Computing at Grand Valley State University and serves as Director of the NSF IUCRC Center on Pervasive AI (PPI, Michigan Site). He is the recipient of the prestigious 2018 President of Tunisia Distinguished Research Award, the University Distinguished Teaching Award, the University Distinguished Digital Education Award, the College of Engineering and Computer Science Distinguished Research Award, four Best Paper awards, including an IEEE 10-Year Most Influential Paper award (2011-2021), and his AI-based software refactoring invention, licensed and deployed by several Fortune 500 companies, and selected as one of the Top 8 inventions at the University of Michigan for 2018 (including the three campuses) by the UM Technology Transfer Office. He received grants from industry and federal agencies and published over 175 papers in top journals and conferences. His h-index is 46, as reported by Google Scholar.