UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA MECÁNICA

QUANTUM MACHINE LEARNING FOR PREDICTIVE MAINTENANCE

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL MECÁNICO

CRISTIAN ANTONIO MAC-KAY CISTERNAS

PROFESOR GUÍA:
ENRIQUE LÓPEZ DROGUETT

MIEMBROS DE LA COMISIÓN:
VIVIANA MERUANE NARANJO
BENJAMIN HERRMANN PRIESNITZ

SANTIAGO DE CHILE
2023

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL MECÁNICO
POR: CRISTIAN ANTONIO MAC-KAY CISTERNAS
AÑO: 2023
PROF. GUÍA: ENRIQUE LÓPEZ DROGUETT

MACHINE LEARNING CUÁNTICO PARA EL MANTENIMIENTO PREDICTIVO

Durante los últimos años, el aprendizaje de máquinas ha sido utilizado para resolver problemas sobre mantenimiento predictivo. En este contexto, se han desarrollado aplicaciones en dos principales tópicos: Diagnóstico y Pronóstico.

El objetivo de desarrollar algoritmos de aprendizaje de máquinas para el diagnóstico es predecir si un componente o sistema mecánico está fallando o no. Por otro lado, el objetivo de las aplicaciones del aprendizaje de máquinas sobre el pronóstico es predecir cuánta vida útil tiene un componente o un sistema mecánico.

Sin embargo, en muchos casos, las técnicas actuales son demasiado caras o no lo suficientemente fiables para utilizarlas en la industria. Una causa de ello es que hay aplicaciones para las cuales estos algoritmos son difíciles de entrenar, debido al gran volumen de datos, a la complejidad de los sistemas o a su multidimensionalidad.

En este contexto surge la idea de explorar las capacidades del aprendizaje de máquinas cuántico para el mantenimiento predictivo. Debido a que la computación cuántica permite realizar un número indefinido de tareas simultáneamente debido a los principios cuánticos en los que se basa.

En el desarrollo de este trabajo, se exploran las capacidades de diferentes modelos de aprendizaje de máquinas cuántico en búsqueda de aquellos que ofrezcan resultados prometedores. Estos modelos fueron probados con dos casos de estudio pertinentes al mantenimiento predictivo y cuyos resultados se compararan con los resultados obtenidos con técnicas clásicas de aprendizaje de máquinas. Para desarrollar el tema, se utiliza en Python utilizando Jupyter Notebook, IBM Quantum Lab y Google Colab, permitiendo de esa forma utilizar entornos de ejecución en donde se integren librerías útiles del aprendizaje de máquinas clásico, como Scikitlearn, Pytorch, Pandas, entre otras, con bibliotecas utilizadas para la computación cuántica como Qiskit.

DEGREE TOPIC ABSTRACT TO OBTAIN
THE MECHANICAL ENGINEER TITLE
BY: CRISTIAN ANTONIO MAC-KAY CISTERNAS
YEAR: 2023
GUIDE PROF.: ENRIQUE LÓPEZ DROGUETT

QUANTUM MACHINE LEARNING FOR PREDICTIVE MAINTENANCE

During the last few years, machine learning has been used to solve predictive maintenance problems. In this context, there have been developed applications in two main topics: Diagnostics and Prognostics.

The goal of developing machine learning algorithms for diagnostics is to predict whether or not a component or mechanical system is failing. On the other hand, the goal of machine learning applications on prognostics is to predict the remaining useful lifetime of a component or mechanical system.

However, in many cases, current techniques are too expensive or not reliable enough to be used in industry. One reason for this is that there are applications for which these algorithms are difficult to train, due to the large volume of data, the complexity of the systems or their multidimensionality.

In this context, the idea of exploring the capabilities of quantum machine learning for predictive maintenance arises. Because quantum computing allows an indefinite number of tasks to be performed simultaneously due to the quantum principles on which it is based.

In the development of this work, the capabilities of different quantum machine learning models are explored to find those that offer promising results. These models were tested with two relevant cases of study to predictive maintenance and whose results will be compared with the results obtained with classical machine learning techniques. To develop the topic, Python is used in Jupyter Notebook, IBM Quantum Lab and Google Colab, thus allowing using execution environments where useful libraries of classical machine learning, such as Scikitlearn, Pytorch, Pandas, among others, are integrated with libraries used for quantum computing such as Qiskit.

A mis seres queridos, quienes son como las estrellas brillando en mi camino, iluminando mi sendero hacia el éxito. Gracias por ser mi fortaleza en los momentos difíciles, por ser la energía que impulsa mis sueños, por darme la oportunidad de ejecutar mis deseos.

Acknowledgments

The following text is written in Spanish for those who supported me during this process.

“ Estimados seres queridos que han marcado mi vida durante este proceso. Ocupo este espacio para expresarles mi gratitud infinita. Por estar a mi lado en esta aventura llena de momentos. Procedo a dedicarles unas breves palabras a cada uno de ustedes, empezando por mi amada familia.

Papá, eres mi roca inquebrantable y mi guía fiel, siempre estuviste ahí para apoyarme en mis metas y desafíos, a pesar de lo difíciles que pudieran parecer, nunca te rendiste. Mamá, me has brindado amor incondicional y has inculcado en mí muchos de los valores que hoy tengo con orgullo, gracias por enseñarme a querer y estar a mi lado incondicionalmente. Carol, mi querida hermana, gracias por las risas compartidas y por tu preocupación y apoyo constante hacia mí, pues sin tu iniciativa este proceso no hubiera comenzado. Daniela, mi amor, gracias por tu compañía y por ser mi luz, me inspiras y me apoyas en los momentos más oscuros siendo mi hogar y mi todo, te quiero con todo mi corazón. Tsuki, mi pequeña gatita y fiel compañera, gracias por tu amor y tus ronroneos, eres mi compañía en las tardes solitarias y un regalo en mi vida. Gracias a todos por hacer de mi hogar un lugar lleno de amor.

Amigos y compañeros de universidad, gracias por las risas, los aprendizajes y los desafíos. Hicieron que mi tiempo en la universidad fuera inolvidable. Gracias por su amistad y por ser parte de mi camino, tanto a aquellos que permanecen en mi vida como a aquellos con quienes separamos nuestros caminos.

Profesores, gracias por su paciencia y dedicación. Me guiaron y me enseñaron en mi formación. Fueron faros en mi camino hacia el conocimiento. Me enseñaron a ver más allá de lo que mis ojos alcanzaban. Tanto a ustedes como a la universidad, gracias por ofrecerme una formación de excelencia, aprendí tanto dentro como fuera de las aulas, con el rigor necesario para mi formación. Gracias Universidad de Chile por ser también un hogar en esta etapa de mi vida, espero poder seguir nutriéndome de todo lo que me brindaron y seguir creciendo con estas enseñanzas.

Este trabajo significa la culminación de una etapa importante, la conclusión de mi carrera universitaria. Este texto es solo un pequeño eco de mi profunda gratitud, cada uno de ustedes ha sido un pilar fundamental en mi camino y estoy agradecido por su presencia constante en mi vida. Espero poder retribuir en algún momento todo lo que ustedes han hecho por mí. ¡Gracias por todo! ”

Table of content

1. Introduction **1**
1.1. Objectives . . . 2
1.2. Scopes . . . 2

2. Background **4**
2.1. Machine Learning General Concepts . . . 4
2.1.1. Supervised learning . . . 4
2.1.2. Unsupervised learning . . . 4
2.1.3. Overfitting . . . 5
2.1.4. Evaluation metrics . . . 5
2.1.5. Obtaining time parameters . . . 8
2.1.5.1. Time windows . . . 8
2.1.5.2. Overlap . . . 8
2.1.5.3. Temporal parameters . . . 8
2.1.6. Dimensionality reduction . . . 10
2.1.7. One hot encoding . . . 12
2.1.8. Support Vector Classifier . . . 12
2.1.9. Neural Networks . . . 15
2.2. Classical computing . . . 18
2.3. Quantum computing . . . 22
2.3.1. Background . . . 22
2.3.1.1. Quantum Bits . . . 22
2.3.1.2. Quantum superposition . . . 23
2.3.1.3. Observer effect . . . 23
2.3.1.4. Quantum interference . . . 25
2.3.1.5. Quantum entanglement . . . 26
2.3.1.6. Quantum Circuits . . . 27
2.3.1.7. Quantum Gates . . . 28
2.3.1.7.1. Bit Flip Pauli Gate (X-Gate) . . . 29
2.3.1.7.2. Hadamard Gate (H-Gate) . . . 29
2.3.1.7.3. Rotation Gates . . . 30
2.3.1.7.4. Unitary Gate (U-Gate) . . . 32
2.3.1.7.5. Phase (P-Gate) . . . 33
2.3.1.7.6. C-NOT Gate . . . 33
2.3.1.8. Qubits Measurements . . . 34
2.3.2. Quantum computers . . . 35

2.3.3. Simulators 37
2.3.4. IBM Qiskit 37
2.3.4.1. Backends 38
2.4. Programming Resources 39
2.5. Quantum Machine Learning 41
2.5.1. General Concepts 42
2.5.1.1. Features Maps 42
2.5.1.2. Ansatz 44
2.5.1.3. Optimizer 45
2.5.1.4. Quantum Neural Network 46
2.5.2. Models used 48
2.5.2.1. Variatonal Quantum Classifier 48
2.5.2.2. Quantum Neural Network Classifier 49
2.5.2.3. Quantum Neural Network with Pytorch Classifier . . . 50
2.5.2.4. Quantum Support Vector Classifier 52
2.6. Used cases of study 53
2.6.1. MFPT 53
2.6.2. C-MAPSS 55

3. Methodology **57**
3.1. Selection of case studies and data exploration 57
3.2. Initial data preparation 58
3.2.1. Assembling datasets 58
3.2.1.1. MFPT 58
3.2.1.2. C-MAPSS 60
3.2.1.3. Sensor selection 61
3.3. Selection of used models 64
3.4. Implement classical model to cases of study 64
3.5. Implement quantum models to cases of study. 65
3.5.1. Variational Quantum Classifier 65
3.5.2. Quantum Neural Network Classifier 67
3.5.3. Quantum Neural Network with Pytorch Classifier 68
3.5.4. Quantum Support Vector Classifier 68

4. Results **70**
4.1. Data processing results 70
4.1.1. Dimensionality reduction 73
4.2. Classical Support Vector Classifier Results 74
4.2.1. MFPT Fault Dataset Manipulation 74
4.2.2. C-MAPSS Aircraft Engine Simulator 74
4.2.2.1. FD001 Statistics Features 74
4.2.2.2. FD001 Sensors Features 75
4.3. Variational Quantum Classifier Results 75
4.3.1. MFPT Fault Dataset Manipulation 76
4.3.2. C-MAPSS Aircraft Engine Simulator 77
4.3.2.1. FD001 Statistics Features 77
4.3.2.2. FD001 Sensors Features 79

4.4. Quantum Neural Network Classifier Results 80
4.4.1. C-MAPSS Aircraft Engine Simulator 80
4.4.1.1. FD001 Statistics Features 81
4.4.1.2. FD001 Sensors Features 82
4.5. Quantum Neural Network with Pytorch Classifier Results 83
4.5.1. C-MAPSS Aircraft Engine Simulator 83
4.5.1.1. FD001 Statistics Features 83
4.6. Quantum Support Vector Classifier Results 84
4.6.1. C-MAPSS Aircraft Engine Simulator 84
4.6.1.1. FD001 Sensors Features 84
4.7. Unused Quantum Scructures 85

5. Discussions 87
5.1. Variational Quantum Classifier for MFPT Dataset 87
5.2. Variational Quantum Classifier for CMAPSS Statistics Features Dataset 88
5.3. Variational Quantum Classifier for CMAPSS Sensors Features Dataset 89
5.4. Quantum Neural Network Classifier for CMAPSS Statistics Features Dataset 90
5.5. Quantum Neural Network Classifier for CMAPSS Sensors Features Dataset 91
5.6. Quantum Neural Network with Pytorch Classifier for CMAPSS Sensors Features Dataset 92
5.7. Quantum Support Vector Classifier for CMAPSS Sensors Features Dataset 93
5.8. Classical Results vs Quantum Results 94

6. Conclusions 97

7. Proposed work 99

Bibliography 101

Annex A. Datasets Features Plots 103
A.1. MFPT Features plots 103
A.2. C-MAPSS FD001 Features plots 106

Annex B. Models Confusion Matrices 113
B.1. Variational Quantum Classifier 113
B.1.1. MFPT Fault Dataset 113
B.1.1.1. FD001 Statistics Features 117
B.1.1.2. FD001 Sensors Features 124
B.2. Quantum Neural Network Classifier 130
B.2.0.1. FD001 Statistics Features 130
B.2.0.2. FD001 Sensors Features 132
B.3. Pytorch + Quantum Neural Network Classifier 135
B.4. Quantum Support Vector Classifier 137

Annex C. Codes 138
C.1. Classical SVC 138
C.2. Variational Quantum Classifier 138

C.3. Quantum Neural Network Classifier 140
C.4. Quantum Neural Network with Pytorch Classifier 141
C.5. Quantum Support Vector Classifier 142

List of Tables

2.1. One-hot encoding example 12
3.1. MFPT Data after redefining sampling rate and measured time. 59
3.2. MFPT Data Example after grouping in time windows. 59
3.3. MFPT Data Example after stacking time windows for every health condition. 60
3.4. MFPT Data Example after calculating new features. 60
3.5. C-MAPSS Raw Datasets Conditions. 60
3.6. C-MAPSS FD001 Raw Dataset 61
3.7. C-MAPSS FD001 Raw Dataset with Estimated RUL 62
3.8. C-MAPSS FD001 Dataset after columns droping. 62
3.9. C-MAPSS FD001 Dataset After PCA Feature Reduction 62
3.10. C-MAPSS FD001 RUL Dataset 63
3.11. C-MAPSS FD001 PCA Features from Sensors Dataset for Classification . 63
3.12. C-MAPSS FD001 Statistical Features Dataset 63
3.13. C-MAPSS FD001 Statistical Features after PCA Reduction with Labels . 64
4.1. Datasets PCA Reduction Results 73
4.2. Classical Predictions Results 75
4.3. VQC for MPFT Dataset Results 77
4.4. VQC for C-MAPSS FD001 with Statistics Features Results 78
4.5. VQC for C-MAPSS FD001 with Sensor Features Results 80
4.6. QNN for C-MAPSS FD001 with Statistics Features Results 81
4.7. QNN for C-MAPSS FD001 with Sensors Features Results 82
4.8. Pytorch + QNN for C-MAPSS FD001 with Statistics Features Results . 84
4.9. QVSC for C-MAPSS FD001 with Sensors Features Results 85
4.10. Unused Structures 86

List of Figures

2.1.1. Example of labeled and unlabeled data for classification and clustering. . 5
2.1.2. Overfitting example. 5
2.1.3. Confusion matrix example. 6
2.1.4. Hyperplane illustrations. 13
2.1.5. Optimal hyperplane example. 13
2.1.6. Kernel Example. 14
2.1.7. Artificial Neuron Scheme. 16
2.1.8. Artificial Neural Network Scheme. 16
2.1.9. Activation Functions. 17
2.2.1. Classical Logic Gates. 19
2.2.2. Schematic diagram of a classic transistor. 19
2.2.3. Scheme of AND operation with transistors. 20
2.2.4. Scheme of OR operation with transistors. 20
2.2.5. Scheme of XOR operation with transistors. 20
2.2.6. Half Adder circuit diagram. 21
2.2.7. Schematic diagram of the ALU TI 7400. 21
2.3.1. Comparision between a classical bit and a quantum bit. 22
2.3.2. Classical noise cancel example. 25
2.3.3. Multiple states superposition. 26
2.3.4. Quantum Circuit Example. [11] 27
2.3.5. Illustration of an angle representation for a Quantum State. 30
2.3.6. Illustration of states rotations on Blech Sphere. 31
2.3.7. An inside view of IBM Quantum System One. 36
2.5.1. Classical Neural Network vs Quantum Neural Network Comparison . . . 47
2.5.2. VQC Example Circuit 49
2.5.3. QNN Classifier Example Circuit 50
2.6.1. MFPT Fault Dataset Manipulation 54
2.6.2. C-MAPSS Aircraft Engine Simulator 55
3.5.1. VQC FeatureMap and Ansatz Example 66
3.5.2. VQC Callback Graph Example 67
4.1.1. MFPT Raw Dataset Plot 70
4.1.2. MFPT Dataset Mean Plot 71
4.1.3. MFPT Dataset Variance Plot 71
4.1.4. MFPT Dataset RMS Plot 71
4.1.5. C-MAPSS Aircraft Correlation Matrix Between Features 72
4.1.6. C-MAPSS FD001 Train Dataset LPC Outlet Temperature Plots 73
4.1.7. C-MAPSS FD001 Test Dataset HPT Coolant Bleed Plots 73

4.2.1. Confussion Matrixes for Classical SVC applied to MFPT 74
4.2.2. Confussion Matrixes for Classical SVC applied to C-MAPSS FD001 with Statistics Features 74
4.2.3. Confussion Matrixes for Classical SVC applied to C-MAPSS FD001 with Statistics Features 75
4.3.1. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to MFPT 76
4.3.2. Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and COBYLA applied to MFPT 76
4.3.3. Confussion Matrixes for VQC with RawFeatures, EfficientSU2 and COBYLA applied to MFPT 76
4.3.4. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Statistics Features 77
4.3.5. Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Sensor Features 78
4.3.6. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Sensor Features 78
4.3.7. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Sensor Features 79
4.3.8. Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Sensor Features 79
4.3.9. Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Sensor Features 80
4.4.1. Confussion Matrixes for QNN with ZZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Statistics Features 81
4.4.2. Confussion Matrixes for QNN with ZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Statistics Features 81
4.4.3. Confussion Matrixes for QNN with ZZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Sensor Features 82
4.4.4. Confussion Matrixes for QNN with ZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Sensor Features 82
4.5.1. Confussion Matrixes for Pytorch + OpflowQNN with ZZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features 83
4.5.2. Confussion Matrixes for Pytorch + CircuitQNN with ZZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features 83
4.6.1. Confussion Matrixes for Quantum SVC with ZFeatureMap applied to C-MAPSS FD001 with Sensors Features 84
4.6.2. Confussion Matrixes for Quantum SVC with ZZFeatureMap applied to C-MAPSS FD001 with Sensors Features 85
A.1.1. MFPT Dataset Mean Plot 103
A.1.2. MFPT Dataset Variance Plot 103
A.1.3. MFPT Dataset RMS Plot 104
A.1.4. MFPT Dataset Peak Plot 104
A.5. MFPT Dataset Valley Plot 104
A.6. MFPT Dataset Peak to Peak Plot 105

A.7. MFPT Dataset Crest Factor Plot 105
A.8. MFPT Dataset Kurtosis Plot 105
A.9. MFPT Dataset Skewness Plot 106
A.2.1. C-MAPSS FD001 Dataset LPC Outlet Temperature Mean Plot 106
A.2.2. C-MAPSS FD001 Dataset LPC Outlet Temperature Standard Deviation Plot 106
A.2.3. C-MAPSS FD001 Dataset HPC Outlet Temperature Mean Plot 107
A.2.4. C-MAPSS FD001 Dataset HPC Outlet Temperature Standard Deviation Plot 107
A.2.5. C-MAPSS FD001 Dataset LPT Outlet Temperature Mean Plot 107
A.2.6. C-MAPSS FD001 Dataset LPT Outlet Temperature Standard Deviation Plot 108
A.2.7. C-MAPSS FD001 Dataset Bypass Duct Pressure Mean Plot 108
A.2.8. C-MAPSS FD001 Dataset Bypass Duct Pressure Standard Deviation Plot 108
A.2.9. C-MAPSS FD001 Dataset HPC Outlet Pressure Mean Plot 109
A.2.10. C-MAPSS FD001 Dataset HPC Outlet Pressure Standard Deviation Plot 109
A.2.11. C-MAPSS FD001 Dataset Corrected Fan Speed Mean Plot 109
A.2.12. C-MAPSS FD001 Dataset Corrected Fan Speed Standard Deviation Plot 110
A.2.13. C-MAPSS FD001 Dataset Bypass Ratio Mean Plot 110
A.2.14. C-MAPSS FD001 Dataset Bypass Ratio Standard Deviation Plot 110
A.2.15. C-MAPSS FD001 Dataset Bleed Enthalpy Mean Plot 111
A.2.16. C-MAPSS FD001 Dataset Bleed Enthalpy Standard Deviation Plot . . . 111
A.2.17. C-MAPSS FD001 Dataset HPT Coolant Bleed Mean Plot 111
A.2.18. C-MAPSS FD001 Dataset HPT Coolant Bleed Standard Deviation Plot . 112
A.2.19. C-MAPSS FD001 Dataset HPT Coolant Bleed Mean Plot 112
A.2.20. C-MAPSS FD001 Dataset HPT Coolant Bleed Standard Deviation Plot . 112
B.1.1. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to MFPT 113
B.1.2. Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and COBYLA applied to MFPT 114
B.1.3. Confussion Matrixes for VQC with RawFeatures, EfficientSU2 and COBYLA applied to MFPT 114
B.1.4. Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and COBYLA applied to MFPT 114
B.1.5. Confussion Matrixes for VQC with RawFeatures, RealAmplitudes and COBYLA applied to MFPT 115
B.1.6. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SLSQP applied to MFPT 115
B.1.7. Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and SLSQP applied to MFPT 115
B.1.8. Confussion Matrixes for VQC with RawFeatures, RealAmplitudes and SLSQP applied to MFPT 116
B.1.9. Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and SLSQP applied to MFPT 116
B.1.10. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SPSA applied to MFPT 116

B.1.11. Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and SPSA applied to MFPT . . . 117
B.1.12. Confussion Matrixes for VQC with RawFeatures, EfficientSU2 and SPSA applied to MFPT . . . 117
B.1.13. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Statistics Features . . . 117
B.1.14. Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Statistics Features . . . 118
B.1.15. Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Statistics Features . . . 118
B.1.16. Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Statistics Features . . . 118
B.1.17. Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Statistics Features . . . 119
B.1.18. Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Statistics Features . . . 119
B.1.19. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Statistics Features . . . 119
B.1.20. Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Statistics Features . . . 120
B.1.21. Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Statistics Features . . . 120
B.1.22. Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Statistics Features . . . 120
B.1.23. Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Statistics Features . . . 121
B.1.24. Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Statistics Features . . . 121
B.1.25. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Statistics Features . . . 121
B.1.26. Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Statistics Features . . . 122
B.1.27. Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Statistics Features . . . 122
B.1.28. Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Statistics Features . . . 122
B.1.29. Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Statistics Features . . . 123
B.1.30. Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Statistics Features . . . 123
B.1.31. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Sensor Features . . . 124
B.1.32. Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Sensor Features . . . 124
B.1.33. Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Sensor Features . . . 125

B.1.34. Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Sensor Features 125
B.1.35. Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Sensor Features 125
B.1.36. Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Sensor Features 126
B.1.37. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Sensor Features 126
B.1.38. Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Sensor Features 126
B.1.39. Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Sensor Features 127
B.1.40. Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Sensor Features 127
B.1.41. Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Sensor Features 127
B.1.42. Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Sensor Features 128
B.1.43. Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Sensor Features 128
B.1.44. Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Sensor Features 128
B.1.45. Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Sensor Features 129
B.1.46. Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Sensor Features 129
B.1.47. Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Sensor Features 129
B.1.48. Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Sensor Features 130
B.2.1. Confussion Matrixes for QNN with ZZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Statistics Features 130
B.2.2. Confussion Matrixes for QNN with ZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Statistics Features 131
B.2.3. Confussion Matrixes for QNN with ZZFeatureMap, EfficientSU2 to C-MAPSS FD001 with Statistics Features 131
B.2.4. Confussion Matrixes for QNN with ZFeatureMap, EfficientSU2 to C-MAPSS FD001 with Statistics Features 131
B.2.5. Confussion Matrixes for QNN with RawFeatureVector, EfficientSU2 to C-MAPSS FD001 with Statistics Features 132
B.2.6. Confussion Matrixes for QNN with RawFeatureVector, RealAmplitudes to C-MAPSS FD001 with Statistics Features 132
B.2.7. Confussion Matrixes for QNN with ZZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Sensor Features 132
B.2.8. Confussion Matrixes for QNN with ZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Sensor Features 133

B.2.9. Confussion Matrixes for QNN with ZZFeatureMap, EfficientSU2 to C-MAPSS FD001 with Sensor Features 133
B.2.10. Confussion Matrixes for QNN with ZFeatureMap, EfficientSU2 to C-MAPSS FD001 with Sensor Features 133
B.2.11. Confussion Matrixes for QNN with RawFeatureVector, EfficientSU2 to C-MAPSS FD001 with Sensor Features 134
B.2.12. Confussion Matrixes for QNN with RawFeatureVector, RealAmplitudes to C-MAPSS FD001 with Sensor Features 134
B.3.1. Confussion Matrixes for Pytorch + OpflowQNN with ZZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features 135
B.3.2. Confussion Matrixes for Pytorch + CircuitQNN with ZZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features 135
B.3.3. Confussion Matrixes for Pytorch + OpflowQNN with ZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features 136
B.3.4. Confussion Matrixes for Pytorch + CircuitQNN with ZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features 136
B.3.5. Confussion Matrixes for Pytorch + OpflowQNN with RawFeatureVector, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features 136
B.3.6. Confussion Matrixes for Pytorch + CircuitQNN with RawFeatureVector, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features 137
B.4.1. Confussion Matrixes for Quantum SVC with ZFeatureMap applied to C-MAPSS FD001 with Sensors Features 137
B.4.2. Confussion Matrixes for Quantum SVC with ZZFeatureMap applied to C-MAPSS FD001 with Sensors Features 137

1 | Introduction

Throughout the technological development, there has been an evolution of the maintenance techniques used in the different engineering industries. These techniques, which allow to operate with greater efficiency the different stages of the productive processes, have evolved in an accelerated way starting from corrective strategies, to techniques which consist of scheduled works such as preventive or predictive maintenance.

The strategies consisting of scheduled maintenance works are aimed at maximizing the uptime of a system's components and thus optimizing the available resources. This objective is achieved by finding the right time to repair or replace a component, thus avoiding unplanned shutdowns that compromise production and operational safety, as well as premature replacement of still functional components.

In this context, maintenance strategies have shifted towards predictive maintenance techniques, which optimize the planning of maintenance tasks, adding information on reliability that allows to decide more effectively when to execute the respective maintenance tasks. These techniques are performed through continuous monitoring of the components or equipment of a production process, using tools such as accelerometers, thermographs, ultrasonic sensors, among others.

The previously mentioned predictive strategy has evolved, hand in hand with technological and theoretical development, towards increasingly sophisticated techniques that allow processing the information obtained from continuous monitoring to perform analysis and predictions through computational techniques such as Machine Learning or Deep Learning. These techniques allow a computer to learn, from the information provided by equipment monitoring, to perform various useful tasks in order to obtain a deeper understanding of the condition of the equipment or components.

However, during the development and advancement of these computational machine learning techniques, it has been found that there are applications where these algorithms are difficult to implement, due to the complexity of the problems to be solved. This represents an important challenge, since it is of vital importance to obtain increasingly better results, so that they can be applied with reliability in industries that operate under critical safety conditions, in this case, in the context of maintenance and reliability.

On the other hand, in recent years, a new computational tool has emerged which

generates interest for its promising potential: quantum computing. The potential that characterizes quantum computing is explained by the quantum principles on which it is based, allowing to execute an indefinite number of superimposed tasks.

Nowadays, the first quantum processing units are already available through open APIs (IBM's Qiskit, among others), which also contain methods for simulating quantum processing units. Because of this, it is already possible to explore the capabilities that this new technology offers when used to implement machine learning techniques in the context of maintenance and reliability for predictive strategies. This area of study arising from the convergence between machine learning and quantum computing is called quantum machine learning.

Motivated by the previously described, the capabilities of quantum machine learning applied to techniques used for reliability and maintenance will be explored in this degree work.

1.1. Objectives

Overall Objective

The overall objective of the present work is to explore and implement new quantum machine learning algorithms and models in the context of predictive maintenance.

Specific Objectives

To achieve the overall objective of this work, the following specific objectives were proposed:

- Select relevant cases of studies, both in the literature and in the community's ongoing research.
- Select quantum machine learning techniques, based on the principles of quantum computing and on the existing machine learning algorithms.
- Apply traditional machine learning algorithms to selected cases.
- Apply the proposed quantum machine learning algorithms to selected cases.
- Train, evaluate and compare both techniques used for a same case study.

1.2. Scopes

The following work involves the implementation of novel quantum machine learning models to be used in predictive maintenance. The results obtained from the training and evaluation of these models will be compared with classical machine learning techniques,

until finding a model or models that present competitive results with respect to the performance already achieved with classical techniques. For this, it will be necessary to know how quantum computing works, differentiating it from classical computing and how it is used to run the models.

2 | Background

In this chapter, an explanation and literature review on quantum computing, machine learning and maintenance is performed. Subsequently, the datasets used in this work are presented.

2.1. Machine Learning General Concepts

Machine learning is a branch of computer science that consists of the study and development of models that allow a system to learn, based on experience, to perform certain tasks without being explicitly programmed to do so. This learning is achieved by means of an algorithm that trains the system using a set of data from a monitoring of system variables. One of the main tasks sought using machine learning is the prediction of the classes to which a data set belongs, which in this context of predictive maintenance, can be used to classify the health states of a system.

Among the main classifications of machine learning algorithms, there is one that is based on the type of supervision that is performed on the algorithms at the time of training. We will review two of the main categories of this classification: supervised learning and unsupervised learning.

2.1.1. Supervised learning

In supervised learning, the objective is to find patterns in the data sets that are linked to attributes that define the meaning of each data. These attributes are called labels, which in general are the classes to which the data belong and which must be predicted by the model. One of the typical supervised learning tasks are classification tasks, in which it is predicted, after training, to which class the data belong. [1]

2.1.2. Unsupervised learning

In the same way as in supervised learning, in the unsupervised case, the goal is to find patterns in the dataset. However, in this case the data are not linked to labels, so this pattern search is performed through clusters of data. [1]

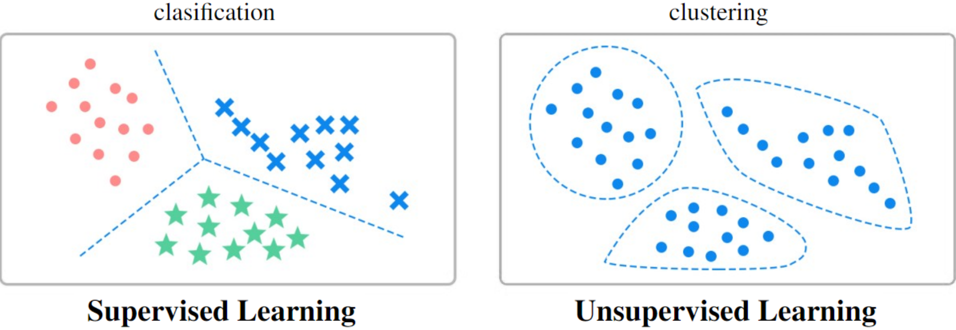


Figure 2.1.1: Example of labeled and unlabeled data for classification and clustering.

2.1.3. Overfitting

One of the most important challenges when training predictive models is to generalize the behavior of the data used. When a model loses its generality during training, it will adjust to a particular behavior found in the data on which it was trained. Because of this, when executing the task for which the model was trained, but using a different data set, poor performances will be obtained because the model fails to fit well with data that is not the training data, while when performing the same task but using the actual training data, an almost perfect performance is obtained. This phenomenon is known as overfitting. [2]

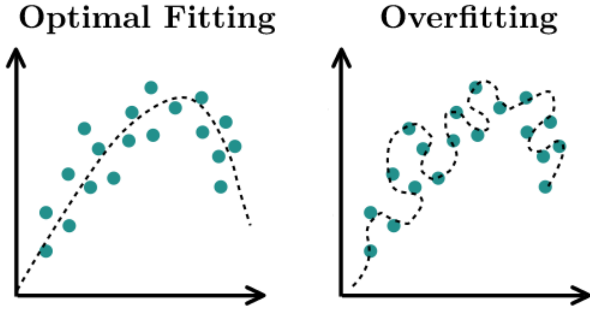


Figure 2.1.2: Overfitting example.

2.1.4. Evaluation metrics

To evaluate the performance of a model after training, different metrics are used depending on the type of task performed. Usually, for a classification model, the most commonly used metric to evaluate its performance is the so-called confusion matrix. This metric consists of a matrix that graphically shows the count of correctly and incorrectly classified data. In the figure 2.1.3, an example of a confusion matrix is presented.[3]

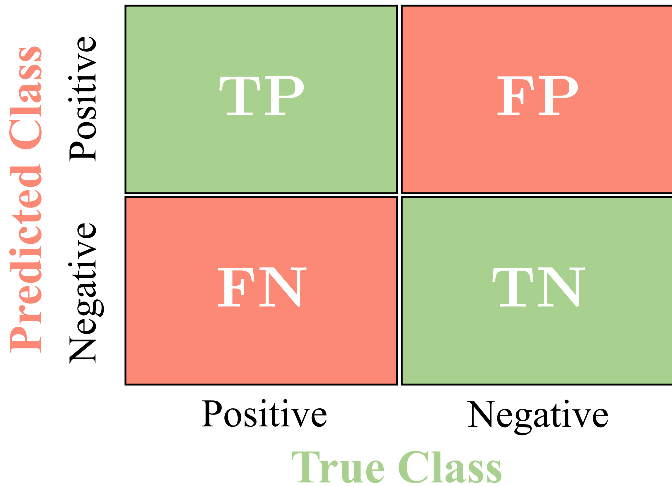


Figure 2.1.3: Confusion matrix example.

In the example matrix, there are two classes with which the data are labeled: Positive and Negative. When performing the prediction of the labels in the classification, the results are grouped into 4 subgroups defined as follows:

- **True positive (TP):** Represents the amount of data that were well classified as positive.
- **False positive (FP):** Represents the amount of data that were misclassified as positive.
- **False negative (FN):** Represents the amount of data that were misclassified as negative.
- **True negative (TN):** It represents the amount of data that were well classified as negative.

From these 4 groups, it is possible to obtain additional evaluation metrics, which are described below: [3]

Accuracy

The accuracy represents the ratio of classifications performed correctly by the model with respect to the total number of predictions made. This metric is represented by the equation 2.1 presented below.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

Expression for Accuracy calculation.

However, this measure does not reflect the presence or absence of errors in the model, such as the model predicting Negative classes well but Positive classes very poorly, which is why the following evaluation metrics emerge.

Precision

Precision is similar to Accuracy, but only considers correct predictions of the Positive class, so it represents the rate of correctly performed classifications of the Positive class with respect to all predictions that were predicted as Positive. This metric is represented by the equation 2.2 presented below.

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

Expression for calculating the Precision.

Recall

Recall, also known as Sensivity, is also similar to Accuracy, but only considers the correct predictions of the Positive class and the actual amount of data classified as positive. It therefore represents the rate of correct classifications of the Positive class relative to the actual amount of data classified as Positive. This metric is represented by the equation 2.3 presented below.

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

Expression for Recall calculation.

Specificity

Specifity is the analogous metric to Recall but for cases with Negative labels. This metric considers the correct predictions of the Negative class and the actual amount of Negative data, thus representing the rate of correctly performed classifications of the Negative class with respect to the actual amount of data classified as Negative. This metric is represented by the equation 2.4 presented below.

$$Specifity = \frac{TN}{TN + FP} \tag{2.4}$$

Expression for Specifity calculation.

Negative Predictive Value (NPV)

Finally, the last metric is the Negative Predictive Value, which is an analog to Precision but with the data classified as Negative, thus representing the rate of correctly

performed classifications of the Negative class with respect to all predictions that were predicted as Negative. This metric is represented by the equation 2.5 presented below.

$$NPV = \frac{TN}{TN + FN} \tag{2.5}$$

Expresión para el cálculo del NPV.

2.1.5. Obtaining time parameters

This is a technique used to compress the information of the cases, without losing it, into a dataset that allows a more efficient training, through the separation into time windows, the use of overlap and the calculation of new parameters.

2.1.5.1. Time windows

In general, in Machine Learning model training, the more samples used, the better the learning. However, it is necessary to be efficient when distributing the data in order to optimize execution times without losing information.

For this reason, in problems where the data to be used consist of time series, it is common to regroup them in time windows of data with a certain length. This window length is equivalent to a small amount of time compared to the total time in which the data were obtained. However, defining a very small amount of window length results in the windows not containing enough information, so it is necessary to balance the length of the windows with the amount of windows generated.

2.1.5.2. Overlap

The overlap consists of a strategy that improves training performances when using time windows. This technique consists of allowing the constructed time windows to share some data.

2.1.5.3. Temporal parameters

Temporal parameters are metrics that are calculated from each of the temporal windows defined in a particular case. This will allow to build a new dataset, with which the Machine Learning model will be trained more efficiently and without loss of information. The important metrics for this work are defined as follows.

Let X be a temporal vector of length n: $X = (x_1, x_2, ..., x_n)$, the following temporal parameters are obtained:

- **Mean** (μ): Represents a central measure of a data set obtained by dividing the sum of the values by the number of elements in the set.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- **Variance** (σ): Describes the dispersion of a set of data around its mean. The greater the variance, the greater the dispersion of the data and vice versa. In other words, the variance measures how far the data are from the mean.

$$\sigma = \frac{1}{n}\sum_{i=1}^{n}(\mu - x_i)^2$$

- **Root mean square (RMS)**: It is used to calculate the square root of the squared values mean in a data set. It is especially useful for describing electrical or mechanical signals, where negative and positive values can cancel each other out. By squaring all values, negative values are eliminated and ensures that positive values are accounted for in the average.

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

- **Peak**: It is used to describe the maximum value in a data set. It is useful to describe the maximum amplitude of a signal.

$$peak = max(X)$$

- **Peak to peak (P2P)**: It is used to describe the difference between the maximum and minimum values in a data set. It is calculated by taking the difference between the maximum value and the minimum value in the set. It is useful to understand the total amplitude of a signal, including both positive and negative values.

$$P2P = max(X) - min(X)$$

- **Crest factor**: It is used to describe the relationship between the peak value and the RMS value of a signal. It is useful to understand the waveform of a signal and to compare different signals with each other. For example, a signal with a high crest factor, i.e. a large peak to RMS ratio, has a "sharper" waveform and contains more energy in the peak values. On the other hand, a signal with a low crest factor, i.e. a small peak to RMS ratio, has a "smoother" waveform and contains less energy in the peak values.

$$crest\ factor = \frac{peak}{RMS}$$

- **Kurtosis**: It is used to describe the shape of a data distribution and the concentration of values around the mean. If the kurtosis is equal to zero, the distribution

has a typical bell shape and is said to be mesokurtic. If the kurtosis is greater than zero, the distribution has more concentration around the mean than a normal distribution, and is said to be leptokurtic or "pointed". If the kurtosis is less than zero, the distribution has less concentration around the mean than a normal distribution, and is said to be platykurtic or "flattened". [4]

$$kurtosis = \frac{\sum_{i=1}^{n}(x_i - \mu)^4}{n \cdot \sigma^2}$$

- **Skewness**: It is used to describe the skewness of a data distribution. If skewness is equal to zero, the distribution is symmetric, meaning that the mean and mode are equal and the distribution is balanced on both sides of the mean. If skewness is greater than zero, the distribution is skewed to the right, meaning that the tail of the distribution extends to the right of the mean. If the skewness is less than zero, the distribution is skewed to the left, meaning that the tail of the distribution extends to the left of the mean. Skewness is useful for understanding the shape of a distribution and can help identify outliers or extremes. [4]

$$skewness = \frac{m_3}{m_2^{3/2}} \qquad \text{where: } m_i = \frac{1}{n}\sum_{j=1}^{n}(x_j - \mu)^i$$

2.1.6. Dimensionality reduction

The dimensionality of a dataset is defined by how many values exist for each measurement or *sample*. These values coming, in this context, from continuous monitoring using sensors are called *features*. In this way, each *sample* can be understood as a vector of dimension equal to the number of *features* in the dataset.

There are cases where the available data contains a large number of features, so dimensionality reduction methods are typically used to improve the model training process by reducing execution times. For this, there are several dimensionality reduction methods such as: *Principal components analysis (PCA)*, *Linear discriminant analysis (LDA)*, among others.

The method used in this work is the *Principal Component Analysis (PCA)*, which consists of applying a series of mathematical transformations to the data to bring them to a lower dimensional space. In general terms, this method consists of the following, for a set of vectors $\{X_n\}$, $n \in \{1...N\}$ d, the q main axes w_j , $j \in \{1...q\}$ are those orthonormal axes in which the variance contained under their projection is maximum. Where, the w_j vectors are given by the q eigenvectors that have the largest eigenvalues found in the covariance matrix S.

The covariance matrix S , is given by the expression of the equation 2.6, where $\bar{t}$ is the sample mean and λ_j the respective eigenvalue.

$$S = \sum_{n} \frac{(t_n - \bar{t})(t_n - \bar{t})^T}{N} \qquad \text{such as: } Sw_j = \lambda_j w_j \tag{2.6}$$

Expression for the covariance matrix of the PCA method.

Thus, the vector $x_n = W^T(t_n - \bar{t})$ with $W = (w_1, w_2, ..., w_q)$, is the representation of the original vector t_n reduced to a space of dimension q. [5]

To define to which value of the new dimension q the data is reduced, an analysis of variances is performed. To illustrate this analysis, let us take as an example the following covariance matrix for a data set of 3 variables:

$$S = \begin{bmatrix} 1.34373 & -0.16015 & 0.18647 \\ -0.16015 & 0.61920 & -0.12668 \\ 0.18647 & -0.12668 & 1.48554 \end{bmatrix}$$

In this case, the variances of the three variables are on the diagonal, and their sum corresponds to the overall variance, which in this case, has a value of 3.448.

However, as previously mentioned, the PCA method performs a new representation of the data with a different dimension, for this, the method replaces the original variables with new variables, called principal components coming from the principal axes w_j previously mentioned. These axes, being orthogonal, do not present covariances, and their variances coming from the covariance matrix in the new space of dimension q correspond to the eigenvalues of the matrix. Therefore, if we define for this transformation, a new dimension equal to the original dimension, it is possible to perform an analysis of which dimensions provide the most information to the data. Then, considering the previous example but on these new orthogonal axes, the following covariance matrix is obtained:

$$S = \begin{bmatrix} 1.65135 & 0.00000 & 0.00000 \\ 0.00000 & 1.22028 & 0.00000 \\ 0.00000 & 0.00000 & 0.57684 \end{bmatrix}$$

The sum of this last matrix diagonal is also 3.448, however, it is now possible to calculate the importance of each dimension. For this, the percentages of each variance are calculated:

- First dimension: $1.651/3.448 = 47.9\%$
- Second dimension: $1.220/3.448 = 35.4\%$
- Third dimension: $0.577/3.448 = 16.7\%$

This allows to decide how many dimensions to reduce to. For example, if in this case it is decided to reduce the dimension of the data to 2, 16.7% of variance would be eliminated, which would translate into a loss of information by reducing the dimensionality. However, there are cases where the percentage of variance of a dimension is negligible compared to the rest, so it is easier to decide to reduce the dimensionality

without losing information. In general, and in this work, it is decided to keep at least 95% of the variance.

2.1.7. One hot encoding

Sometimes, especially in classification tasks, the data to be worked with contains labels that are not numeric values, but strings of characters such as a name or a characteristic. For example, an attribute such as color may have values such as ‘red’, ‘blue’ or ‘green’.

However, some algorithms cannot work with category data directly, so it is necessary to translate them into numerical values. For this purpose, method *One-hot encoding* is used.

This method performs a representation of each attribute of a category using new variables using only binary values. For example, for the case of the color attribute mentioned above, 3 binary variables are needed, which will take the value 1 to indicate the color value of the data and the value 0 to indicate that the data does not meet that characteristic, as exemplified in table 2.1.

Table 2.1: One-hot encoding example

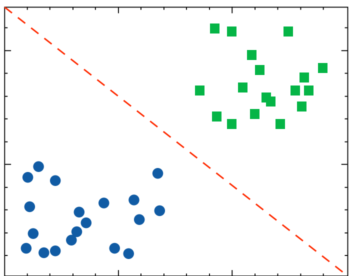
	red	blue	green
Data classified as color red	(1 , 0 , 0)		
Data classified as color blue	(0 , 1 , 0)		
Data classified as color green	(0 , 0 , 1)		

2.1.8. Support Vector Classifier

One of the most popular classification algorithms is the Support Vector Classifier (SVC). This algorithm belongs to a set of supervised learning methods for classification, regression and outlier detection tasks, called Support Vector Machines (SVMs).

Support Vector Machines correspond to algorithms that calculate the optimal hyperplane capable of separating the classes of data in their space. Examples of hyperplanes in different dimensions are as illustrated in figure 2.1.4.

A hyperplane in 2D **A hyperplane in 3D**



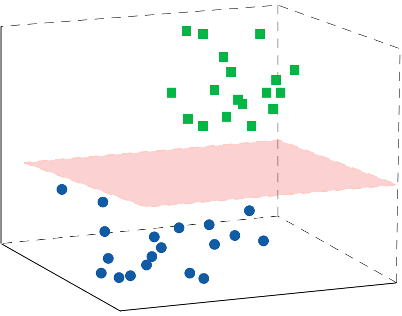


Figure 2.1.4: Hyperplane illustrations.

There are infinite hyperplanes in the n-dimensional space of the data, so there can also be infinite hyperplanes that divide the data into their respective classes. For this reason, the optimal hyperplane is defined as the one that maximizes the margins, that is, the distance between the hyperplane and the data of each class, as illustrated in Figure 2.1.5.

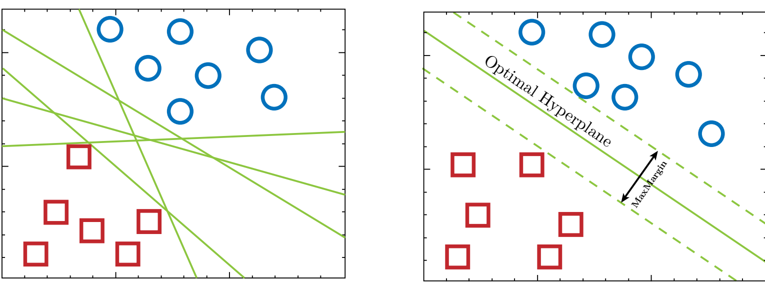


Figure 2.1.5: Optimal hyperplane example.

The search for optimal margins is related, by means of an optimization problem, to different parameters. In general, these parameters depend on the type of algorithm, however, the most usual ones are:

- **Regularization parameter (C)** : Controls the balance between the training data. In general, a high value of the regularization parameter would mean a smaller margin for the hyperplane.
- **Kernel** : It corresponds to a function $K(x)$ used to perform a dimensionality increase. This increase is performed because in some cases, it is not possible to find hyperplanes that separate the classes of the data in their original dimension, so they are moved to a higher dimensional space in which it is possible to find the hyperplanes, as illustrated in figure 2.1.6.
- **Kernel parameters** : They correspond to the parameters of the function $K(x)$ used as Kernel, which must be adjusted to find the optimal hyperplane.

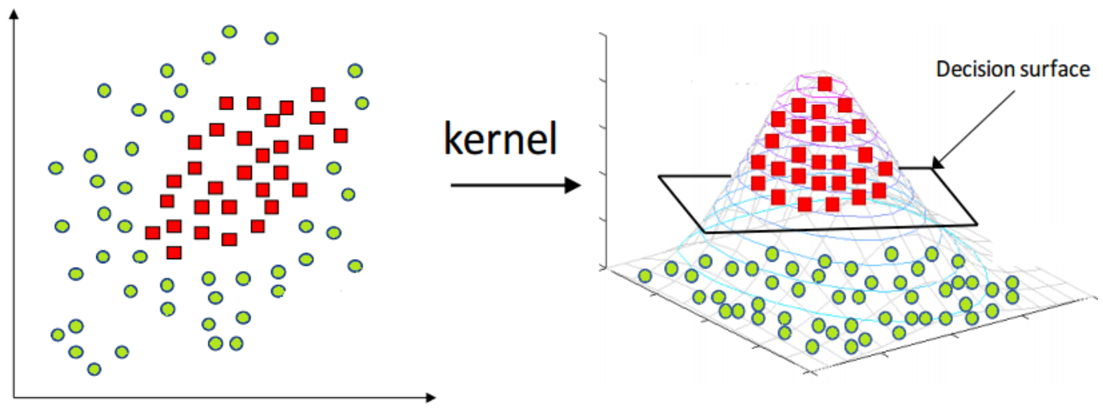


Figure 2.1.6: Kernel Example.

As for the most commonly used Kernel functions, there are some such as the Linear Kernel, the Polynomial Kernel or the Radial basis function Kernel (RBF). In this particular case, the RBF Kernel of equation 2.7 is used. [6]

$$K(x_1, x_2) = exp(-\gamma \left\| x_1 - x_2 \right\|^2) \qquad \text{where: } \gamma = \frac{1}{2\sigma^2} \qquad \text{and } \sigma \text{ as a free parameter} \tag{2.7}$$

RBF Kernel.

Among the SVMs algorithms, there is the one that is important for this work, the Support Vector Classifier. This classification algorithm has different variations, but in this context, the "C-Support Vector Classifier" is used.

In the "C-Support Vector Classifier" algorithm, given a training vector $x_i \in R^n$ with $i = 1, ..., l$ in two data classes and a vector indicator $y \in R^l$ such that $y_i \in \{1, -1\}$, the following optimization problem is solved:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2} w^T w + C \sum_{i=1}^{l} \xi_i \\ \text{subject to} \quad & y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i, \\ \text{with} \quad & \xi_i \geq 0, i = 1, ..., l \end{aligned} \tag{2.8}$$

C-Support Vector Classsifier Primal Optimization Problem.

In this case, $\phi(x_i)$ represents a function that maps the values of the training vector x_i to a higher dimensional space and the regularization parameter C is positive. However, due to the possible higher dimension of the variable vector w, the following case corresponding to the dual problem of the previous optimization problem is usually solved:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T Q \alpha - e^T \alpha \\ \text{subject to} \quad & y^T \alpha = 0, \\ \text{with} \quad & 0 \geq \alpha_i \geq 0, i = 1, ..., l \end{aligned} \tag{2.9}$$

C-Support Vector Classsifier Dual Optimization Problem.

In this new optimization problem, $e = [1, ..., 1]^T$ is a vector of ones, and Q is a semidefinite positive matrix such that $Q_{ij} = y_i y_j K(x_i, x_j)$, where $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the Kernel function mentioned above.

By finding the solution of the dual problem of equation 2.9, using the relation between the primal and dual problem, it is obtained that the variable vector w satisfies the relation of equation 2.10.

$$w = \sum_{i=1}^{l} y_i \alpha_i \phi(x_i) \tag{2.10}$$

C-Support Vector Classsifier Optimal w vector.

And the decision function of the problem is the one of the equation 2.11. Thus, finally, the values of $y_i\alpha$ $\forall i$, the values of b constants, label names, support vectors, and kernel parameters were stored. [7]

$$sgn(w^T \phi(x) + b) = sgn\left(\sum_{i=1}^{l} y_i \alpha_i K(x_i, x) + b\right) \tag{2.11}$$

C-Support Vector Decision Function.

2.1.9. Neural Networks

Neural Networks, also known as Artifitial Neural Networks, are a computational model whose architecture is based on units called artificial neurons. These units receive their name because of their similarity to their biological counterpart, due to the way in which neurons are connected to each other.

These models, like those previously mentioned, rely on the use of training data to learn and improve their accuracy over time. When these learning algorithms are fine-tuned for accuracy, they are powerful tools that allow us to perform tasks such as data classification.

The structure of an artificial neuron, as illustrated in figure 2.1.7, takes as input x_n parameters that can come from initial data or data from other neurons. Then, each input parameter has its corresponding weight w_{nj} by which it is multiplied and then summed using a transfer function. Finally, the output of the neuron consists of

the result of applying an activation function to the sum obtained from the transfer function.

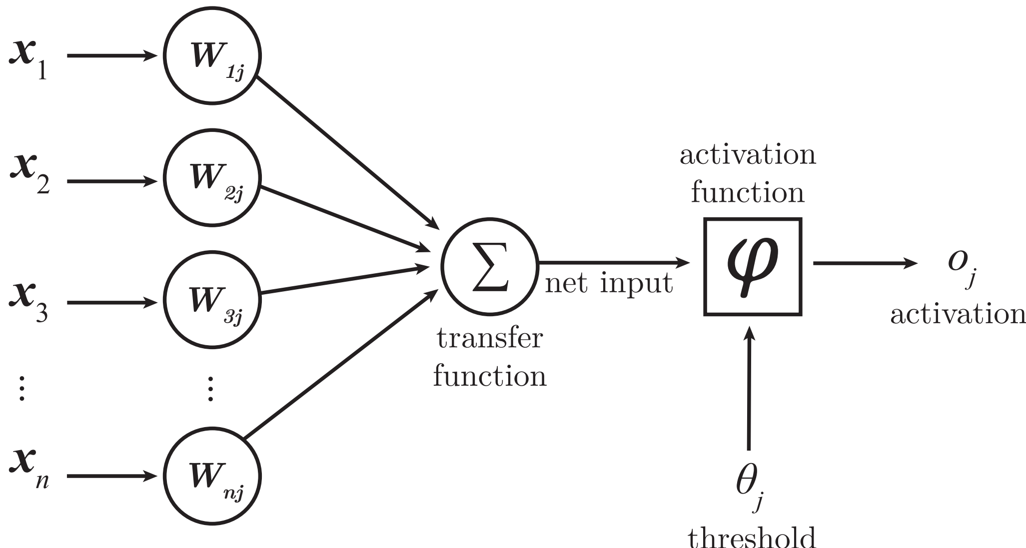


Figure 2.1.7: Artificial Neuron Scheme.

A neural network bases its architecture on a collection of connected artificial neurons, which are commonly referred as the nodes of the neural network. This collection of artificial neurons consists of a series of layers with nodes, within which there is an input layer, one or more hidden layers and an output layer, as illustrated in Figure 2.1.8 . Each node of the network layers is connected to other nodes of its contiguous layers, and depending on its weight and threshold it is determined whether it passes information to the next layer of the network.

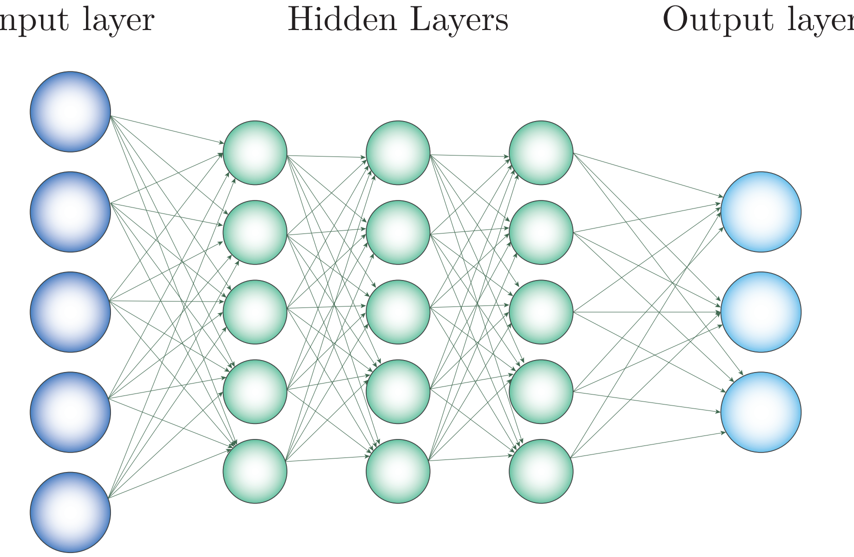


Figure 2.1.8: Artificial Neural Network Scheme.

Each of the previously mentioned layers has a defined number of neurons, and are named according to their position inside the network. In detail, each layer has its own function within the network:

- **Input Layer:** Being the first one, it corresponds to the features of the dataset data, therefore, if there are k features in the data, this layer would have k dimensions.
- **Hidden Layer:** These are the layers that follow the input layers, and are made up of N_i neurons, where N_i is a hyperparameter. In these layers each neuron has its own weights, threshold, and activation function, generating an output for each neuron. The result of this layer consists of a vector with each neuron output of dimension N_i.
- **Output Layer:** It corresponds to the last layer, which takes the output of the last hidden layer and generates the final result of the neural network, again using neurons. The dimensions of this layer match the dimensions of the result being sought. For example, if N different classes of data are being classified, this layer will have N neurons, one for each class.

The activation function used in the network consists of a nonlinear function that transforms the output of each neuron to a range of defined values, generating the output of each neuron. Some of the commonly used functions are the following, illustrated in the graphs in Figure 2.1.9:

- **ReLU (Rectified Linear Unit):** $f(u) = max(0, u)$, where $f : R \rightarrow R_0^+$.
- **Sigmoid:** $f(u) = \frac{1}{1+\exp(-u)}$, where $f : R \rightarrow [0, 1]$
- **Tanh:** $f(u) = tanh(u)$, where $f : R \rightarrow [-1, 1]$

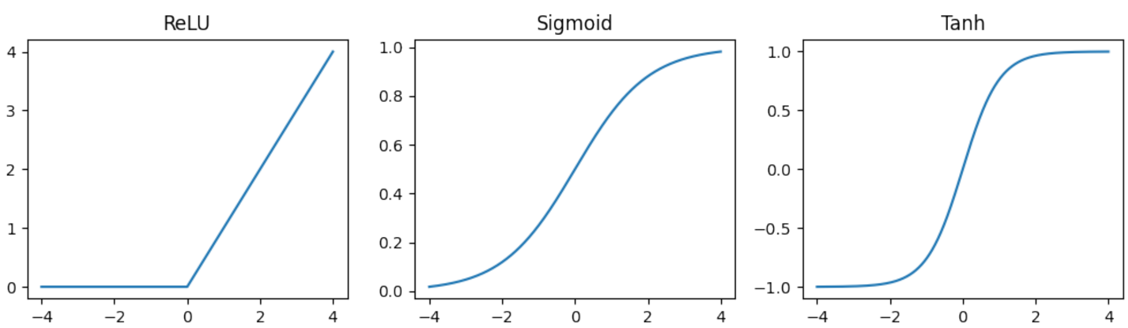


Figure 2.1.9: Activation Functions.

In addition to layers, neurons and activation functions, neural networks have a number of additional hyperparameters. These hyperparameters are fixed variables or conditions used during neural network training, among which are:

- **Epochs:** Corresponds to the number of training cycles, in other words, it is the number of times the model is trained with the training data.
- **Initialization weights:** They correspond to the initial weight values of the net.
- **Error function and Optmizer:** It corresponds to the algorithm for updating the weights and thresholds, and is the one that allows the neural networks to be trained. The choice of one optimizer or another brings with it the choice of hyperparameters specific to each optimizer. But a common hyperparameter among them corresponds to the Learning Rate.
- **Learning Rate:** Indicates how much influence each update of the weights and threshold has. Thus, if this value is too large, the next update will end up modifying the values too much, which may prevent the model from converging. Some optimizers modify this value as they train in order to converge faster.

2.2. Classical computing

In order to understand how quantum computing works, it is necessary to go back to its classical counterpart and then make the analogy. First, it is necessary to understand how a classical processor works.

Everything mentioned above in the Machine Learning section works based on its most fundamental unit: the binary number system. This is because all current classical electronics is based on this binary system. This numeric system allows to represent any integer by two symbols (0 and 1), analogously to how in the decimal system ten symbols (from 0 to 9) are used to represent any integer.

The binary system turns out to be the best way to represent quantities with a single signal in a computer, because it is used to do the following sort of "translation":

- If there is current in an electrical circuit, it is interpreted as a 1.
- If there is no current in an electrical circuit, it is interpreted as a 0.

These electrical circuits are typically conductors. For example, with 5 conductors, using this "translation" it is possible to count from 0 to 15 because with 5 binary quantities it is possible to represent up to the binary number 10000 which in decimal is the number 15.

Once this form of numerical representation is achieved through electrical circuits, it is possible to perform arithmetic operations. For this purpose, special components called logic gates are used, which are based on Boolean algebra and mathematical logic. Some of the best known logic gates are shown in the table of Figure 2.2.1.

Gate Name	NOT	AND	OR	NAND	NOR	XOR	XNOR
Algebraic Expressions	$C = \overline{A}$	$C = A \cdot B$	$C = A + B$	$C = \overline{A \cdot B}$	$C = \overline{A + B}$	$C = A \oplus B$	$C = \overline{A \oplus B}$
Gate Symbol							

Truth Table

NOT Input	NOT Output	AND Input		AND Output	OR Input		OR Output	NAND Input		NAND Output	NOR Input		NOR Output	XOR Input		XOR Output	XNOR Input		XNOR Output
A	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1
		1	0	0	1	0	1	1	0	1	1	0	0	1	0	1	1	0	0
1	0	0	1	0	0	1	1	0	1	1	0	1	0	0	1	1	0	1	0
		1	1	1	1	1	1	1	1	0	1	1	0	1	1	0	1	1	1

Figure 2.2.1: Classical Logic Gates.

In order to put this into practice, for example to perform the *AND*operation, at a very beginning a component called *Relay*, then it was passed to the *vacuum valves*, and nowadays the *transistores*are used. There are many types of transistors, however for a better understanding, it will be explained using the one that works as a "bridge or gate", represented in the illustration of the figure 2.2.2.

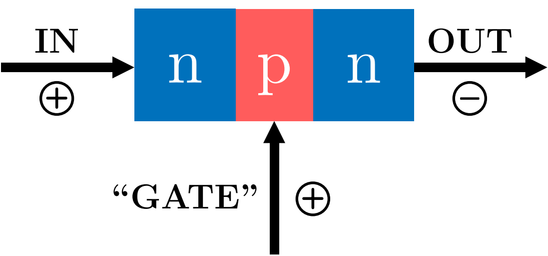


Figure 2.2.2: Schematic diagram of a classic transistor.

If the gate of the transistor is supplied with electricity, it opens the way for the input current to flow to the output of the transistor. In this way, by performing different combinations, it is possible to perform classical logic operations. For example, in the figure 2.2.3 shows how by means of two consecutive transistors it is possible to represent the AND operation. Also, the figure 2.2.4 represent the OR operation using also two transistors.

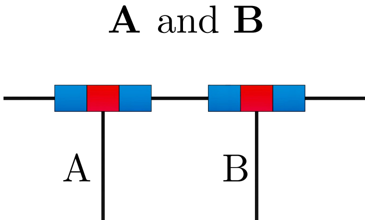


Figure 2.2.3: Scheme of AND operation with transistors.

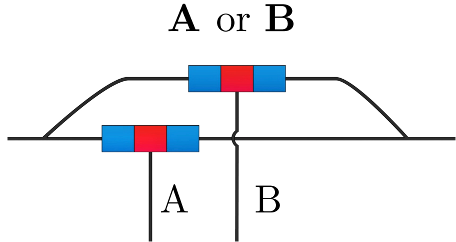


Figure 2.2.4: Scheme of OR operation with transistors.

By combining these operations, it is possible to construct other operations by increasing their complexity, as is the case of the XOR operation illustrated in Figure 2.2.5.

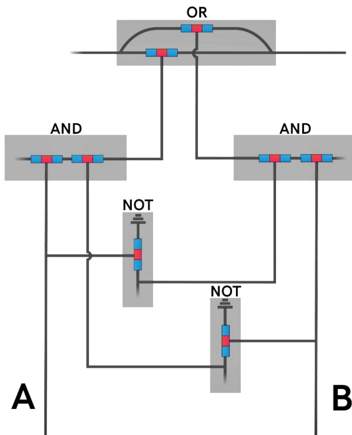


Figure 2.2.5: Scheme of XOR operation with transistors.

Having understood the above, instead of using illustrative representations of the transistors, the symbols of each logic gate tabulated in the table in figure 2.2.1 will be used to simplify the schematics. In this way, it is possible to explain how arithmetic operations are performed by these transistors.

For this purpose, it is illustrated below in figure 2.2.6 how by combining two logic gates in one circuit, it is possible to perform an addition. This circuit is known as *Half adder*.

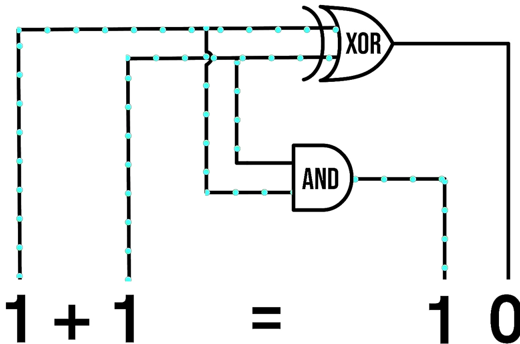


Figure 2.2.6: Half Adder circuit diagram.

As in the case of the *Half adder*, it is possible to make more combinations in this way with more logic gates using more conductors, for example, to sum up to 255 using 8 conductors.

With these logic gates, it is also possible to achieve other operations such as subtraction, negation, increment, decrement, along with others that are not achieved by themselves, but are combinations of the previous operations, thus achieving multiplication and division.

Logic gates and their operations are fundamental and are performed in computer processors, in a special package called *ALU (Arithmetic Logic Unit)*, which contains a large number of logic gates such as the one illustrated in the figure 2.2.7.

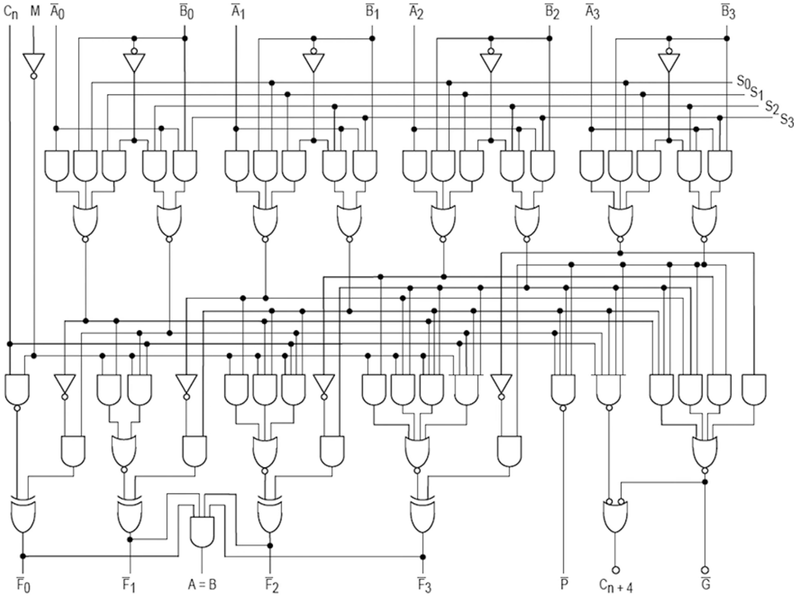


Figure 2.2.7: Schematic diagram of the ALU TI 7400.

In addition, in the processors there are other units such as the FPU (Floating Point Unit), which is used to perform arithmetic calculations with decimals, using signals that

indicate the digits and others that indicate the place of the decimal, or also memory units that store the input and output values of the operations.

However, not only numbers can be represented in classical computing using the binary system, but also other elements such as colors, images, text, among others.

2.3. Quantum computing

Quantum computing provides a new form of computing that differs from classical computing and represents a new paradigm for computing. It is based on fundamental principles of quantum mechanics such as superposition, quantum interference and quantum entanglement. To understand these principles, it is necessary to review some basic concepts of quantum computing.

2.3.1. Background

2.3.1.1. Quantum Bits

Classically, traditional computing works with traditional bits that represent the state of a system with two possible states, 0 and 1. In quantum computing there are also these bits, called *Qubits*, and unlike the classical conception, in the quantum case the bits are not restricted to be in each of these states, but can even be in both states at the same time. This behavior is illustrated in Figure 2.3.1.

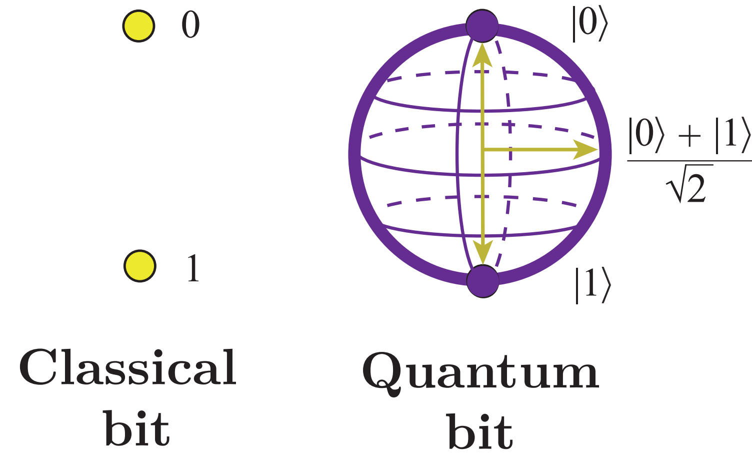


Figure 2.3.1: Comparision between a classical bit and a quantum bit.

Mathematically a qubit is described, as shown in the equation 2.12, by a 2 by 1 matrix, with complex numbers in its components. This matrix in quantum mechanics is represented using dirac notation and is called *ket*. This ket describes rather the state of the qubit and the probabilities with which the quantum bit could be found in each

state. [8]

$$|\psi\rangle = [c_0, c_1]^T \ , \ \text{donde:} \ \ |c_0|^2 + |c_1|^2 = 1 \tag{2.12}$$

Ket representation for a quantum state.

However, also a set of qubits can form a quantum system. The possible states of the system are given by the tensor product that represents all possible combinations of the individual states, so this time the ket describing the possible states in this new system are represented as in equation 2.13. [8]

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes ... \otimes |\psi_N\rangle \tag{2.13}$$

Ket representation for a combination of quantum states.

2.3.1.2. Quantum superposition

The quantum state of a system is explained by one of the main principles of quantum mechanics: The superposition principle. This principle postulates that a particle (or a system) with possible states represented by elements of vectors in the space $|\psi_n\rangle$, can also exist in a complex linear combination of these possible states, represented in the equation 2.14 [9]

$$|\psi\rangle = \sum_{i}^{N} \alpha_i \cdot |\psi_i\rangle \tag{2.14}$$

Possible states of a combination.

However, this alone is not so special, since a classical computer configured in such a way that its bits can take values between 0 and 1 is an ordinary analog computer, which is not efficient and is barely more powerful than an ordinary computer. This is why the power of quantum computing is based in part on taking advantage of a particular type of superposition that allows exponentially many logical states at once. [10]

This means that a qubit system can exist in multiple states at the same time through this complex linear combination, coming from this particular type of superposition, whose possible states are determined by the combinations given by Equation 2.13. This marks the first fundamental difference with classical computing, since this principle is the one that allows, as mentioned above, that quantum bits can represent the values 0 and 1, or linear combinations of both, and these linear combinations are known as **superposition states**. [11]

2.3.1.3. Observer effect

On the other hand, one of the most important phenomena to understand the behavior of qubits is the **observer effect**, to understand this, it is necessary to dig deeper into

the quantum states and their probabilities. This phenomenon implies that, unless its value is observed, a qubit is in a state of superposition of 0 and 1, but once its value is observed, it is obtained that the qubit is in state 0 or state 1. The probabilities that a qubit turns out to be in one value or another are not necessarily 50:50, but can have any probability distribution, and this probability distribution that a qubit has when it is observed depends on its quantum state. [9]

As previously illustrated in equation 2.12, one way to represent quantum state vectors is by means of the *ket* of the Dirac notation, for example, the 0 state is represented as $|0\rangle = [1, 0]^T$, and state 1 is represented as $|1\rangle = [0, 1]^T$. It is possible to perform a linear combination of these two quantum states, respecting the restriction present in the equation 2.12, which means that the amplitudes of these vectors are proportional to the probabilities, and the sum of their squares must represent the probability of 100%. To do this it is necessary to add weights to the quantum states as in equation 2.15.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} 1 \cdot \alpha + 0 \cdot \beta \\ 0 \cdot \alpha + 1 \cdot \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{2.15}$$

Superposition state equation with weights.

These coefficients must satisfy that $\alpha^2 + \beta^2 = 1$. So, for example, if a state is desired in which both coefficients are equal, the coefficients must be equal to $\frac{1}{\sqrt{2}}$, which would give a state like the one in equation 2.16. That a qubit is in this superposition state means that when measuring it, it is possible to find it in both state $|0\rangle$ and state $|1\rangle$ with 50% probability.

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \begin{bmatrix} 1 \cdot \frac{1}{\sqrt{2}} + 0 \cdot \frac{1}{\sqrt{2}} \\ 0 \cdot \frac{1}{\sqrt{2}} + 1 \cdot \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.16}$$

Superposition state equation with equal weigths.

Now, if you want a qubit to be in a superposition state that represents, for example, a 25% probability of being in the $|0\rangle$ state and 75% of being in the $|1\rangle$ state, it has to be satisfied that $\alpha^2 + \beta^2 = 1$ with $\alpha^2 = 1/4$, obtaining that α should be $1/2$, and β should be $\sqrt{3}/2$. In this case, the state is the one illustrated in equation 2.17.

$$|\psi\rangle = 1/2|0\rangle + \sqrt{3}/2|1\rangle = \begin{bmatrix} 1/2 \\ \sqrt{3}/2 \end{bmatrix} \tag{2.17}$$

Superposition state equation with different weigths.

Until now, the notion of quantum state has been explained for a binary quantum system, i.e., of a single qubit. Until we observe a qubit, it will be in a superposition state, but once it is observed, there are different probabilities of measuring 0 or 1. This means that when multiple measurements are made on multiple binary systems of a single qubit in identical states, they will not deliver the same result, instead, when observed, it has α^2 probability of delivering 0 and β^2 probability of delivering 1. This is the phenomenon known as the observer effect, since the mere observation of a

phenomenon changes the observed phenomenon, in this case, the observation of a qubit changes its state from a superposition of $|0\rangle$ and $|1\rangle$ states to one or the other of the $|0\rangle$ and $|1\rangle$ states.

For example, if a qubit is in the state represented in the equation 2.17, observing or measuring it will push the qubit out of its superposition state and collapse it with probability 25% to the state $|0\rangle$ and 75% to the state $|1\rangle$. Once observed, and consequently collapsed into one of the states, a new observation or measurement will now cause the qubit to be found with a probability of 100% in one of the two states. [3]

2.3.1.4. Quantum interference

On the other hand, one of the principles of quantum mechanics used in this context is the quantum interference principle. To understand how this principle is used in this context, it is useful to study the classical analog: noise cancellation.

Noise cancellation is accomplished by employing the superposition and interference principle to reduce the amplitude of the unwanted noise, generating a signal of approximately the same frequency and amplitude as the noise but offset by π or an odd multiple of π, as shown in the illustration in Figure 2.3.2. This superposition results in interference and an output that significantly reduces the noise with respect to the original signal. [11]

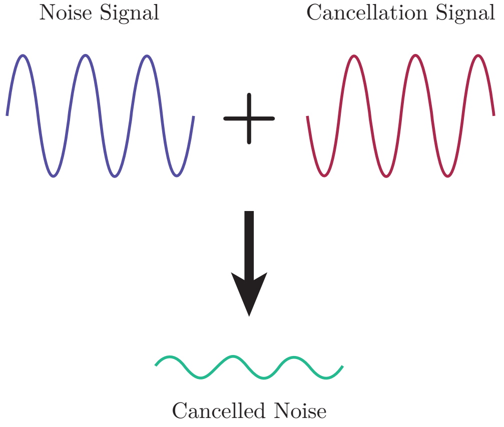


Figure 2.3.2: Classical noise cancel example.

Although this processing is performed by digital circuits, amplitude and phase are continuous variables that can never match perfectly in these circuits, so the noise of the original signal is not completely canceled. In the case of a quantum computer, the processing is performed in a very similar way, however, this time it deals with quantum states and not with classical signals such as sound.

In the interference used in quantum computing, a superposition of all possible computational states is prepared, as illustrated in Figure 2.3.3. This superposition is used

as the input of a **quantum circuit**, which will be explained in later sections, that selectively interferes the components of the superposition according to an algorithm, canceling unwanted amplitudes and phases to obtain the desired solution as the output of the quantum circuit. [11]

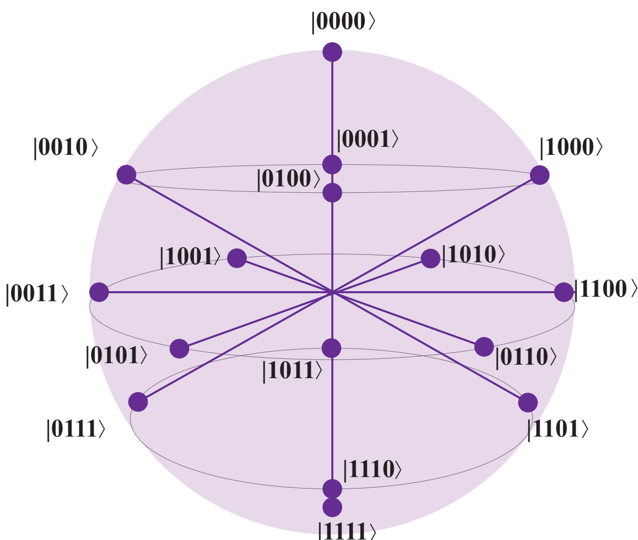


Figure 2.3.3: Multiple states superposition.

In summary, quantum interference is what allows to bias a quantum system, directing it toward desired states by destructive interference patterns to eliminate states that lead to incorrect responses and to reinforce those that take the desired states. [3]

2.3.1.5. Quantum entanglement

Another principle of quantum mechanics present in quantum computing is quantum entanglement. This principle results in a strong correlation between quantum particles, remaining perfectly correlated even when they are separated by large distances. More in detail, the possible states in a system of quantum particles will be a combination of the individual states of each particle, however these are not independent from each other, i.e. the state of a particle belonging to the system is not defined by itself but is entangled with the states of the rest of the particles in the system. [12]

In the context of quantum computing, this principle refers to those states of a system of qubits, in which the combined state of the qubits contains more information than the individual states of each qubit. Most of the states of a qubit, superposed between 0 and 1, contained in a set of several qubits, are entangled. These entangled states are the most valuable and useful superpositions for quantum computation. [10]

In a quantum computer, the entangled states are states of the computer as a whole, which do not coincide with any “analog” or “digital” state of the individual qubits. This is why a quantum computer is significantly more powerful than any other classical

computer, since, for example, the entangled states between qubits can be used for the phenomenon of "quantum teleportation", where an entangled state shared between two qubits can be manipulated to transfer information from one qubit to another, regardless of the physical proximity of the qubits. [11]

2.3.1.6. Quantum Circuits

Analogous to the case of classical computing seen previously, quantum computing is built on the basis of quantum operations with qubits combined in a circuit. In this case, these are known as quantum circuits.

A quantum circuit consists of a computational routine with quantum operations on quantum data contained in qubits, combined with classical operations to perform measurements and preprocessing of the classical data before encoding them into quantum bits. An example of a quantum circuit is illustrated in figure 2.3.4. [11]

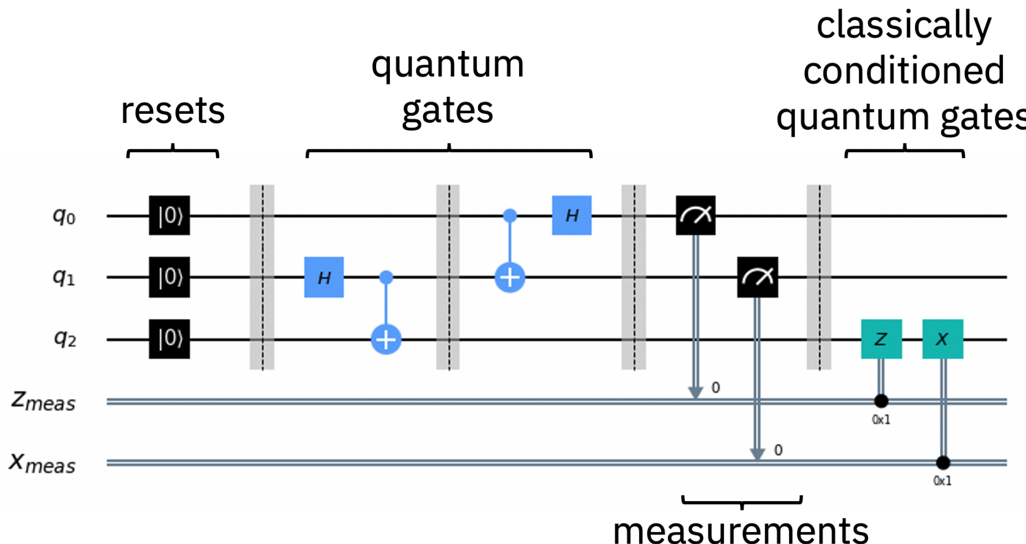


Figure 2.3.4: Quantum Circuit Example. [11]

Each of the horizontal lines represented in the illustration of the figure 2.3.4, represents a qubit, being the left end of the circuit, the initial quantum data in each qubit, and the right end, the quantum outputs generated by the calculations performed by the circuit. The operations performed on the qubits are represented by boxes and they can be qubits measurements or quantum operations which are called **quantum gates**.

In synthesis, a quantum circuit allows a quantum computer to take classical information, encode it into quantum information, take advantage of principles such as quantum interference, quantum entanglement or the superposition principle, and then translate the quantum result into a classical solution that solves the given problem.

2.3.1.7. Quantum Gates

It follows from the section on quantum circuits, the concept of Quantum Gates. This concept is analogous to the classical logic gates seen in the classical computing section and consists of primitive logic operations on quantum data, which represent reversible transformations that preserve the information stored in the qubits. [11]

These transformations represent the core of a quantum circuit, and consequently, of all quantum computing. To understand why it is so fundamental, it is important to understand what was mentioned in previous sections: Classical computers work by sending pulses of electricity through a circuit, if an electric pulse is received in a given time, it is interpreted as True, and if the pulse is not received it is interpreted as False, or rather 1 and 0 using the binary system.

All computation is based on this circuit operation, including quantum computation in a more complex and sophisticated way with quantum circuits and in a non-binary way, to make the step towards the quantum case, it is essential to keep in mind the expressions of the equations 2.15 and 2.12 on a quantum state $|psi\rangle$.

While the state of a classical bit is a boolean, i.e., 0 or 1 for False or True, in the quantum case, the state of a quantum bit is the superposition of $|1\rangle$ and $|0\rangle$ quantum states pondered by weights α and β. In that superposition state, the quantum system is neither 0 nor 1 unless measured. Only when the qubit is measured does the state collapse to a 0 or 1 state. The square of the α and β weights denote the probabilities of measuring 0 or 1. To translate this into computational information, it is necessary to translate it into two-component arrays, which are much more complex numerical values than boolean or binary data of 0 and 1. This is why in the classical case, it is possible to use simple logical operations on Boolean values such as the operations *not* or *or*, however, it is not possible to do it in such a simple way in the quantum case with values that represent probability distributions within a vector.

To illustrate the impossibility of performing these operations in a simple way in the quantum case, the state of equation 2.17 is examined. In this state, the Qubit has a 25% probability of being in the $|0\rangle$ state and a 75% probability of being in the $|1\rangle$ state. The vector representing this superposition state is [$1/2$, $\sqrt{3}/2$], and to invert its components, it is necessary to multiply it by a counterclockwise rotation matrix, as indicated in the equation 2.18.

$$R(90^o) \cdot \begin{bmatrix} 1/2 \\ \sqrt{3}/2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1/2 \\ \sqrt{3}/2 \end{bmatrix} = \begin{bmatrix} \sqrt{3}/2 \\ 1/2 \end{bmatrix} \tag{2.18}$$

State Vector Rotation Example

It is possible to perform this exercise with an infinite number of superposition states with different probability distributions, but it is impossible to list all the results in a truth table as in classical logical operations. However, from this example, it is possible to understand the behavior of one of the most fundamental quantum gates in quantum

computing, the **X-Gate**. [3]

Just as classical computation has a set of Boolean operators, quantum computation also has a set of operators, which in this case are the aforementioned Quantum Gates. Some of the fundamental Quantum Gates are the following:

2.3.1.7.1. Bit Flip Pauli Gate (X-Gate)

As previously mentioned, the X-Gate quantum logic gate is a fundamental operator in quantum computing that performs the function of inverting the state of a quantum bit. This gate operates on an individual qubit and changes the state $|0\rangle$ to $|1\rangle$ and vice versa. The representative matrix of the X-Gate is a unitary matrix of dimension 2x2, which is used to transform the state vector of the qubit. The action of the X-Gate, represented in the equation 2.19, on the quantum state is a unitary operation, which means that it maintains the norm of the state vector and thus preserves the total probability of the system. The X-Gate is an essential tool in the construction of quantum algorithms and in the implementation of logical operations in quantum computing. [3]

$$X \cdot |\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 0 \cdot c_0 + 1 \cdot c_1 \\ 1 \cdot c_0 + 0 \cdot c_1 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_0 \end{bmatrix} \tag{2.19}$$

X-Gate over a state.

2.3.1.7.2. Hadamard Gate (H-Gate)

The H-Gate quantum logic gate, also known as the Hadamard gate, is a fundamental operation in quantum computing that has the ability to create superpositions of quantum states. This gate applies a unitary transformation to a qubit, taking it from a $|0\rangle$ or $|1\rangle$ ground state to a superposition state. The matrix representing the H-Gate comes from an outer product between the $|+\rangle$ and $|-\rangle$ states, represented in the expression of the equation 2.20, with the basis state vectors.

$$|+\rangle = \frac{|0> + |1>}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \quad , \quad |-\rangle = \frac{|0> - |1>}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.20}$$

Superposition states $|+\rangle$ and $|-\rangle$.

The $|+\rangle$ and $|-\rangle$ states are quantum superposition states and are represented as column vectors in the basis system of quantum states. The $|+\rangle$ state is a balanced superposition of the $|0\rangle$ and $|1\rangle$ basis states while the $|-\rangle$ state is an unbalanced superposition of the basis states. These states are useful in quantum computation to perform quantum operations and to describe the behavior of quantum systems in general.

$$H = |+\rangle\langle 0| + |-\rangle\langle 1| = \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \text{ , where: } |a\rangle\langle b| = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \cdot \begin{bmatrix} b_0 \; b_1 \; ... \; b_n \end{bmatrix} \tag{2.21}$$

H-Gate Matrix.

Applying the matrix resultant from the outer product of the superposition states $|+\rangle$ and $|-\rangle$, represented in the equation 2.21, on a basis state vector results in a superposition state with equal probabilities of finding the qubit in either of the two basis states, as represented in the equation 2.22. The H-Gate is widely used in quantum algorithms for the creation of superpositions and the implementation of quantum search and classification algorithms. [3]

$$H \cdot |0\rangle = \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \tag{2.22}$$

H-Gate over the state $|0\rangle$.

2.3.1.7.3. Rotation Gates

Another of the most important quantum gates are the rotation gates. These operations are useful to change the probabilities determined by the quantum state in which the Qubits or quantum systems are located.

From the equation 2.15, it is possible to specify the amplitude of the qubit's probabilities during its initialization. In order to get the desired probability, the qubit must be initialized in a state that has amplitudes equal to the square roots of the probabilities sought. outside the initialization of the qubit

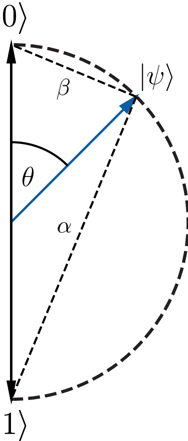


Figure 2.3.5: Illustration of an angle representation for a Quantum State.

To control the probability distributions of a state at qubit initialization, rather than specifying the exact probabilities, it is possible to determine the probabilities by means of an angle θ. This value represents the angle between the base vector of state $|0\rangle$ and the state of the qubit $|\psi\rangle$ on the Bloch sphere, as illustrated in Figure 2.3.5. This angle controls the proximity of the top of the vector to the top or bottom of the system, and these proximities represent the amplitudes of the probabilities whose squares are the probabilities of measuring 0 or 1 respectively. Thus, it is possible to derive the values of α and β and even the state $|\psi\rangle$ through the expression of the equation 2.23

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix} \tag{2.23}$$

Angle representation for a Quantum State.

It is possible to use this interpretation to rotate the state of the qubit and consequently change its probability distribution. Considering that θ is the angle between the state $|0\rangle$ and the state of the qubit $|psi\rangle$, a rotation of the vector $|0\rangle$ at an angle θ transforms it into $|psi\rangle$. Then, the operation $|\psi\rangle\langle 0|$ denotes this part of the transformation. On the other hand, the state of the Qubit labeled as $|\psi'\rangle$ represents the rotation of the state $|1\rangle$ at angle [illegible] econd part of the transformation, [illegible]

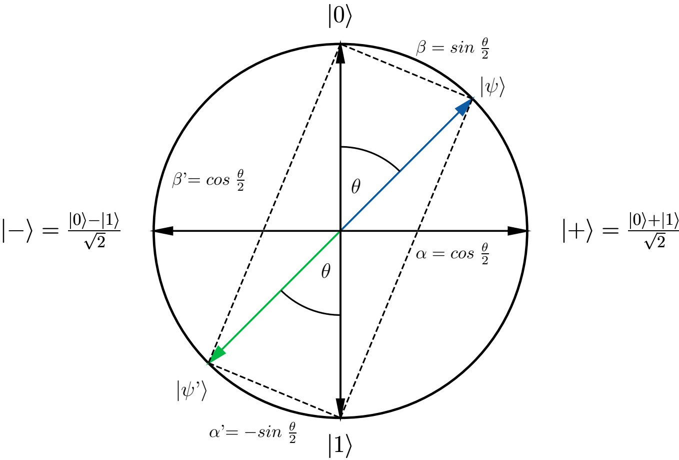


Figure 2.3.6: Illustration of states rotations on Blech Sphere.

With these two transformations, is constructed the matrix that represents the rotation operation applicable to a qubit, which is represented in the equation 2.24 and it is one of the matrices used in the so-called Rotation Gates.

$$R_y = |\psi\rangle\langle 0| + |\psi'\rangle\langle 1| = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} -sin\frac{\theta}{2} \\ cos\frac{\theta}{2} \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} cos\frac{\theta}{2} & -sin\frac{\theta}{2} \\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix} \tag{2.24}$$

R_y Rotation Gate Matrix.

Since this transformation rotates the Qubit around the y axis of the quantum system, this function takes the θ angle in radians as the first parameter, where the value of 2π denotes a rotation of 360°. The second parameter of this function is the position of the Qubit to which the gate will be applied. However, it should be careful, since the θ angle does not stop when it "reaches" the $|1\rangle$ state, so it might be possible to rotate the state beyond it, and instead of increasing the probability of measuring 1, it would be decreased. However, the R_y gate is easily reversible, applying another R_y gate but at a $-\theta$ angle. [3]

Analogous to the quantum gate R_y, there is the operation R_x, which rotates the Qubit but this time around the X-axis and using the matrix of the equation 2.25. In the same way, there is the R_z opetation which rotates the Qubit around the Z-axis, using the matrix of the equation 2.26. [10]

$$R_x = \begin{bmatrix} cos\frac{\theta}{2} & -i\ sin\frac{\theta}{2} \\ -i\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix} \tag{2.25}$$

R_x Rotation Gate Matrix.

$$R_z = \begin{bmatrix} e^{-i\phi/2} & 0 \\ 0 & e^{i\phi/2} \end{bmatrix} \tag{2.26}$$

R_z Rotation Gate Matrix.

2.3.1.7.4. Unitary Gate (U-Gate)

The U-gate in quantum computing is a type of unitary operator used to perform any of the transformations on the state of a single qubit. This gate has three parameters, two angles describing the position of the state on the Bloch sphere and the global phase of the state, which can be adjusted to control the nature of the transformation performed on the qubit. In the mathematical representation, the U gate is represented, just like the operations of a single qubit, as a 2 x 2 matrix represented in the equation 2.27 and its action on a qubit is realized through a matrix product. [10]

$$U(\theta, \phi, \lambda) = \begin{bmatrix} cos\frac{\theta}{2} & -e^{i\lambda}\ sin\frac{\theta}{2} \\ e^{i\phi}\ sin\frac{\theta}{2} & e^{i(\phi+\lambda)}\ cos\frac{\theta}{2} \end{bmatrix} \tag{2.27}$$

U-Gate Matrix.

In summary, the U-gate is an important and highly versatile component in the implementation of quantum algorithms and problem solving in the field of quantum computing, since it can represent other Quantum Gates, such as those in the equation 2.28.

$$U(\theta, -\frac{\pi}{2}, \frac{\pi}{2}) = R_y(\theta) \quad , \quad U(\theta, 0, 0) = R_x(\theta) \tag{2.28}$$

U-Gate examples.

2.3.1.7.5. Phase (P-Gate)

The Phase Gate is a quantum logic gate used in quantum computing. It is defined as a rotation matrix in qubit space that applies a phase to a quantum state. The phase is a complex factor that determines the direction in which a quantum state moves in state space. The Phase Gate applies a phase of $e^{i\lambda}$ to a $|1\rangle$ state and does not affect the $|0\rangle$ state, using the matrix of the equation 2.29. When the applied phase i $-\pi$, this gate is equivalent to the R_z Gate represented in the 2.26 equation. [10]

$$P(\lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix} \tag{2.29}$$

P-Gate Matrix.

2.3.1.7.6. C-NOT Gate

Until now, only the operation of quantum gates on a single qubit has been explained, however, the space of a quantum processor grows exponentially with the number of qubits, so for n qubits, the vector space has dimension 2^n and to describe the quantum states in this system the tensor product is used, which for any two operators A and B, is the one represented in the equation 2.30, where A_{jk} and B_{lm} are the matrix elements of A and B respectively. [10]

$$A \otimes B = \begin{pmatrix} A_{00}\begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} & A_{01}\begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} \\ \\ A_{10}\begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} & A_{11}\begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} \end{pmatrix}, \tag{2.30}$$

Tensor product between matrixes.

$$\text{With } \; v = \begin{bmatrix} v_o \\ v_1 \\ \vdots \\ v_n \end{bmatrix} \text{ and } w = \begin{bmatrix} w_o \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$$

$$\text{, then } \; v \otimes w = \begin{bmatrix} v_0 w_0 \, , & v_0 w_1 & \cdots & v_0 w_n \, , & v_1 w_0 \, , & v_1 w_1 & \cdots & v_1 w_n & \cdots & v_n w_n \end{bmatrix}^{\mathrm{T}} \tag{2.31}$$

Tensor product between two vectors.

Let be a pair of qubits $|a\rangle$ and $|b\rangle$ described by the states $|a\rangle = a_0|0\rangle + a_1|1\rangle$ y $|b\rangle = b_0|0\rangle + b_1|1\rangle$, it is possible to represent the states of this quantum system in an expression, such as in the equation 2.32, using the tensor product between two vectors denoted in the equation 2.31. [3]

$$|ab\rangle = |a\rangle \otimes |b\rangle = a_0b_0|0\rangle|0\rangle + a_0b_1|0\rangle|1\rangle + a_1b_0|1\rangle|0\rangle + a_1b_1|1\rangle|1\rangle = \begin{bmatrix} a_0b_0 \\ a_0b_1 \\ a_1b_0 \\ a_1b_1 \end{bmatrix} \tag{2.32}$$

Tensor product between two state vectors.

A multi-qubit quantum gate involves its application on a qubit based on the state of other qubits, for example, in the case of changing the state of a second qubit when the first qubit is in the $|0\rangle$ state. This kind of quantum gate is known as a controlled gate and uses the concept of entanglement to relate the state of two or more qubits.

The CNOT quantum gate is one of the most basic and essential controlled quantum gates in quantum computing. It is a binary quantum gate that operates on two qubits and has the ability to invert the state of one of them depending on the state of the other. This gate is defined by a unitary matrix acting on a pair of qubits. In mathematical terms, the action of the CNOT gate on a pair of qubits q_0 and q_1 can be expressed as the transformation denoted in the equation 2.33, where the operator X is the X-Gate and the operator I is the identity operator, which produces no change in the quantum state on which it is applied. [11]

$$C_X \ q_0, q_1 = I \otimes |0\rangle\langle 0| + X \otimes |1\rangle\langle 1| = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{2.33}$$

C-NOT Gate Matrix.

The effect of this quantum gate is that if the control qubit is in a $|0\rangle$ state, the target qubit is left unchanged, while if the control qubit is in a $|1\rangle$ state, the state of the target qubit is reversed. The CNOT gate is a reversible gate and is universal, meaning that it can be used to implement any unitary operation in a quantum system. In addition, it is a gate that forms the basis for the implementation of quantum algorithms, such as quantum factorization and Grover's algorithm. [10]

2.3.1.8. Qubits Measurements

Measurement in quantum computing is a fundamental process that allows obtaining information about the quantum state of a system. It is fundamental to remember that in a quantum system, qubits are in a superposed quantum state, that is, in a linear combination of basic states and the measurement of a qubit collapses its superposed state to one of its basic states, in a probabilistic way.

In this context, like the quantum gates, it is possible to execute an operation on the qubits that perform an observation. This observation is performed in the standard

basis, also known as Z-Basis classical bit basis, which is the basis where the basic states are $|0\rangle$ and $|1\rangle$, which correspond to the logical states "0" and "1" in classical computing.

This operation can be used to perform any type of measurement in combination with quantum gates. It is important to note that once a measurement is performed on a qubit, its state becomes deterministic and cannot be changed again until an additional quantum operation is performed on it. [10]

Unlike quantum gates, this measurement operation extracts partial information about the state of a qubit or quantum system, often losing the phase, in order to represent it as a classical bit and write it to some readout device. This is the typical way of getting information from quantum data to a classical device. [11]

2.3.2. Quantum computers

Quantum computers are an emerging technology that promises to revolutionize the way calculations are performed and information is processed. Unlike classical computers, which use binary bits that can only be in a "1" or "0" state, quantum computers use qubits, which can be in several states simultaneously. This allows them to perform many operations in parallel, which means they can solve problems much faster than classical computers.

In the same way that classical transistors can be compared to electrical switches, which will turn on or off to represent a bit, qubits can be compared to joysticks that can be at various angles simultaneously to represent various states. This allows them to perform many operations in parallel and thus speed up information processing.

The basic components of a quantum computer include qubits, which can be physical or logical, and a control and readout system that allows the qubits to interact with other components and the environment. Physical qubits can be constructed using materials such as nitrogen in diamond, silicon or graphite, as well as various technologies including neutral atoms, trapped ions, superconducting circuits, semiconductor quantum dots and photons. Each type of qubit has its own strengths and weaknesses, and the choice of qubit type depends on the specific use for which the quantum computer is being built. For example, atom-based qubits can be very accurate and stable, but are difficult to manipulate and scale to large sizes. On the other hand, qubits based on superconductors are easier to manipulate and scale, but may be less accurate and stable. [11]

In addition to the choice of qubit type, the physical construction of qubits also requires that certain stringent environmental conditions be achieved. Quantum qubits are very sensitive to particle interference and environmental energy, which can affect their ability to perform precise operations. Therefore, it is necessary to keep quantum computers in controlled environments with extremely low temperatures and to protect them from external disturbances.

Currently, there are several companies that are developing and manufacturing quan-

tum computers and their components, including IBM, Google and Intel. These companies are working on creating larger and more powerful quantum computers, with the aim of addressing challenges in areas such as materials simulation, route optimization and artificial intelligence.

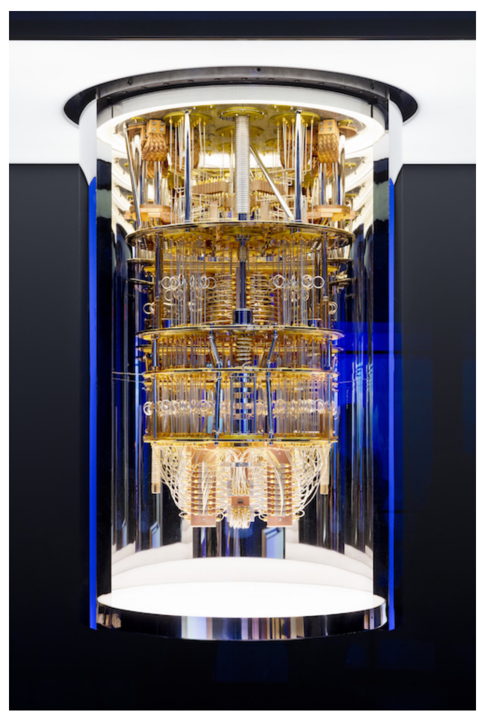


Figure 2.3.7: An inside view of IBM Quantum System One.

For example, the IBM Quantum System One, illustrated in the image in Figure 2.3.7 is a quantum computing system designed to be a scalable and reliable platform for the development and use of quantum computing. It is a self-contained system consisting of a series of quantum and classical components that work together to perform quantum computations. The quantum part of the system includes qubits that enable superposition and entanglement, two unique quantum properties that allow much faster and more efficient information processing than classical systems.

In addition to the qubits, the IBM Quantum System One also includes a set of control and measurement devices that allow users to program and measure the quantum states of the qubits. It also includes quantum cooling and thermal control systems to keep the qubits in optimal conditions for use. This classic IBM Quantum System One system consists of a series of computers and communication devices that allow users to interact with the quantum system and receive results of quantum calculations. It also includes specialized software that allows users to program and control the quantum system. [10]

In summary, quantum computers are a developing technology that have the potential to revolutionize the way information is processed and computations are performed. With continued research and development, it is likely to see an increase in the size and power of these computers in the near future..

2.3.3. Simulators

Quantum computer simulators are computational tools that emulate the behavior of quantum systems. These simulators are designed to emulate the dynamics and properties of quantum systems and to enable exploration and analysis of these systems.

These simulators work through a combination of mathematical theories and computational algorithms using the mathematical formalism of quantum mechanics, which describes the behavior of quantum systems in terms of quantum states and unitary operations, to represent a quantum system. With this formalism, the simulators numerically solve the temporal dynamics of a quantum system represented in mathematical terms, using specific computational algorithms. These algorithms can be simple, such as the numerical integration of differential equations, or more complex, such as the algorithms of quantum control theory.

In addition, these simulators generally use a matrix representation of quantum states and unitary operations, which allows their efficient processing on a classical computer.

In terms of composition, quantum computer simulators are usually composed of a combination of hardware and software. The hardware can be a classical computer or a specific quantum computer, while the software includes the algorithms and theoretical models needed for the simulation.

Some examples of quantum computer simulators are found in IBM Qiskit, a free software platform that allows users to simulate and code on quantum computers. IBM Qiskit offers a wide range of tools and resources for simulation, optimization and programming of quantum computers, including a quantum compiler, quantum simulators and a library of quantum algorithms.

In short, simulators work by mathematically representing a quantum system, numerically solving its temporal dynamics and using efficient computational algorithms for processing. This allows researchers and developers to simulate and understand the behavior of quantum systems and to design quantum algorithms and technologies, even when the technology is not available.

2.3.4. IBM Qiskit

IBM Qiskit is an open source programming platform and software tools for developing and running quantum algorithms on quantum computing systems. It is composed of a series of modules covering different aspects of quantum algorithm development, including building quantum circuits, simulating quantum systems and running algorithms on real quantum computing systems.

The quantum circuit construction module allows users to create and define quantum circuits using a wide variety of quantum gates and operations. It also provides a

visual interface to facilitate circuit creation and verification. On the other hand, the quantum systems simulation module allows users to simulate the behavior of quantum circuits in a classical environment, providing a way to test and verify algorithms prior to their execution in a real quantum computing system. Finally, the algorithm execution module allows users to run the algorithms on real quantum computing systems, both on local systems and in the cloud. In addition, it also provides tools for the analysis and visualization of the results obtained.

All these modules are importable and executable in a virtual Python environment, just like any library, making IBM Qiskit a comprehensive and complete tool for the development and execution of quantum algorithms, allowing users to experiment with quantum technology and discover new applications in this constantly evolving field. [11]

2.3.4.1. Backends

Qiskit backends are the devices or platforms that run the quantum programs written with the Qiskit framework. These backends include quantum simulators in software as well as real quantum devices, such as superconducting quantum processors and trace ion-based processors. Qiskit backends communicate with Qiskit programs through a standardized application programming interface (API), allowing users to write quantum programs once and run them on different backends without having to modify the code.

In the case of actual quantum devices, IBM offers a variety of quantum systems based on superconducting qubit technology. These systems are developed on IBM Quantum System One and are built using the world's leading components such as quantum processors, cryogenic components and classical computing technology. The actual quantum systems determine their capacity according to the types of processors found in each of them, which are classified into the following families: [10]

- **Eagle:** This processor family incorporates the most scalable technologies compared to the previous ones, with 127 Qubits.
- **Hummingbird:** This class of processors uses a heavy-hexagonal layour of qubits and have up to 65 Qubits.
- **Egret:** These processors are based on the innovations of tunable couplers to a 33-qubit platform, resulting in faster and higher fidelity two-qubit gates.
- **Falcon:** This family of processors offers a considerable amount of mid-scale circuitry and are useful for demonstrating performance and scalability improvements prior to using larger devices.
- **Canary:** These devices are small designs containing from 5 to 16 Qubits.

On the other hand, in the case of simulators, IBM offers a collection of high-performance simulators to run quantum circuits and algorithms in a realistic way even

simulating the noise effects present in a quantum computer. Among these are the following simulators:

- **Statevector:** It simulates a quantum circuit by calculating the wave function of the state vector of a qubit when quantum gates and instructions are applied. This simulator also supports general noise simulations. It supports up to 32 Qubits.
- **Stabilizer:** It is a Clifford circuits simulator and can also simulate the noise evolution of a quantum device. It supports up to 5000 Qubits.
- **Extended stabilizer:** Simulates the action of a quantum circuit using a ranked-stabilizer decomposition. Supports up to 63 Qubits
- **MPS:** It performs the simulation of the states using tensors, with the so-called Matrix Product State representation. It is more efficient with weakly intertwined states and supports up to 100 Qubits.
- **QASM:** It is a multi-purpose simulator, useful both for simulating circuits and the noise of quantum devices. This simulator supports up to 32 Qubits.
- **Automatic:** Default simulation method. Automatically selects the simulation method based on the circuit and the noise model.

On the other hand, Qiskit provides the possibility of defining quantum-classical workloads in near real-time, to create and customize the development of applications efficiently. For this, before running the work on a quantum device, core functions called primitives are used, which perform basic quantum computing tasks and are used as an entry point. Two primitives are currently available: [10]

- **Estimator:** It allows efficient calculation and interpretation of the variances and expectation values of the quantum operators used in the algorithms.
- **Sampler:** This function takes a circuit as input and generates a readout of the quasi-likelihoods using error mitigation.

In summary, both primitives and backends are the execution platform for quantum programs written with Qiskit, either in the form of quantum simulators in software or real quantum devices. These backends allow users to run their quantum programs on a wide variety of quantum platforms and devices, facilitating research and development in the field of quantum computing.

2.4. Programming Resources

The application and execution of quantum models requires long times, so to carry out this work, Python programming was used using three programming environments: Jupyter Notebook, Google Colab and IBM Quantum Lab.

Jupyter Notebook is an interactive programming environment that allows users to create and share code notebooks that combine code, text and visualizations in a single document. Jupyter Notebook is the most common implementation of this environment and supports several programming languages, including Python.

Google Colab is a free service from Google that allows users to create and run Jupyter Notebooks in the cloud. Like Jupyter Notebook, Colab allows you to write and run code, add text and visualizations. The advantage of Colab is that it runs on Google cloud servers, which means you don't need to install software on your own computer and can access more powerful computational resources. In addition, Colab also offers paid plans or free access to Google's graphics processing units (GPUs) and tensor processing units (TPUs), which are useful for deep learning and large-scale data processing.

IBM Quantum Lab is an online platform that enables users to learn and experiment with quantum computing using IBM quantum computing systems. The platform provides a graphical user interface for creating and running quantum programs in the cloud, as well as a series of tutorials and educational resources for learning the basics of quantum computing. In addition, the platform also provides an interface for creating and running Jupyter notebooks with either quantum simulators or real quantum hardware.

Each of the programming environments requires different treatment to install the libraries used in this context, however, in all of them a virtual environment was used with the following Python libraries:

- **Qiskit:** It is the main library used to apply the quantum models throughout this work, it includes the possibility to work with qubits and quantum circuits and run them in quantum simulators or in real quantum hardware.
- **Qiksit Machine Learning:** It provides tools and algorithms for applying machine learning in quantum computing. The library is used to design and run machine learning models in quantum systems, and also to simulate and train machine learning models in quantum and classical environments.
- **Scikit Learn:** It is used to apply machine learning algorithms to data sets. It provides tools for classification, regression, clustering and dimensionality reduction, as well as data preprocessing. The whole section2.1 of general Machine Learning concepts can be applied using this library.
- **Qiskit aer:** It is used to simulate the behavior of quantum systems and to validate quantum algorithms in the simulation environment described in the section 2.3.3. It also allows users to model and analyze errors in quantum computing, making it a useful tool for those who wish to experiment with quantum computing without needing access to quantum hardware and for those who wish to develop and test quantum algorithms in a controlled simulation environment.
- **Pytorch:** It is used to create neural networks and apply deep learning techniques to tasks in image processing, natural language processing and other areas. The library allows users to build and train deep learning models using a variety of

network architectures, and provides tools for optimization, parallel processing and data visualization.

- **Tensorflow:** Like Pytorch, this library is used to create and train machine learning models. The library is widely used in image processing, natural language processing and other areas, and allows users to build and train deep neural networks using a variety of network architectures. TensorFlow also provides tools for optimization, parallel processing and data visualization.

- **Pandas:** It is one of the useful libraries for data preprocessing, used for data analysis and manipulation. It is used to work with structured data sets, such as spreadsheets or databases, and allows users to read and write data in various formats, including CSV, Excel and SQL databases. Pandas also provides tools for filtering, transforming and grouping data, and for working with missing data.

- **Numpy:** It is another library used for data preprocessing because it is used to work with numerical matrices and vectors. It provides tools to perform complex mathematical operations on matrices and vectors, including linear algebra, statistics and Fourier transform operations. NumPy is also used to generate random numbers and to work with structured data.

- **Matplotlib:** It is used to create graphs and data visualizations. It is used to generate 2D and 3D charts, including line charts, scatter plots, bar charts and pie charts. Matplotlib allows users to customize the appearance of graphs and add labels, titles and legends. It also integrates well with other Python libraries, such as NumPy and Pandas.

- **Pylatexenc:** Allows you to encode and decode text strings with special characters used in LaTeX. It is used to process and manipulate technical, scientific or mathematical documents that have been formatted in LaTeX.

- **QuTip:** It is used to simulate open and closed quantum systems. It provides tools to calculate quantum properties, such as the wave function, reduced state and time evolution of a quantum system. QuTiP is also used to simulate the coupling of quantum systems to electromagnetic fields and to perform quantum tomography calculations.

2.5. Quantum Machine Learning

Quantum machine learning is an emerging branch of machine learning that uses quantum computing to improve the efficiency and accuracy of machine learning models.

Unlike classical machine learning, which relies on conventional bit-based computing, quantum machine learning uses qubits. Qubits allow the use of quantum algorithms, such as the quantum Fourier transform, to perform complex operations more efficiently than classical algorithms.

Quantum machine learning can also take advantage of the properties of superposition, interference and quantum entanglement which can significantly increase the processing power of a machine learning model. Despite these advantages, quantum machine learning still faces many challenges, such as the need for highly sophisticated quantum hardware and the difficulty of designing optimal quantum algorithms.

Although quantum machine learning is at an early stage of development, some promising algorithms have already been developed that have shown significant improvements over their classical counterparts. For example, the Grover classification algorithm has been shown to be more efficient than the classical binary search algorithm.

Overall, quantum machine learning has the potential to significantly transform the field of machine learning, and is likely to play an increasingly important role in advanced machine learning applications in the future.

2.5.1. General Concepts

During this section, the fundamental concepts and tools on which Quantum Machine Learning models are built will be explained.

2.5.1.1. Features Maps

In Qiskit, a feature map is a tool used in quantum data encoding that is used to transform input data into a high-dimensional quantum feature space. A feature map acts as an additional layer of processing before the input data is fed to a quantum classification algorithm.

This implies that these feature maps can be used to map the classical data of a dataset to quantum states. Feature maps are mathematical functions that transform the input data into a quantum representation. This is achieved by mapping a set of features from the dataset to a set of qubits, which are encoded into quantum states using a rotation gate.

In more technical terms, a feature map is a transformation that takes data from a low-dimensional space and maps it to a higher-dimensional Hilbert space, which is commonly referred to as the feature space, as in the expression 2.34. In this context, this transformation is a mapping from a classical space to a quantum space using a quantum circuit. This process increases the complexity of the feature space and allows a quantum classification algorithm to detect more complex patterns in the input data. [13]

$$\phi : \mathbb{R}^d \to \mathcal{H} \text{ , where: } \underset{\substack{\text{Classical}\\\text{features}}}{\overrightarrow{x}} \xrightarrow{\phi(\overrightarrow{x})} \underset{\substack{\text{Quantum state}\\\text{vector}}}{|\Phi(\overrightarrow{x})\rangle\langle\Phi(\overrightarrow{x})|} \text{ for quantum cases} \tag{2.34}$$

Feature Map Transformation.

Feature maps are used in the context of quantum data classification, which is a type of quantum machine learning problem. In quantum data classification, the input data is represented by qubits, and the goal is to design a quantum algorithm that can classify the input data into one of several categories.

In this context and for some applications, it is essential to know how a quantum circuit will be structured without having explicit information about its size or specific characteristics. To overcome this limitation, Qiskit has developed an approach in which the quantum circuit itself can be built dynamically, once all the necessary information is available. In this way, greater flexibility and adaptability is achieved in the construction of quantum circuits for different applications. In this way, Qiskit offers several different types of circuits usable as feature maps, each designed to work with different types of input data and different quantum classification problems. Some examples of feature maps are as follows: [11]

- **PauliFeatureMap:** Encode input data $\overrightarrow{x} \in \mathbb{R}^n$ according to the expression 2.35, where the variable $P_i \in \{I, X, Y, Z\}$ denotes the Pauli matrices, the index S describes the connectivity between different qubits or datapoints: $S \in \{\binom{n}{k}\ combinations,\ k = 1, ...n\}$

$$U_{\Phi(\vec{x})} = \exp\left(i \sum_{S \subseteq [n]} \phi_S(\vec{x}) \prod_{i \in S} P_i\right)\ , \text{ where: } \phi_S(\vec{x}) = \begin{cases} x_0 \text{ if } k = 1 \\ \prod_{j \in S} (\pi - x_j) \text{ otherwise} \end{cases} \tag{2.35}$$

Pauli Feature Map Transformation.

- **ZFeatureMap:** Encode input data $\overrightarrow{x} \in \mathbb{R}^n$ according to the same expression 2.35, but for the case $k = 1$ and $P_0 = Z$

- **ZZFeatureMap:** Encode input data $\overrightarrow{x} \in \mathbb{R}^n$ according to the same expression 2.35, but for the case $k = 2$, $P_0 = Z$ and $P_{0,1} = ZZ$

- **SatePreparation:** It is used to prepare classical data inputs to specific quantum states in a quantum circuit. This object takes as input an amplitude vector representing the desired quantum state and returns a quantum circuit that prepares that state. To understand how this works, it is important to note that any quantum state can be expressed as a linear combination of the states of the computational basis. For example, a qubit can be expressed as a linear combination of the states $|0\rangle$ and $|1\rangle$, as in the equation 2.15, where α y β are the amplitude coefficients describing the quantum state. StatePreparation uses a technique known as "preparation amplitude", which consists of preparing a quantum state using a series of quantum gates (Hadamard, rotations and CNOT) acting on an initial state, which in this case will be of complex amplitudes. In this way a quantum circuit is dynamically constructed that prepares a specific quantum state from a given amplitude vector.

- **RawFeatureVector:** This circuit serves as a set of instructions for preparing state vectors with a defined number of dimensions. The execution of the circuit depends on the availability of all the parameters necessary for its definition. In the field of Machine Learning, this circuit can be used to load training data into qubit states. Unlike applying a kernel transformation, this circuit provides raw feature vectors. However, it is important to note that it is not compatible with optimizers that rely on gradient calculations.

It should be noted that in the feaures maps inspired by the Pauli transform, as well as in the StatePreparation, the resulting quantum circuit to encode classical data into quantum data consists of a number of qubits equal to the number of features present in the dataset. However, using RawFeatureVector as a featuremap, a circuit with a number of qubits equal to the logarithm in base 2 of the number of features is obtained.

In summary, feature maps are an important tool in quantum data processing in Qiskit. By transforming the input data into a high-dimensional quantum feature space, feature maps enable quantum classification algorithms to detect more complex patterns in the input data, which can significantly improve the accuracy of quantum data classification.

2.5.1.2. Ansatz

In physics and mathematics, an ansatz is an initial guess or conjecture about the form or structure of a solution to a given problem. It is generally used when there is no known exact solution to the problem and a more creative approach is required to find an approximate solution.

For example, in the context of quantum mechanics, an ansatz is often used to approximate the solution of a wave equation, which makes it possible to describe the behavior of subatomic particles. The ansatz is used to propose a wave function that fits the conditions of the problem and is then adjusted by mathematical techniques to determine the most accurate solution.

In this case of quantum computing and in particular in the Qiskit library, the ansatz are used to prepare a quantum state and consist of a parameterized family of quantum circuits. These circuits can be modified by varying their parameters to generate different quantum states.

In Qiskit, there are several predefined ansatz in the library that can be used to prepare quantum states for various applications, such as:

- **RealAmplitudes**
- **EfficientSU2**
- **PauliTwoDesign**

These are just a few examples of the ansatz available in Qiskit, and the choice of ansatz will depend on the problem at hand and the architecture of the quantum device used.

2.5.1.3. Optimizer

In the context of prediction tasks, once the predictions of a quantum circuit are obtained, a classical optimization routine is performed to adjust the values of the circuit to improve the accuracy of the predictions. This process is repeated several times until a desired level of accuracy is reached.

The main task of the classical optimization routine is to minimize the value of the loss function associated with the quantum circuit. The loss function is defined as a measure of the discrepancy between the predictions of the quantum circuit and the experimental data. In other words, the loss function reflects the distance between the theoretical predictions and the experimental results, and its minimization results in higher accuracy of the predictions. Among the most commonly used loss functions are the following, where y_i is the data label, x_i is the feature data used for training, y'_i is the predicted label and N is the train data length:

- **Absolute Error:** This loss function measures the absolute difference between the predicted values and the true values.

$$L(y, y') = ||y'_i - y_i|| \tag{2.36}$$

Absolute Error Loss Function.

- **Mean Squared Error:** This function measures the average of the quadratic difference between the predicted values and the true values.

$$L(y, y') = \frac{1}{N}\sum_{i=1}^{N} ||y'_i - y_i||^2 \tag{2.37}$$

MSE Loss Function.

- **Cross Entropy:** This loss function measures the discrepancy between the predicted probability distribution and the true probability distribution.

$$L(y, y') = -\sum_{i=1}^{N} [\, y \cdot log(y'_i) + (1 - y_i) \cdot log(1 - y'_i)\,] \tag{2.38}$$

Cross Entropy Loss Function.

- **Cross Entropy Sigmoid:** This loss function is similar to the cross-entropy loss function, but is used in combination with a sigmoid function in the output layer of neural networks.

$$L(y, y') = -[\, y \cdot log(y'_i) + (1 - y_i) \cdot log(1 - y'_i)\,] \tag{2.39}$$

Cross Entropy Sigmoid Loss Function.

The choice of the appropriate optimizer will depend on the specific characteristics of the problem to be solved and the available resources, and will be crucial to ensure the effectiveness and efficiency of the optimization process and improve the accuracy of quantum predictions. There are different optimizers to perform this task, among which are:

- **COBYLA** (Constrained Optimization By Linear Approximation optimizer.)
- **SPSA** (Simultaneous Perturbation Stochastic Approximation optimizer.)
- **SLSQP** (Sequential Least Squares Programming optimizer.)
- **LBFGSB** (Limited Broyden–Fletcher–Goldfarb–Shanno optimizer.)
- **PBFGS** (Parallelized Limited-memory BFGS optimizer.)
- **ADAM** (A gradient-based optimization algorithm that is relies on adaptive estimates of lower-order moments)
- **AQGD** (Analytic Quantum Gradient Descent with Epochs optimizer.)
- **CG** (Conjugate Gradient optimizer.)
- **GSLS** (Gaussian-smoothed Line Search.)
- **NELDER MEAD** (Gaussian-smoothed Line Search.)
- **NFT** (Nakanishi-Fujii-Todo algorithm.)
- **POWELL** (A conjugate direction method for unconstrained optimization.)
- **TNC** (Truncated Newton optimizer.)
- **UMDA** (Continuous Univariate Marginal Distribution Algorithm.)

2.5.1.4. Quantum Neural Network

As mentioned in the previous section on classical computing, classical neural networks are models inspired by neurons in the human brain that can be trained to recognize patterns in data and solve complex problems. These networks are based on a series of interconnected nodes, or neurons, organized in a layered structure, with parameters that can be learned by applying machine learning or deep learning strategies.

In the field of quantum machine learning, the motivation is to integrate concepts from quantum computing and classical machine learning to create new and better learning schemes. Quantum neural networks apply this generic principle by combining classical neural networks with parameterized quantum circuits.

QNNs are models that can be trained to find hidden patterns in datasets, similar to classical neural networks. These models receive as input classical data in a quantum state, and then process it with quantum gates parameterized by trainable weights. The output of measuring this quantum state is used to train the weights by backpropagation through a loss function.

In a QNN, neurons are represented by qubits, so they are based on parameterized quantum circuits that can be trained variationally using classical optimizers. These circuits are composed of feature maps and an ansatz with trainable weights, with which the values of the weights that minimize the cost function associated with the learning task in question are searched.

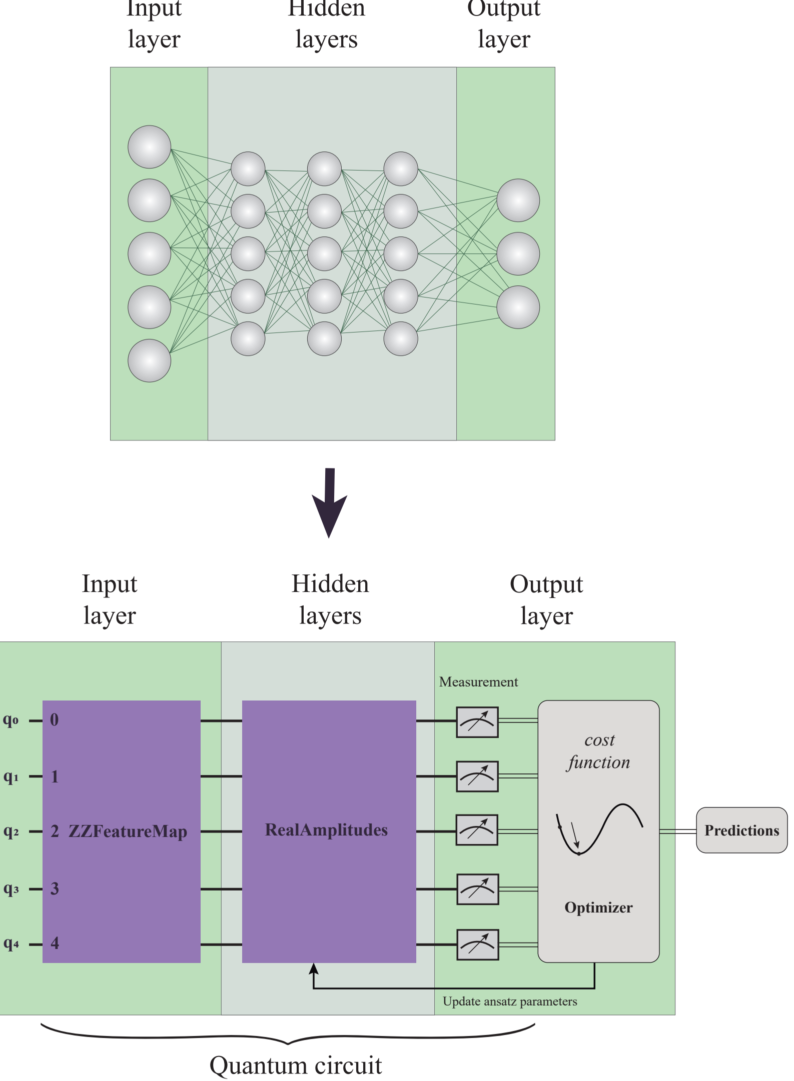


Figure 2.5.1: Classical Neural Network vs Quantum Neural Network Comparison

In this way, the analogy present in the figure **??** could be made, where in comparison to a classical neural network, in the input layer would be the datasets encoded to quan-

tum states by means of a Feature Map, in the hidden layer would be the parameterized Ansatz circuits, while in the output layer would be the final measurements made after the optimization to the qubits of the neural network. All this is replicable in IBM Qiskit by means of the different available circuits and optimizers, in addition there are some quantum neural networks in Qiskit, in particular for this work, the following ones are used: [14]

- **OpflowQNN:** This is a technique that uses a neural network to process information through a quantum circuit. This circuit is usually combined, it is composed of a FeatureMap, which sets the input parameters for the network, and an Ansatz, which determines the weight parameters. By applying this technique, the neural network is able to calculate the expectation value for the prediction desired.
- **CircuitQNN:** Like OpflowQNN, this quantum neural network can take input and weight parameters from a FeatureMap and an Ansatz to produce samples from the measurement. In addition, CircuitQNN allows to specify an interpretation function to post-process the samples, whose functionality is to assign some measured number to another unassigned number, or to a tuple of unassigned integers, which are used as new indices for the output matrix. In this way, information is obtained about the probability distribution of the quantum neural network results and how they relate to the input values and network parameters.
- **EstimatorQNN:** In the same way as the previous networks, this one takes input parameters through quantum circuits with FeatureMap and Ansatz to estimate the neural network cost function and calculate the approximate gradients.

2.5.2. Models used

This section presents the quantum models used in this work. These models are based on the properties of quantum mechanics and have proven to be useful in a variety of applications, including data classification. The models presented include the Variational Quantum Classifier, Quantum Neural Network Classifier, Quantum Neural Network with Pytorch Classifier and Quantum Support Vector Classifier. Each model is explained in detail, including its structure and operation.

2.5.2.1. Variatonal Quantum Classifier

The Variational Quantum Classifier (VQC) is a quantum machine learning algorithm that uses quantum circuits to classify data. VQC is based on supervised classification, in which the algorithm is provided with a training data set labeled with known categories, and the algorithm learns to classify new data into those categories. Like any classification model, VQC has a training and a testing stage, similar to classical machine learning models.

This model has a hybrid architecture that uses both quantum and classical resources to perform classification tasks because it is composed of two main parts: a quantum circuit that functions as a classifier, such as the one in Figure **??**, and a classical

optimization algorithm that adjusts the parameters of the quantum circuit to improve

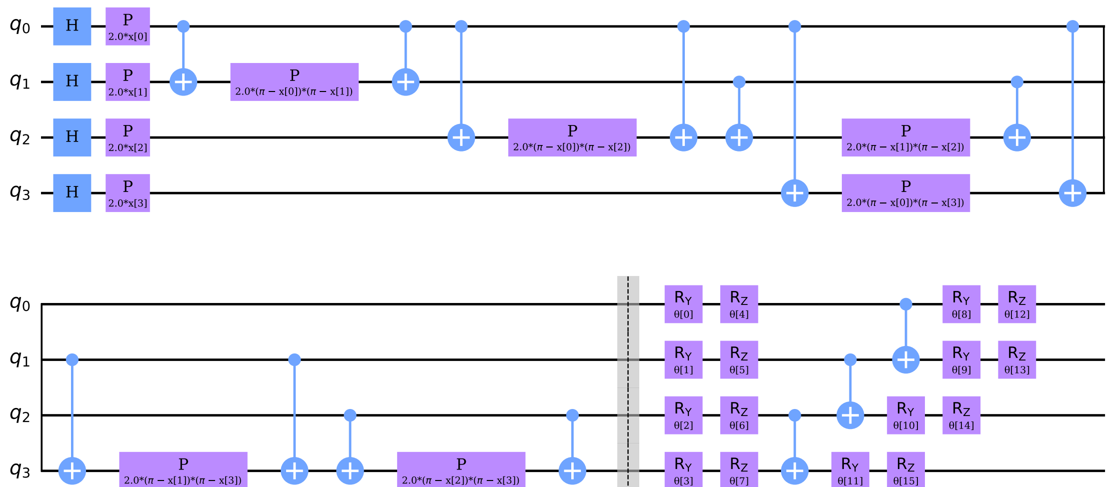


Figure 2.5.2: VQC Example Circuit

The training process is divided into four steps. First, the training data are loaded into qubits of a quantum circuit using FeatureMaps. Then, they are passed to a variational quantum circuit, using the Ansatz, which performs the classification task. After performing the classification, the parameters of the variational circuit are adjusted using a classical optimization task, such as COBYLA, to improve the classification accuracy by minimizing the value of the loss function of the equation 2.40. Subsequently, the quantum circuit is measured and the classical optimization algorithm is continued to find the optimal parameters of the variational quantum circuit by returning to the Ansatz recursively to improve the classification accuracy. [11]

$$L(\theta) = \frac{1}{N}\sum_{i=1}^{N} ||f(x_i,\theta) - y_i||^2 \text{ , where } \begin{cases} f(x_i,\theta) & : \text{Ansatz output} \\ x_i & : \text{Train data} \\ y_i & : \text{Data label} \\ \theta & : \text{Ansatz parameters} \\ N & : \text{Train Data Length} \end{cases} \quad (2.40)$$

Mean Squared Error used as VQC Loss Function.

In summary, Qiskit's Variational Quantum Classifier uses a quantum circuit and a classical optimization algorithm to classify input data. The quantum circuit is trained with labeled data and then used to classify new data into known categories.

2.5.2.2. Quantum Neural Network Classifier

Qiskit's Quantum Neural Network Classifier (QNN Classifier) is a quantum machine learning classification model that uses a quantum circuit to perform classification. In

this paper, the QNN Classifier is implemented in Qiskit using the Qiskit Machine Learning library.

This model receives a quantum neural network like those seen previously, which are constructed from a circuit like the one in figure 2.5.3 with a FeatureMap and an Ansatz. In principle, a quantum neural network can have a one-dimensional or a multidimensional output. In the first case, the output of the neural network is expected to be in the range [-1, +1], which makes it suitable for binary classification. In the second case, the neural network output represents a probability distribution, which means that each input in the output is non-negative and sums one. To handle this situation, two different approaches can be used to interpret the results: using one-hot encoding or not using one-hot encoding. If one-hot encoding is used, each probability vector resulting from the neural network is considered a single sample and the loss function is applied to the entire vector. If one-hot encoding is not used, each entry of the probability vector is considered an individual sample and the loss function is applied to the index, weighting with the co

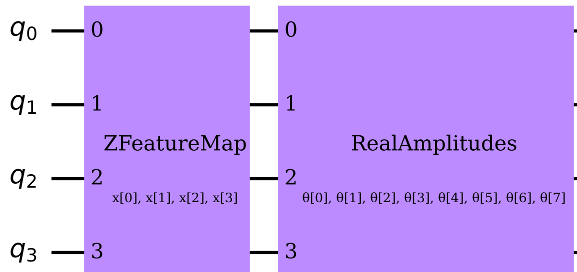


Figure 2.5.3: QNN Classifier Example Circuit

The loss function is optimized using a classical optimization algorithm, such as COBYLA. These loss functions can be the Mean Squared Error as in the case of VQC, or Absolute Error, Cross Entropy or Cross Entropy Sigmoid. By default, the Mean Squared Error is used.

Subsequently, as in any quantum circuit, measurements are performed on the qubits to obtain the results and, like any classification model, the model evaluation stage is followed by the test dataset.

2.5.2.3. Quantum Neural Network with Pytorch Classifier

Torch Connector is a library that allows users to connect PyTorch, a popular deep learning framework, with simulators and quantum hardware. This allows researchers to experiment with hybrid classical-quantum neural networks that can harness the power of both classical and quantum computing. In this way, it is possible to integrate any Quantum Neural Network built in Qiskit into the Pytorch workflow.

In this work, quantum neural networks such as OpflowQNN and CircuitQNNN are

used and trained using PyTorch's automatic differentiation engine. This engine is an essential tool in the PyTorch framework, which allows users to automatically compute gradients of mathematical functions. Automatic differentiation is a technique for calculating derivatives of a function by recording all operations performed on the function, including elementary operations such as addition, multiplication, and exponentiation. In particular, this Pytorch engine allows users to define a mathematical function using PyTorch tensors and tensor operations, and then automatically compute, using backpropagation, the gradients of the function with respect to the input variables. The gradients computed by the automatic differentiation engine are needed to train neural networks using deep learning techniques, such as stochastic gradient descent. [15]

In the case of using the EstimatorQNN network, the workflow is as follows:

1. Encoding data into Torch Tensors
2. Define the circuits to be used for the FeatureMap and Ansatz.
3. Set up QNN con el circuito cuántico definido.
4. Set up the Pytorch module, giving to Torch Connector the QNN weights as initial weights.
5. Choose the optimizer function and the loss function from Pytorch package .
6. Training the QNN with Torch Connector module and the optimizer rutine.
7. Evaluate the model performance.

On the other hand, in case of using the SamplerQNN network, a more specific setup is required for the Pytorch engine to successfully perform the backpropagation. The workflow in this case is as follows:

1. Encoding data into Torch Tensors
2. Define the circuits to be used for the FeatureMap and Ansatz.
3. Set up QNN con el circuito cuántico definido, esta vez prestando atención a su inicialización, we must make sure that we are returning a dense array of probabilities in the network's forward pass, para esto el parámetro *sparse* del SamplerQNN debe estar seteado como False. Also in this QNN the output can be set up in different formats, and an optional post-processing step can be used to interpret the sampler's output, but we must remember to explicitly provide the desired output shape to the network.
4. Set up the Pytorch module, giving to Torch Connector the QNN weights as initial weights.
5. Choose the optimizer function and the loss function from Pytorch package .
6. Training the QNN with Torch Connector module and the optimizer rutine.
7. Evaluate the model performance.

2.5.2.4. Quantum Support Vector Classifier

Quantum Support Vector Classifier (QSVC) is a quantum classification algorithm that uses the theory of quantum mechanics to classify data. QSVC is based on the classical Support Vector Machine (SVM) algorithm, but uses the power of quantum computing to perform more efficient and accurate computations using a Quantum Kernel.

As mentioned in the 2.1.8 section of the Support Vector Classifier, in many datasets the distribution of your data can be better interpreted in a higher dimensional feature space by using a Kernel function $k(\vec{x}_i, \vec{x}_j) = \langle f(\vec{x}_i), f(\vec{x}_j)\rangle$ where k is the kernel function, $\vec{x}_i, \vec{x}_j$ are n dimensional inputs, f is a map from n-dimension to m-dimension space and $\langle a, b\rangle$ denotes the dot product. Sin embargo, in the context of Quantum Machine Learning using the Qiskit library, a quantum feature map is used. $\phi(\vec{x})$ to map a vector of classical features $\vec{x}$ to a quantum Hilbert space, $|\phi(\vec{x})\rangle\langle\phi(\vec{x})|$, whereby the Kernel Function can be described by the matrix of the equation 2.41. [11]

$$K_{ij} = \left|\langle\phi^{\dagger}(\vec{x}_j)|\phi(\vec{x}_i)\rangle\right|^2 \tag{2.41}$$

Quantum Kernel Matrix Representation.

QSVC operates in a quantum Hilbert space, which would potentially perform more accurate and efficient classification of data. Instead of representing data in terms of vectors and matrices, QSVC represents data as quantum states and uses quantum operations on quantum circuits to classify them. This is how this model uses a Kernel built based on FeatureMaps, such as those mentioned above: ZFeatureMap, ZZFeatureMap or RealAmplitudes, trough the FidelityQuantumKernel function available in Qiskit. [11]

Another particularity of this classification method is the implementation of the *Pegasos* algorithm, which consists of a stochastic sub-gradient descent algorithm for solving the optimization problem cast by Support Vector Machines. This algorithm performs a stochastic gradient descent with a specific stepsize in the primal optimization problem such as the 2.8 equation of the 2.1.8 section of Support Vector Classifier. At its start, when starting to iterate, this algorithm sets the first vector w_1 as a zero vector and in a next randomly chosen t iteration, it replaces the objective function of the primal problem with an approximation based on the training data (x_{i_t}, y_{i_t}) that has been performed in the chosen t iteration, obtaining a function like the one in the 2.42 equation.

$$f(w\,;(x,y)) = \frac{\lambda}{2}w^T w + C\sum_{i=1}^{l}\xi_i(w;(x,y)) \rightarrow f(w\,;i_t) = \frac{\lambda}{2}w^T w + C\,\xi_i(w\,;i_t) \tag{2.42}$$

Pegasos Algorithm aproximation to objective function example.

Subsequently, the algorithm considers the sub-gradient of the objective function approximation obtained from the problem, such as the equation 2.43. It then updates

the values of the vector w such that $w_{t+1} \leftarrow w_t - \eta_t \nabla_t$ using a step size of $n_t = 1/(\lambda t)$. In this way, after a predetermined number of iterations, the objective function of the primary optimization problem of the model is minimized. [16]

$$\nabla_t =_t^{\mathrm{T}} w_t - \mathbb{1}[\, y_{i_t} \langle w_t, x_{i_t} \rangle < 1\,]\; y_{i_t} x_{i_t} \text{ , where: } \mathbb{1}[\, y\langle w, x \rangle < 1\,] = \begin{cases} 1 & \text{, if true} \\ 0 & \text{, otherwise} \end{cases} \qquad (2.43)$$

Pegasos Algorithm subgradient of aproximated objetive function example.

In summary, one of the main advantages of QSVC is its ability to handle nonlinear and high-dimensional data. Unlike classical classification algorithms, QSVC could classify data that cannot be linearly separated in a finite-dimensional space using the quantum kernel. Moreover, a classical optimization algorithm is employed in this case, which promises to succeed in solving the optimization problem of SVC. Another advantage of QSVC is its ability to perform multiple classification tasks at the same time, which makes it especially useful for machine learning applications where fast and accurate classification is required.

2.6. Used cases of study

In this study, two widely recognized datasets in predictive maintenance research will be used: the MFPT (Machinery Failure Prevention Technology) organization's bearing dataset known as Fault Dataset Manipulation, and the NASA dataset called C-MAPSS (Commercial Modular Aero Propulsion System Simulation). These datasets will be used to test and evaluate the performance of quantum machine learning algorithms in the early detection of bearing and turbine system failures.

The MFPT bearing dataset consists of vibration measurements of bearings subjected to different types of failures, while the NASA C-MAPSS dataset consists of measurements of turbine sensors under different operating conditions. Both datasets have been used in numerous studies on failure diagnosis techniques in rotating machinery systems, leading to improvements in the efficiency, reliability and safety of these systems.

In this study, quantum machine learning techniques will be used to analyze and process data from both data sets to detect early failures in bearing and turbine systems. The use of quantum techniques in the detection of faults in rotating machinery systems is an emerging and promising field in predictive maintenance research, and the results of this study are expected to contribute significantly to the advancement in this area.

2.6.1. MFPT

The Machinery Failure Prevention Technology (MFPT) bearing dataset known as Fault Dataset Manipulation is a valuable resource for research in the field of predictive

maintenance of bearing systems. This dataset consists of vibration measurements of bearings that have been subjected to different types of failures, such as the presence of cracks and wear. When the rolling elements pass or impact with any of these defects, high frequency vibrations are produced that can be recorded, for example, using an accelerometer or a transducer. These measurements were taken under different operating conditions, allowing researchers to study how operating conditions affect the detection of bearing faults.

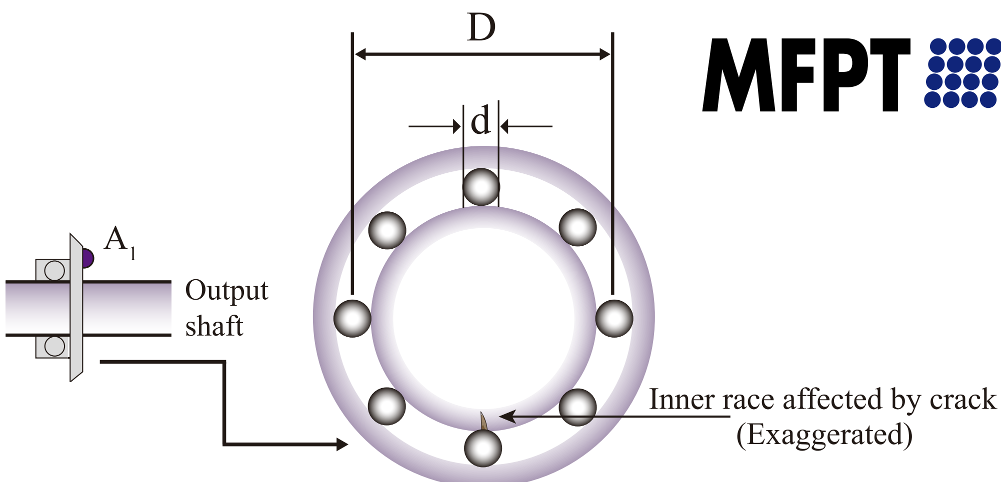


Figure 2.6.1: MFPT Fault Dataset Manipulation

The MFPT dataset has been used in numerous studies on bearing failure diagnosis techniques, including the use of machine learning techniques and the creation of simulation models for early detection of bearing failures. In addition, various methodologies have been developed for processing and analyzing this data, which has enabled researchers to improve the accuracy and efficiency of their diagnostic algorithms.

This data set has been fundamental for the development of predictive maintenance techniques in bearing systems, which has made it possible to prevent equipment failures and reduce maintenance costs. In addition, it has been used in the creation of real-time bearing vibration monitoring systems, which allow machine operators to detect early bearing failures and take preventive measures to avoid catastrophic failures.

In detail, this dataset consists of experimental vibration measurements of ball bearings under different damage states: failures located in the outer race groove, in the inner race groove or in the rolling elements. These measurements consist of accelerations, which can be evidenced by means of vibration analysis when a rolling element passes through one of these failures. In the case of this work, a dataset with three states is used: failure in the outer race, failure in the inner race and nominal health state. [17]

2.6.2. C-MAPSS

NASA's Commercial Modular Aero-Propulsion System Simulation (C-MAPSS) dataset is an important resource for research in the field of aeronautical propulsion system failure monitoring and diagnosis. This dataset was created through turbofan engine simulations and consists of several datasets, each representing a different engine failure scenario.

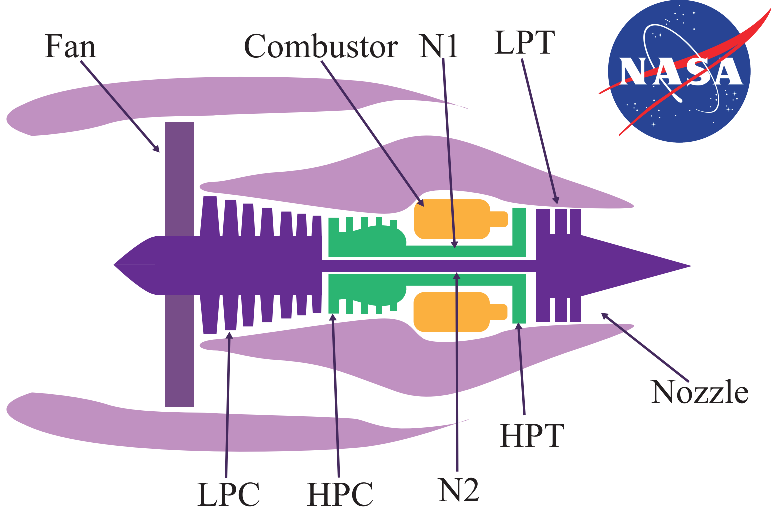


Figure 2.6.2: C-MAPSS Aircraft Engine Simulator

C-MAPSS data sets have been used in a wide range of research, including the development of fault diagnosis techniques using machine learning and the study of the relationship between diagnostic signals and engine failures. In addition, various methodologies have been developed for the analysis and processing of these data, which has enabled researchers to improve the accuracy and efficiency of their algorithms.

These data sets are widely used as a benchmark to evaluate and compare different diagnostic and predictive maintenance techniques in the field of aeronautical engineering. The data include measurements of several engine variables, such as exhaust gas temperature and compressor pressure, which allow researchers to develop diagnostic and failure prediction algorithms. These sets further consist of multiple multivariate time series, where each data set is divided into training and test subsets. Each time series comes from a different engine, i.e., the data can be considered from a fleet of engines of the same type, where each starts with different degrees of initial wear and manufacturing variation that are unknown to the user. This wear and variation is considered normal, i.e., it is not considered a fault condition, in addition there are three operating configurations that have a substantial effect on engine performance, which are also included in the data. The motor operates normally at the beginning of each time series and develops a fault at some point during the series.

In this case, the C-MAPSS dataset used consists of multiple simulations of turbofan

engine degradation under different conditions, both mechanical and operational. In this sense, the dataset is composed of more than 27000 time series of the turbofan sensors that, thanks to the turbofan simulation, could be associated to the RUL of the engine. The features of the dataset are measurements of 21 sensors, at 100 units of the turbofan engine, each one for a number of cycles between 100 and 200. [18]

3 | Methodology

3.1. Selection of case studies and data exploration

The process of searching datasets for predictive maintenance tasks with quantum computing involves several steps, from the selection of search criteria to the evaluation and final selection of the most suitable datasets. This process is described below:

1. Selection of search criteria: The first step in the search for datasets is to define search criteria that fit the specific requirements of the project. These criteria may include aspects such as the availability of detailed information on system performance, data quality, relevance of the dataset to the system to be studied, among others.

2. Dataset search: Once the search criteria have been defined, the search for datasets that meet these criteria begins. This search may involve the review of scientific literature, the consultation of data repositories, the search for information in companies, among others.

3. Dataset evaluation: Once the datasets that satisfy the search criteria have been identified, they are evaluated to determine which are the most suitable for the project. This evaluation may involve reviewing the quality of the data, the availability of detailed information about the system and its maintenance, the relevance of the dataset for the system to be studied, among others.

4. Final dataset selection: Finally, the final selection of the datasets to be used for the project is made. This process must take into account the results of the evaluation of each dataset, as well as the specific needs of the project.

In the present case of the search for datasets for predictive maintenance tasks with quantum computing, the MFPT bearing datasets and the NASA CMAPSS dataset were chosen. These datasets were selected after a careful evaluation that took into account aspects such as data quality, availability of detailed information about the system, relevance of the dataset for mechanical systems and the popularity that these datasets carry being considered benchmark cases.

3.2. Initial data preparation

Data preparation is a fundamental step in the Machine Learning model training process. Before data can be used to train a model, a series of cleaning, transformation and preprocessing tasks need to be performed to ensure that the data is suitable for use in the model. The quality and relevance of the training data has a major impact on the performance and accuracy of the resulting model.

3.2.1. Assembling datasets

It is essential for this work, to properly assemble the datasets that allow the models to be executed with the current resources of quantum computing, without leaving aside that these are useful for the comparison of the models to which they are delivered. The following describes the process of assembling the datasets of this work.

3.2.1.1. MFPT

First, the data to be used to assemble the dataset must be selected. In this case, the following data available on the MFPT website are selected:

- **Baseline condition** at 270 lbs of load, input shaft rate of 25 Hz, sample rate of 97,656 sps, measured for 6 seconds
- **Outer race fault condition** at 25 lbs of load, input shaft rate 25 Hz, sample rate of 48,828 sps, measured for 3 seconds
- **Inner race fault condition** at 50 lbs of load, input shaft rate of 25 Hz, sample rate of 48,828 sps, measured for 3 seconds

Subsequently, 3 seconds of measurement are taken from each of these measured data sets, and the number of samples of each one is standardized, redefining for each of the health states a sample rate of 4882 sps, in order to reduce the large volume of data for the models to be used. To do this, simply the first of every 20 consecutive data of the Baseline condition data is taken, and in the cases of Outer race fault condition and Inner race fault condition the first of every 10 data is taken, thus obtaining data like those of the table 3.1, where the time is in units of second and the rest of the data is in units of acceleration “g”.

Then, we proceed to split the training and test data in a ratio of 80:20 for each of the data sets for each condition. Thus, the following data sets are obtained:

- Train Split Baseline Condition Acelerations
- Test Split Baseline Condition Acelerations
- Train Split Inner Race Fault Condition Acelerations

Table 3.1: MFPT Data after redefining sampling rate and measured time.

Index	**Time**	**Baseline Condition**	**Inner Race Condition**	**Outer Race Condition**
0	0.000000	0.831588	-0.069897	-0.176256
1	0.000205	-1.511027	-1.207542	-0.231410
2	0.000410	1.177196	-0.403396	-1.607862
⋮	⋮	⋮	⋮	⋮
14646	2.999508	-0.912831	-0.911982	0.720799
14647	2.999713	-0.917827	-0.895886	-1.212071
14648	2.999918	-0.416342	-0.830530	-0.005353

- Test Split Inner Race Fault Condition Acelerations
- Train Split Outer Race Fault Condition Acelerations
- Test Split Outer Race Fault Condition Acelerations

Then, small subsamples of data are obtained in these new sets, for this, they are grouped in temporal windows of 96 samples with an overlap of 48 samples, that is to say, half of the data of each window is shared with its contiguous windows. In this way, windows like the one in table 3.2 are obtained, where an example is shown for the case of the Baseline Condition training data.

Table 3.2: MFPT Data Example after grouping in time windows.

Time Windows with Train Healthy Condition Data					
Window Index	**1st Window Component**	**2nd Window Component**	**...**	**95th Window Component**	**96th Window Component**
0	[0.8315881	-1.5110275	...	-1.9110523	-0.1569234]
1	[0.06379319	0.4264816	...	-1.0001166	-0.8816726]
⋮	⋮	⋮	⋱	⋮	⋮
241	[-0.8036293	-1.8802995	...	0.2404058	-0.8571472]
242	[0.6064907	0.5147141	...	0.2293921	0.1730686]

Subsequently, all windows of all data groups are stacked regardless of their health condition and a label corresponding to their condition is added, represented by 0 for Healthy health status, 1 for Inner Race failure and 2 for Outter Race failure, as exemplified in table 3.3.

Table 3.3: MFPT Data Example after stacking time windows for every health condition.

MFPT Labeled Time Windows Dataset Example			
Index	**Window**	**Label**	**Condition**
0	[0.831588 , -1.511027 , ... , -1.911052 , -0.156923]	0	Healthy
1	[0.063793 , 0.426481 , ... , -1.000116 , -0.881672]	0	Healthy
⋮	⋮	⋮	⋮
243	[-0.069896 , -1.207542 , ... , 0.094397 , -0.618919]	0	Healthy
244	[2.882126 , 1.810279 , ... , -0.481353 , -0.453111]	1	Outer Race
⋮	⋮	⋮	⋮
485	[-1.087795 , -0.390770 , ... , -0.867816 , -0.490348]	1	Outer Race
486	[-0.176255 , -0.231410 , ... , -0.117901 , 2.098567]	2	Inner Race
⋮	⋮	⋮	⋮
727	[-0.157723 , -0.399753 , ... , -5.339973 , 2.405596]	2	Inner Race
728	[0.164306 , -1.747766 , ... , -0.577141 , -2.264203]	2	Inner Race

Then, temporal parameters are calculated to be used as features of the final dataset with which the models are worked. The mean, variance, rms, peak, valley, peak2peak, crest factor, kurtosis and skewness are calculated for each Time Windows, thus obtaining a dataset like the one in table 3.4.

Table 3.4: MFPT Data Example after calculating new features.

MFPT Labeled Time Windows Dataset Example										
Index	**Mean**	**Variance**	**RMS**	**Peak**	**Valley**	**Peak to peak**	**Crest factor**	**Kurtosis**	**Skewness**	**Label**
0	-0.137602	0.964180	0.991521	2.287067	-1.911052	4.198119	2.306623	-0.694730	0.150808	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
244	-0.057800	1.064843	1.033530	2.882126	-2.433998	5.316124	2.788623	0.897920	0.617413	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
728	-0.139224	3.448500	1.862225	9.495661	-5.339973	14.835634	5.099093	8.822974	2.148120	2

3.2.1.2. C-MAPSS

The C-MAPSS Aircraft Engine Simulator contains different datasets according to the operational conditions and failure modes of the turbofan engine. These conditions are summarized in the table 3.5, and for this work we choose to work with the dataset FD001 because it is a smaller and more specific subset, although it is still representative for a classification task.

Table 3.5: C-MAPSS Raw Datasets Conditions.

Dataset	**Train trajectories**	**Test trajectories**	**Amount of operational condition**	**Amount of fault modes**
FD001	100	100	One (Sea level)	ONE (HPC Degradation)
FD002	260	259	Six	ONE (HPC Degradation)
FD003	100	100	One (Sea level)	TWO (HPC and Fan Degradation)
FD004	248	249	Six	TWO (HPC and Fan Degradation)

After choosing the dataset FD001, which is seen as in the table 3.6, it is necessary to understand the content of the columns of the dataset, for this the headers of the 21 sensors of the dataset are labeled, among which are: [1]

1. Fan Inlet Temperature
2. LPC Outlet Temperature
3. HPC Outlet Temperature
4. LPT Outlet Temperature
5. Fan Inlet Pressure
6. Bypass Duct Pressure
7. HPC Outlet Pressure
8. Physical Fan Speed
9. Physical Core Speed
10. Engine Pressure Ratio
11. HPC Outlet Static Pressure
12. Fuel Flow Ratio
13. Corrected Fan Speed
14. Corrected Core Speed
15. Bypass Ratio
16. Fuel Burner Air Ratio
17. Bleed Enthalpy
18. Demanded Fan Speed
19. Demanded Fan Conversion Speed
20. HPT Coolant Bleed
21. LPT Coolant Bleed

Table 3.6: C-MAPSS FD001 Raw Dataset

Index	**Unit**	**Cycles**	**Altitude**	**Air Speed**	**Throttle Angle**	**Sensor 1: Fain inlet temperature**	**Sensor 2: LPC outlet temperature**	**...**	**Sensor 20: HPT Coolant bleed**	**Sensor 21: LPT Coolant bleed**
0	1	1	-0.007	-0.0004	100	518.67	641.82	...	39.06	23.419
1	1	2	0.0019	-0.0003	100	518.67	642.15	...	39.00	23.423
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
191	1	192	0.0003	0.0002	100	518.67	643.38	...	38.32	23.415
192	2	1	0.0012	-0.0007	100	518.67	642.32	...	39.14	22.973
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
478	2	287	-0.003	0.0006	100	518.67	643.21	...	38.74	23.552
479	3	1	0.015	0.0002	100	518.67	642.53	...	38.93	23.165
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
20630	100	200	0.001	-0.0002	100	518.67	643.85	...	38.37	23.0522

3.2.1.3. Sensor selection

Although the C-MAPSS simulator contains a dataset with the actual RULs corresponding to each unit, in this step an estimated RUL is calculated by subtracting from the maximum cycles present in the dataset, which is 200 for unit 100, the cycle in which each data is found. Thus, the RUL is added as an additional feature in this step such as the table 3.7. Once obtained the RUL colume, with the method *Dataframe.corr()* the correlation of each of the columns of the dataset with the estimated RUL is obtained, thus eliminating those columns that do not have a strong correlation with the RUL, decreasing the number of features, considering operating conditions and sensor measurements, from 25 to 12.

[1] Where LPC is Low Pressure Compressor, HPC is High Pressure Compressor, LPT is Low Pressure Turbine, HPT is High Pressure Turbine

Table 3.7: C-MAPSS FD001 Raw Dataset with Estimated RUL

Index	Unit	Cycles	Altitude	Air Speed	Throttle Angle	Sensor 1: Fain inlet temperature	...	Sensor 21: LPT Coolant bleed	Estimated RUL
0	1	1	-0.007	-0.0004	100	518.67	...	23.419	199 cycles
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
20630	100	200	0.001	-0.0002	100	518.67	...	23.0522	0 cycles

The deleted columns correspond to Altitude, Air speed, Throttle Angle, Fan Inlet Temperature, Fan Inlet Pressure, Bypass Duct Pressure, Physical Core Speed, Engine Pressure Ratio, Corrected Core Speed, Fuel Burner Air Ratio, Demanded Fan Speed and Demanded Fan Conversion Speed. In addition, the column generated from the estimated RUL is also detached, resulting in a dataset like the one in the table below. 3.8.

Table 3.8: C-MAPSS FD001 Dataset after columns droping.

Index	Unit	Cycles	Sensor 1: LPC Oulet temperature	...	Sensor 13: LPT Coolant bleed
0	1	1	641.82	...	23.419
⋮	⋮	⋮	⋮	⋱	⋮
20630	100	200	643.85	...	23.0522

Once the number of features in the dataset is reduced with the sensor measurements, further dimensionality reduction is performed. This time using the Scikit Learn PCA method. First, the variances of each of the features of the dataset are calculated and it is decided to reduce the number of features according to the information of how many features obtain a variance of 0.95 or higher, thus reducing the features of the dataset as in the table below 3.9.

Table 3.9: C-MAPSS FD001 Dataset After PCA Feature Reduction

Index	PCA Feature 1	PCA Feature 2	PCA Feature 3	PCA Feature 4
0	-109.9947	46.1568	2.1783	3.5121
⋮	⋮	⋮	⋮	⋮
20630	95.1002	-44.5401	16.9480	-2.8483

Subsequently, the actual RUL data are loaded for each engine unit from the FD001 dataset. This dataset contains the real RUL because it comes from a simulation, which is useful for a more accurate training. In this case, a dataset of the same length as that of the sensors is built, but with the RUL per unit, thus obtaining a dataset like the one in the table below 3.10.

Table 3.10: C-MAPSS FD001 RUL Dataset

Index	RUL
0	112
⋮	⋮
20630	20

To the last two datasets of tables 3.9 and 3.10, *MinMaxScaler()* are performed to normalize their data and proceed to assemble the first dataset for classification, present in table 3.11, joining both datasets but replacing the RUL by health status labels, where a RUL greater than 0.4 is considered as a Healthy health status, and a RUL less than 0.4 is considered as a Degraded health status.

Table 3.11: C-MAPSS FD001 PCA Features from Sensors Dataset for Classification

Index	PCA Feature 1	PCA Feature 2	PCA Feature 3	PCA Feature 4	Label
0	-109.9947	46.1568	2.1783	3.5121	Healthy
⋮	⋮	⋮	⋮	⋮	⋮
20630	95.1002	-44.5401	16.9480	-2.8483	Degraded

AAdditionally, another dataset is constructed by calculating temporal parameters using as time windows the cycles of each unit. The parameters to be calculated are the average and the standard deviation for each sensor whose correlation with the RUL turns out to be strong, i.e. from the dataset of table 3.8, obtaining a dataset like the one in table 3.12.

Table 3.12: C-MAPSS FD001 Statistical Features Dataset

Index	Feature 1: LPC Outlet Temperature Mean	Feature 2: LPC Outlet Temperature Standar Deviation	...	Feature 25: LPT Coolant Bleed Mean	Featuer 26: LPT Coolant Bleed Standar Deviation
0	642.6210	0.4867	...	23.3063	0.1051
⋮	⋮	⋮	⋱	⋮	⋮
100	642.7413	0.4604	...	23.2693	0.1028

Finally, *MinMaxScaler()* is also performed to normalize the data, PCA reduction to reduce the features, and the second dataset is assembled for the classification, present in the table 3.13, joining the dataset with statistical features with the RUL dataset, this time by units, and replacing the RUL by health status labels, where again a RUL greater than 0.4 is considered as a Healthy health status, and a RUL less than 0.4 is considered as a Degraded health status.

Table 3.13: C-MAPSS FD001 Statistical Features after PCA Reduction with Labels

Index	PCA Feature 1	PCA Feature 2	PCA Feature 3	PCA Feature 4	PCA Feature 5	Label
0	0.3909	-0.4672	0.5142	0.0955	-0.0672	Healthy
⋮	⋮	⋮	⋱	⋮	⋮	⋮
100	-0.2171	-0.3251	-0.0494	-0.2178	0.0457	Degraded

All this dataset processing is also performed on the test data present in the datasets provided by NASA.

3.3. Selection of used models

In this work, a study on the use of quantum machine learning models for predictive maintenance is carried out. For this purpose, a rigorous selection of quantum models was carried out, based on technical and scientific criteria, to determine which were the most suitable to address the classification tasks posed in the selected datasets.

After a thorough review of the scientific literature, four IBM Qiskit quantum machine learning models were chosen, namely: Variational Quantum Classifier, Quantum Neural Network Classifier, Quantum Neural Network Classifier using Torchconector and Quantum Support Vector Classifier.

In addition, a classical model for comparison was selected: the Support Vector Classifier present in Scikit Learn. This choice made it possible to contrast the results obtained by the quantum models with a classical model widely used in the scientific community.

It is important to note that one of the main challenges faced in this work is the limited availability of online quantum computing resources through IBM Quantum Lab, providing only hardware with a small number of qubits. Therefore, simulations were performed in Qiskit to train and evaluate the performance of quantum models with more qubits in which to encode dataset features. This approach was considered appropriate, since it allowed the training and evaluation of the selected quantum models to be carried out.

3.4. Implement classical model to cases of study

The implementation of the classical model chosen in this work is quite simple. This is because the Support Vector Classifier method is available in the Scikit Learn library.

In particular, default parameters are used, which are:

- Regularization parameter as C = 1.0
- Radial basis function Kernel

In this way, the model is trained by means of the method *.fit(X,y)* using the training sets of each case study. Then the evaluation of the model is performed with both the training and testing datasets to obtain the evaluation metrics.

Finally, the actual labels of the datasets are compared with the predicted labels using the *.predict(X)* method and the confusion matrices are made using the functions available in Scikit Learn.

3.5. Implement quantum models to cases of study.

This section explains how the four quantum machine learning models chosen in this work are implemented: Variational Quantum Classifier (VQC), QNN Classifier, QNN Classifier with TorchConnector and Quantum Support Vector Classifier (QSVC). The methodology used to implement each model is described in detail, including the design of quantum circuits, the selection of optimization algorithms and the encoding of input data.

3.5.1. Variational Quantum Classifier

First, the datasets of each of the case studies must be loaded, since this model is implemented to perform classification tasks with both case studies. Subsequently, the volume of data to be used must be adjusted because the training cannot be completed with very large datasets, which is why in the case of the dataset with the PCA features of the C-MAPSS FD001 sensors, 50% of the data is taken, using the function *train_test_split()* from Scikit Learn. On the other hand, for the MPFT classification case, it is not necessary to make this cut in the dataset.

Subsequently, we start to build the quantum circuit to be used in this model. First of all, we choose how to encode the data features to the qubits by means of a Feature Map. In this step, the number of repetitions of the FeatureMap circuit is chosen, generally between 1 and 3 repetitions, and the Feature Map is selected from the following options:

1. ZZFeatureMap
2. ZFeatureMap
3. RawFeatureVector

The number of qubits of the circuit is specified, in the cases of ZZFeatureMap and ZFeatureMap, the number of qubit is chosen as the number of features of the dataset. On the other hand, in the case of RawFeatureVector, the function defines the number of qubits as the logarithm base 2 of the number of features of the dataset, so if the dataset contains a number of features that is not in the domain of the $log_2()$ function, a PCA reduction is performed to the nearest value that meets this constraint.

Subsequently, the choice of the circuit for the Ansatz is made, where its number of input qubits must match the output qubits of the FeatureMap. We must also specify how many times the Ansatz circuit is repeated in this step, which is typically set between 1 and 3 repetitions. The circuits from which the Ansatz are chosen are as follows:

1. RealAmplitudes
2. EfficientSU2
3. RawFeatureVector

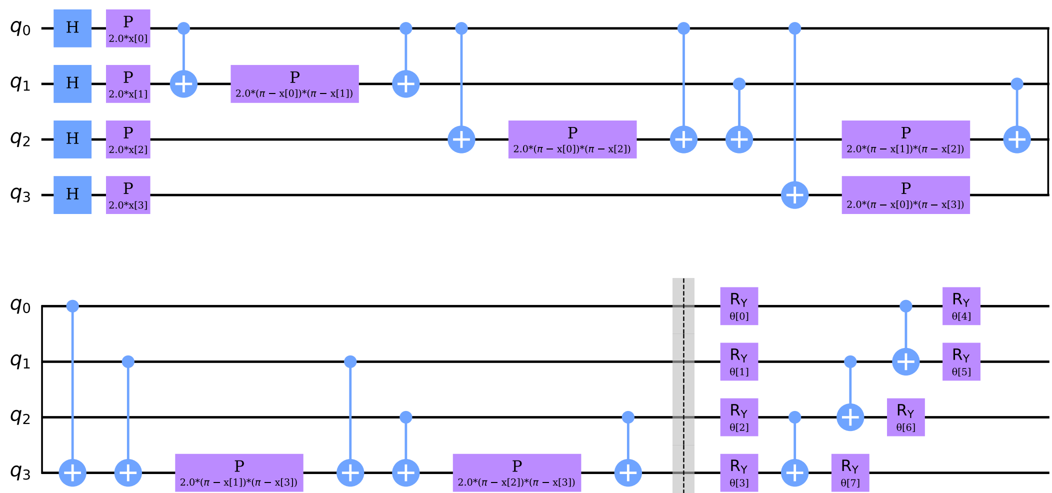


Figure 3.5.1: VQC FeatureMap and Ansatz Example

The complete model circuit can be visualized as in the example in figure 3.5.1, where a ZZFeatureMap is used to encode the data and a RealAmplitudes as Ansatz.

Then, the optimizer to be used is selected, typically the COBYLA optimizer present in the Qiskit library, and the maximum number of iterations of the optimizer is specified, which is typically chosen between 100 and 200 iterations.

In the next case it is defined where the classifier is trained, in a simulator or in a real quantum computer. Due to the limitation in getting real hardware with enough memory and qubits for the features, we choose to use simulators. In particular, *AerSimulator* is used and initialized by the function *QuantumInstance()*.

The last step before training the model is to define a callback function, which VQC will use to evaluate the objective function at each iteration in the iteration process. This function is defined with the weights and the values of the objective function in those weights as inputs, generating a plot like the one in figure 3.5.2 during the training of the model.

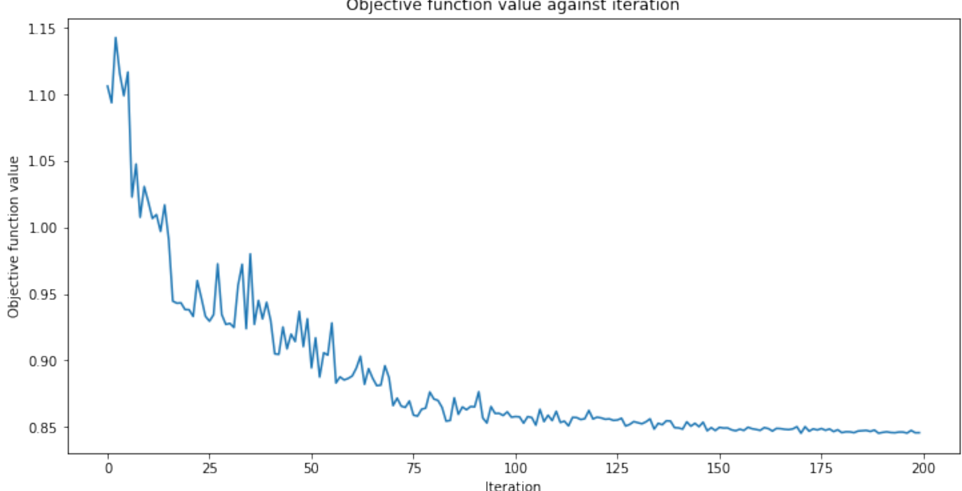


Figure 3.5.2: VQC Callback Graph Example

Subsequently, the function is used to build the Variational Quantum Classifier in Qiskit, providing the assembled circuit, the optimizer, the simulator and the callback function to perform the training. Once the training is finished, we proceed to calculate the different evaluation metrics and to plot the confusion matrices, both with the train split and the test split for each case study.

3.5.2. Quantum Neural Network Classifier

First, like the previous model, it is necessary to load the case study datasets, however, in this model only the C-MAPSS dataset is used since the model is used for classification in both cases. Then, the size of the dataset used for training must be adjusted due to the fact that training cannot be completed with very large datasets. As in the previous model, the C-MAPSS FD001 sensor dataset is reduced to use the 50% of the data using the function*train_test_split* from Scikit Learn.

Then, as in VQC, quantum circuits and the number of Qubits for the FeatureMap and Ansatz as well as an optimizer must be selected. For the Feature Map, typically the same as for the VQC is used, as well as for the Ansatz, and for the optimizer, typically COBYLA is used. Subsequently, a quantum neural network is created from the quantum circuit, typically with Qiskit's *EstimatorQNNN* network, which receives the quantum circuit, and the ansatz parameters to be used as the neural network weights.

As in the previous model, the last step before training the model is to define a callback function. This function is defined with the weights and the values of the objective function in those weights as inputs, generating a plot like the one in figure 3.5.2 during the training of the model.

The model is trained using the method *.fit(X,y)*, with the training sets for each case

study. The model is then evaluated using both the training sets and the test sets to obtain the corresponding evaluation metrics.

Finally, the actual labels of the datasets are compared with the predicted labels using the method *.predict(X)*, and the confusion matrices are created using the functions provided by Scikit Learn.

3.5.3. Quantum Neural Network with Pytorch Classifier

For this model we follow similar to the previous models, however, the optimization and training routine is performed using the Pytorch library methods. As in all models, the first step is the data import, in this case the C-MAPSS datasets are used, from which a 50% of data is taken to speed up the training routine. In this case, the labels must be normalized according to the type of neural network to be used, we have the following cases:

1. For the CircuitQNN network, labels are required to be 0 or 1.
2. For the OpflowQNN network, labels are required to be -1 or +1.

Subsequently, the training and test data must be encoded as Pytorch tensors by means of the function *Tensor()*

To continue, the neural network must be built. To do so, the network to be used and the circuit with which it will be built are chosen, as in the previous models:

1. ZZFeatureMap, ZFeatureMap or RawFeatureVector
2. RealAmplitudes, EfficientSU2 or RawFeatureVector

The difference with respect to the previous model lies in this point, because now a Pytorch optimization routine is used, typically the LBFGS optimizer with a MSELoss loss function, which is used to initialize the TorchConnector to subsequently perform the training using the TorchConnector's *.train()* method.

After training is completed, various evaluation metrics are calculated and corresponding confusion matrices are created for both the training set and the test set for each case study. These matrices are represented visually by plots as in the previous models.

3.5.4. Quantum Support Vector Classifier

The first step for this model is the import of the data, in this case the C-MAPSS datasets are used, from which a 50% of data is taken to speed up the training routine.

In this case, the data must be *MinMaxScaler* in a range between 0 and π to ensure the compatibility of the encoding of this method.

With the dataset used, we have 4 features, so 4 qubits will be used in the circuit. In addition we must specify the number of steps performed in the training, typically 100 steps are used. In addition we must specify the value of a hyperparameter which acts as a positive regularization parameter, where a small value of this parameter means lower training weights and prevents overfitting, while a large value improves the training performance drastically, typically a regularization parameter of 1000 is used.

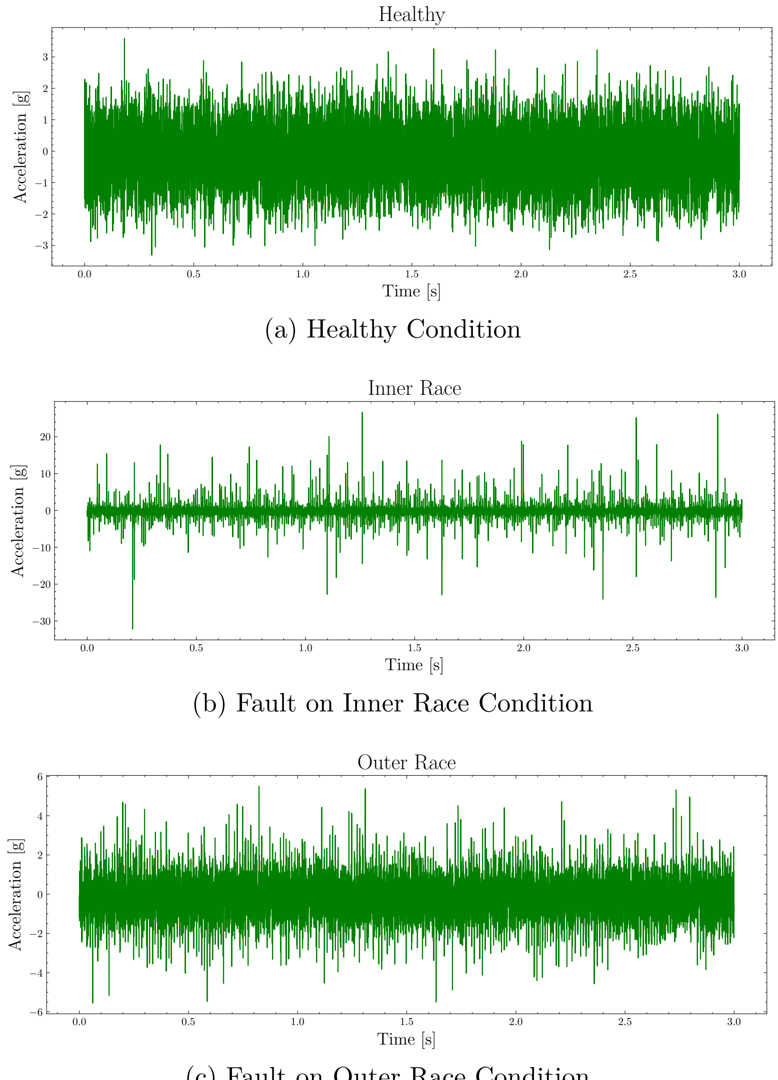
Then, a FeatureMap is chosen for the quantum circuit, among which can be ZFeatureMap or ZZFeatureMap. Once this is defined, we proceed to transform this quantum circuit into a QuantumKernel that is able to separate the classes of the dataset using the function *FidelityQuantumKernel* of Qiskit.

Subsequently, this Quantum Kernel is delivered to the function *PegasosQSVC* of Qiskit which is compatible with all the interface of a Scikit Learn SVC as the one used in the classical classification model. Finally this model is trained using the *.fit* method and the model is evaluated with both the train split and the split test together with the confusion matrices.

4 | Results

4.1. Data processing results

The graphs obtained during the process of preparing the datasets to be used in the models are presented below. First, the raw acceleration signals of the constructed MFPT dataset are presented in Figure 4.1.1.



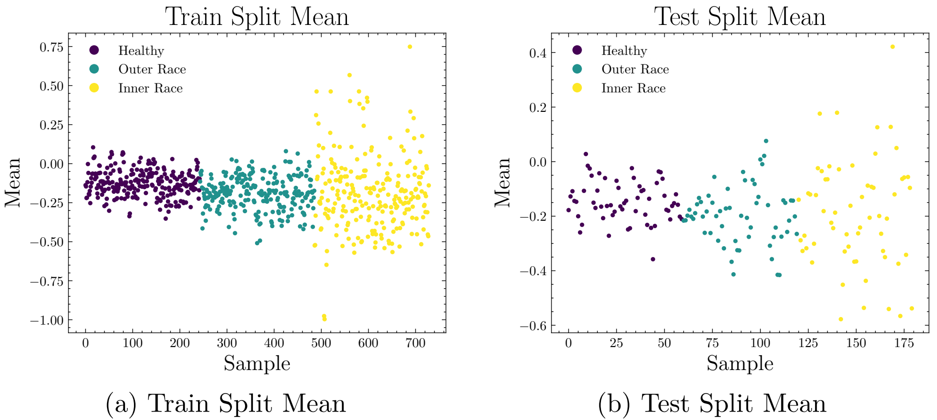
(a) Healthy Condition

(b) Fault on Inner Race Condition

(c) Fault on Outer Race Condition

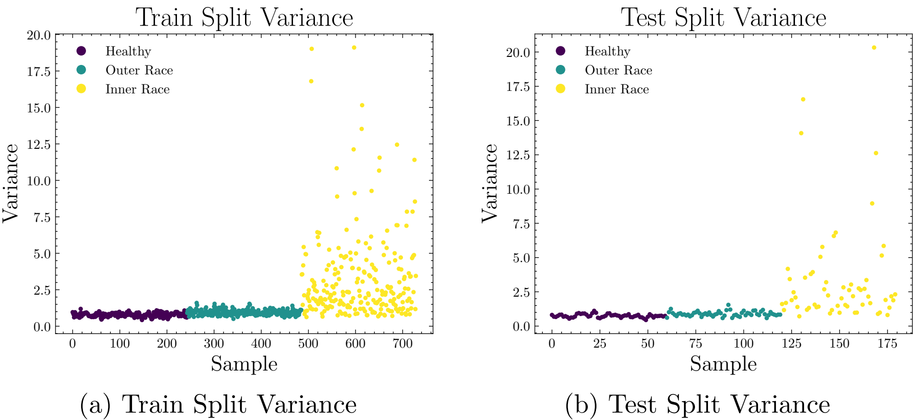
Figure 4.1.1: MFPT Raw Dataset Plot

Subsequently, from Figures 4.1.2 to 4.1.4 are some of the feature plots calculated for the MFPT dataset, the plots of all the calculated features can be found in Annex A.



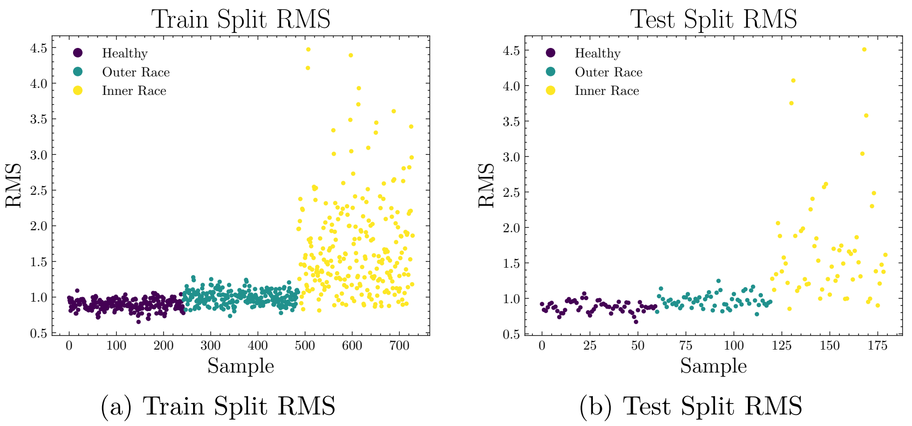
(a) Train Split Mean (b) Test Split Mean

Figure 4.1.2: MFPT Dataset Mean Plot



(a) Train Split Variance (b) Test Split Variance

Figure 4.1.3: MFPT Dataset Variance Plot



(a) Train Split RMS (b) Test Split RMS

Figure 4.1.4: MFPT Dataset RMS Plot

Another important result during the data processing process, this time of the C-

MAPSS FD001 data, is the correlation matrix of features in Figure 4.1.5.

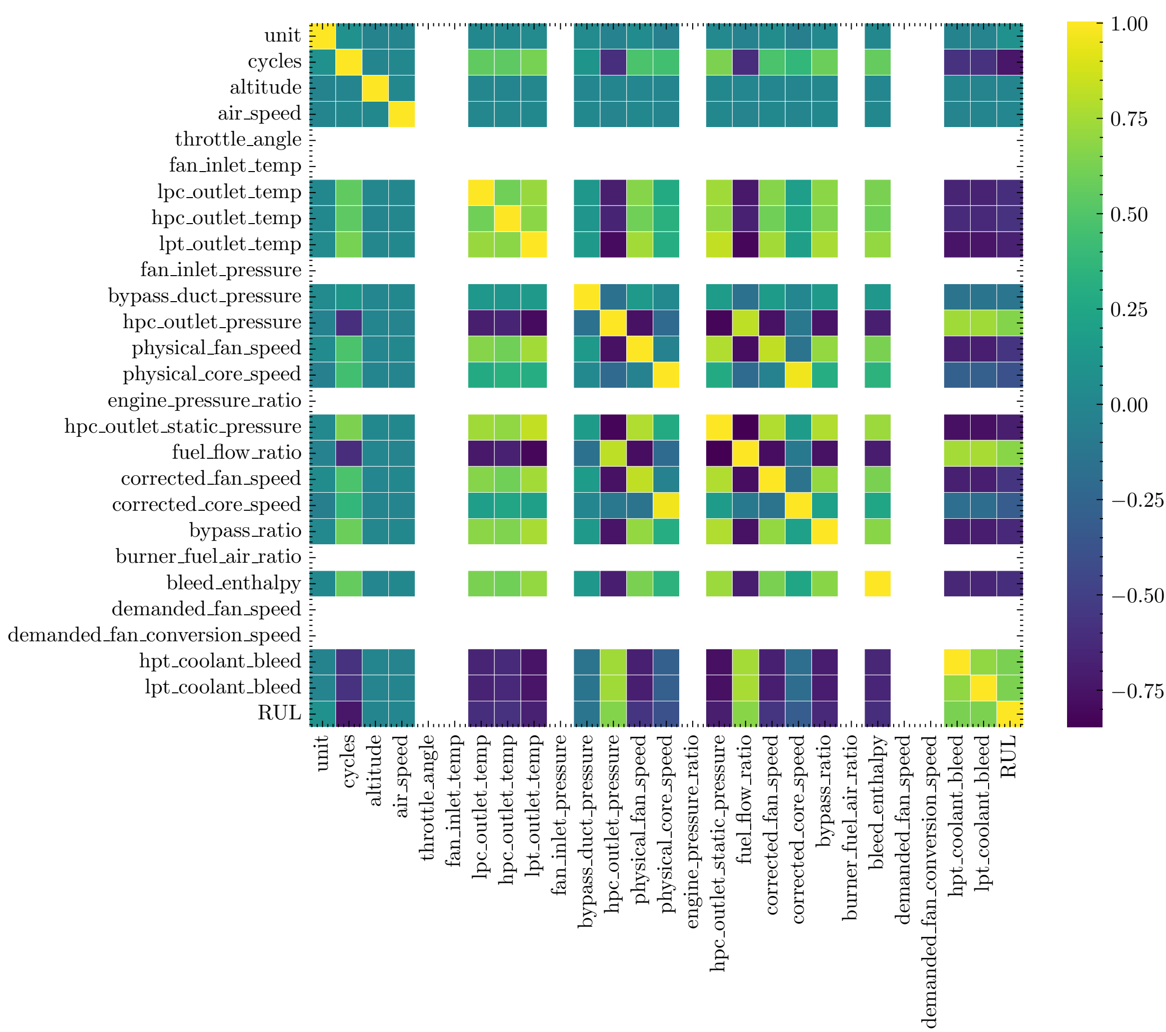
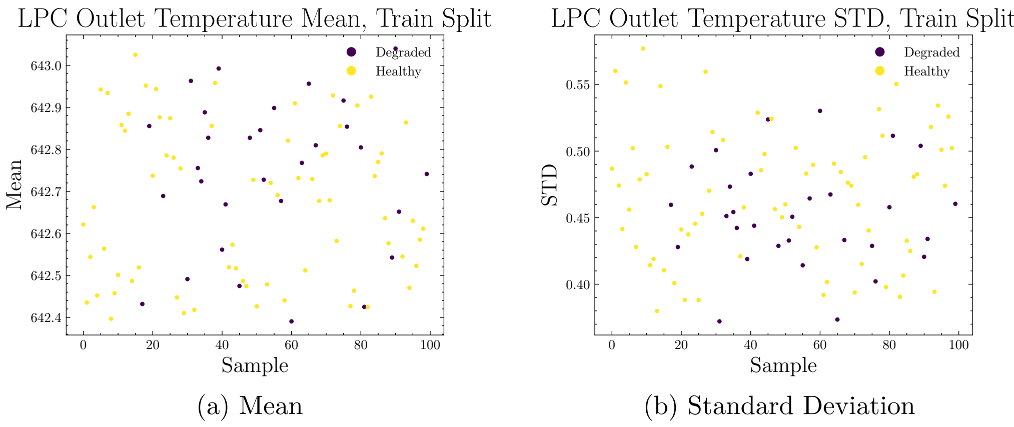


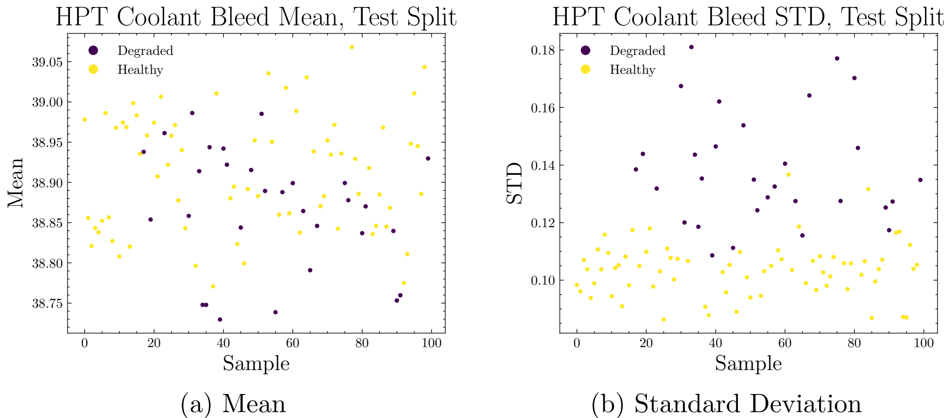
Figure 4.1.5: C-MAPSS Aircraft Correlation Matrix Between Features

Subsequently, in Figures 4.1.6 and 4.1.7 are some of the feature plots calculated for the C-MAPSS FD001 dataset by cycles of each unit, the plots of the totality of the calculated features are in Annex A.



(a) Mean (b) Standard Deviation

Figure 4.1.6: C-MAPSS FD001 Train Dataset LPC Outlet Temperature Plots



(a) Mean (b) Standard Deviation

Figure 4.1.7: C-MAPSS FD001 Test Dataset HPT Coolant Bleed Plots

4.1.1. Dimensionality reduction

The results of the variances of the three datasets at the time when each one is subjected to feature reduction with PCA are shown in the table 4.1.

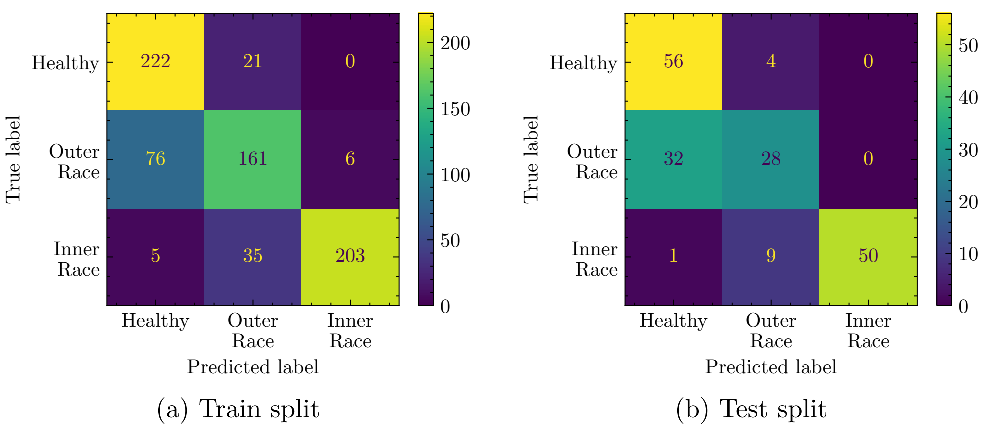
Table 4.1: Datasets PCA Reduction Results

Dataset	**PCA Features Quantity**	**Variance Reached**
MFPT	8	99.99%
C-MAPSS Sensor Features	4	98.36%
C-MAPSS Statistical Features	5	95.88%

4.2. Classical Support Vector Classifier Results

The confusion matrices and evaluation metrics for each of the case studies using Classical Support Vector Classifier are presented below.

4.2.1. MFPT Fault Dataset Manipulation



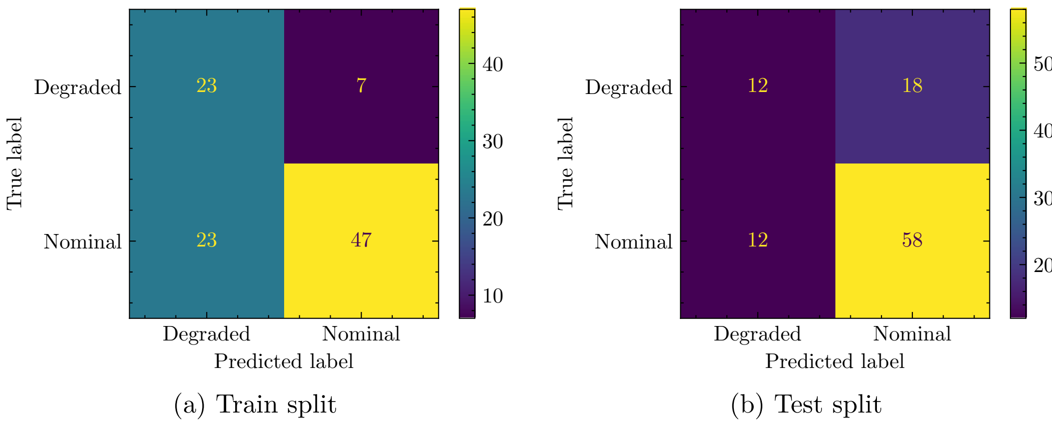
(a) Train split

(b) Test split

Figure 4.2.1: Confussion Matrixes for Classical SVC applied to MFPT

4.2.2. C-MAPSS Aircraft Engine Simulator

4.2.2.1. FD001 Statistics Features

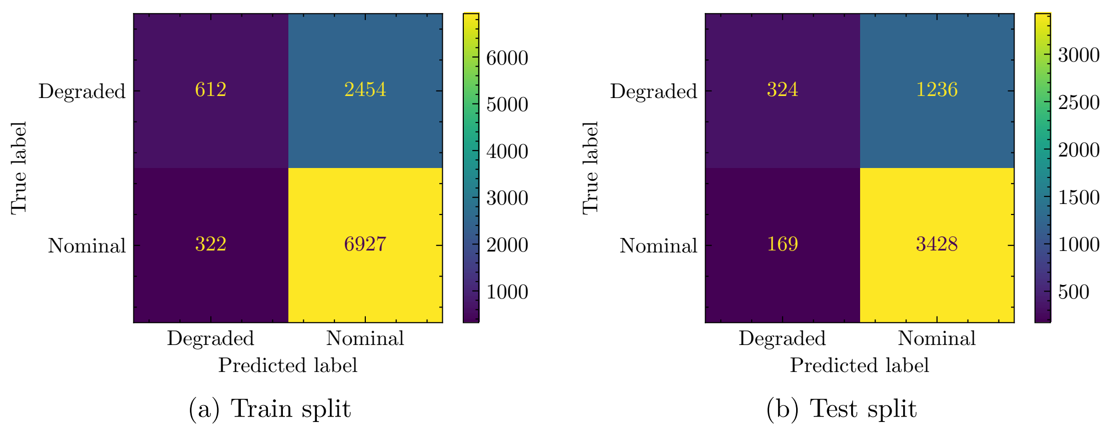


(a) Train split

(b) Test split

Figure 4.2.2: Confussion Matrixes for Classical SVC applied to C-MAPSS FD001 with Statistics Features

4.2.2.2. FD001 Sensors Features



(a) Train split

(b) Test split

Figure 4.2.3: Confussion Matrixes for Classical SVC applied to C-MAPSS FD001 with Statistics Features

The results of the model applied to the datasets are shown in table 4.2.

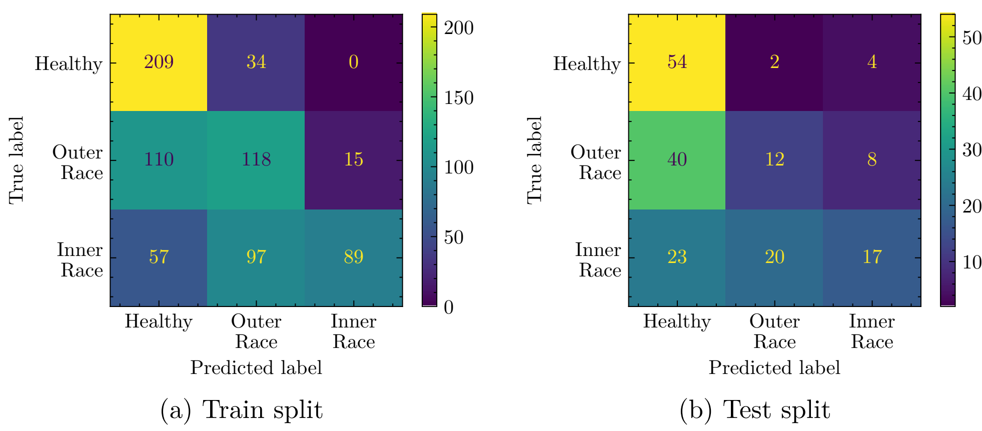
Table 4.2: Classical Predictions Results

Classical SVC Results													
Dataset	**Accuracy**		**Precision**		**Recall**		**Specificity**		**NPV**		**Training Time**		
	Train	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	**Jupyter Notebook Intel i7 2.20GHz 8 GB RAM**	**IBM Quantum Lab Intel Xeon 2.60GHz 31 GB RAM**	**Google Colab Intel Xeon 2.20 Ghz 13 GB RAM**
MFPT Statistical Features	0.80	0.74	-	-	-	-	-	-	-	-	0.017 seconds	0.012 seconds	0.009 seconds
C-MAPSS FD001 Statistical Features	0.70	0.70	0.77	0.40	0.50	0.50	0.87	0.76	0.67	0.83	0.001 seconds	0.002 seconds	0.002 seconds
C-MAPSS FD001 Sensor Features	0.73	0.73	0.20	0.21	0.66	0.66	0.74	0.73	0.96	0.95	2.966 seconds	3.987 seconds	4.171 seconds

4.3. Variational Quantum Classifier Results

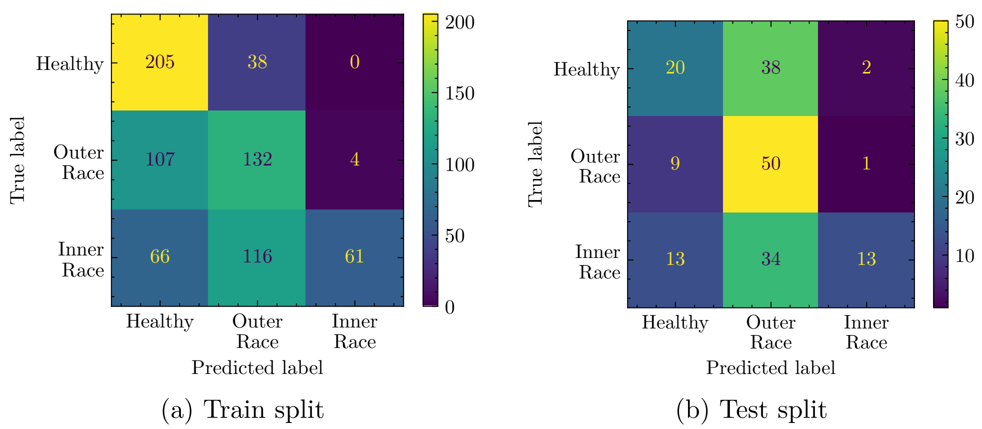
Some of the confusion matrices and evaluation metrics for each of the case studies using Variational Quantum Classifier are presented below. The full set of confusion matrices can be found in the Annex B.

4.3.1. MFPT Fault Dataset Manipulation



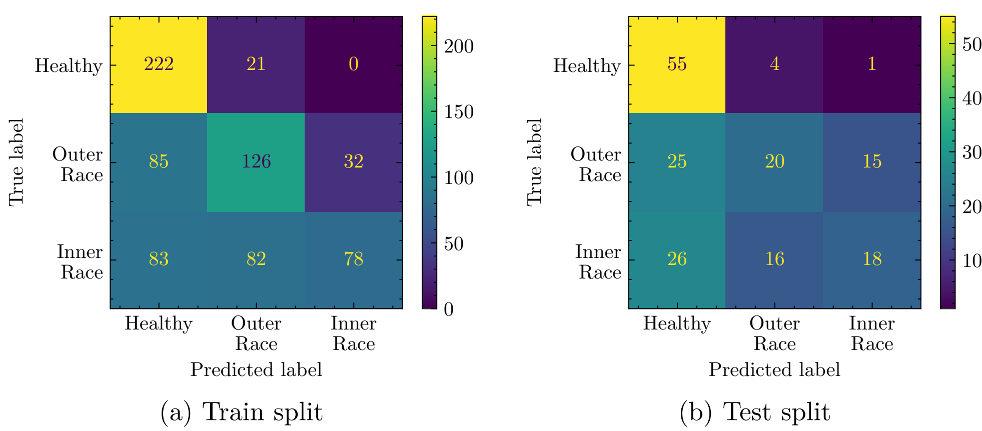
(a) Train split (b) Test split

Figure 4.3.1: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to MFPT



(a) Train split (b) Test split

Figure 4.3.2: Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and COBYLA applied to MFPT



(a) Train split (b) Test split

Figure 4.3.3: Confussion Matrixes for VQC with RawFeatures, EfficientSU2 and COBYLA applied to MFPT

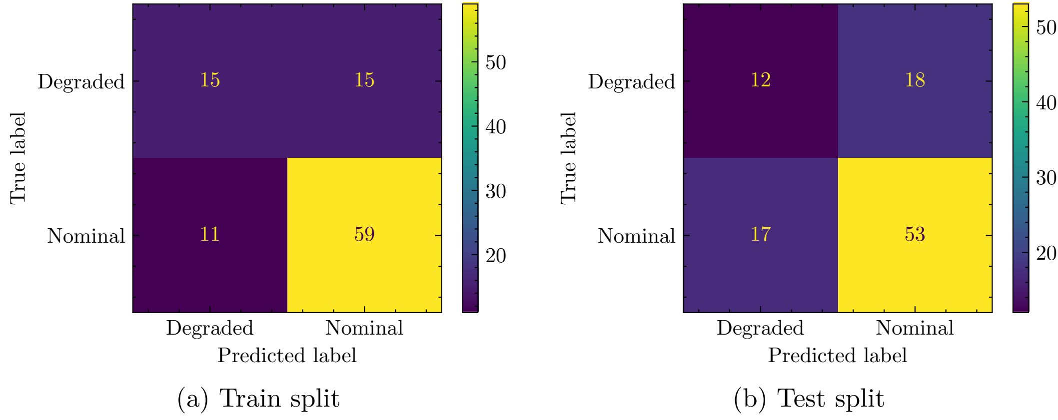
Table 4.3 shows the results of the model applied for this dataset.

Table 4.3: VQC for MPFT Dataset Results

VQC for MFPT Dataset. Executed on IBM Quantum Lab, 31 GB RAM					
Feature Map	Ansatz	Optimizer	Accuracy		Training Time
			Train	Test	
ZZFeatureMap	RealAmplitudes	COBYLA	0.55	0.46	7384 seconds
ZZFeatureMap	EfficientSU2	COBYLA	0.57	0.46	8340 seconds
ZFeatureMap	RealAmplitudes	COBYLA	0.56	0.47	5417 seconds
ZFeatureMap	EfficientSU2	COBYLA	0.59	0.47	6118 seconds
RawFeatures	RealAmplitudes	COBYLA	0.55	0.48	6296 seconds
RawFeatures	EfficientSU2	COBYLA	0.58	0.42	7112 seconds
ZZFeatureMap	RealAmplitudes	SLSQP	0.55	0.47	18892 seconds
ZZFeatureMap	EfficientSU2	SLSQP	0.56	0.46	21337 seconds
ZFeatureMap	RealAmplitudes	SLSQP	0.56	0.45	13859 seconds
ZFeatureMap	EfficientSU2	SLSQP	0.60	0.46	15652 seconds
RawFeatures	RealAmplitudes	SLSQP	0.56	0.48	16109 seconds
RawFeatures	EfficientSU2	SLSQP	0.58	0.53	18195 seconds
ZZFeatureMap	RealAmplitudes	SPSA	0.64	0.54	17242 seconds
ZZFeatureMap	EfficientSU2	SPSA	0.67	0.54	22418 seconds
ZFeatureMap	RealAmplitudes	SPSA	0.66	0.56	14561 seconds
ZFeatureMap	EfficientSU2	SPSA	0.68	0.55	15572 seconds
RawFeatures	RealAmplitudes	SPSA	0.61	0.54	17004 seconds
RawFeatures	EfficientSU2	SPSA	0.65	0.59	19205 seconds

4.3.2. C-MAPSS Aircraft Engine Simulator

4.3.2.1. FD001 Statistics Features



(a) Train split

(b) Test split

Figure 4.3.4: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Statistics Features

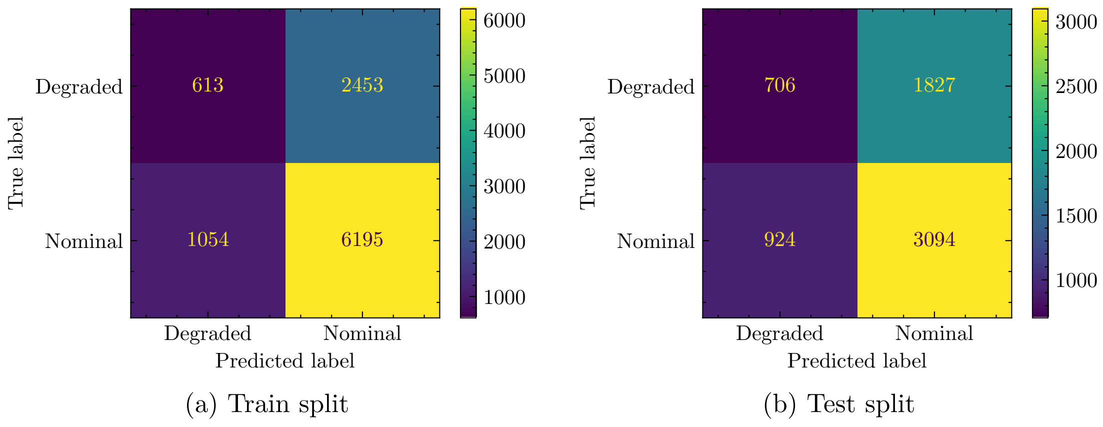


Figure 4.3.5: Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Sensor Features

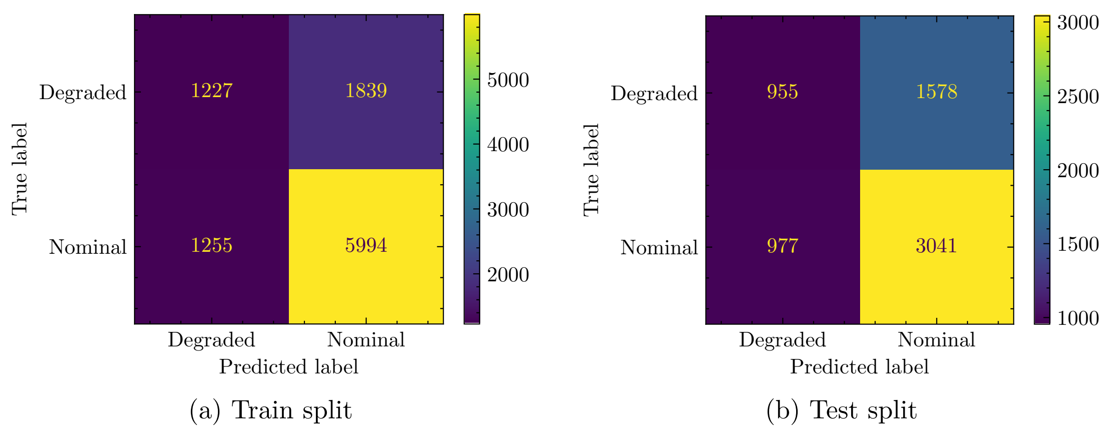


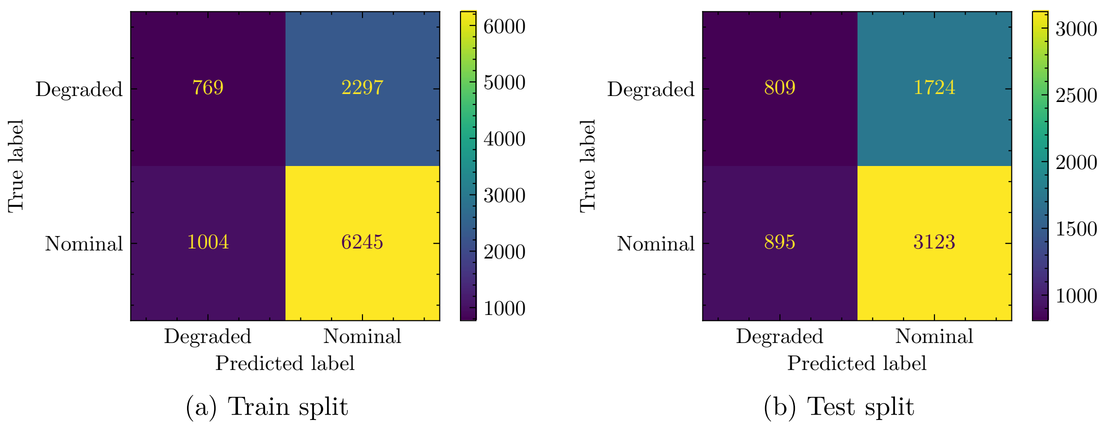
Figure 4.3.6: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Sensor Features

Table 4.4 shows the results of the model applied for this dataset.

Table 4.4: VQC for C-MAPSS FD001 with Statistics Features Results

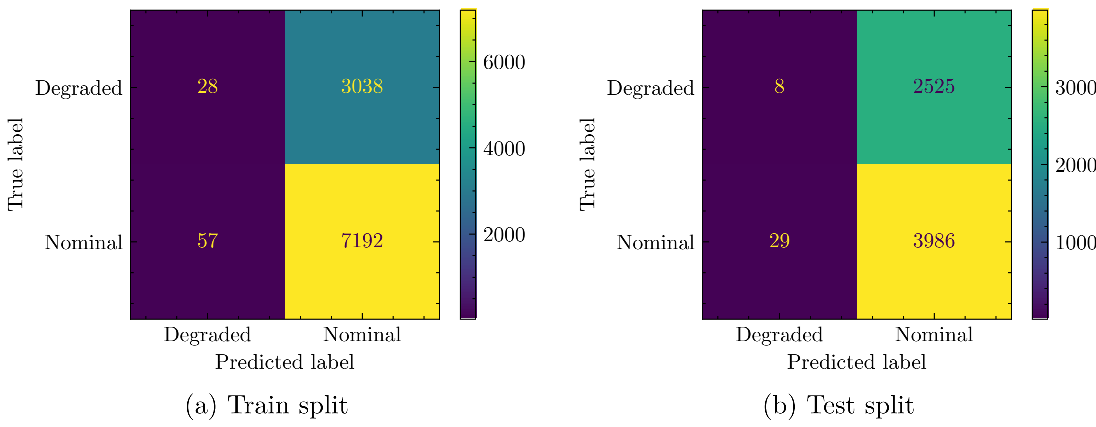
VQC for C-MAPSS FD001 Statistical Features Dataset. Executed on IBM Quantum Lab, 31 GB RAM													
Feature Map	**Ansatz**	**Optimizer**	**Accuracy**		**Precision**		**Recall**		**Specificity**		**NPV**		**Training Time**
			Train	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	
ZZFeatureMap	EfficientSU2	COBYLA	0.74	0.65	0.50	0.40	0.58	0.41	0.80	0.75	0.84	0.76	1219 seconds
ZZFeatureMap	RealAmplitudes	COBYLA	0.76	0.67	0.70	0.57	0.58	0.47	0.86	0.80	0.79	0.73	1079 seconds
ZFeatureMap	EfficientSU2	COBYLA	0.75	0.66	0.80	0.43	0.56	0.43	0.89	0.76	0.73	0.76	1152 seconds
ZFeatureMap	RealAmplitudes	COBYLA	0.73	0.62	0.77	0.43	0.53	0.38	0.88	0.74	0.71	0.70	1035 seconds
RawFeatureVector	EfficientSU2	COBYLA	0.76	0.65	0.70	0.33	0.58	0.40	0.86	0.73	0.79	0.79	1253 seconds
RawFeatureVector	RealAmplitudes	COBYLA	0.78	0.61	0.97	0.33	0.58	0.34	0.98	0.72	0.70	0.73	1128 seconds
ZZFeatureMap	EfficientSU2	SLSQP	0.75	0.61	0.80	0.33	0.56	0.34	0.89	0.72	0.73	0.73	3522 seconds
ZZFeatureMap	RealAmplitudes	SLSQP	0.74	0.65	0.67	0.33	0.56	0.40	0.84	0.73	0.77	0.79	3118 seconds
ZFeatureMap	EfficientSU2	SLSQP	0.79	0.61	0.90	0.33	0.60	0.34	0.95	0.72	0.74	0.73	2584 seconds
ZFeatureMap	RealAmplitudes	SLSQP	0.75	0.60	0.80	0.17	0.56	0.25	0.89	0.69	0.73	0.79	2287 seconds
RawFeatureVector	EfficientSU2	SLSQP	0.76	0.71	0.70	0.77	0.58	0.51	0.86	0.87	0.79	0.69	2996 seconds
RawFeatureVector	RealAmplitudes	SLSQP	0.75	0.61	0.80	0.77	0.56	0.51	0.89	0.87	0.73	0.69	2659 seconds
ZZFeatureMap	EfficientSU2	SPSA	0.83	0.77	0.97	0.87	0.64	0.58	0.98	0.93	0.77	0.73	3513 seconds
ZZFeatureMap	RealAmplitudes	SPSA	0.80	0.77	0.80	0.90	0.63	0.57	0.90	0.94	0.80	0.71	3015 seconds
ZFeatureMap	EfficientSU2	SPSA	0.83	0.76	0.97	0.70	0.64	0.58	0.98	0.86	0.77	0.79	2463 seconds
ZFeatureMap	RealAmplitudes	SPSA	0.80	0.72	0.80	0.70	0.63	0.53	0.90	0.85	0.80	0.73	2379 seconds
RawFeatureVector	EfficientSU2	SPSA	0.81	0.75	0.77	0.80	0.66	0.56	0.89	0.89	0.83	0.73	2835 seconds
RawFeatureVector	RealAmplitudes	SPSA	0.77	0.71	0.87	0.57	0.58	0.52	0.93	0.81	0.73	0.77	2782 seconds

4.3.2.2. FD001 Sensors Features



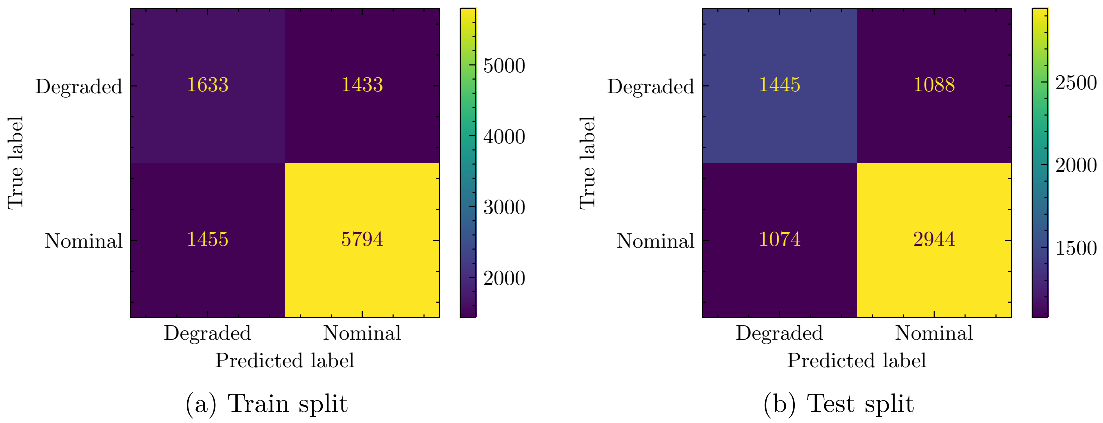
(a) Train split (b) Test split

Figure 4.3.7: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Sensor Features



(a) Train split (b) Test split

Figure 4.3.8: Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

Figure 4.3.9: Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Sensor Features

Table 4.5 shows the results of the model applied for this dataset.

Table 4.5: VQC for C-MAPSS FD001 with Sensor Features Results

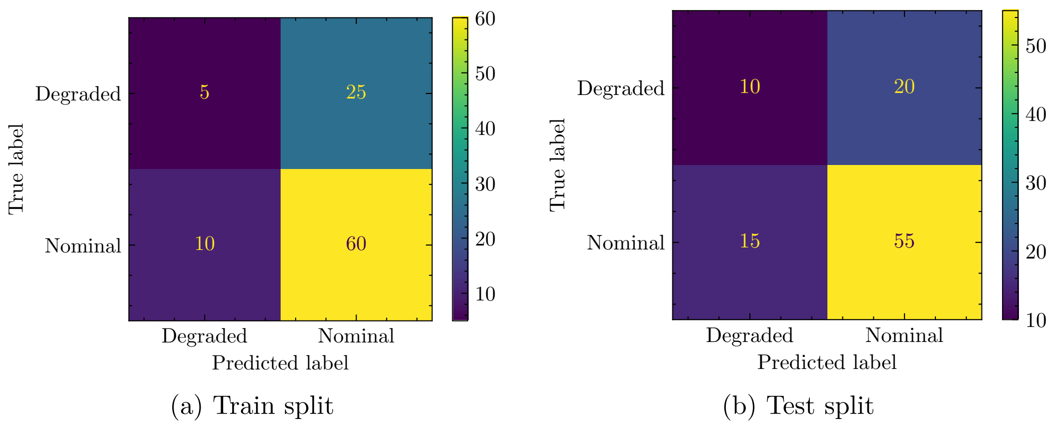
VQC for C-MAPSS FD001 Sensor Features Dataset. Executed on Jupyter Notebook, Intel i7 2.20GHz, 8 GB RAM													
Feature Map	**Ansatz**	**Optimizer**	**Accuracy**		**Precision**		**Recall**		**Specificity**		**NPV**		**Training Time**
			Train	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	
ZZFeatureMap	EfficientSU2	COBYLA	0.68	0.60	0.25	0.32	0.43	0.47	0.73	0.64	0.86	0.78	98085 seconds
ZZFeatureMap	RealAmplitudes	COBYLA	0.68	0.67	0.16	0.54	0.40	0.58	0.72	0.72	0.90	0.75	91022 seconds
ZFeatureMap	EfficientSU2	COBYLA	0.70	0.61	0.01	0.00	0.33	0.22	0.70	0.61	0.99	0.99	71961 seconds
ZFeatureMap	RealAmplitudes	COBYLA	0.70	0.61	0.00	0.00	0.18	0.14	0.70	0.61	1.00	0.99	78659 seconds
RawFeatureVector	RealAmplitudes	COBYLA	0.66	0.58	0.20	0.28	0.37	0.43	0.72	0.63	0.85	0.77	40453 seconds
RawFeatureVector	EfficientSU2	COBYLA	0.69	0.62	0.33	0.44	0.47	0.51	0.75	0.68	0.84	0.73	42536 seconds
ZZFeatureMap	EfficientSU2	SLSQP	0.70	0.61	0.40	0.38	0.49	0.49	0.77	0.66	0.83	0.76	283392 seconds
ZZFeatureMap	RealAmplitudes	SLSQP	0.69	0.62	0.37	0.44	0.47	0.51	0.76	0.68	0.82	0.73	250885 seconds
ZFeatureMap	EfficientSU2	SLSQP	0.70	0.61	0.38	0.38	0.49	0.49	0.76	0.66	0.83	0.76	207917 seconds
ZFeatureMap	RealAmplitudes	SLSQP	0.70	0.61	0.37	0.38	0.49.	0.49	0.76	0.66	0.84	0.76	184030 seconds
RawFeatureVector	EfficientSU2	SLSQP	0.67	0.57	0.29	0.31	0.42	0.42	0.74	0.63	0.83	0.73	241068 seconds
RawFeatureVector	RealAmplitudes	SLSQP	0.66	0.54	0.19	0.21	0.37	0.35	0.72	0.60	0.86	0.75	213952 seconds
ZZFeatureMap	EfficientSU2	SPSA	0.76	0.71	0.69	0.64	0.58	0.62	0.86	0.77	0.79	0.76	282668 seconds
ZZFeatureMap	RealAmplitudes	SPSA	0.73	0.70	0.54	0.56	0.55	0.62	0.81	0.74	0.81	0.79	242597 seconds
ZFeatureMap	EfficientSU2	SPSA	0.75	0.71	0.65	0.64	0.57	0.62	0.84	0.77	0.79	0.76	198181 seconds
ZFeatureMap	RealAmplitudes	SPSA	0.73	0.59	0.54	0.26	0.55	0.45	0.81	0.63	0.81	0.80	191422 seconds
RawFeatureVector	EfficientSU2	SPSA	0.75	0.69	0.65	0.65	0.57	0.59	0.84	0.77	0.79	0.71	228114 seconds
RawFeatureVector	RealAmplitudes	SPSA	0.72	0.67	0.53	0.57	0.53	0.57	0.80	0.73	0.80	0.73	223849 seconds

4.4. Quantum Neural Network Classifier Results

Some of the confusion matrices and evaluation metrics for each of the case studies using Quantum Neural Network Classifier are presented below. The full set of confusion matrices can be found in the Annex B.

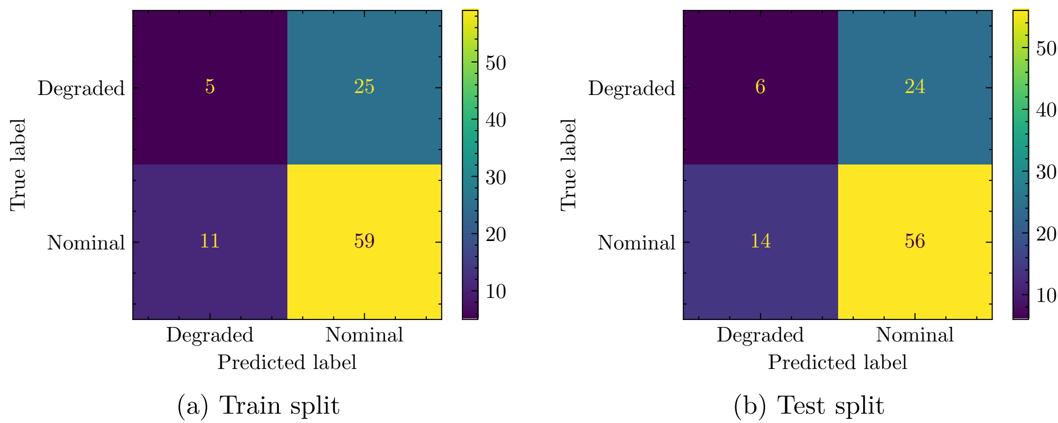
4.4.1. C-MAPSS Aircraft Engine Simulator

4.4.1.1. FD001 Statistics Features



(a) Train split (b) Test split

Figure 4.4.1: Confussion Matrixes for QNN with ZZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Statistics Features



(a) Train split (b) Test split

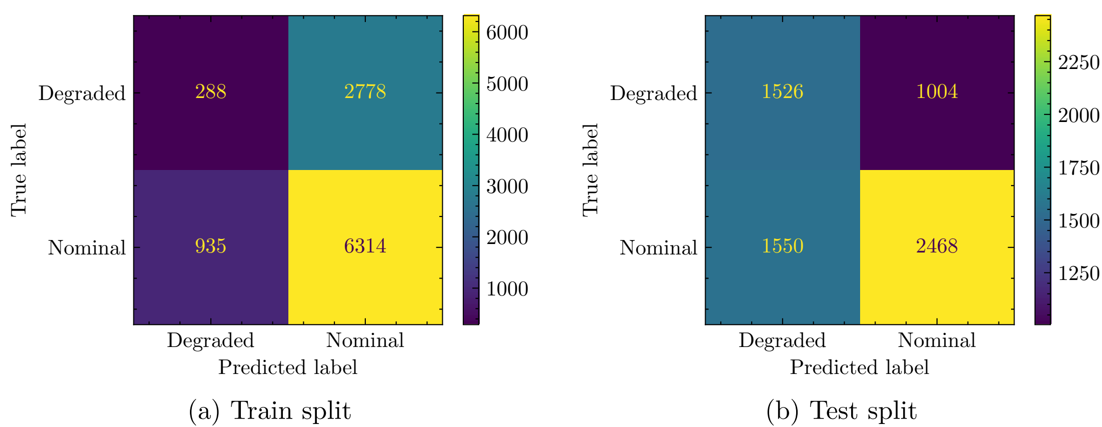
Figure 4.4.2: Confussion Matrixes for QNN with ZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Statistics Features

Table 4.6 shows the results of the model applied to this dataset.

Table 4.6: QNN for C-MAPSS FD001 with Statistics Features Results

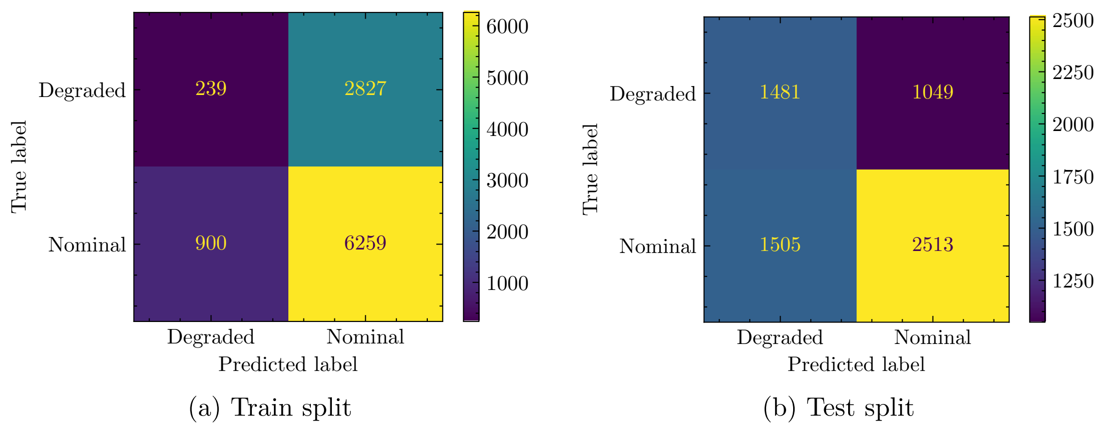
QNN for C-MAPSS FD001 Statistics Features Dataset. Executed on Jupyter Notebook, Intel i7 2.20GHz, 8 GB RAM													
Feature Map	**Ansatz**	**Accuracy**		**Precision**		**Recall**		**Specificity**		**NPV**		**Training Time**
		Train	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	
ZZFeatureMap	RealAmplitudes	0.65	0.63	0.17	0.33	0.33	0.40	0.71	0.73	0.86	0.79	967 seconds
ZFeatureMap	RealAmplitudes	0.64	0.62	0.17	0.20	0.31	0.30	0.70	0.70	0.84	0.80	835 seconds
RawFeatureVector	RealAmplitudes	0.65	0.57	0.33	0.20	0.40	0.24	0.73	0.68	0.79	0.73	982 seconds
ZZFeatureMap	EfficientSU2	0.67	0.63	0.60	0.63	0.46	0.42	0.80	0.80	0.70	0.63	864 seconds
ZFeatureMap	EfficientSU2	0.68	0.62	0.57	0.43	0.47	0.38	0.80	0.74	0.73	0.70	829 seconds
RawFeatureVector	EfficientSU2	0.68	0.61	0.50	0.33	0.47	0.34	0.78	0.72	0.76	0.73	994 seconds

4.4.1.2. FD001 Sensors Features



(a) Train split (b) Test split

Figure 4.4.3: Confussion Matrixes for QNN with ZZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Sensor Features



(a) Train split (b) Test split

Figure 4.4.4: Confussion Matrixes for QNN with ZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Sensor Features

Table 4.7 shows the results of the model applied for this dataset.

Table 4.7: QNN for C-MAPSS FD001 with Sensors Features Results

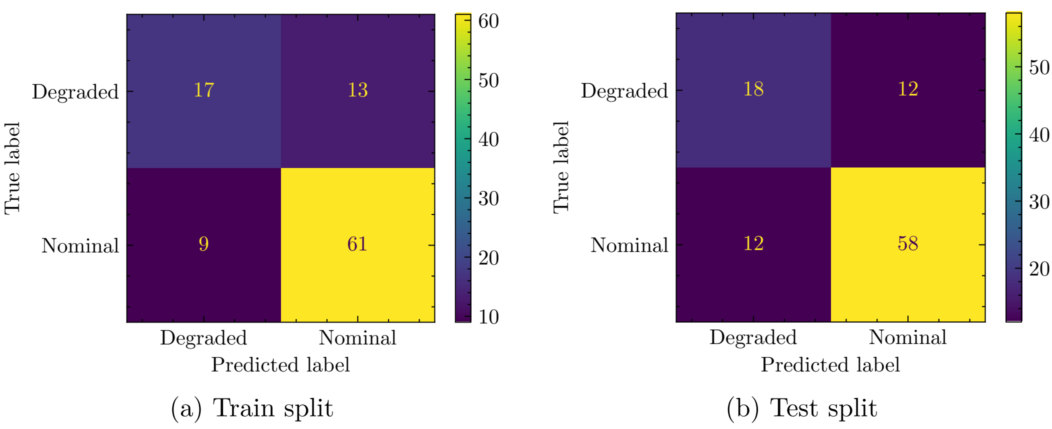
QNN for C-MAPSS FD001 Sensors Features Dataset. Executed on Jupyter Notebook, Intel i7 2.20GHz, 8 GB RAM													
Feature Map	**Ansatz**	**Accuracy**		**Precision**		**Recall**		**Specificity**		**NPV**		**Training Time**
		Train	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	
ZZFeatureMap	RealAmplitudes	0.64	0.61	0.09	0.60	0.24	0.50	0.69	0.71	0.87	0.61	61886 seconds
ZFeatureMap	RealAmplitudes	0.63	0.61	0.08	0.59	0.21	0.50	0.69	0.71	0.87	0.63	53438 seconds
RawFeatureVector	RealAmplitudes	0.63	0.56	0.21	0.34	0.31	0.41	0.71	0.63	0.81	0.70	62845 seconds
ZZFeatureMap	EfficientSU2	0.64	0.61	0.12	0.41	0.27	0.49	0.70	0.67	0.86	0.73	55294 seconds
ZFeatureMap	EfficientSU2	0.67	0.60	0.06	0.31	0.25	0.47	0.70	0.64	0.93	0.78	53054 seconds
RawFeatureVector	EfficientSU2	0.68	0.60	0.28	0.35	0.44	0.48	0.74	0.65	0.85	0.76	63613 seconds

4.5. Quantum Neural Network with Pytorch Classifier Results

Some of the confusion matrices and evaluation metrics for each of the case studies using Quantum Neural Network with Pytorch Classifier are presented below. The full set of matrices can be found in the Annex B.

4.5.1. C-MAPSS Aircraft Engine Simulator

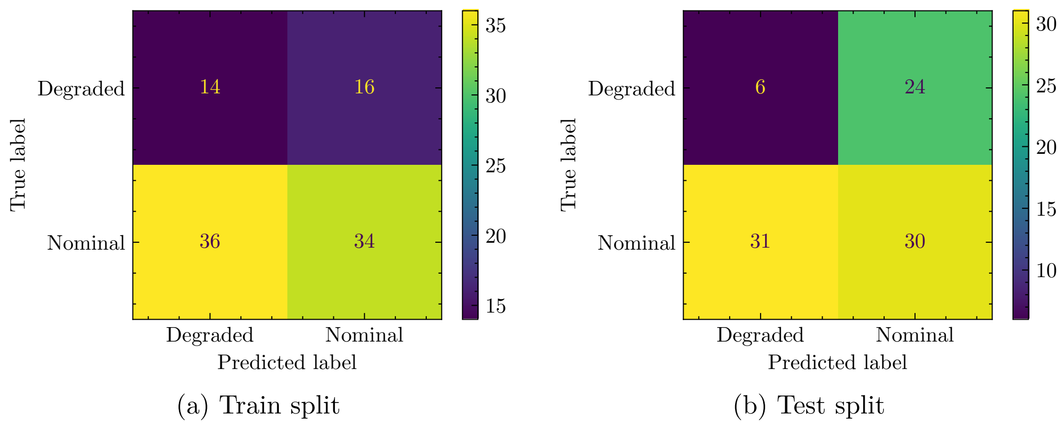
4.5.1.1. FD001 Statistics Features



(a) Train split

(b) Test split

Figure 4.5.1: Confussion Matrixes for Pytorch + OpflowQNN with ZZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features



(a) Train split

(b) Test split

Figure 4.5.2: Confussion Matrixes for Pytorch + CircuitQNN with ZZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features

The results of the model applied to this dataset are shown in table 4.8.

Table 4.8: Pytorch + QNN for C-MAPSS FD001 with Statistics Features Results

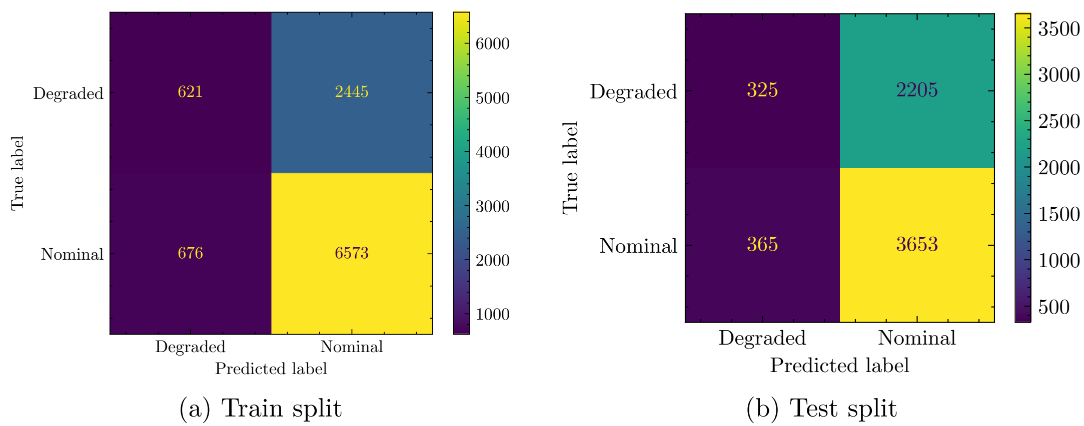
Pytorch + QNN for C-MAPSS FD001 Statistical Features Dataset. Executed on Google Colab Pro, Tesla T4, 16 GB RAM															
Quantum Neural Network	Feature Map	Ansatz	Optimizer	Accuracy		Precision		Recall		Specificity		NPV		Training Time
				Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	
OpflowQNN	ZZFeatureMap	RealAmplitudes	LBFGS	0.78	0.76	0.57	0.60	0.65	0.60	0.82	0.83	0.87	0.83	1620 seconds
CircuitQNN	ZZFeatureMap	RealAmplitudes	LBFGS	0.48	0.45	0.47	0.20	0.28	0.16	0.68	0.56	0.49	0.49	540 seconds
OpflowQNN	ZFeatureMap	RealAmplitudes	LBFGS	0.77	0.75	0.33	0.77	0.77	0.56	0.77	0.88	0.96	0.74	1398 seconds
CircuitQNN	ZFeatureMap	RealAmplitudes	LBFGS	0.49	0.47	0.37	0.33	0.26	0.23	0.67	0.65	0.54	0.53	467 seconds
OpflowQNN	RawFeatureVector	RealAmplitudes	LBFGS	0.78	0.75	0.33	0.60	0.83	0.58	0.77	0.83	0.97	0.81	1754 seconds
CircuitQNN	RawFeatureVector	RealAmplitudes	LBFGS	0.50	0.48	0.30	0.63	0.24	0.32	0.66	0.72	0.59	0.41	548 seconds

4.6. Quantum Support Vector Classifier Results

Some of the confusion matrices and evaluation metrics for each of the case studies using Quantum Support Vector Classifier are presented below.

4.6.1. C-MAPSS Aircraft Engine Simulator

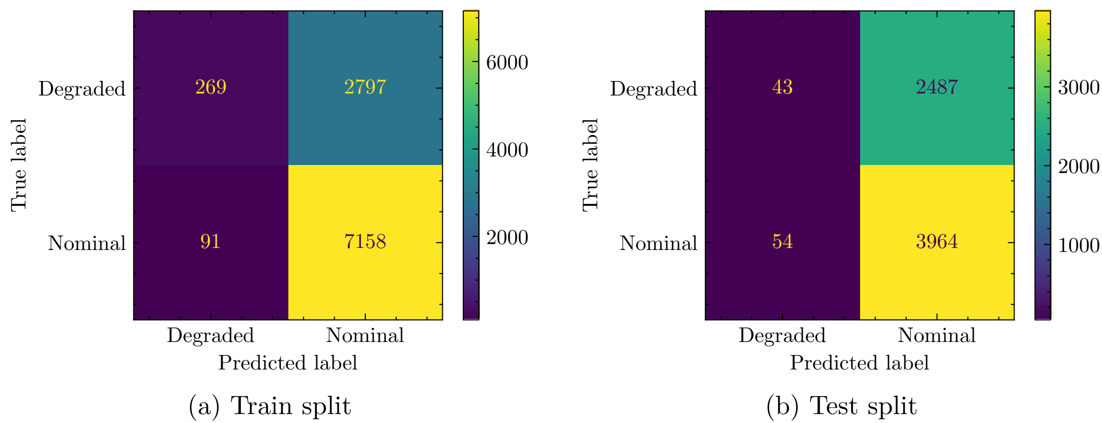
4.6.1.1. FD001 Sensors Features



(a) Train split

(b) Test split

Figure 4.6.1: Confussion Matrixes for Quantum SVC with ZFeatureMap applied to C-MAPSS FD001 with Sensors Features



(a) Train split

(b) Test split

Figure 4.6.2: Confussion Matrixes for Quantum SVC with ZZFeatureMap applied to C-MAPSS FD001 with Sensors Features

Table 4.9 shows the model results applied for this dataset. The FeatureMap RawFeatureVector is not supported due to its unbound parameters.

Table 4.9: QVSC for C-MAPSS FD001 with Sensors Features Results

QSVC for C-MAPSS FD001 Sensors Features Dataset. Executed on Jupyter Notebook, Intel i7 2.20GHz, 8 GB RAM												
Feature Map	**Accuracy**		**Precision**		**Recall**		**Specificity**		**NPV**		**Training Time**	**Testing Time**
	Train	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**		
ZFeatureMap	0.70	0.61	0.20	0.13	0.48	0.47	0.73	0.62	0.91	0.91	7 seconds	889 seconds
ZZFeatureMap	0.72	0.60	0.09	0.02	0.74	0.44	0.72	0.61	0.99	0.99	16 seconds	2043 seconds

4.7. Unused Quantum Scructures

The following table shows the features maps, ansatz and optimizers that could not be implemented in the models and their respective reasons.

Table 4.10: Unused Structures

Structure	**Reason**
Feature Maps	
State Preparation	They don't perform a proper data coding because the large amount of data used and the parameters of the maps.
Pauli FeatureMap	
Ansatzs	
Pauli Two Design Ansatz	Does not get good training performances with the used Feature Maps and the big amount of used data.
Optimizers	
ADAM	The loss function does not converge at allowed training times due the big dimensionality of the data
NFT	
POWELL	
Gradien Descent	
Nelder Mead	
LBFGSB	Each training step took very long times, so it's not possible to complete de training
AQGD	
CG	
GSLS	No class prediction is achieved when used. Too low accuracies are obtained
TNC	
UMDA	

5 | Discussions

Before going deeper into the results, it is important to mention that the accuracy on the test set is the most important result to analyze. This is the performance of the model on data that it has not seen before and is a good indicator of the model's ability to generalize to new data. In addition, attention should also be paid to the training time, which is important to consider the computational cost and resources needed to train the model. The results per model are discussed below for an overall comparison.

5.1. Variational Quantum Classifier for MFPT Dataset

Initially, in this research, combinations of FeatureMaps, Ansatz and optimizers were investigated in a Variational Quantum Classifier (VQC) of Qiskit with the objective of classifying bearing faults in the MFPT dataset. In general, the VQC is a hybrid algorithm that uses a combination of quantum and classical techniques to classify data into two or more categories. The results obtained from this research are summarized in Table 4.3.

Firstly, it can be observed that the choice of feature map, ansatz and optimizer plays a critical role in determining the classification accuracy of the variational quantum classifier. Secondly, it can be observed that the SPSA optimizer generally outperformed the COBYLA and SLSQP optimizers in terms of classification accuracy. Thirdly, the EfficientSU2 ansatz performed better than the RealAmplitudes ansatz across all feature maps and optimizers in terms of classification train accuracy.

Regarding the choice of feature map, the results indicate that the ZZFeatureMap performed slightly worse than the ZFeatureMap and RawFeatures in terms of test accuracy. However, it is interesting to note that the ZZFeatureMap achieved the highest training accuracy with the SPSA optimizer and the EfficientSU2 ansatz. This could suggest that the ZZFeatureMap might be more effective in capturing complex interactions between qubits, which could be useful for more challenging classification tasks.

In terms of the choice of ansatz, the EfficientSU2 ansatz had better train accuracies than the RealAmplitudes ansatz in all cases. This is consistent with previous studies that have shown that the EfficientSU2 ansatz is more expressive than the RealAmplitudes ansatz and can better capture complex quantum correlations.

With respect to the choice of optimizer, the SPSA optimizer consistently outperformed the COBYLA and SLSQP optimizers in terms of classification accuracy. This could be due to the fact that the SPSA optimizer is less sensitive to noise and can better handle optimization in the presence of noise.

It is important to note that the training time varied significantly depending on the choice of feature map, ansatz and optimizer. In general, the SLSQP optimizer had the longest training time, followed by SPSA and COBYLA. However, the training time also depended on the complexity of the feature map and ansatz. For example, using a same optimizer, the ZZFeatureMap and EfficientSU2 ansatz had the longest training times.

Overall, the results suggest that the choice of feature map, ansatz and optimizer can significantly impact the performance of the variational quantum classifier. The RawFeatures feature map with EfficientSU2 ansatz and SPSA optimizer appear to be the most effective choices for achieving high classification accuracy. However, it is important to consider the trade-off between accuracy and training time when selecting the optimal combination of feature map, ansatz and optimizer for a particular classification task.

5.2. Variational Quantum Classifier for CMAPSS Statistics Features Dataset

Based on the results, it can be observed that the ZZFeatureMap combined with EfficientSU2 and SPSA optimizer achieved the highest train accuracy of 0.83, while the combination of ZFeatureMap with RealAmplitudes and COBYLA optimizer yielded the lowest train accuracy of 0.73. On the other hand, also the combination of ZZFeatureMap with EfficientSU2 and SPSA optimizer achieved the highest test accuracy of 0.77, while the combination of ZFeatureMap with RealAmplitudes and SLSQP optimizer yielded the lowest test accuracy of 0.60.

It is worth noting that the highest train accuracy does not necessarily guarantee the highest test accuracy. For instance, the combination of RawFeatures with RealAmplitudes and COBYLA optimizer achieved a high train accuracy of 0.78 but a relatively low test accuracy of 0.61. This observation indicates the importance of evaluating both the train and test accuracy of a model when selecting the best combination of feature maps, ansatz, and optimizer.

Furthermore, the combination of EfficientSU2 ansatz with SPSA optimizer achieved higher accuracy results in most cases, including train accuracy, test accuracy, train recall, test recall, train specificity, test specificity, and train negative predictive value (NPV). This combination appears to be more effective in optimizing the quantum circuit parameters than the other combinations tested.

The choice of feature maps and ansatz also had an impact on the performance of the classifier. For instance, the combination of ZZFeatureMap with RealAmplitudes and

COBYLA optimizer achieved a higher train precision of 0.70, while the combination of ZFeatureMap with EfficientSU2 and COBYLA optimizer achieved a higher train precision of 0.80. This indicates that different feature maps and ansatz may be more suitable for different datasets and classification tasks.

It is also interesting to note that the training time varies significantly among the combinations tested. For example, the combination of ZZFeatureMap with EfficientSU2 and SPSA optimizer took the longest training time of 3513 seconds, while the combination of ZFeatureMap with RealAmplitudes and COBYLA optimizer took the shortest training time of 1035 seconds. Therefore, the choice of optimizer should not only be based on accuracy but also on the computational resources available.

In conclusion, the choice of feature maps, ansatz, and optimizer can significantly impact the performance of a variational quantum classifier. It is important to carefully evaluate the train and test accuracy, as well as other metrics such as precision, recall, specificity, and NPV when selecting the best combination. The combination of EfficientSU2 ansatz with SPSA optimizer appears to be more effective in optimizing the quantum circuit parameters and achieving higher accuracy results in most cases. However, different feature maps and ansatz may be more suitable for different datasets and classification tasks. The training time should also be taken into consideration when selecting the best combination.

5.3. Variational Quantum Classifier for CMAPSS Sensors Features Dataset

The table 4.5 contains the results of several combinations of feature maps, ansatz, and optimizers on a NASA CMAPSS dataset using a Variational Quantum Classifier of Qiskit. The dataset contains health state information, and the performance of the models was evaluated based on train and test accuracy, precision, recall, specificity, and negative predictive value. Additionally, the training time for each combination was also recorded.

One of the first observations from the results is that the ZZFeatureMap and ZFeatureMap feature maps have similar performance in terms of accuracy and other metrics. However, the use of EfficientSU2 ansatz generally results in similar performance than RealAmplitudes ansatz. This trend is consistent across all feature maps and optimizers. For instance, using the ZFeatureMap feature map with the EfficientSU2 ansatz and SLSQP optimizer resulted in a train accuracy of 0.70 and a test accuracy of 0.61, while using RealAmplitudes ansatz with the same feature map and optimizer resulted in a test accuracy of 0.61.

In general, COBYLA optimizer performed worse than SLSQP and SPSA optimizers. The use of SPSA optimizer consistently yielded the highest test accuracy for all feature maps and ansatz. For example, when using the ZZFeatureMap feature map with EfficientSU2 ansatz and SPSA optimizer, a train accuracy of 0.76 and test accuracy of

0.71 were obtained.

The results also showed that the RawFeatureVector feature map performed worse than the ZZFeatureMap and ZFeatureMap feature maps, regardless of the ansatz and optimizer used. However, the performance of the RawFeatureVector feature map can be improved by using the EfficientSU2 ansatz. For instance, when using the RawFeatureVector feature map with EfficientSU2 ansatz and SPSA optimizer, a train accuracy of 0.75 and test accuracy of 0.69 were obtained.

In terms of the performance metrics, the models generally performed better in terms of recall and specificity, compared to precision and negative predictive value. For example, when using the ZFeatureMap feature map with EfficientSU2 ansatz and COBYLA optimizer, a train precision of 0.01 and test precision of 0.00 were obtained, while train recall of 0.33 and test recall of 0.22 were obtained.

Finally, the training time varied significantly across the combinations, with some combinations taking significantly longer than others. For instance, using the ZZFeatureMap feature map with EfficientSU2 ansatz and SLSQP optimizer took 283392 seconds to train, while using the RawFeatureVector feature map with RealAmplitudes ansatz and COBYLA optimizer took only 40453 seconds. Therefore, the choice of feature map, ansatz, and optimizer should also consider the computational resources available.

5.4. Quantum Neural Network Classifier for CMAPSS Statistics Features Dataset

Based on the results, the models using the EfficientSU2 ansatz perform better than those using the RealAmplitudes ansatz, regardless of the feature map used. Specifically, the ZZFeatureMap and ZFeatureMap with EfficientSU2 both achieved higher accuracy, precision, recall, specificity, and negative predictive value compared to their counterparts using RealAmplitudes.

For instance, the ZZFeatureMap with EfficientSU2 achieved a train accuracy of 0.67, while the same feature map with RealAmplitudes only achieved 0.65. Similarly, the ZFeatureMap with EfficientSU2 had a higher test accuracy (0.68) than the same feature map with RealAmplitudes (0.64). These results suggest that the EfficientSU2 ansatz is more suitable for the given dataset.

In terms of feature maps, the models using the RawFeatureVector feature map performed worse compared to those using the ZZFeatureMap or ZFeatureMap, regardless of the ansatz used. For example, the RawFeatureVector with RealAmplitudes achieved the lowest test accuracy among all the models (0.57). Meanwhile, the ZZFeatureMap with EfficientSU2 achieved the highest test accuracy (0.63). This suggests that the RawFeatureVector feature map might not be well-suited for the given dataset, and a more sophisticated feature map might be required to improve the model's performance.

Regarding the optimizers, we can't infer a clear trend from the provided results. The models using the same feature map and ansatz combination with different optimizers achieved similar results. For instance, the ZZFeatureMap with EfficientSU2 achieved similar test accuracy regardless of whether it used the default optimizer or the SPSA optimizer. This indicates that the choice of optimizer might not significantly impact the model's performance on the given dataset.

In terms of individual performance metrics, we can observe that the models achieved high specificity and negative predictive value, indicating a low false positive rate and low false negative rate, respectively. For example, the ZZFeatureMap with RealAmplitudes achieved a high test specificity (0.73) and test negative predictive value (0.79). However, the models' precision and recall were generally lower, indicating that the models might have difficulty correctly identifying all instances of the minority class in the dataset.

Overall, the results suggest that the use of a more sophisticated feature map and EfficientSU2 ansatz can improve the model's performance on the given dataset. However, the performance of the model might still be limited by the choice of features available in the dataset. Therefore, exploring more advanced feature engineering techniques might be necessary to improve the model's performance further. Additionally, we note that the models' training times are relatively low, indicating that quantum neural networks might be a viable alternative to classical machine learning models for this type of dataset.

5.5. Quantum Neural Network Classifier for CMAPSS Sensors Features Dataset

Based on the obtained results, the ZZFeatureMap and ZFeatureMap feature maps with RealAmplitudes and EfficientSU2 ansatzs have very similar performances in terms of test accuracy, precision, recall, specificity, and NPV. However, the ZZFeatureMap with RealAmplitudes has a slightly higher train accuracy and train precision compared to the ZFeatureMap with RealAmplitudes. This may suggest that the ZZFeatureMap is more effective in capturing the underlying relationships in the dataset, leading to a better fitting of the model to the training data. Nonetheless, the difference in performance between the two feature maps is minimal, and thus the choice between them may come down to other factors such as computational efficiency or personal preference.

On the other hand, the RawFeatureVector feature map does not perform as well on the test datasets as the ZZFeatureMap and ZFeatureMap feature maps. This may suggest that the quantum neural network is not able to effectively capture the relevant features from the dataset in their raw form, and that the additional structure provided by the feature maps is necessary for the model to perform well.

Regarding the choice of ansatz, we can observe that the EfficientSU2 ansatz consistently outperforms the RealAmplitudes ansatz in terms of test accuracy and other metrics, except for train precision in the case of ZZFeatureMap. This may suggest that

the EfficientSU2 ansatz is better at capturing the underlying relationships between the features in the dataset, leading to better generalization to unseen data. However, we cannot conclude with certainty that the EfficientSU2 ansatz is always better than the RealAmplitudes ansatz, as the choice of ansatz may also depend on the specific dataset and task.

Finally, it is worth noting that the training times for the different combinations of feature maps and ansatzs vary considerably. The RawFeatureVector with EfficientSU2 combination took the longest time to train, whereas the ZZFeatureMap with EfficientSU2 combination took the shortest time. However, the training time alone should not be the sole criterion for selecting the best combination, as other factors such as accuracy, precision, recall, specificity, and NPV should also be taken into account.

In summary, based on the results provided, we can conclude that the ZZFeatureMap and ZFeatureMap feature maps with EfficientSU2 ansatzs are the most effective combinations for this task. However, the choice between the two feature maps may depend on other factors such as computational efficiency or personal preference. Additionally, it is important to note that training time can vary considerably between different combinations of feature maps and ansatz.

5.6. Quantum Neural Network with Pytorch Classifier for CMAPSS Sensors Features Dataset

Based on the obtained results, we can see that OpflowQNN with ZZFeatureMap and RealAmplitudes Ansatz performed better than other models with the highest test accuracy of 0.76. This indicates that using the OpflowQNN model with the ZZFeatureMap and RealAmplitudes Ansatz can provide good results for health state classification in the CMAPSS dataset.

However, it is interesting to note that CircuitQNN models did not perform as well as OpflowQNN models. For instance, CircuitQNN with ZZFeatureMap and RealAmplitudes Ansatz had a test accuracy of only 0.45, which is significantly lower than the best-performing model.

Another interesting observation is that using the RawFeatureVector as Feature Map instead of ZZFeatureMap or ZFeatureMap did not significantly improve the performance of the models. The OpflowQNN and CircuitQNN models with RawFeatureVector Feature Map had a slightly lower test accuracy compared to models with ZZFeatureMap or ZFeatureMap. This indicates that using the raw feature vector may not provide any additional benefit in this context.

In terms of evaluation metrics, we can see that the models generally had higher precision and specificity scores than recall and sensitivity scores. This suggests that the models are better at correctly identifying negative instances (healthy states) than positive instances (faulty states).

Finally, we can see that the training time varied significantly between models, with OpflowQNN models generally taking longer to train than CircuitQNN models. For instance, OpflowQNN with RawFeatureVector Feature Map, RealAmplitudes Ansatz, and LBFGS optimizer had the longest training time of 1754 seconds, while CircuitQNN with ZFeatureMap, RealAmplitudes Ansatz, and LBFGS optimizer had the shortest training time of 467 seconds. This suggests that the OpflowQNN models may be more computationally expensive and time-consuming to train than CircuitQNN models.

5.7. Quantum Support Vector Classifier for CMAPSS Sensors Features Dataset

Based on the results, two different feature maps where tested: ZFeatureMap and ZZFeatureMap with a Quantum Support Vector Classifier implemented in Qiskit. For each feature map, the performance of the classifier has been evaluated on a health state dataset from NASA's CMAPSS. Specifically, the training and test accuracy, precision, recall, specificity, and negative predictive value (NPV) has been evaluated, as well as the training time.

Starting with the accuracy metric, both feature maps achieved similar training accuracies: 0.70 for ZFeatureMap and 0.72 for ZZFeatureMap. However, the test accuracy was lower for both feature maps, with ZFeatureMap achieving 0.61 and ZZFeatureMap achieving 0.60. This suggests that the model is overfitting to the training data and not generalizing well to new data. Moving on to the precision metric, we see that the results are quite low for both feature maps, with ZFeatureMap achieving 0.20 and ZZFeatureMap achieving 0.09 on the training set. These values suggest that the classifier is not doing a good job of identifying true positives, and is instead predicting more false positives. The test precision values are even lower, with ZFeatureMap achieving 0.13 and ZZFeatureMap achieving 0.02. This further reinforces the overfitting problem mentioned earlier.

The recall metric tells us how well the classifier is able to identify true positives out of all positive instances. For both feature maps, the training recall values are higher than the test recall values, which is consistent with overfitting. ZFeatureMap achieved a training recall of 0.48 and a test recall of 0.47, while ZZFeatureMap achieved a training recall of 0.74 and a test recall of 0.44. These results suggest that the classifier is not doing well at identifying true positives in general, and is particularly poor at doing so on new data.

The specificity metric tells us how well the classifier is able to identify true negatives out of all negative instances. In general, we want high specificity values, as this indicates that the classifier is doing a good job of correctly identifying negative instances. For both feature maps, the training specificity values are higher than the test specificity values. ZFeatureMap achieved a training specificity of 0.73 and a test specificity of 0.62, while ZZFeatureMap achieved a training specificity of 0.72 and a test specificity of 0.61. These results indicate that the classifier is not doing a great job of identifying

true negatives, particularly on new data.

Finally, the negative predictive value (NPV) tells us how well the classifier is able to correctly identify negative instances out of all instances it predicts as negative. For both feature maps, the training NPV values are higher than the test NPV values. ZFeatureMap achieved a training NPV of 0.91 and a test NPV of 0.91, while ZZFeatureMap achieved a training NPV of 0.99 and a test NPV of 0.99. These results suggest that the classifier is better at identifying true negatives than true positives, which is consistent with the low precision values discussed earlier.

In terms of training time, ZFeatureMap took only 7 seconds to train and 889 to evaluate, while ZZFeatureMap took 16 seconds and 2043 to evaluate. However, it's worth noting that the training time can vary based on factors such as the size of the dataset, the number of features, and the complexity of the feature map, so this difference on evaluation time may not be solely due to the feature map used.

Overall, the results suggest that the classifier is not performing well on this dataset, and is particularly struggling with identifying true positives, as evidenced by the low precision values. The overfitting problem is also evident from the higher training performance compared to the test performance, which suggests that the model is not generalizing well to new data. The differences in performance between the two feature maps are not very large, with ZZFeatureMap generally performing slightly worse than ZFeatureMap on most metrics. However, ZZFeatureMap did achieve a higher training recall, which suggests that it may be better at identifying true positives on the training set.

It's worth noting that the performance of the classifier may be improved by tuning hyperparameters such as the regularization parameter and kernel type, or by using a different feature map or preprocessing method. Additionally, it's possible that the dataset itself may not be well-suited to this type of classifier, or that more data is needed to improve performance.

5.8. Classical Results vs Quantum Results

The first two classification models being compared are a classical support vector classifier and a variational quantum classifier over MFPT dataset. The classical support vector classifier achieves a train accuracy of 0.8 and a test accuracy of 0.74. It has a very short training time of less than 0.02 seconds. On the other hand, the variational quantum classifier has a train accuracy of 0.65 and a test accuracy of 0.59. It uses the RawFeatures feature map, EfficientSU2 ansatz, and SPSA optimizer, which appear to be the most effective choices for achieving high classification accuracy. However, the model has a significantly longer training time of 19205 seconds. Another variant of the model was trained at 5417 seconds, which is still significantly longer than the classical support vector classifier.

Overall, the classical support vector classifier for MFPT dataset outperforms the

variational quantum classifier in terms of accuracy and training time. However, it's worth noting that the trade-off between accuracy and training time needs to be considered when selecting the optimal model for a particular classification task. If high accuracy is a priority and training time is not a concern, then the variational quantum classifier with the RawFeatures feature map, EfficientSU2 ansatz, and SPSA optimizer could be a good choice, although it is still very deficient in comparison to the classic model.

In the case of the CMAPSS dataset with the statistics features, the classical support vector classifier achieved a train accuracy and test accuracy of 0.7, which is lower than the variational quantum classifier's accuracy of 0.83 for train accuracy and 0.77 for test accuracy. Therefore, the variational quantum classifier outperforms the classical support vector classifier in terms of accuracy. However, it is important to note that the choice of feature map, ansatz, and optimizer significantly impacted the performance of the variational quantum classifier, and it took significantly longer to train (3513 seconds) compared to the classical support vector classifier (less than 0.02 seconds). The higher accuracy of the variational quantum classifier could be beneficial for certain applications where accuracy is crucial, but the long training time could be a significant drawback, particularly in time-sensitive applications. In contrast, the classical support vector classifier's short training time could be beneficial in applications where real-time classification is required, despite its lower accuracy.

Continuing with the classification of the CMAPSS dataset with the statistics features, there are the results of the QNN Classifier, which only reaches a test accuracy of 0.63, which makes it inferior to both classical SVC and VQC, however, it presents short training times of less than 1000 seconds compared to the VQC times. Therefore, the choice between the classifiers depends on the specific requirements of the application, considering the trade-off between accuracy and training time, because on the other hand, there are the results of the QNN Classifier using Torch connector from Pytorch, which reaches train accuracies of 0.78 and test accuracies of 0.76, with training times up to 1800 seconds depending on the Quantum Neural Network used.

Finally, based on the results from model trained with CMAPSS with Sensor Features dataset, the classical support vector classifier has a train and test accuracy of 0.73, with a training time lower than 4.2 seconds. The variational quantum classifier achieved a higher train accuracy of 0.75 and the same test accuracy of 0.73. However, it took a much longer time to train, at 228114 seconds, compared to the classical support vector classifier. The quantum neural network classifier had lower train and test accuracies, at 0.68 and 0.60, respectively, but still took a significant amount of time to train at 63613 seconds. The quantum support vector classifier achieved a train accuracy of 0.72 and a test accuracy of 0.60. It took a shorter time to train compared to the other quantum models, at 2043 seconds, but there is a concern that the model may be overfitted since its test accuracy is significantly lower than its train accuracy.

Overall, on this last dataset, the classical support vector classifier performed reasonably well with a shorter training time, while the quantum models required much longer training times but achieved comparable or slightly higher accuracies. The choice

of feature map, ansatz, and optimizer significantly impacted the performance of the quantum models, indicating the importance of careful selection of these parameters for achieving high classification accuracy.

6 | Conclusions

In conclusion, this research explored the possibilities of Quantum Machine Learning for Predictive Maintenance, using various classification models and datasets. The objective of the work was to compare classical and quantum machine learning algorithms and evaluate their performance in terms of accuracy and training time.

Due to the limited availability of quantum hardware, all the quantum models were executed on Qiskit simulators, as IBM Quantum labs only provides access to real quantum hardware with limited qubits and memory. As a result, the large datasets used in this research were not feasible to train on real hardware. It is worth noting that the IBM Qiskit library is constantly changing, and it is necessary to be updating the codes and models to test new capabilities of quantum machine learning before the work becomes obsolete.

The analysis of the results revealed that classical computation is still a better option for predictive maintenance due to the short training times and the high accuracies obtained with few resources. However, the trade-off between accuracy and training time needs to be considered when selecting the optimal model for a particular classification task.

In general, the classical support vector classifier performed well with a shorter training time, while the quantum models required much longer training times but achieved comparable or slightly higher accuracies. The choice of feature map, ansatz, and optimizer significantly impacted the performance of the quantum models, indicating the importance of careful selection of these parameters for achieving high classification accuracy.

Although the results obtained in this research are limited by the availability of quantum resources, it is expected that on the next years the quantum resources with better performance will be more accessible for researchers and better results will be achieved. The field of predictive maintenance could benefit greatly from the development of quantum machine learning algorithms that can accurately predict failures before they occur, leading to significant cost savings and increased productivity taking advantage of the physical nature that quantum computing offers, allowing to perform tasks in more efficient and simultaneous ways thanks to the quantum principles on which it is based.

In summary, this research contributes to the ongoing exploration of quantum ma-

chine learning algorithms for predictive maintenance, and highlights the importance of careful selection of parameters for achieving high accuracy in quantum classification models.

7 | Proposed work

Generally speaking, to continue the current work on quantum machine learning models, different combinations of architectures for the models used should be further explored.

First, it is important to continue experimenting with different Featuremaps to transform the input data into quantum state vectors. As in classical machine learning, the way the model receives the data can be crucial for good classification results. Therefore, it is necessary to be aware of new options to implement them and compare their results.

Another important aspect to consider is the Ansatz used in quantum machine learning models. As we have seen, the ansatz, together with the featuremap, is the set of quantum operations performed on the qubits to process the information. It is essential to find an Ansatz that is complex enough to process all the necessary information, but at the same time simple to implement. The search for a suitable Ansatz is an optimization problem that can be solved using machine learning or quantum optimization techniques.

It is also essential to test different and new classical or quantum optimizers to find the one that best suits the classification task at hand. Remembering that optimizers are algorithms that seek to minimize a cost function, which is the measure of the difference between the results obtained and the expected results, it is important to compare and evaluate different optimizers to find the one that generates the best results.

To further improve the research path, it is necessary to run quantum machine learning models on real quantum hardware, as these become increasingly available with more and more qubits to use, rather than using simulators. Simulators are a useful tool for testing models, but they are not fully representative of reality as real quantum devices have limitations, such as read and write errors, decoherence and noise. It is necessary to adapt the models to these limitations to obtain results that are more accurate and representative of reality.

In addition, it is important to keep in mind that quantum technology is constantly evolving. Quantum programming libraries, such as Qiskit, are continually being updated and improved. It is critical to keep up with changes and updates to ensure that the code is always current and optimized for the latest technologies. This is especially important for quantum machine learning models, as the complexity of the algorithms and hardware architecture can change rapidly, rendering those used in the present work

obsolete. It is also important to implement other quantum programming development libraries such as Pennylane or TensorflowQuantum.

In summary, further research in quantum machine learning models involves continuing to experiment with different featuremaps, ansatz and optimizers, as well as running the models on real quantum hardware rather than on simulators and using other libraries for even more comparisons. It is also essential to keep up to date with the latest technologies and updates in the field of quantum computing.

Bibliography

[1] M. Alloghani, D. Al-Jumeily Obe, J. Mustafina, A. Hussain, and A. Aljaaf, *A Systematic Review on Supervised and Unsupervised Machine Learning Algorithms for Data Science*. Springer, 2020.

[2] X. Ying, "An overview of overfitting and its solutions," *Journal of Physics: Conference Series*, vol. 1168, feb 2019.

[3] F. Zickert, *Hands-On Quantum Machine Learning With Python: Volume 1: Get Started*. Independently published, 2021.

[4] D. Zwillinger and S. Kokoska, *CRC Standard Probability and Statistics Tables and Formulae.* Chapman Hall, 2000.

[5] M. E. Tipping and C. M. Bishop, "Mixtures of probabilistic principal component analyzers," *Neural computation*, vol. 11, no. 2, pp. 443–482, 1999.

[6] B. e. a. Schölkopf, *Kernel Methods in Computational Biology.* The MIT Press, 2004.

[7] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," vol. 2, may 2011.

[8] G. S. Martín and E. L. Droguett, "Quantum machine learning for health state diagnosis and prognostics," 2021.

[9] R. Shankar, *Principles of Quantum Mechanics*, pp. 115–122. New York: Plenum Press, 1994.

[10] *IBM Quantum Composer*. https://quantum-computing.ibm.com/, 2021.

[11] G. Aleksandrowicz *et al.*, "Qiskit: An open-source framework for quantum computing," 2021.

[12] D. A. B. Miller, *Quantum Mechanics for Scientists and Engineers*, pp. 426–435. California: Stanford University, 2008.

[13] V. Havlíček, A. D. Córcoles, K. Temme, A. W. Harrow, A. Kandala, J. M. Chow, and J. M. Gambetta, "Supervised learning with quantum-enhanced feature spaces," *Nature*, vol. 567, pp. 209–212, mar 2019.

[14] R. Xia and S. Kais, "Hybrid quantum-classical neural network for calculating ground state energies of molecules," *Entropy*, vol. 22, p. 828, jul 2020.

[15] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*,

2017.

[16] S. Shalev-Shwartz, Y. Singer, and N. Srebro, "Pegasos: Primal estimated sub-gradient solver for svm," in *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, (New York, NY, USA), p. 807–814, Association for Computing Machinery, 2007.

[17] MFPT, *Condition Based Maintenance Fault Database for Testing of Diagnostic and Prognostics Algorithms*. https://www.mfpt.org/fault-data-sets/.

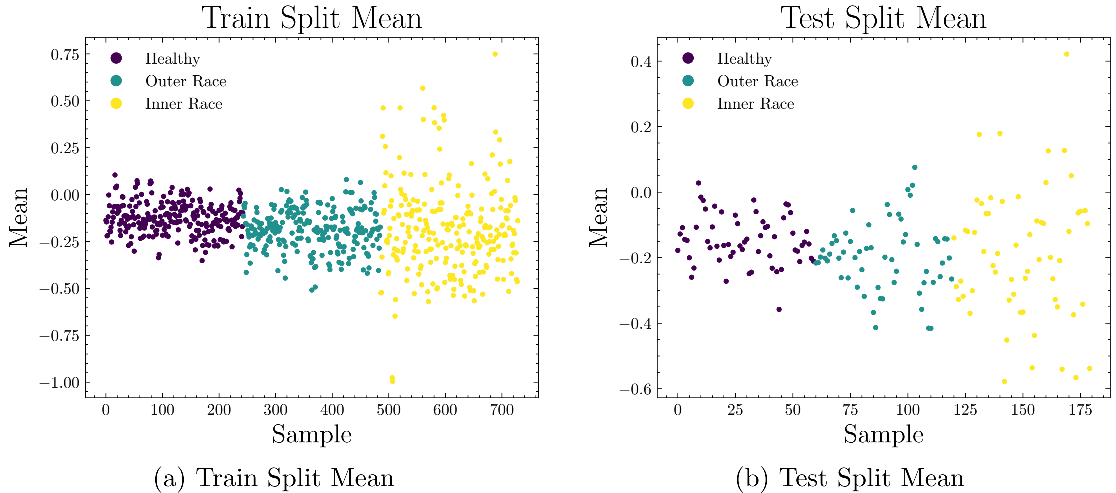
[18] NASA, *C-MAPSS Aircraft Engine Simulator Data*. https://data.nasa.gov/dataset/C-MAPSS-Aircraft-Engine-Simulator-Data/xaut-bemq.

ANNEXES

Annex A | Datasets Features Plots

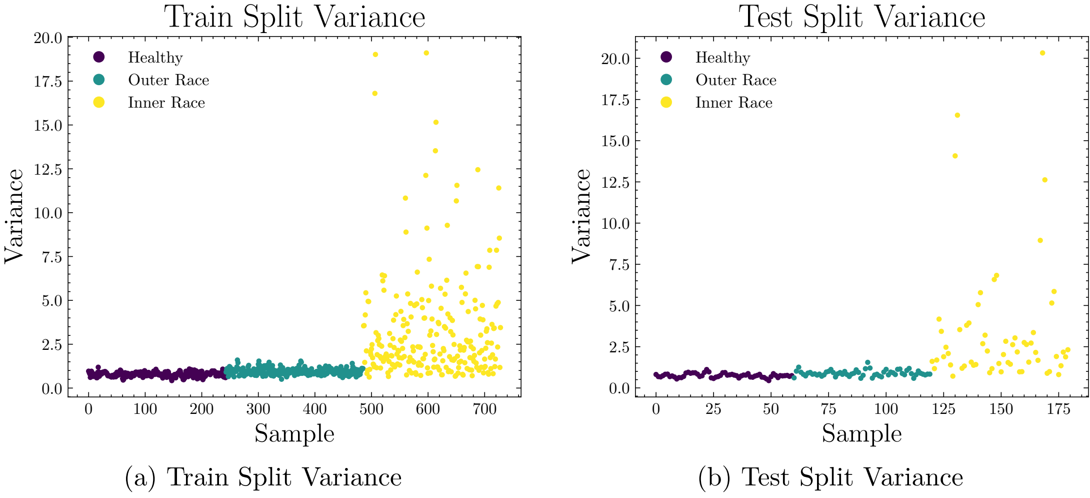
In this Annex, the graphs of the statistical features calculated in the preprocessing of the datasets of each case study are shown.

A.1. MFPT Features plots



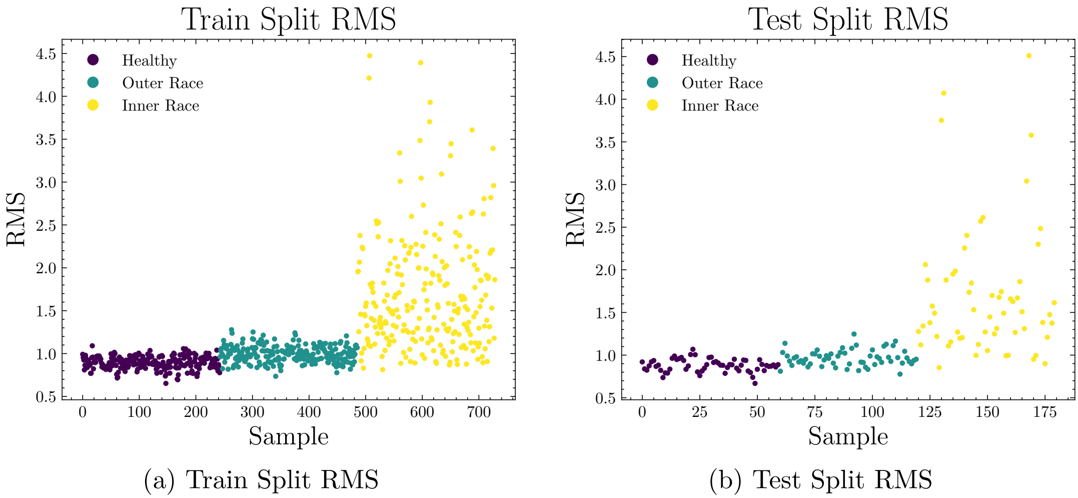
(a) Train Split Mean (b) Test Split Mean

Figure A.1.1: MFPT Dataset Mean Plot



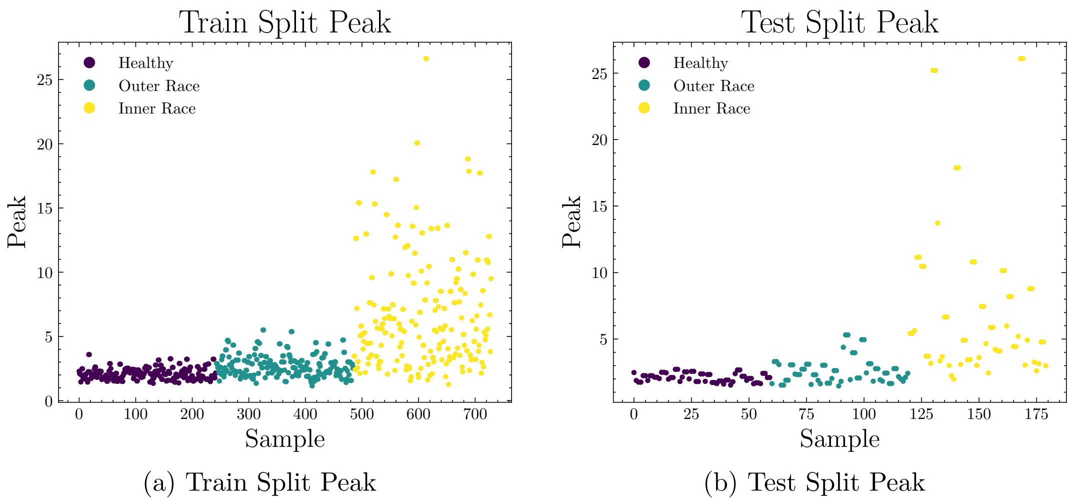
(a) Train Split Variance (b) Test Split Variance

Figure A.1.2: MFPT Dataset Variance Plot



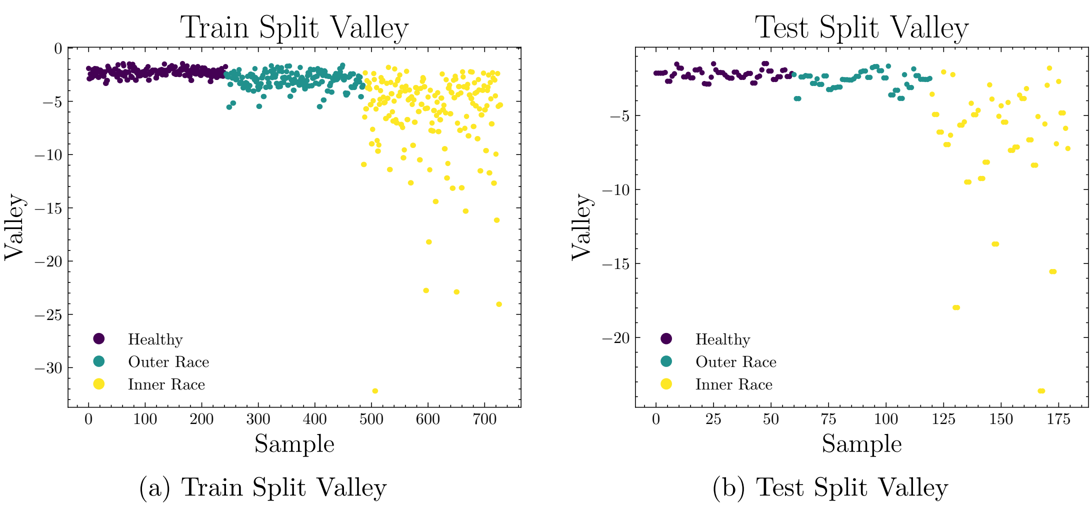
(a) Train Split RMS (b) Test Split RMS

Figure A.1.3: MFPT Dataset RMS Plot



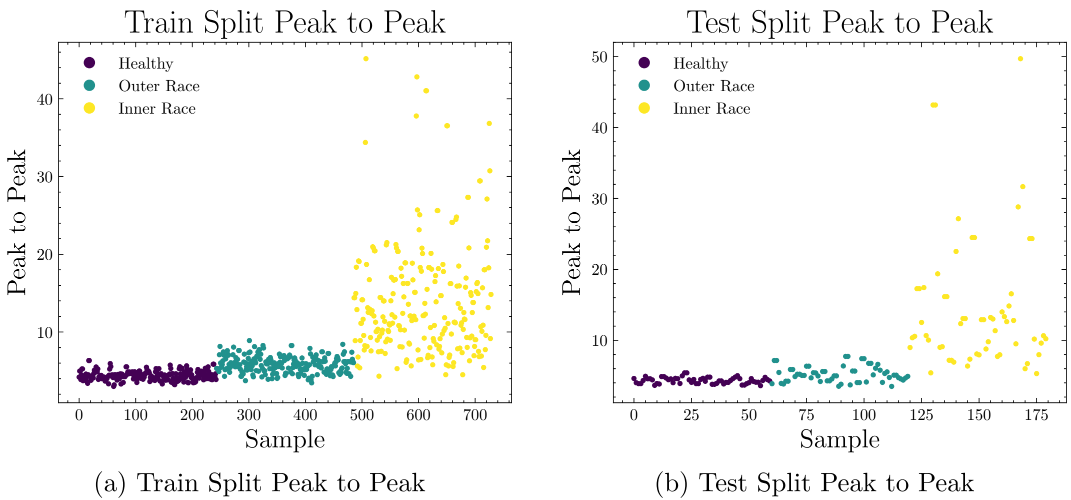
(a) Train Split Peak (b) Test Split Peak

Figure A.1.4: MFPT Dataset Peak Plot



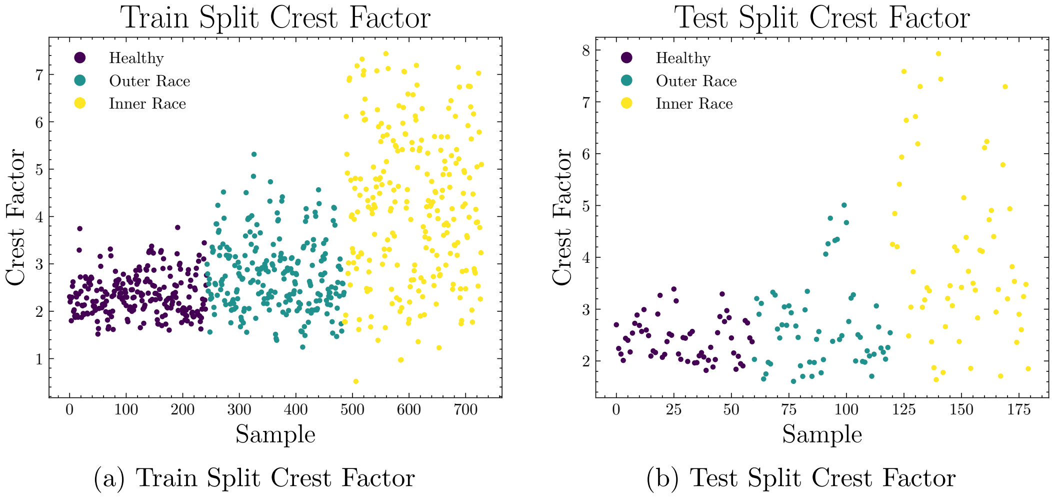
(a) Train Split Valley (b) Test Split Valley

Figure A.5: MFPT Dataset Valley Plot



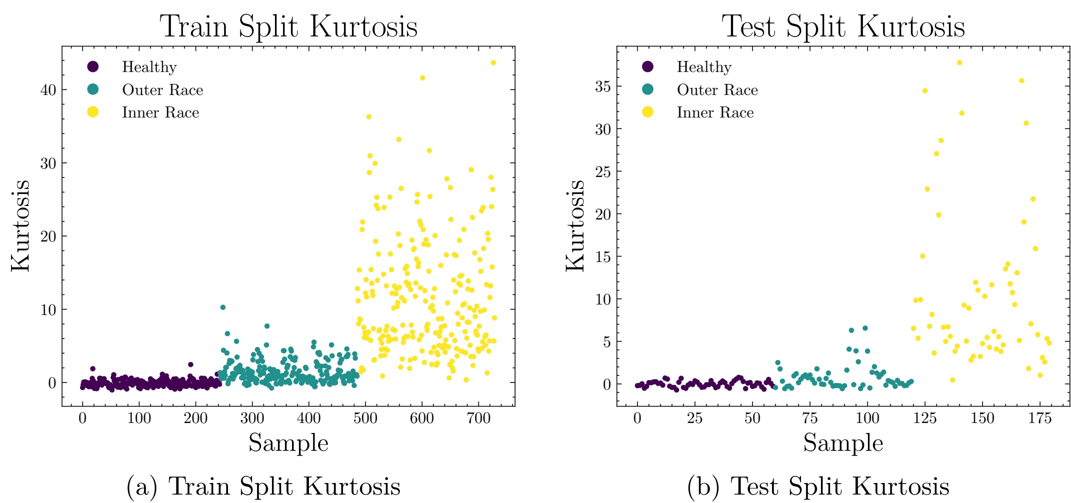
(a) Train Split Peak to Peak (b) Test Split Peak to Peak

Figure A.6: MFPT Dataset Peak to Peak Plot



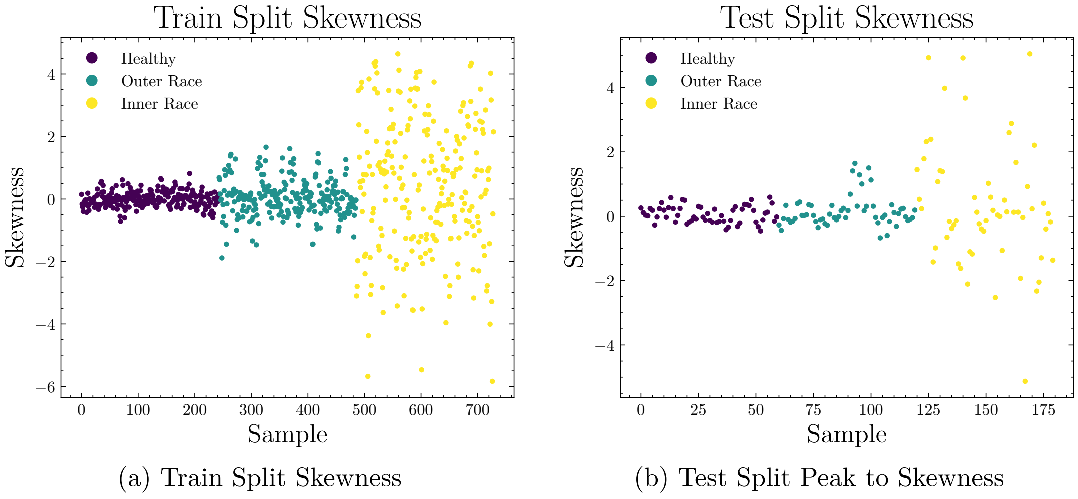
(a) Train Split Crest Factor (b) Test Split Crest Factor

Figure A.7: MFPT Dataset Crest Factor Plot



(a) Train Split Kurtosis (b) Test Split Kurtosis

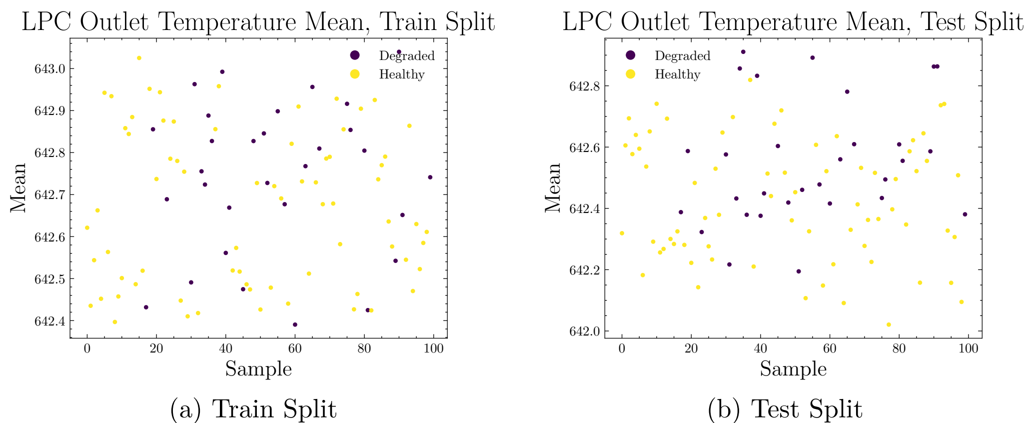
Figure A.8: MFPT Dataset Kurtosis Plot



(a) Train Split Skewness (b) Test Split Peak to Skewness

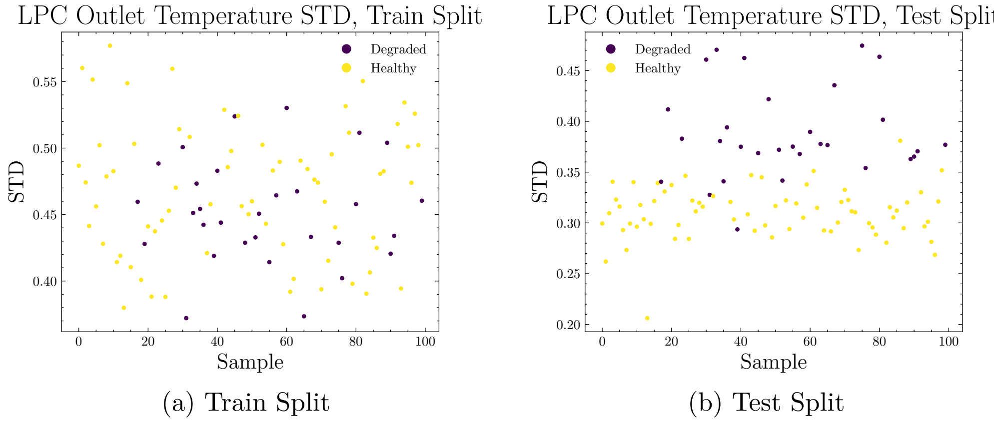
Figure A.9: MFPT Dataset Skewness Plot

A.2. C-MAPSS FD001 Features plots



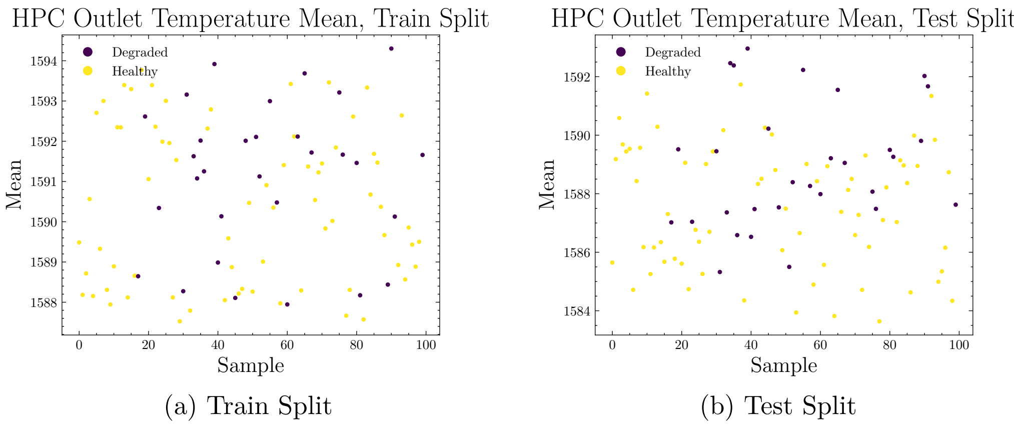
(a) Train Split (b) Test Split

Figure A.2.1: C-MAPSS FD001 Dataset LPC Outlet Temperature Mean Plot



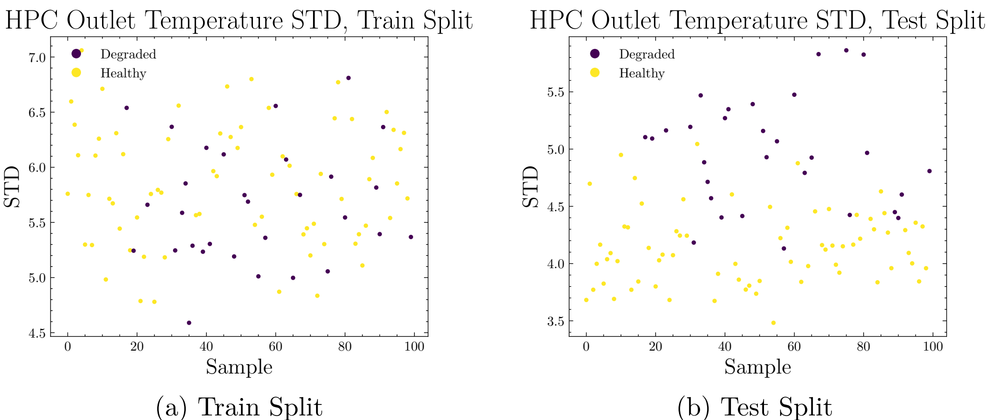
(a) Train Split (b) Test Split

Figure A.2.2: C-MAPSS FD001 Dataset LPC Outlet Temperature Standard Deviation Plot



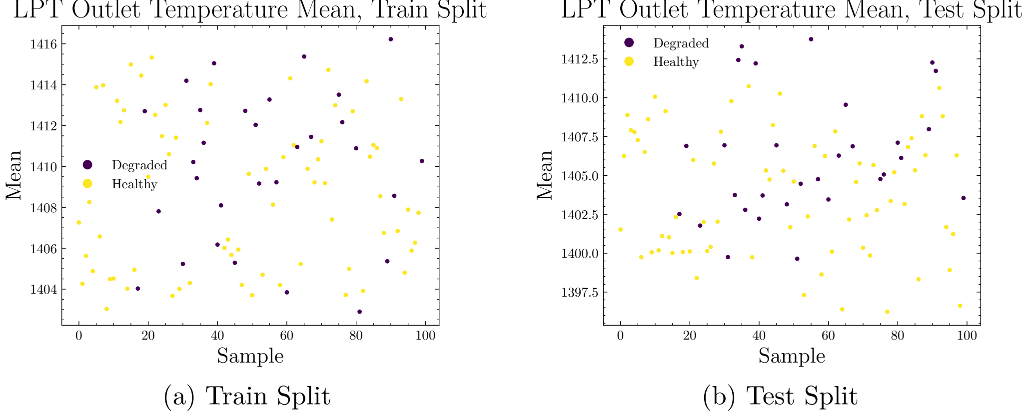
(a) Train Split (b) Test Split

Figure A.2.3: C-MAPSS FD001 Dataset HPC Outlet Temperature Mean Plot



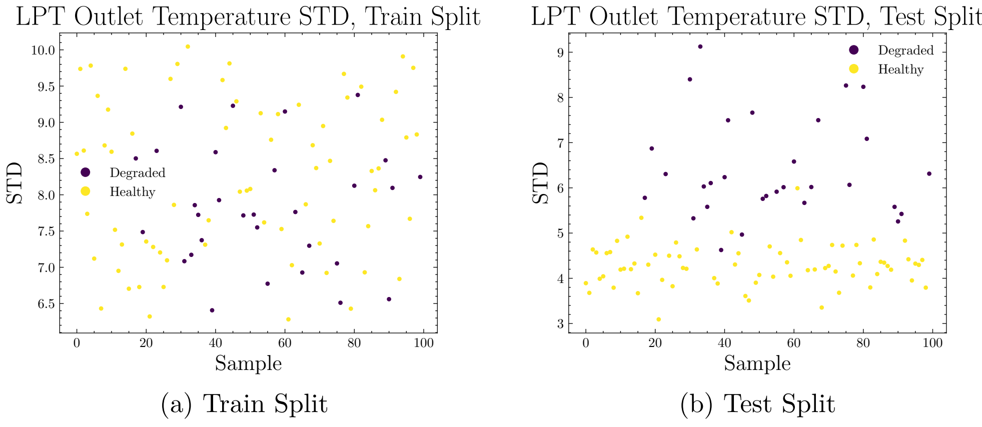
(a) Train Split (b) Test Split

Figure A.2.4: C-MAPSS FD001 Dataset HPC Outlet Temperature Standard Deviation Plot



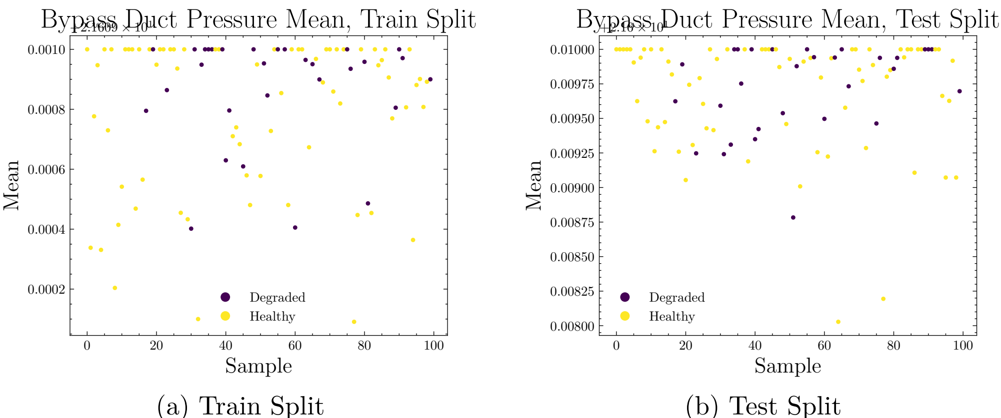
(a) Train Split (b) Test Split

Figure A.2.5: C-MAPSS FD001 Dataset LPT Outlet Temperature Mean Plot



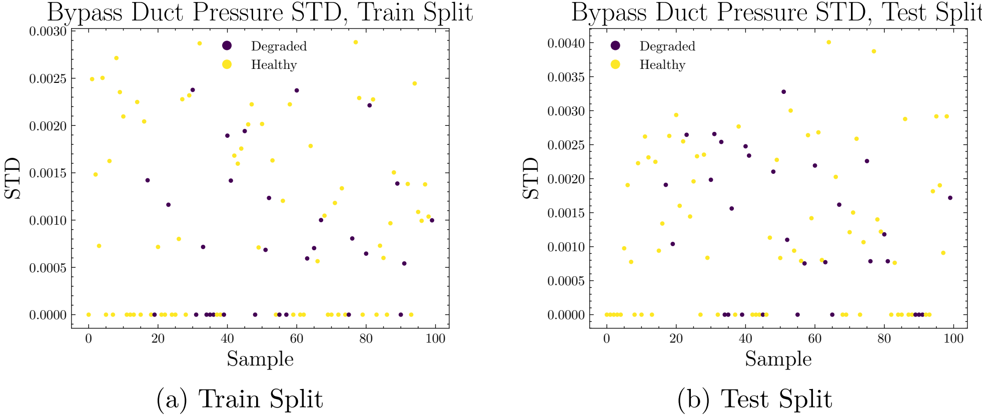
(a) Train Split (b) Test Split

Figure A.2.6: C-MAPSS FD001 Dataset LPT Outlet Temperature Standard Deviation Plot



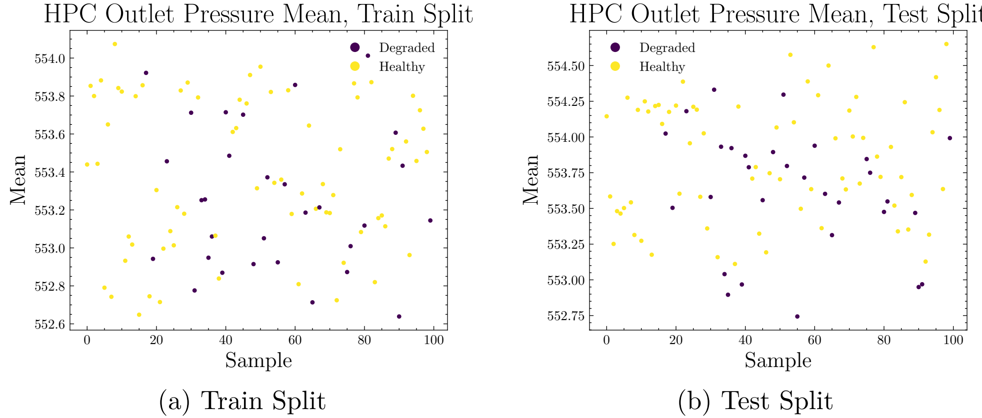
(a) Train Split (b) Test Split

Figure A.2.7: C-MAPSS FD001 Dataset Bypass Duct Pressure Mean Plot



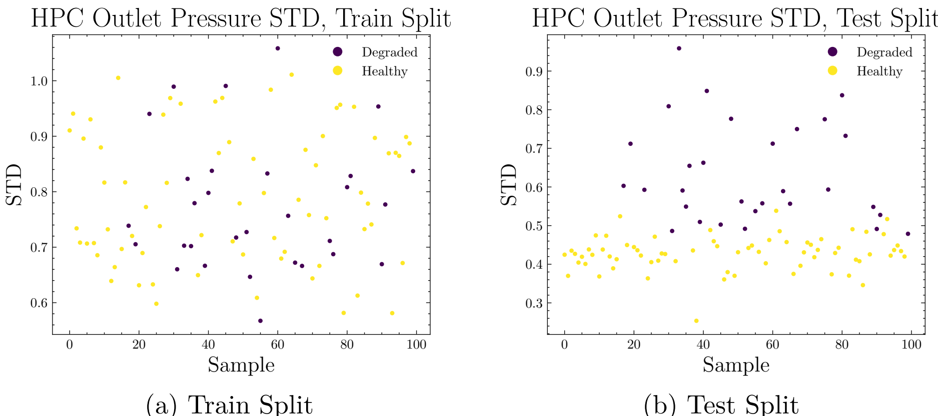
(a) Train Split (b) Test Split

Figure A.2.8: C-MAPSS FD001 Dataset Bypass Duct Pressure Standard Deviation Plot



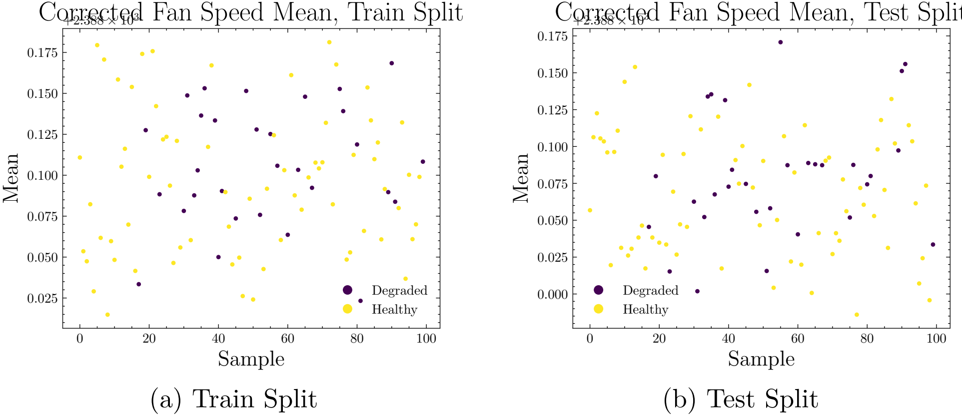
(a) Train Split (b) Test Split

Figure A.2.9: C-MAPSS FD001 Dataset HPC Outlet Pressure Mean Plot



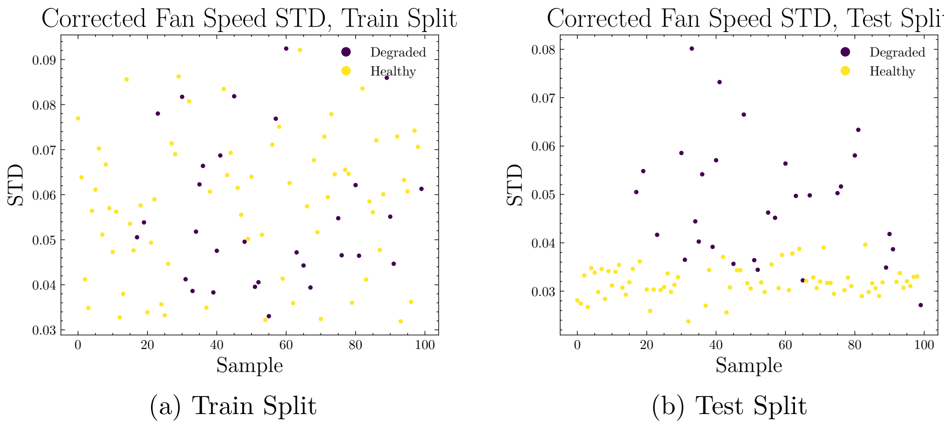
(a) Train Split (b) Test Split

Figure A.2.10: C-MAPSS FD001 Dataset HPC Outlet Pressure Standard Deviation Plot



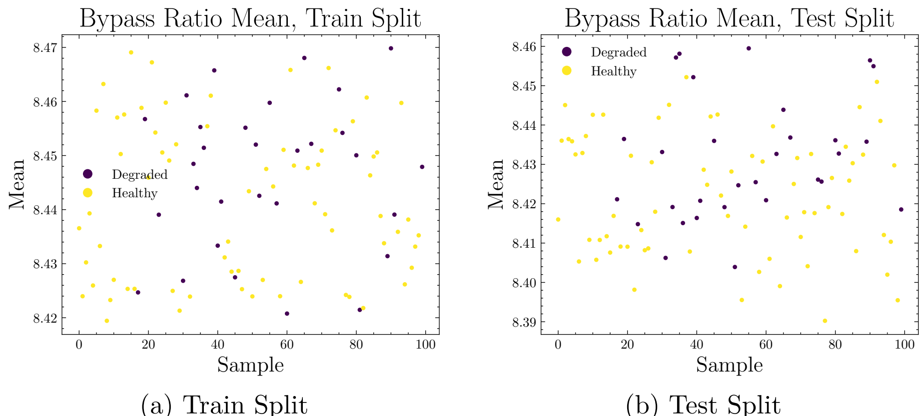
(a) Train Split (b) Test Split

Figure A.2.11: C-MAPSS FD001 Dataset Corrected Fan Speed Mean Plot



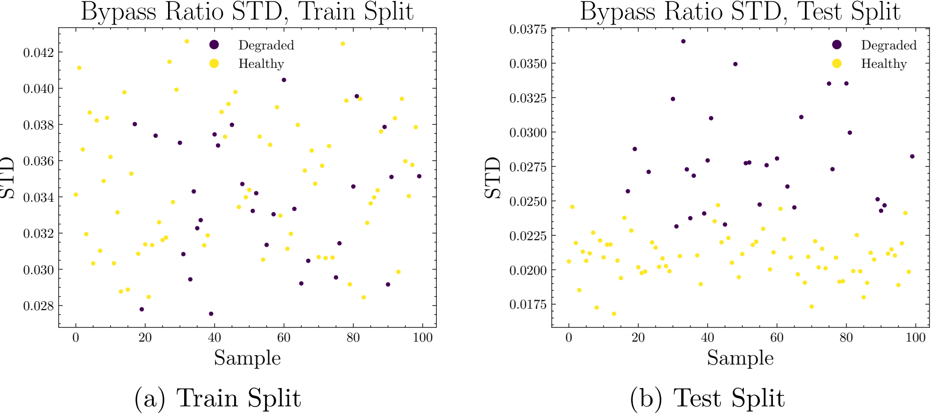
(a) Train Split (b) Test Split

Figure A.2.12: C-MAPSS FD001 Dataset Corrected Fan Speed Standard Deviation Plot



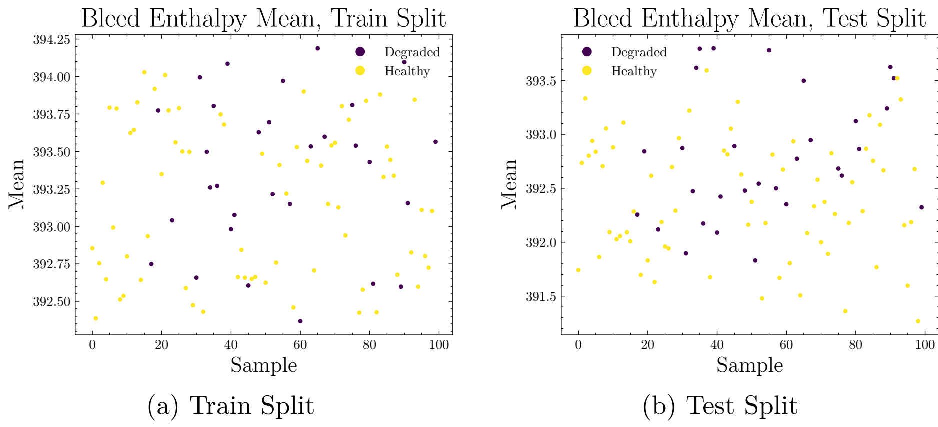
(a) Train Split (b) Test Split

Figure A.2.13: C-MAPSS FD001 Dataset Bypass Ratio Mean Plot



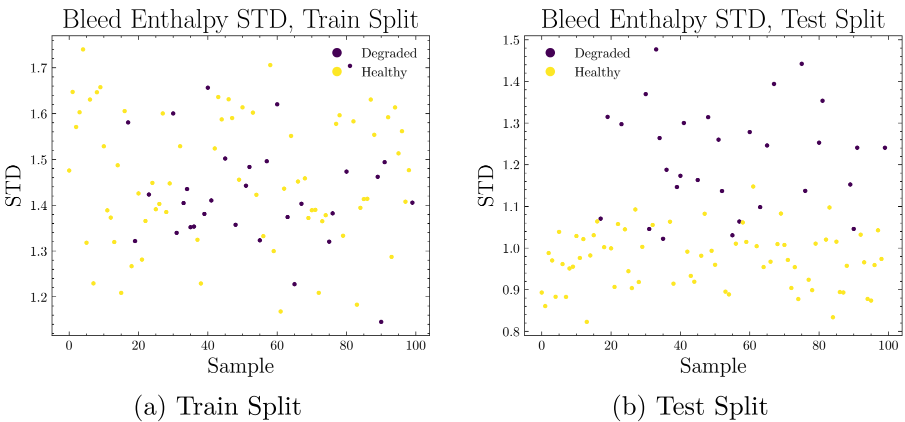
(a) Train Split (b) Test Split

Figure A.2.14: C-MAPSS FD001 Dataset Bypass Ratio Standard Deviation Plot



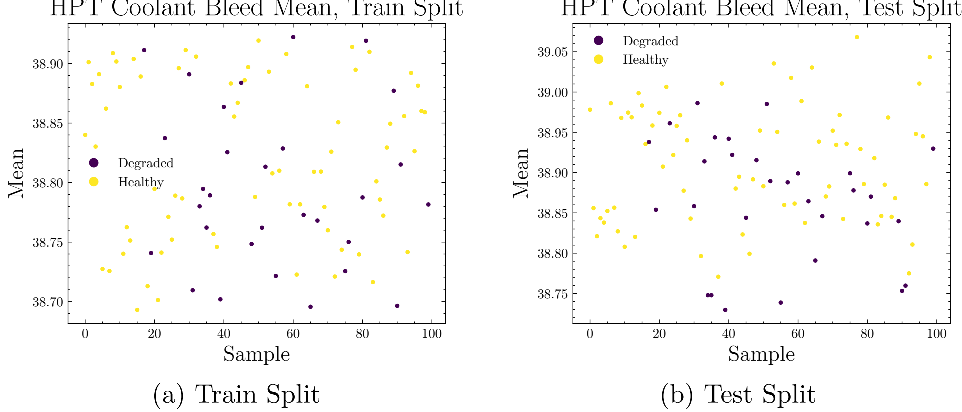
(a) Train Split (b) Test Split

Figure A.2.15: C-MAPSS FD001 Dataset Bleed Enthalpy Mean Plot



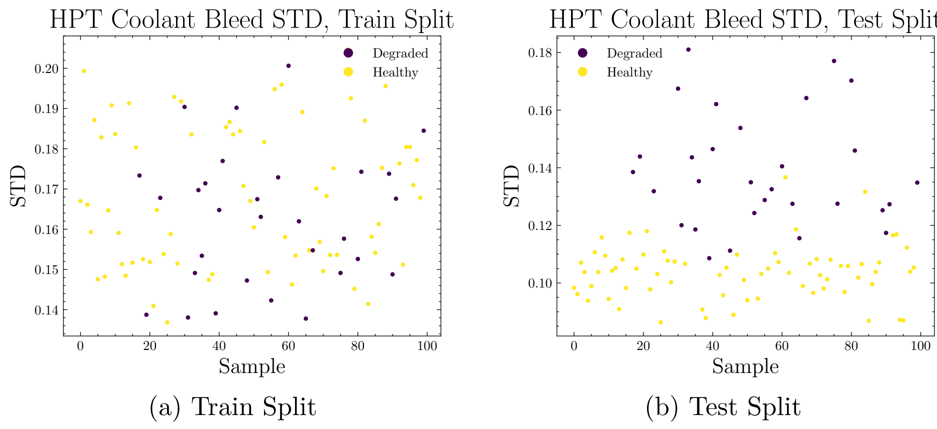
(a) Train Split (b) Test Split

Figure A.2.16: C-MAPSS FD001 Dataset Bleed Enthalpy Standard Deviation Plot



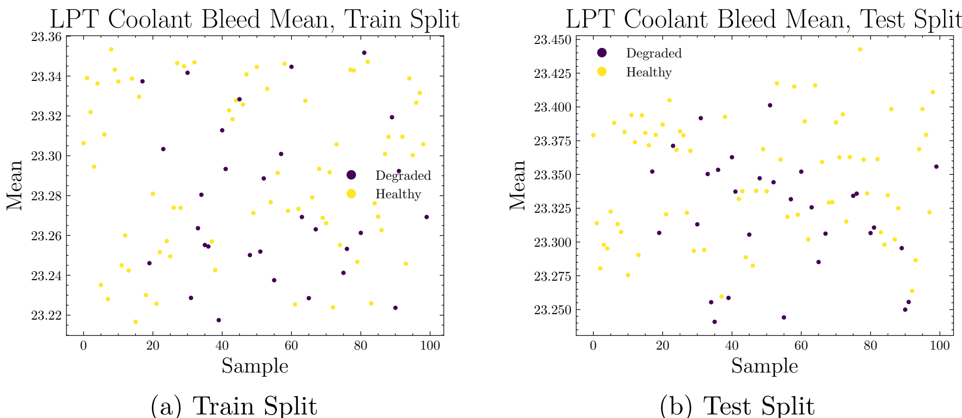
(a) Train Split (b) Test Split

Figure A.2.17: C-MAPSS FD001 Dataset HPT Coolant Bleed Mean Plot



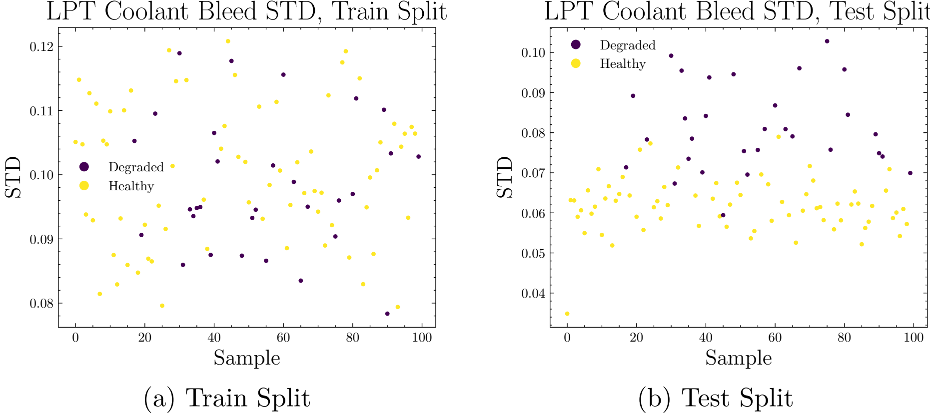
(a) Train Split (b) Test Split

Figure A.2.18: C-MAPSS FD001 Dataset HPT Coolant Bleed Standard Deviation Plot



(a) Train Split (b) Test Split

Figure A.2.19: C-MAPSS FD001 Dataset HPT Coolant Bleed Mean Plot



(a) Train Split (b) Test Split

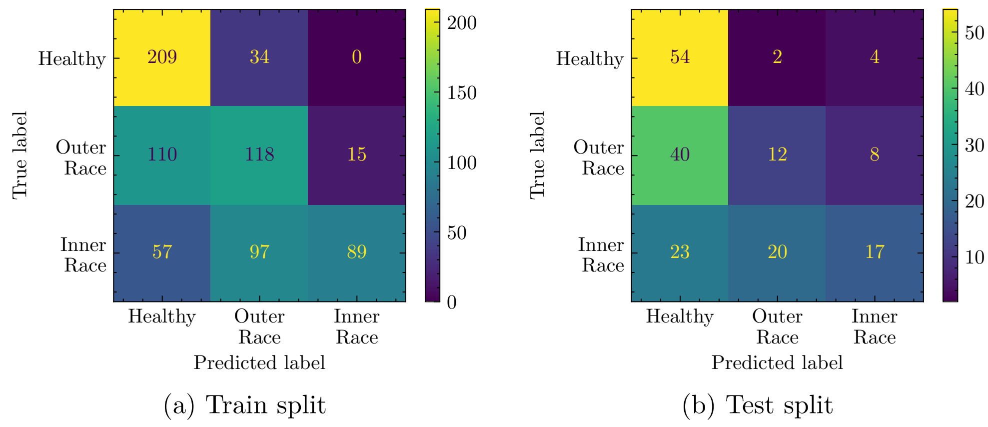
Figure A.2.20: C-MAPSS FD001 Dataset HPT Coolant Bleed Standard Deviation Plot

Annex B | Models Confusion Matrices

In this Annex, the confusion matrices of all the models evaluated with the datasets of each case study are shown.

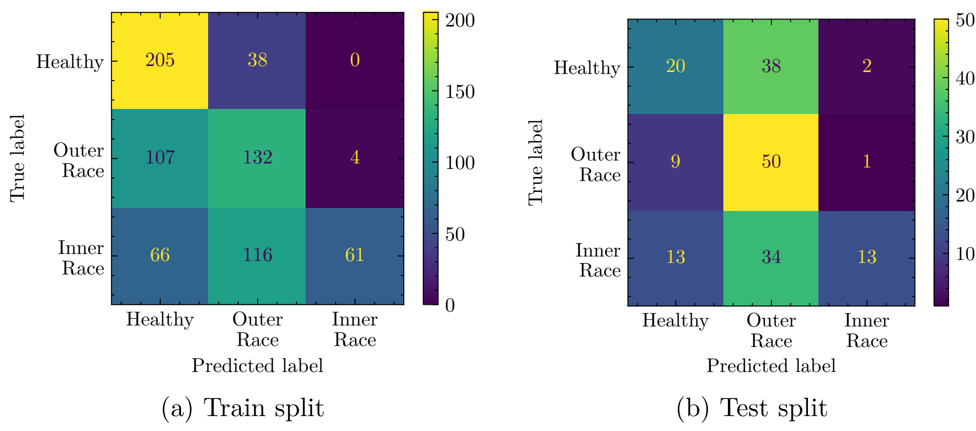
B.1. Variational Quantum Classifier

B.1.1. MFPT Fault Dataset



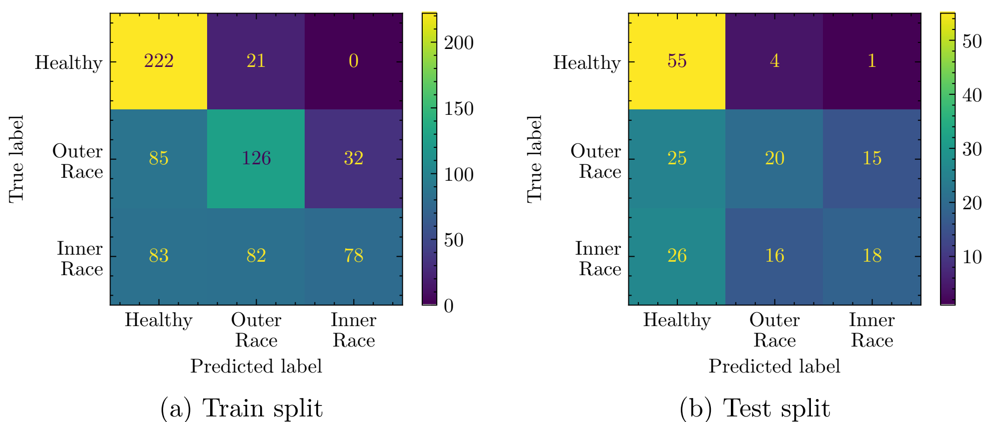
(a) Train split (b) Test split

Figure B.1.1: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to MFPT



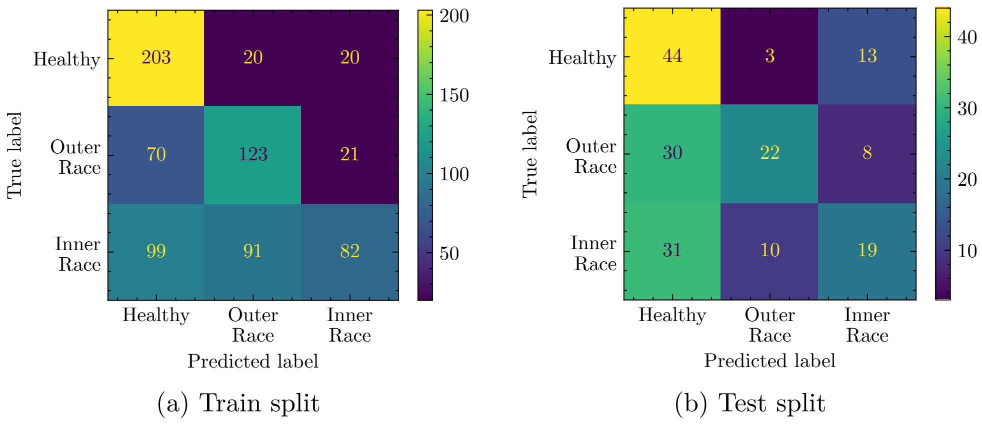
(a) Train split (b) Test split

Figure B.1.2: Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and COBYLA applied to MFPT



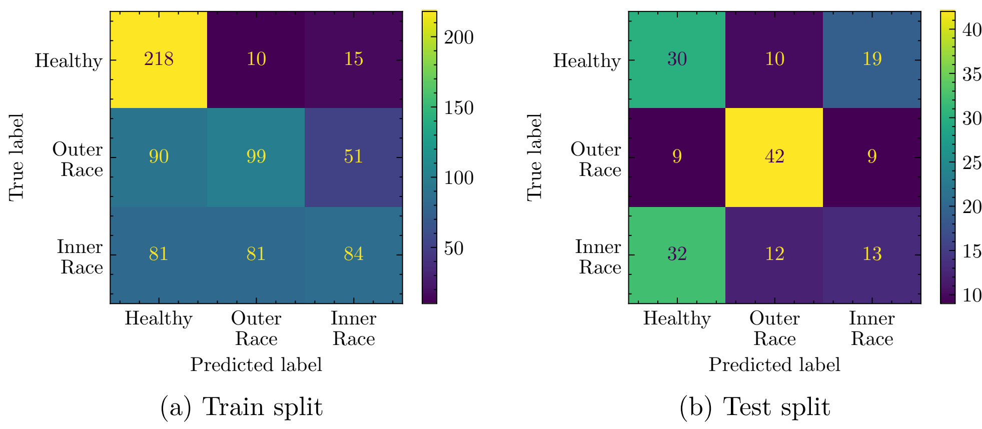
(a) Train split (b) Test split

Figure B.1.3: Confussion Matrixes for VQC with RawFeatures, EfficientSU2 and COBYLA applied to MFPT



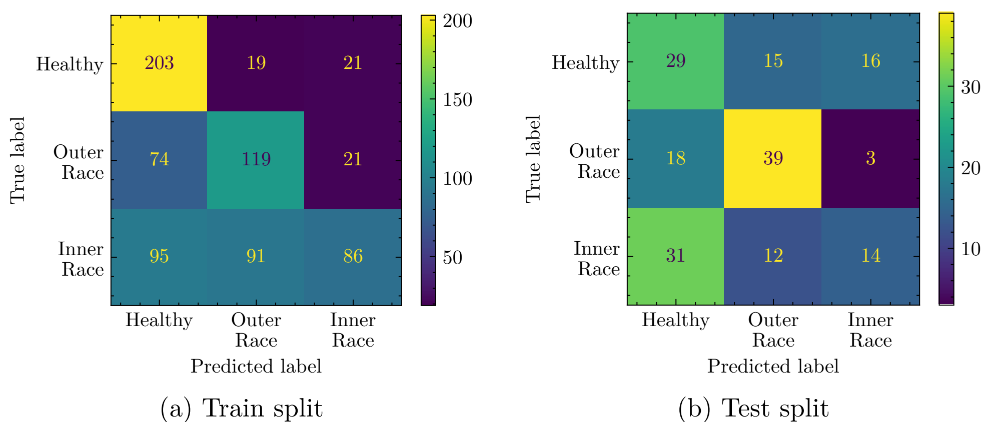
(a) Train split (b) Test split

Figure B.1.4: Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and COBYLA applied to MFPT



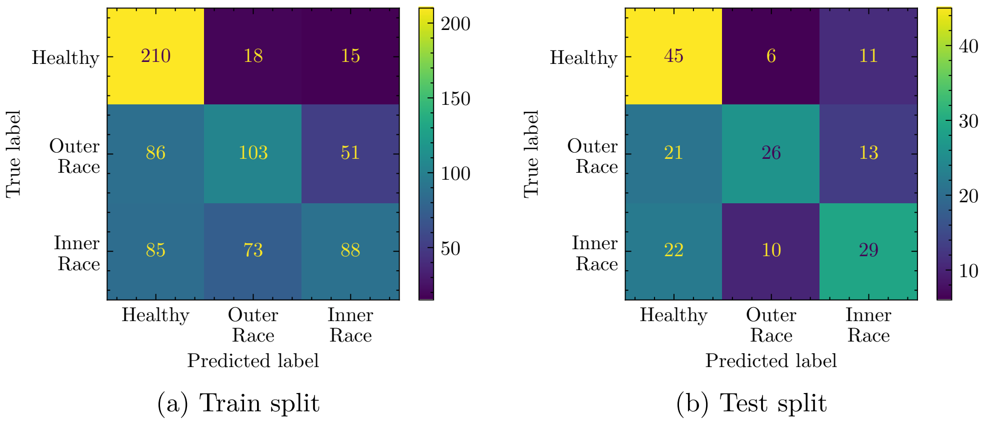
(a) Train split (b) Test split

Figure B.1.5: Confussion Matrixes for VQC with RawFeatures, RealAmplitudes and COBYLA applied to MFPT



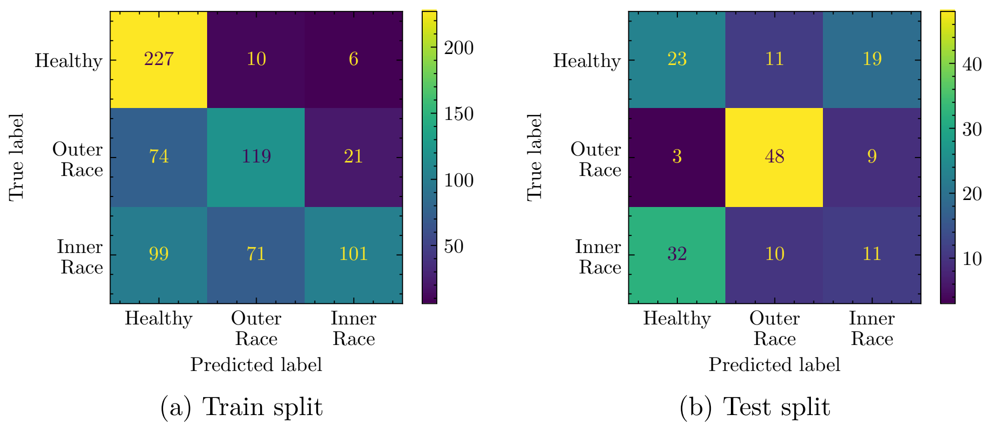
(a) Train split (b) Test split

Figure B.1.6: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SLSQP applied to MFPT



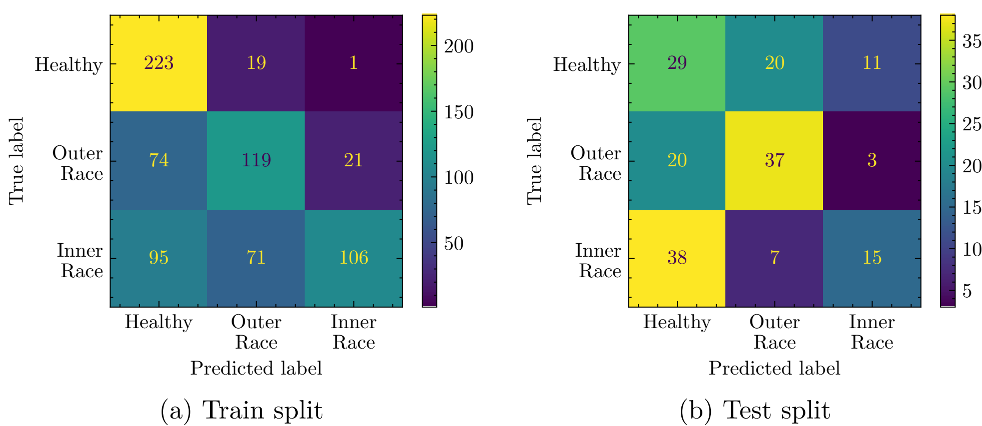
(a) Train split (b) Test split

Figure B.1.7: Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and SLSQP applied to MFPT



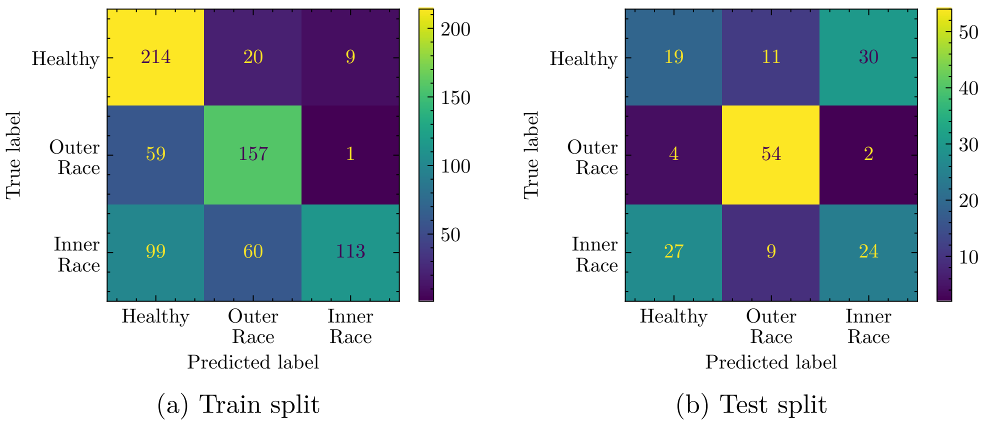
(a) Train split (b) Test split

Figure B.1.8: Confussion Matrixes for VQC with RawFeatures, RealAmplitudes and SLSQP applied to MFPT



(a) Train split (b) Test split

Figure B.1.9: Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and SLSQP applied to MFPT



(a) Train split (b) Test split

Figure B.1.10: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SPSA applied to MFPT

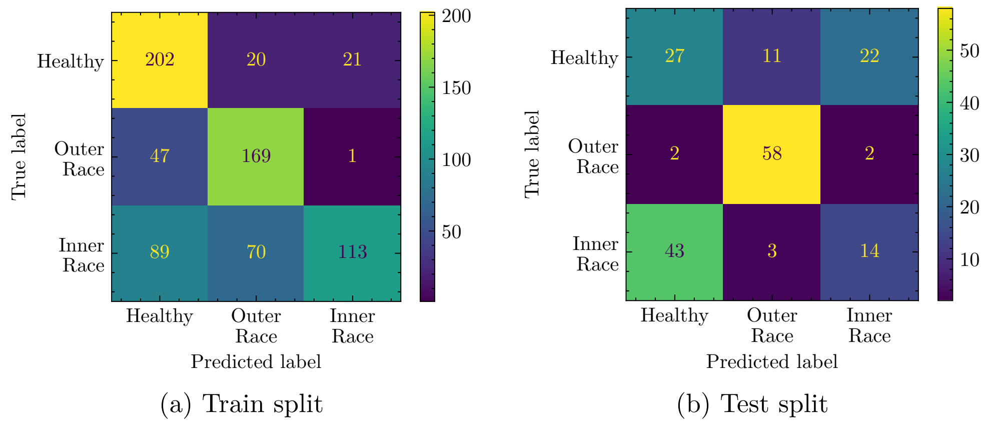


Figure B.1.11: Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and SPSA applied to MFPT

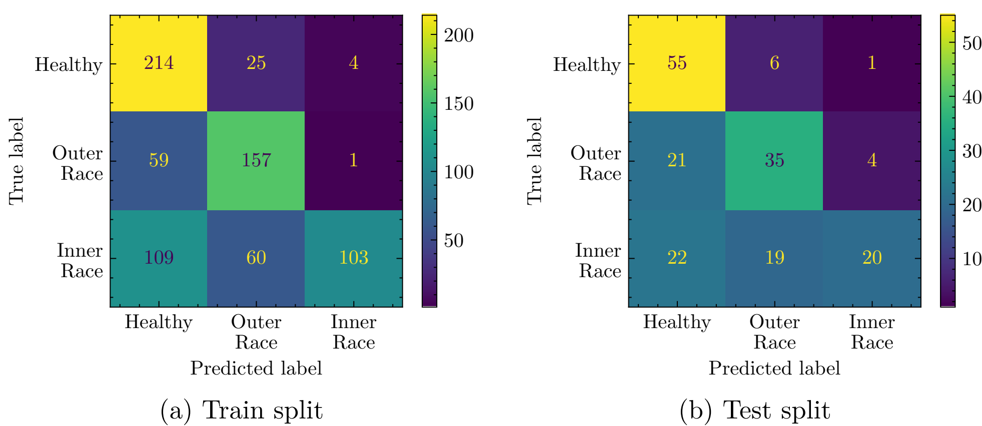


Figure B.1.12: Confussion Matrixes for VQC with RawFeatures, EfficientSU2 and SPSA applied to MFPT

B.1.1.1. FD001 Statistics Features

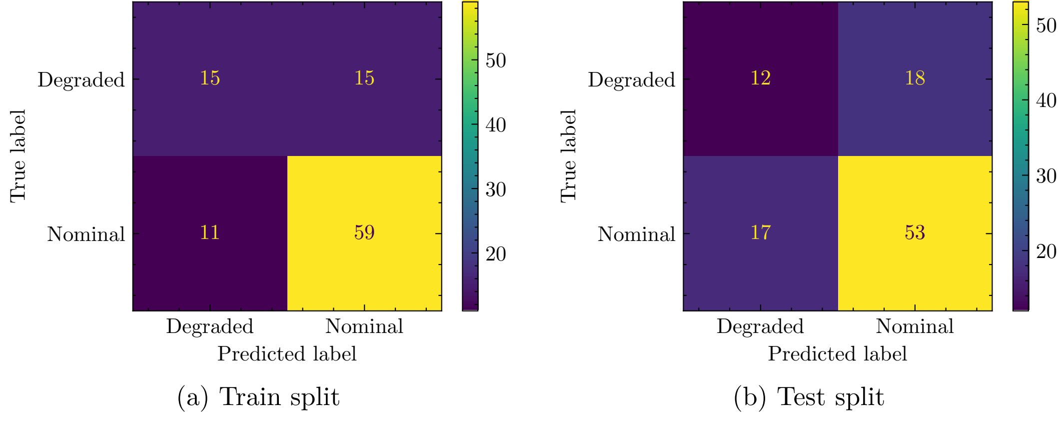
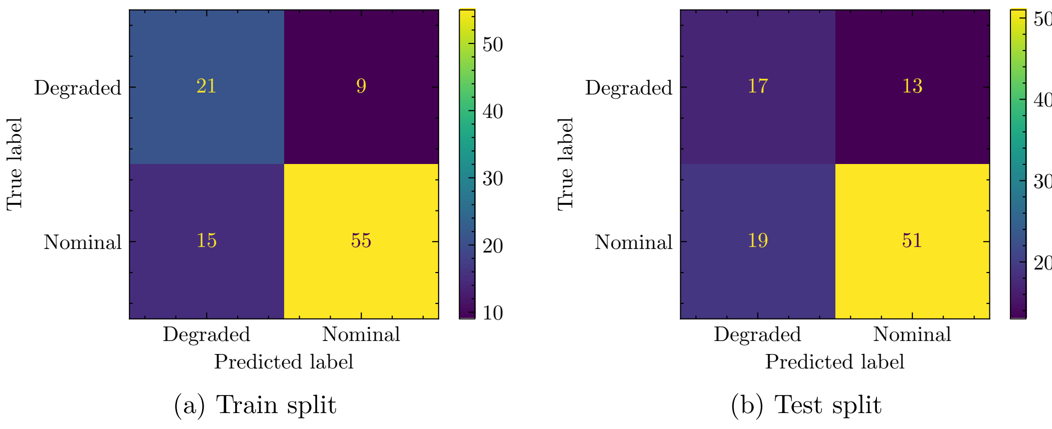
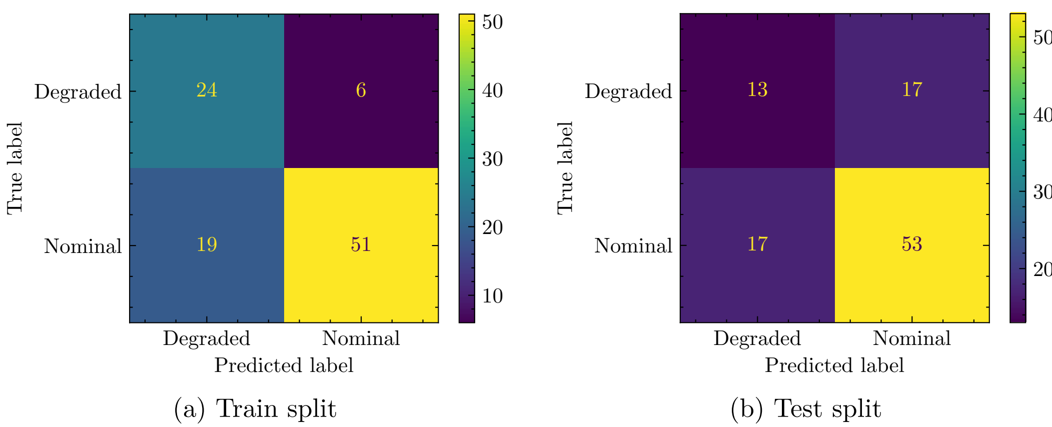


Figure B.1.13: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Statistics Features



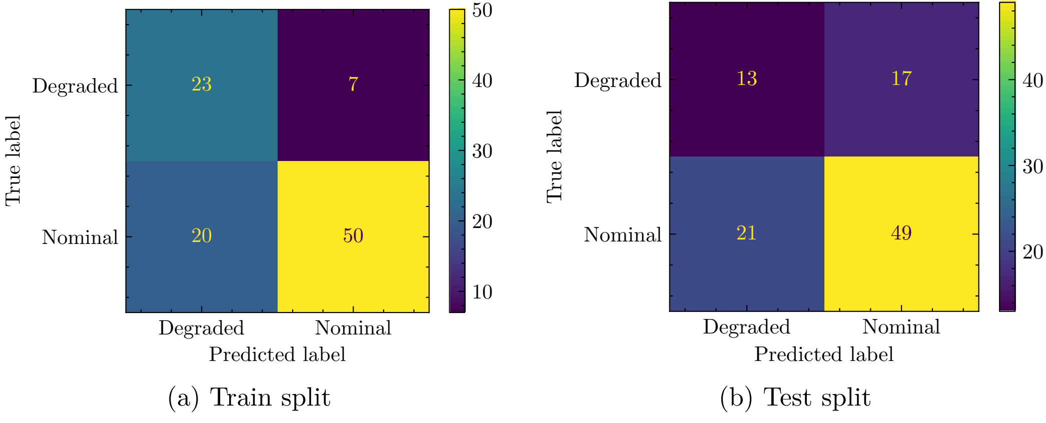
(a) Train split (b) Test split

Figure B.1.14: Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Statistics Features



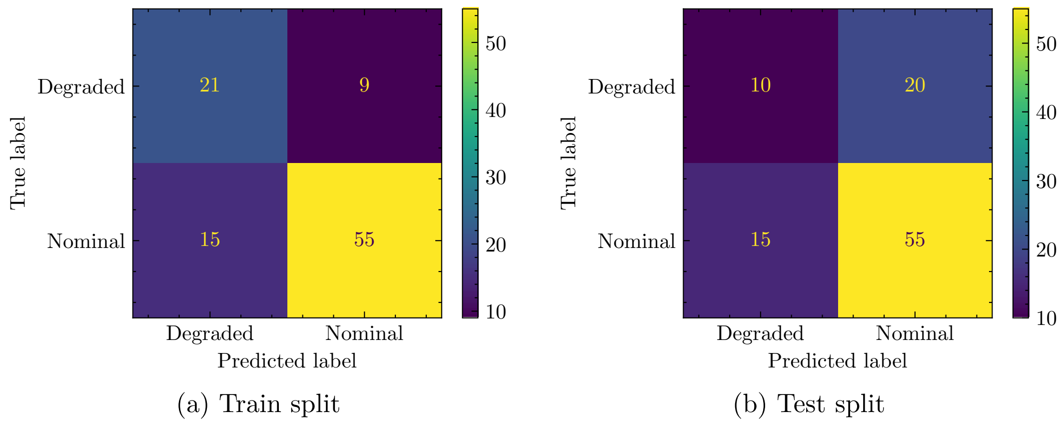
(a) Train split (b) Test split

Figure B.1.15: Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Statistics Features



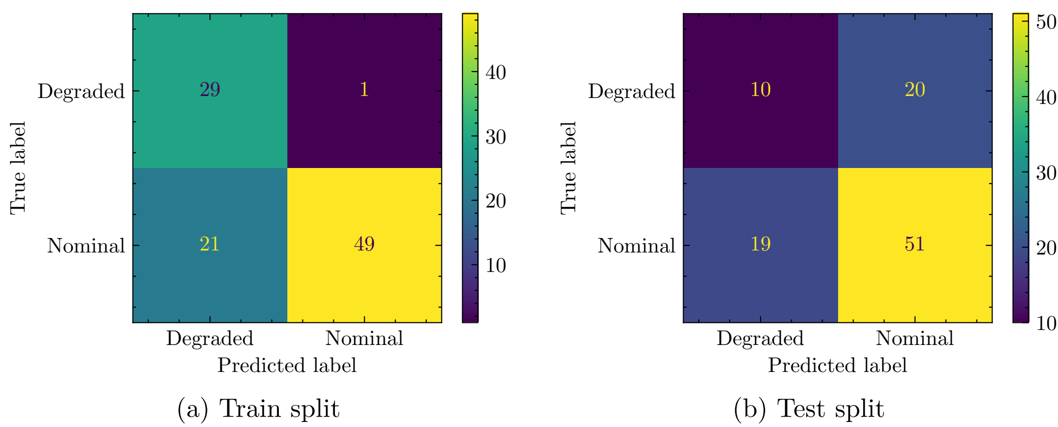
(a) Train split (b) Test split

Figure B.1.16: Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Statistics Features



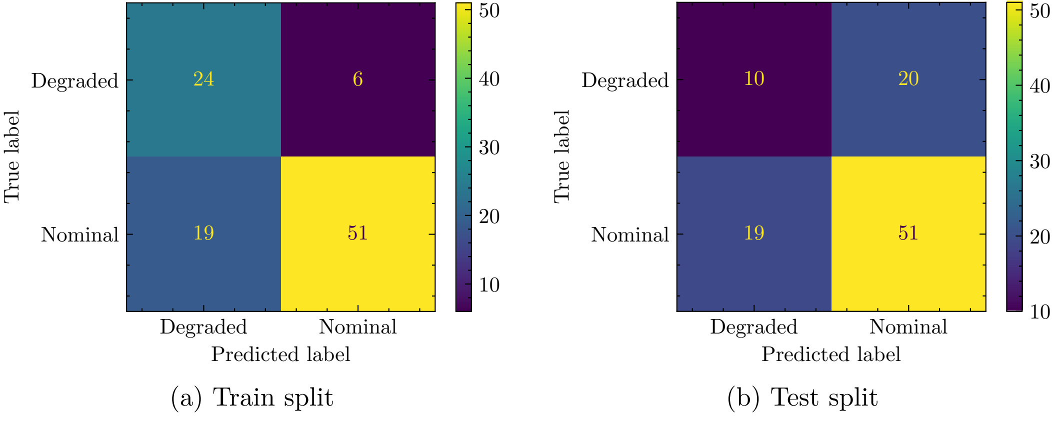
(a) Train split (b) Test split

Figure B.1.17: Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Statistics Features



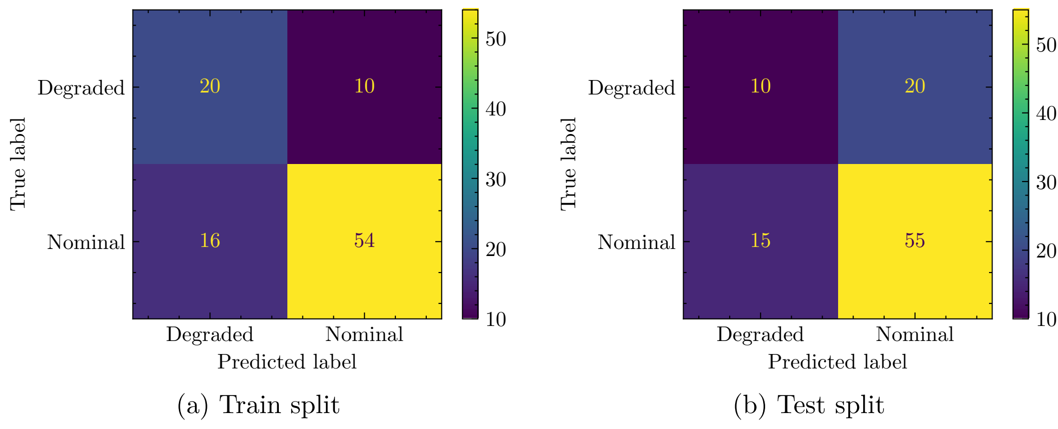
(a) Train split (b) Test split

Figure B.1.18: Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Statistics Features



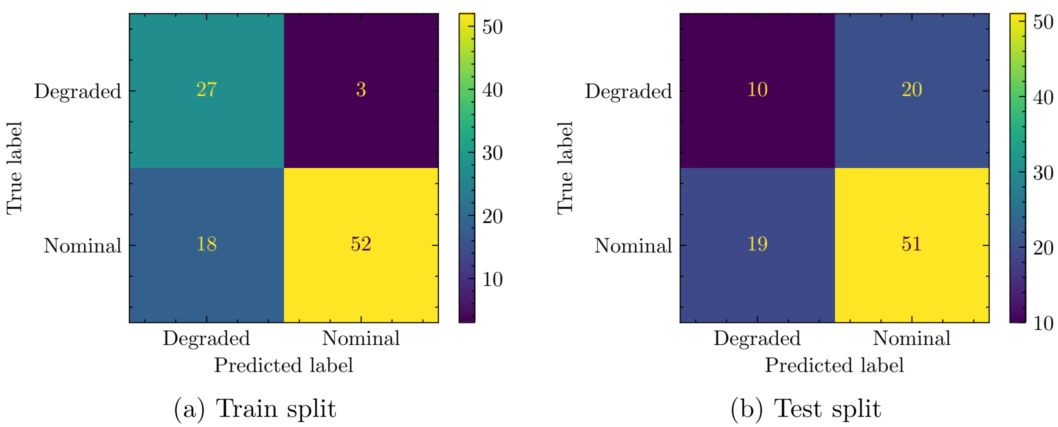
(a) Train split (b) Test split

Figure B.1.19: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Statistics Features



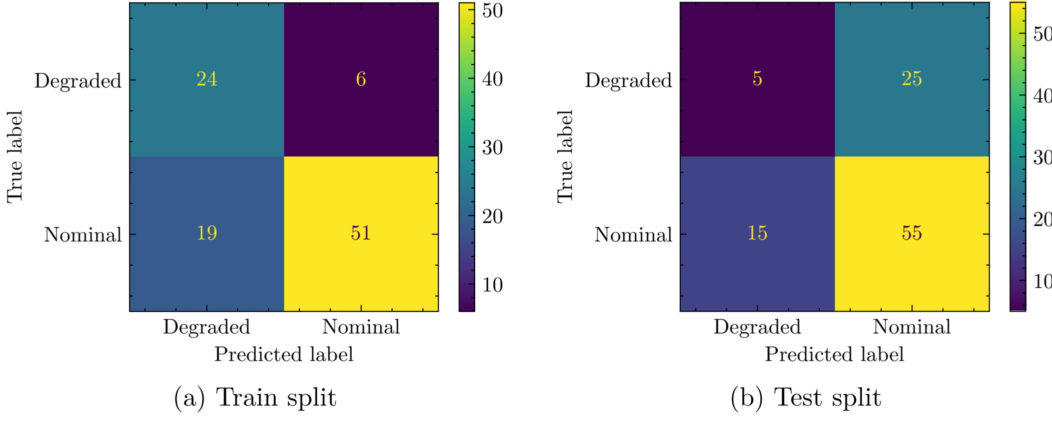
(a) Train split (b) Test split

Figure B.1.20: Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Statistics Features



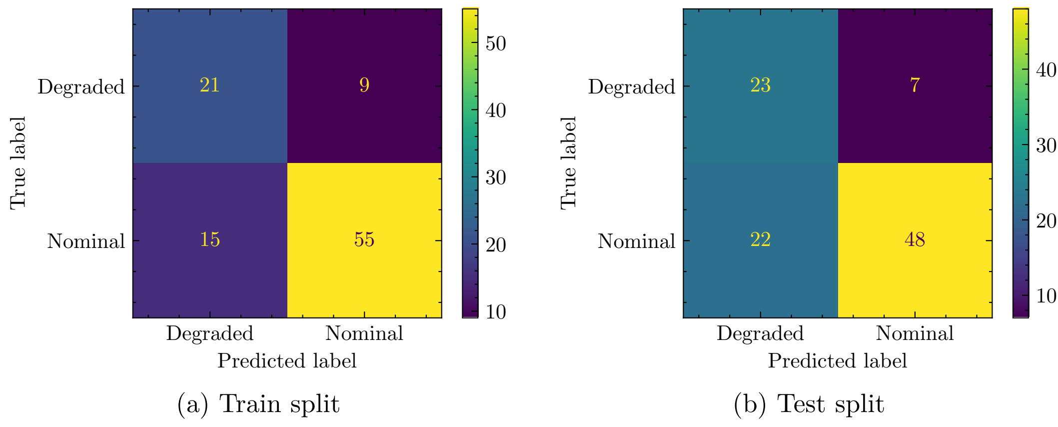
(a) Train split (b) Test split

Figure B.1.21: Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Statistics Features



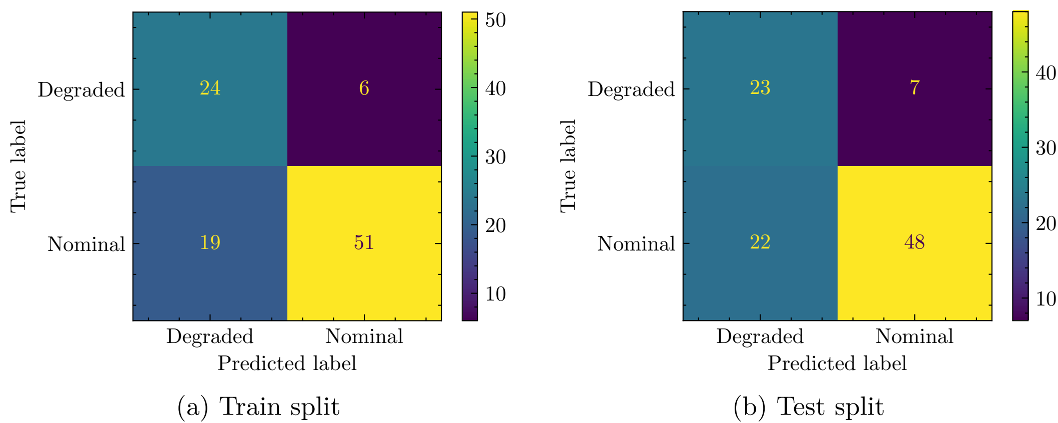
(a) Train split (b) Test split

Figure B.1.22: Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Statistics Features



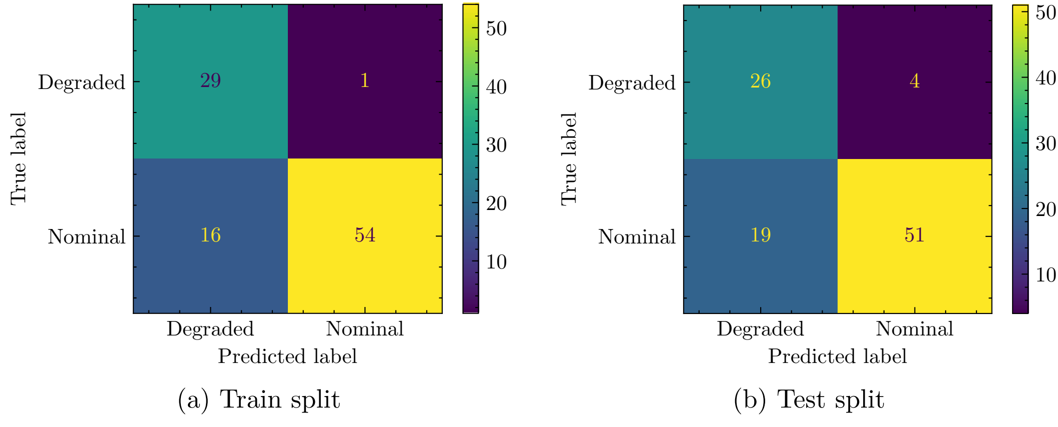
(a) Train split (b) Test split

Figure B.1.23: Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Statistics Features



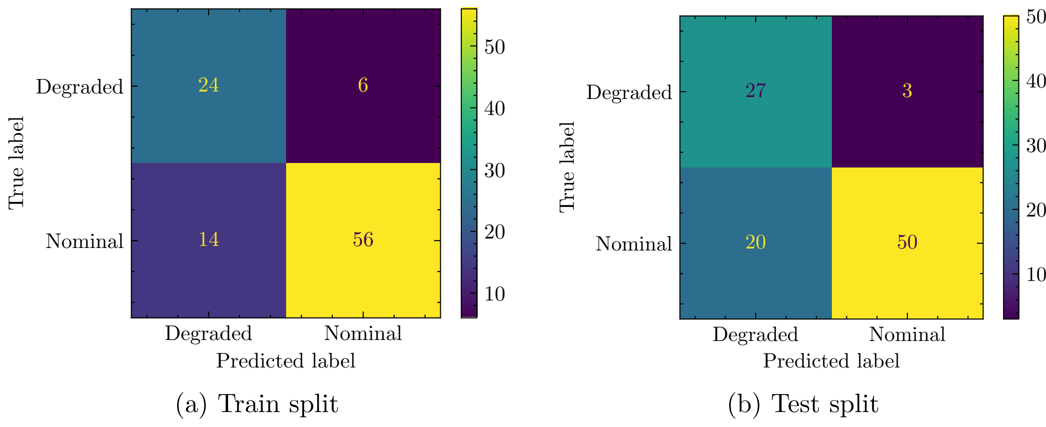
(a) Train split (b) Test split

Figure B.1.24: Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Statistics Features



(a) Train split (b) Test split

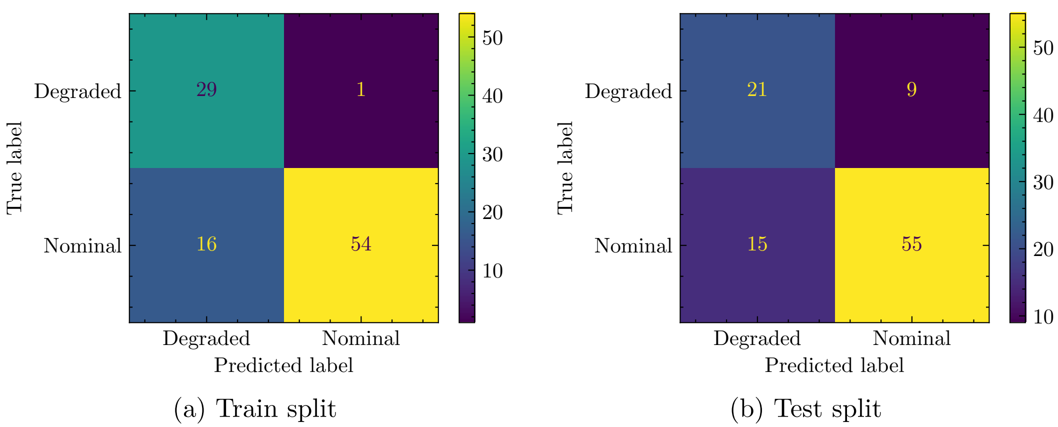
Figure B.1.25: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Statistics Features



(a) Train split

(b) Test split

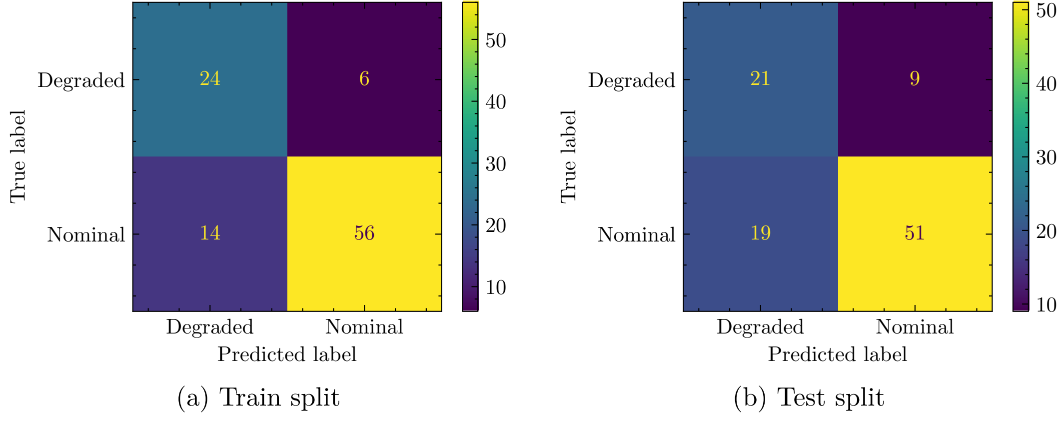
Figure B.1.26: Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Statistics Features



(a) Train split

(b) Test split

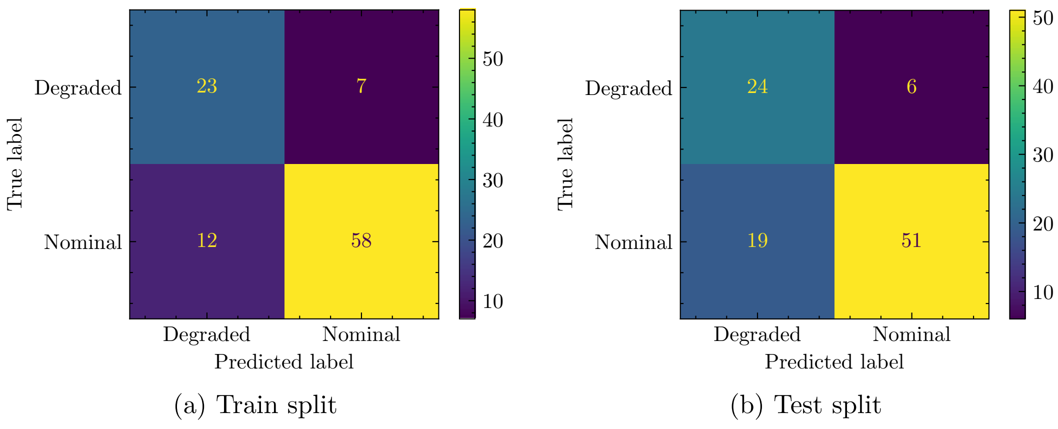
Figure B.1.27: Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Statistics Features



(a) Train split

(b) Test split

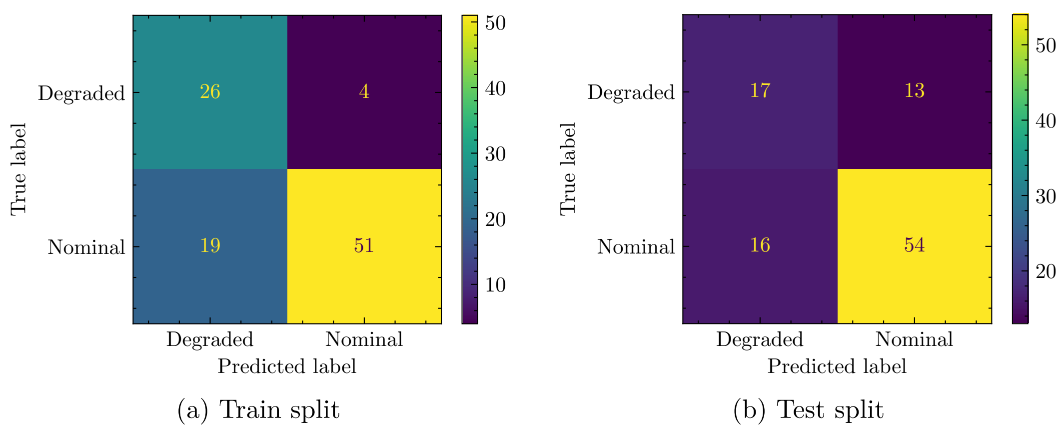
Figure B.1.28: Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Statistics Features



(a) Train split

(b) Test split

Figure B.1.29: Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Statistics Features

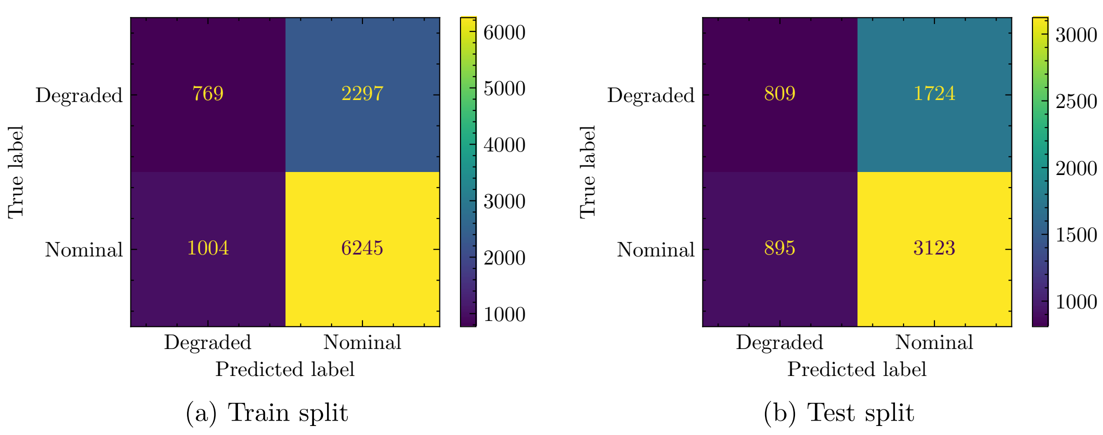


(a) Train split

(b) Test split

Figure B.1.30: Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Statistics Features

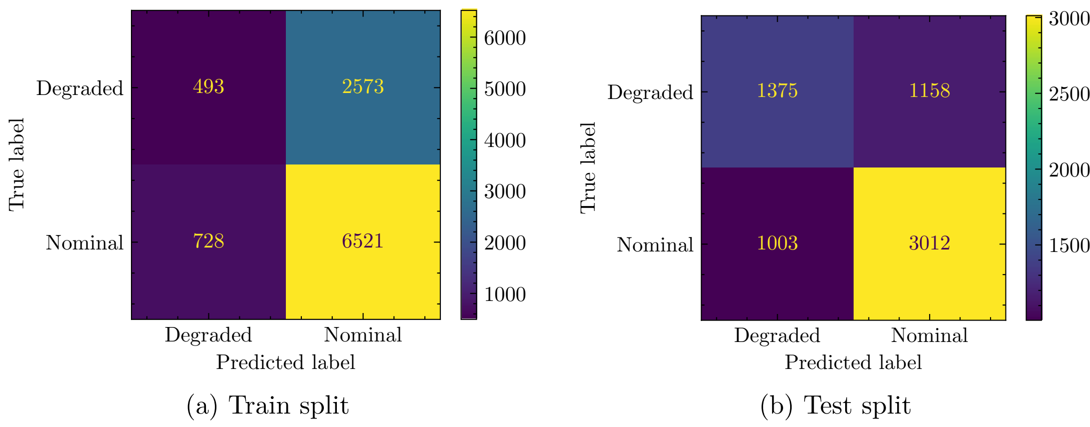
B.1.1.2. FD001 Sensors Features



(a) Train split

(b) Test split

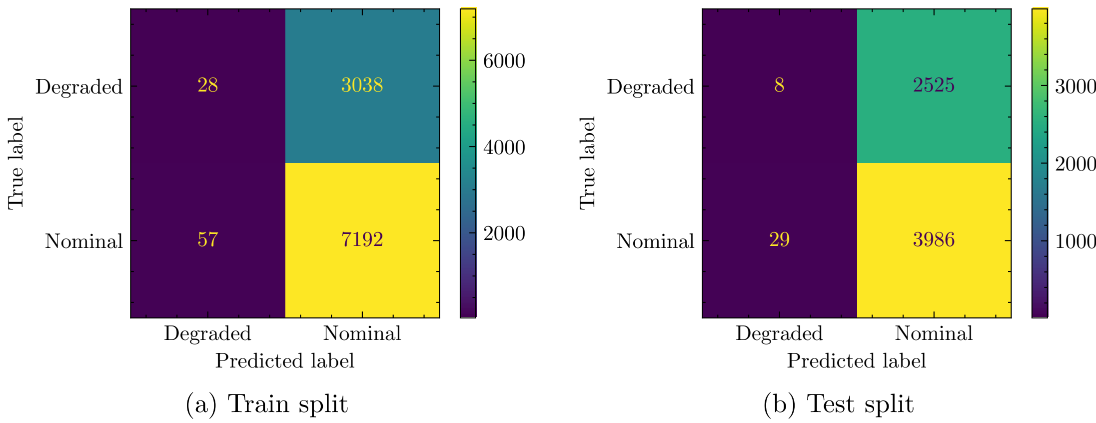
Figure B.1.31: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

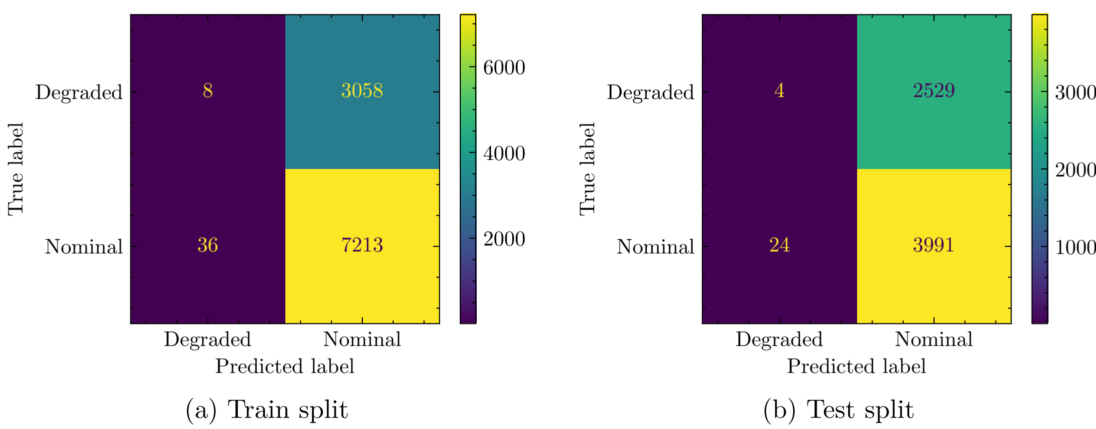
Figure B.1.32: Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

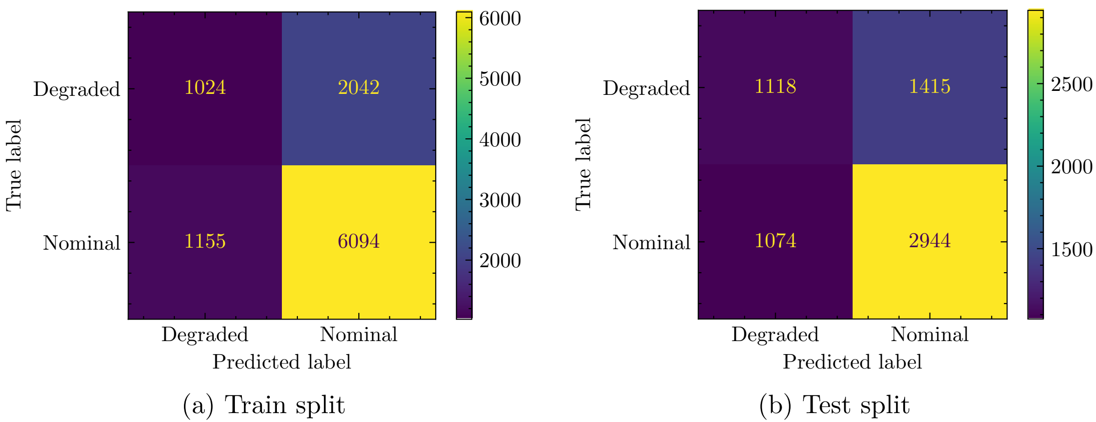
Figure B.1.33: Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

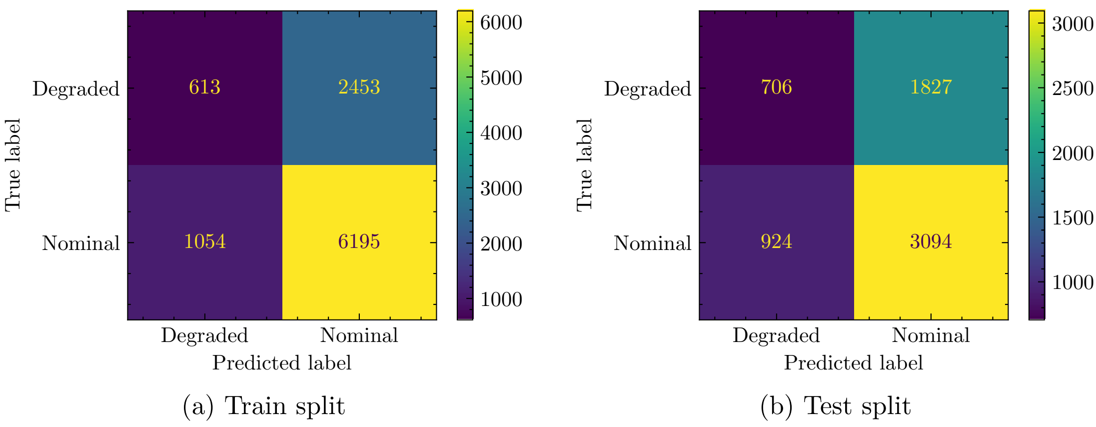
Figure B.1.34: Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

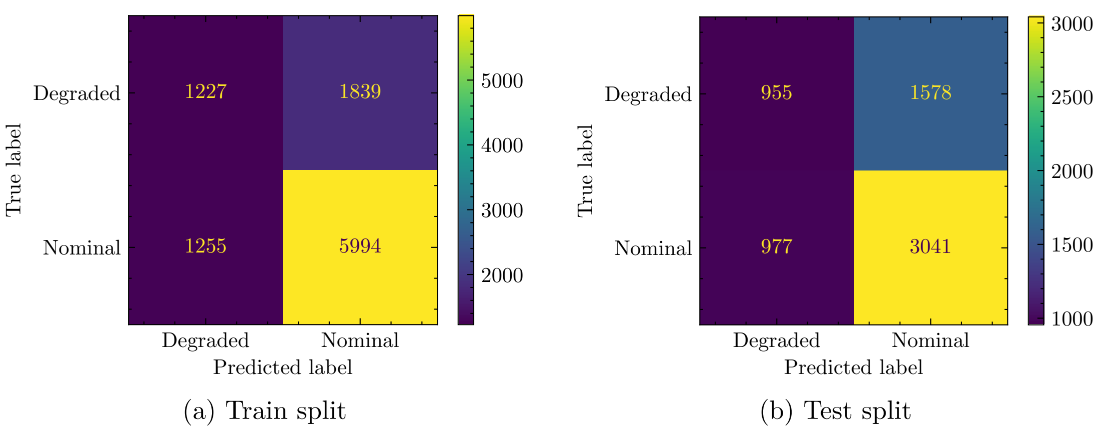
Figure B.1.35: Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and COBYLA applied to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

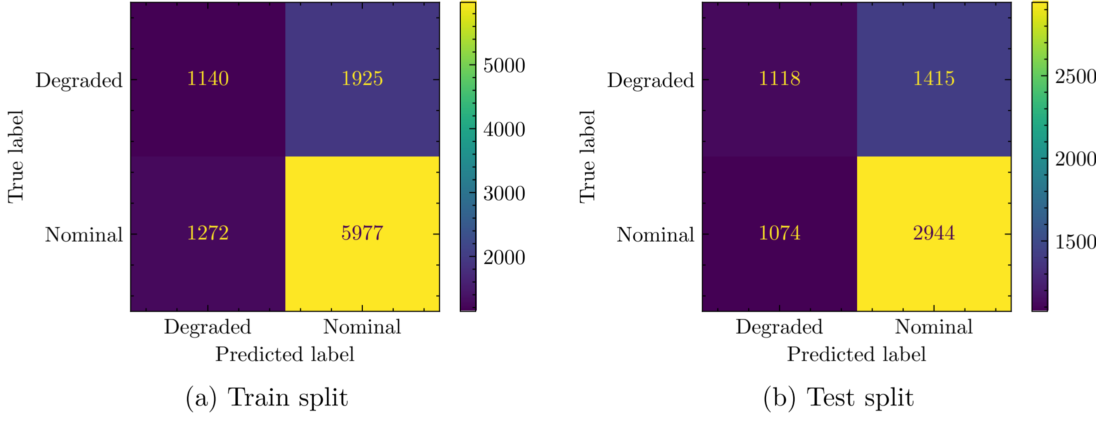
Figure B.1.36: Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and COBYLA applied to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

Figure B.1.37: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

Figure B.1.38: Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Sensor Features

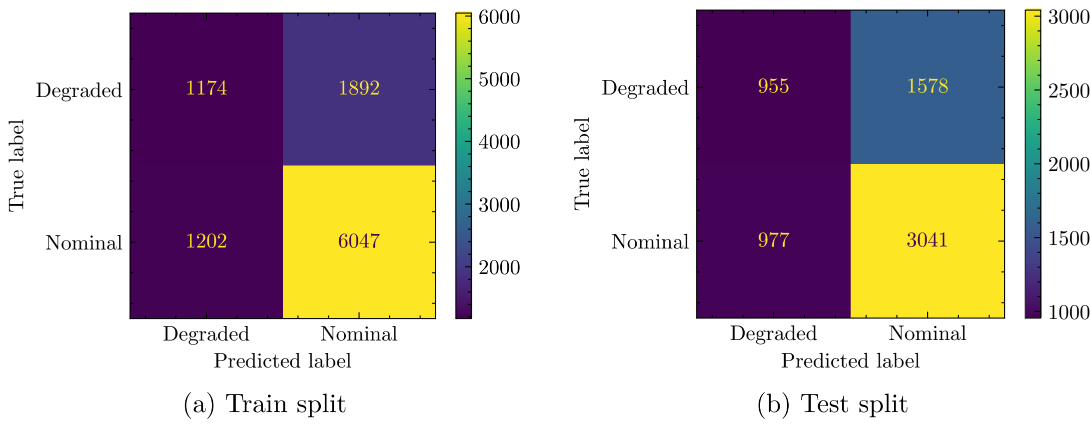


Figure B.1.39: Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Sensor Features

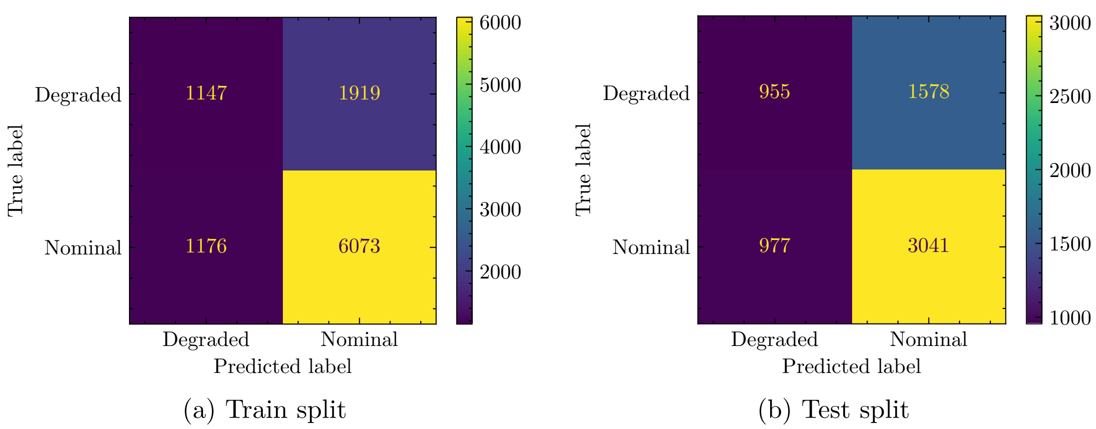


Figure B.1.40: Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Sensor Features

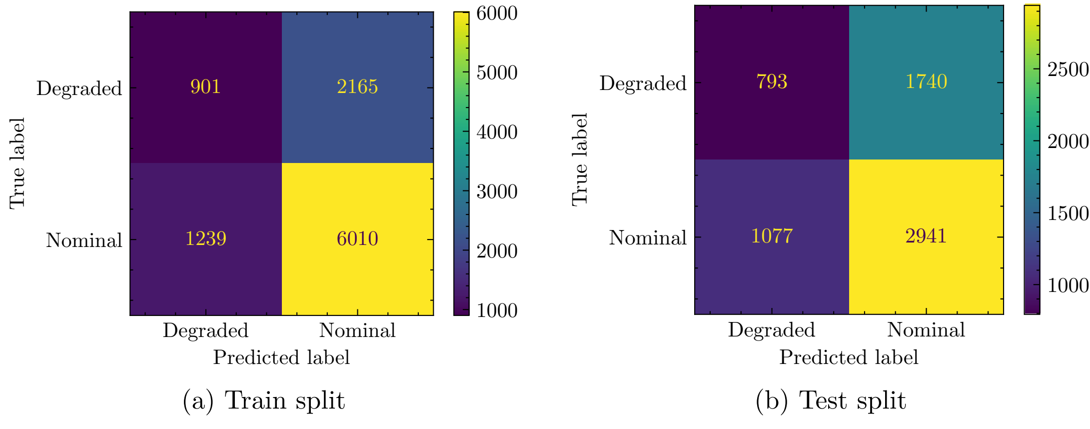


Figure B.1.41: Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and SLSQP applied to C-MAPSS FD001 with Sensor Features

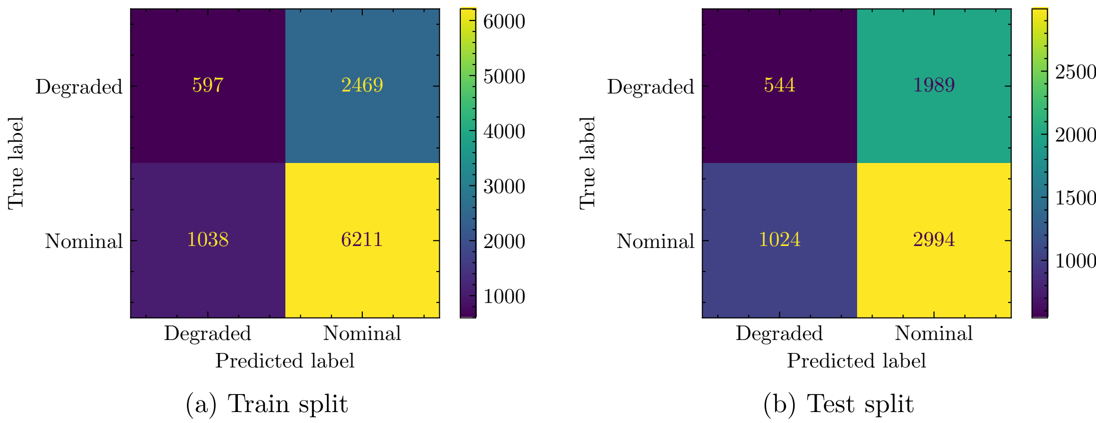


Figure B.1.42: Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and SLSQP applied to C-MAPSS FD001 with Sensor Features

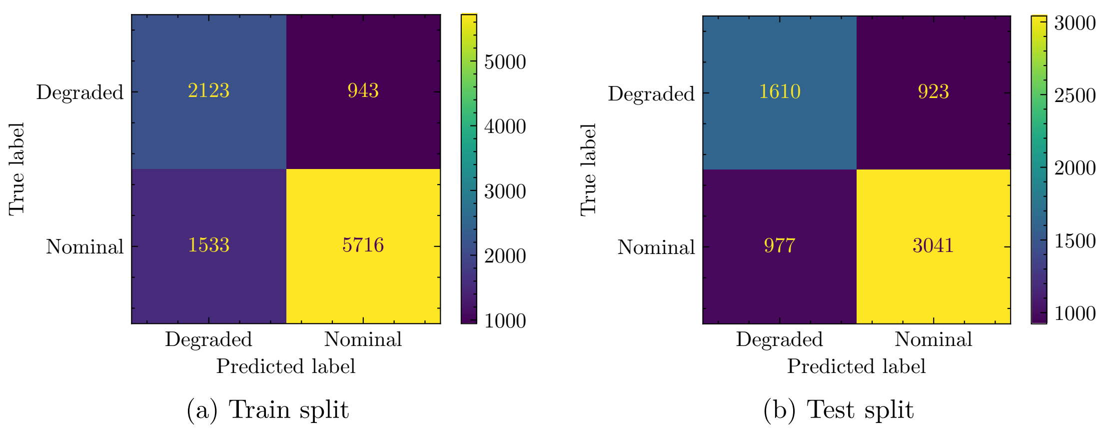


Figure B.1.43: Confussion Matrixes for VQC with ZZFeatureMap, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Sensor Features

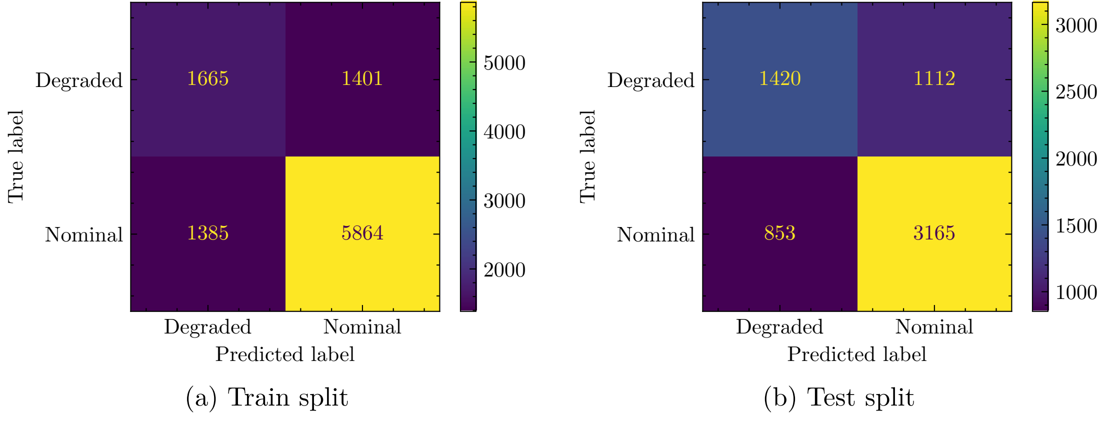


Figure B.1.44: Confussion Matrixes for VQC with ZZFeatureMap, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Sensor Features

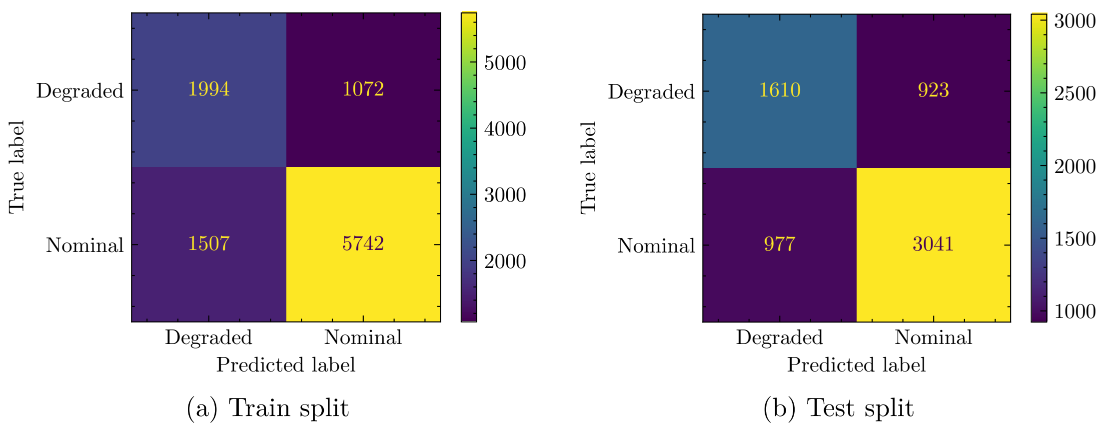


Figure B.1.45: Confussion Matrixes for VQC with ZFeatureMap, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Sensor Features

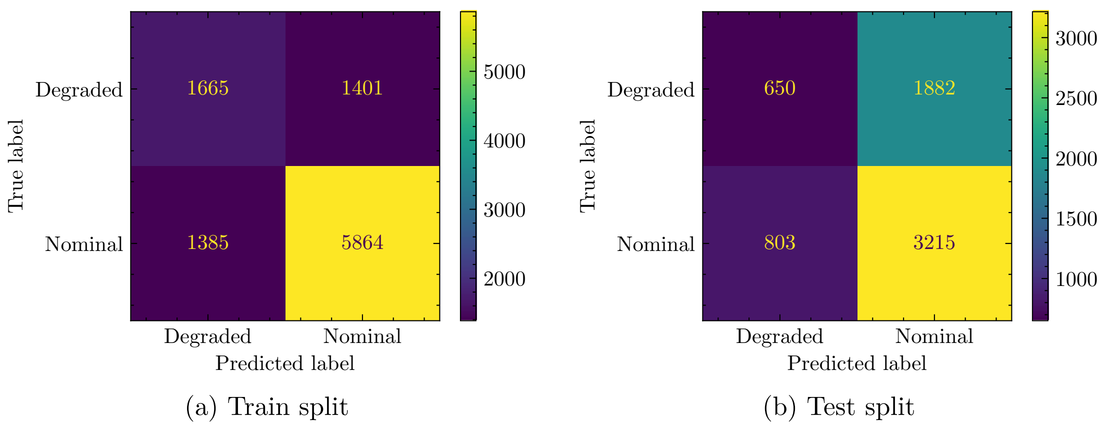


Figure B.1.46: Confussion Matrixes for VQC with ZFeatureMap, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Sensor Features

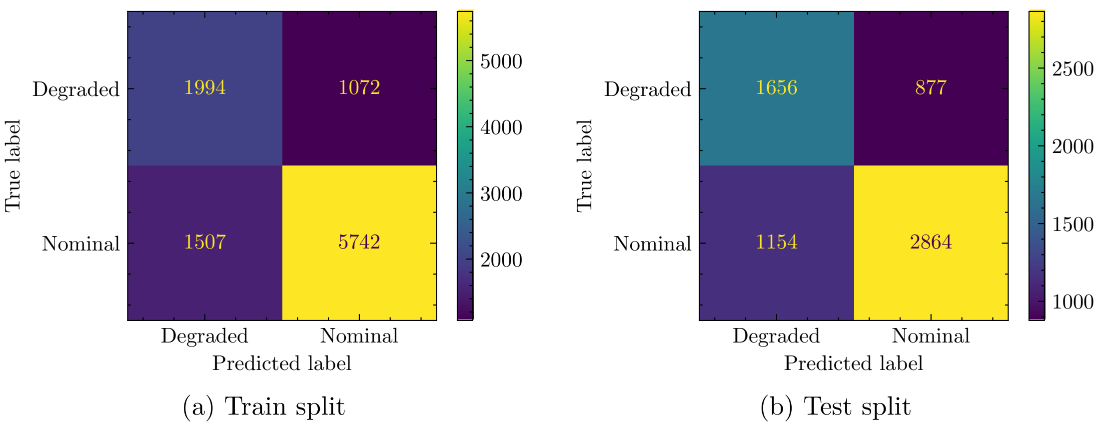
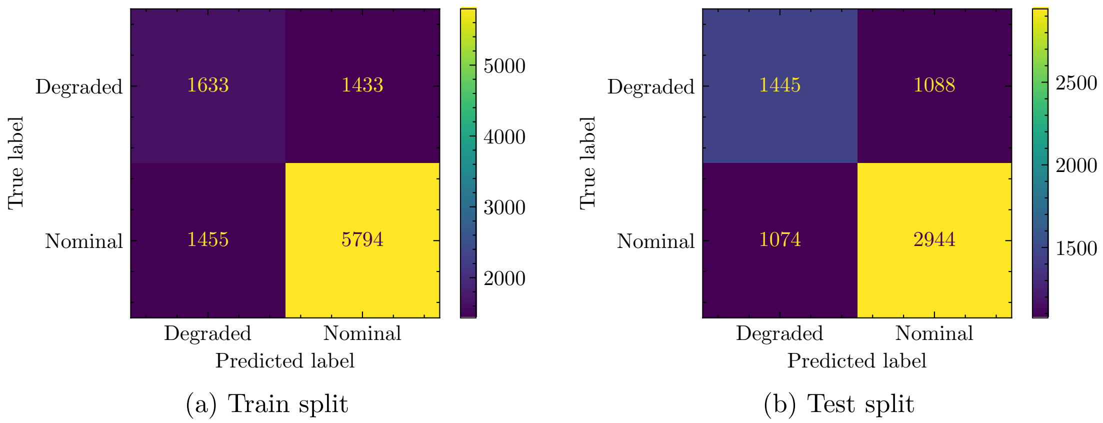


Figure B.1.47: Confussion Matrixes for VQC with RawFeatureVector, EfficientSU2 and SPSA applied to C-MAPSS FD001 with Sensor Features

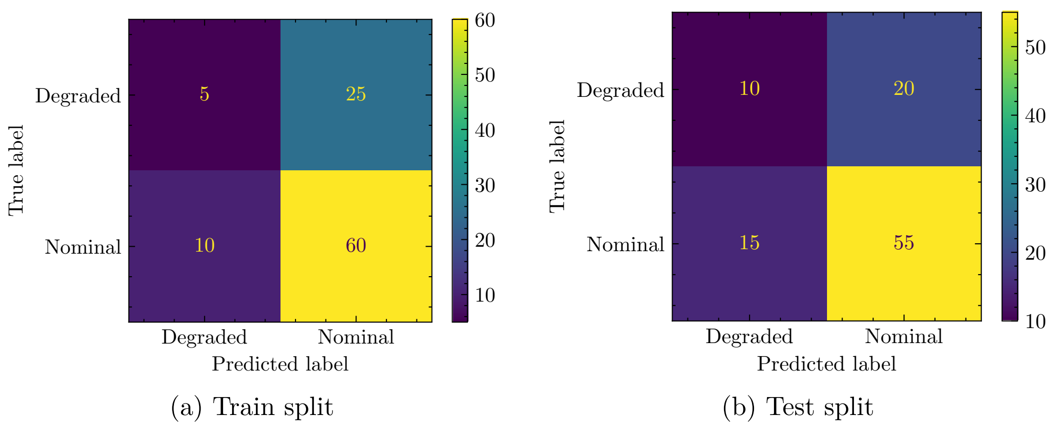


(a) Train split (b) Test split

Figure B.1.48: Confussion Matrixes for VQC with RawFeatureVector, RealAmplitudes and SPSA applied to C-MAPSS FD001 with Sensor Features

B.2. Quantum Neural Network Classifier

B.2.0.1. FD001 Statistics Features



(a) Train split (b) Test split

Figure B.2.1: Confussion Matrixes for QNN with ZZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Statistics Features

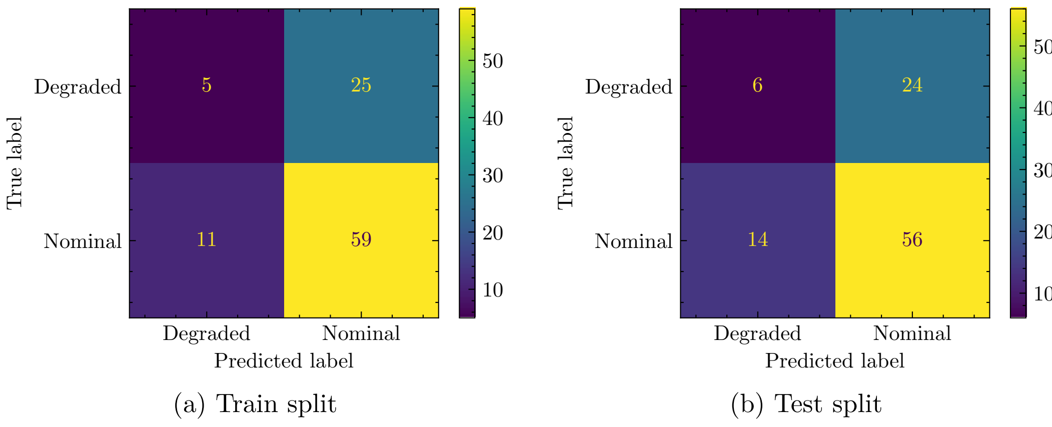


Figure B.2.2: Confussion Matrixes for QNN with ZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Statistics Features

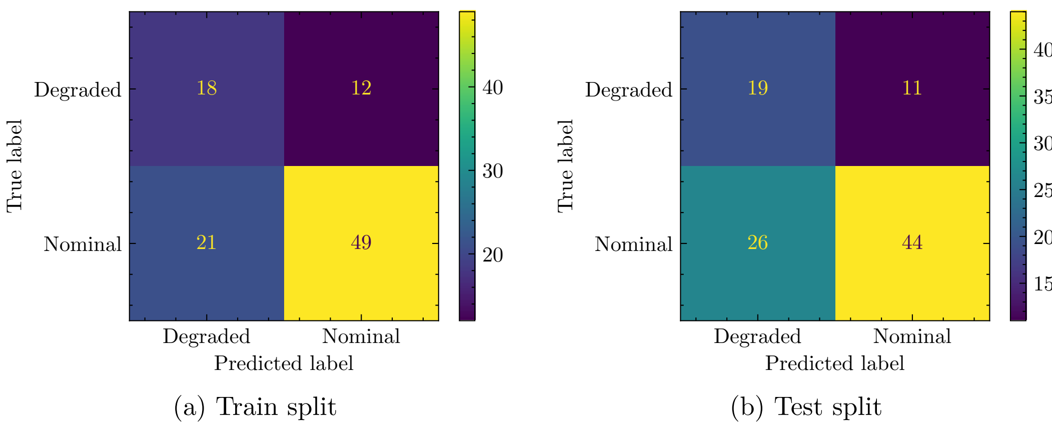


Figure B.2.3: Confussion Matrixes for QNN with ZZFeatureMap, EfficientSU2 to C-MAPSS FD001 with Statistics Features

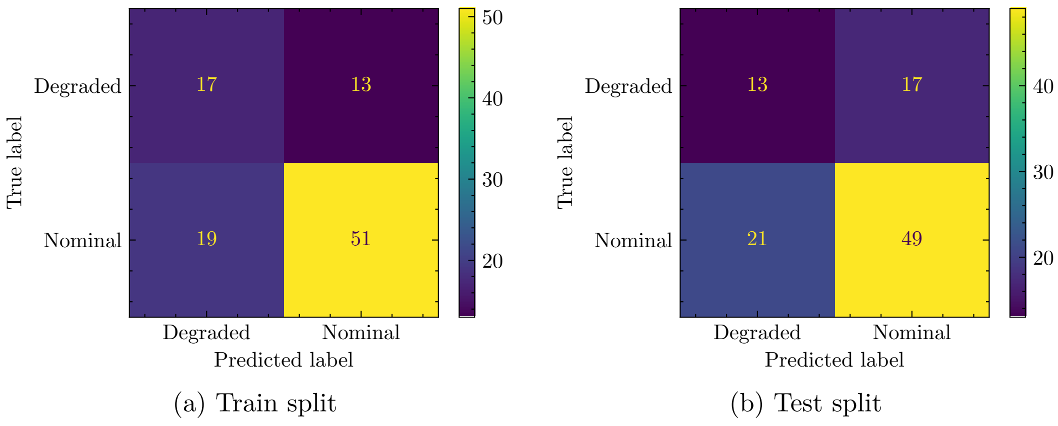
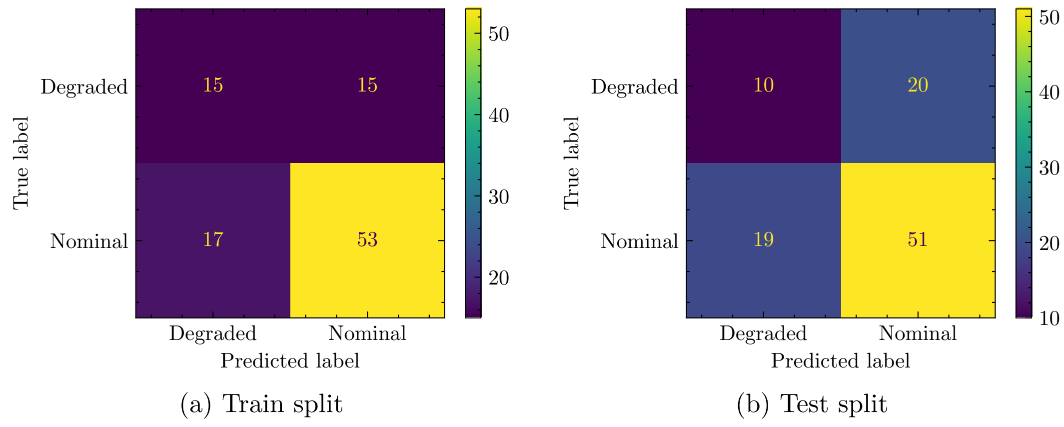
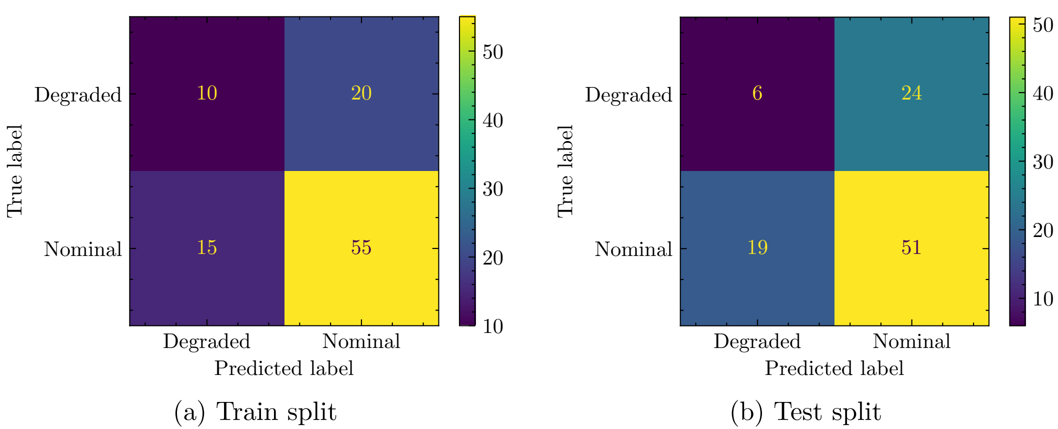


Figure B.2.4: Confussion Matrixes for QNN with ZFeatureMap, EfficientSU2 to C-MAPSS FD001 with Statistics Features



(a) Train split (b) Test split

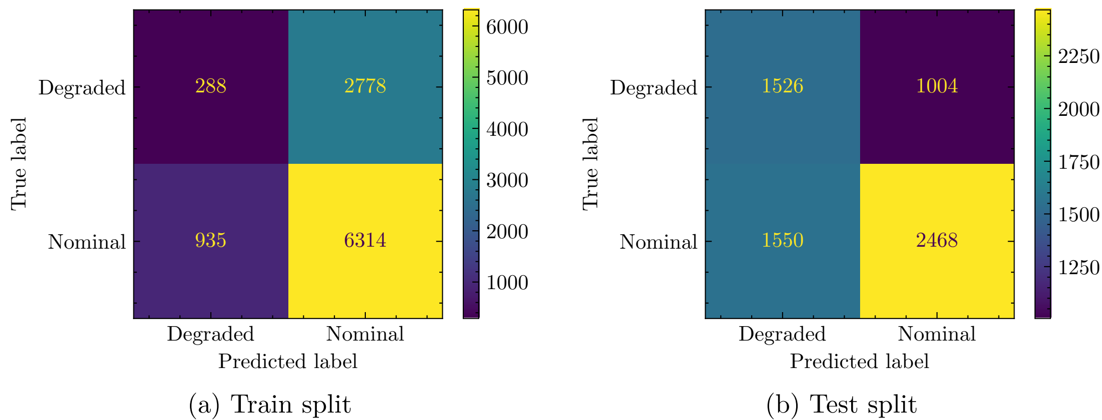
Figure B.2.5: Confussion Matrixes for QNN with RawFeatureVector, EfficientSU2 to C-MAPSS FD001 with Statistics Features



(a) Train split (b) Test split

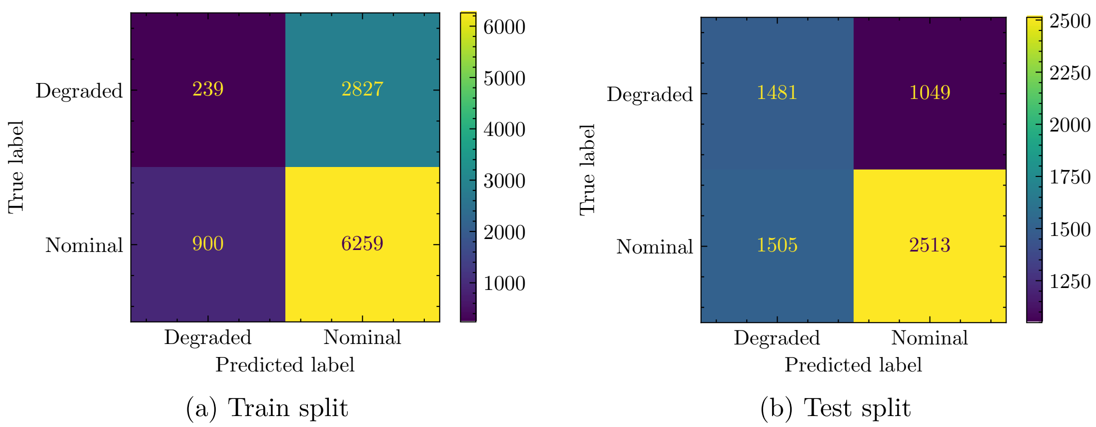
Figure B.2.6: Confussion Matrixes for QNN with RawFeatureVector, RealAmplitudes to C-MAPSS FD001 with Statistics Features

B.2.0.2. FD001 Sensors Features



(a) Train split (b) Test split

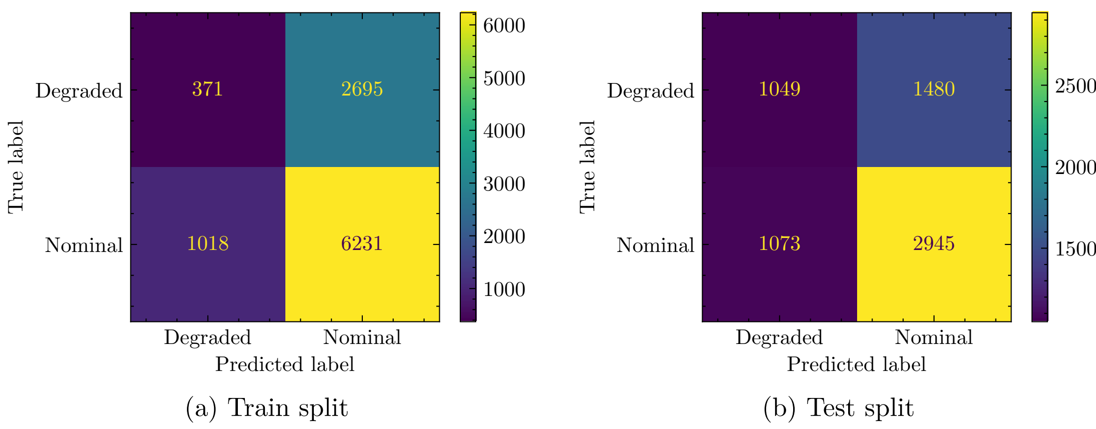
Figure B.2.7: Confussion Matrixes for QNN with ZZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

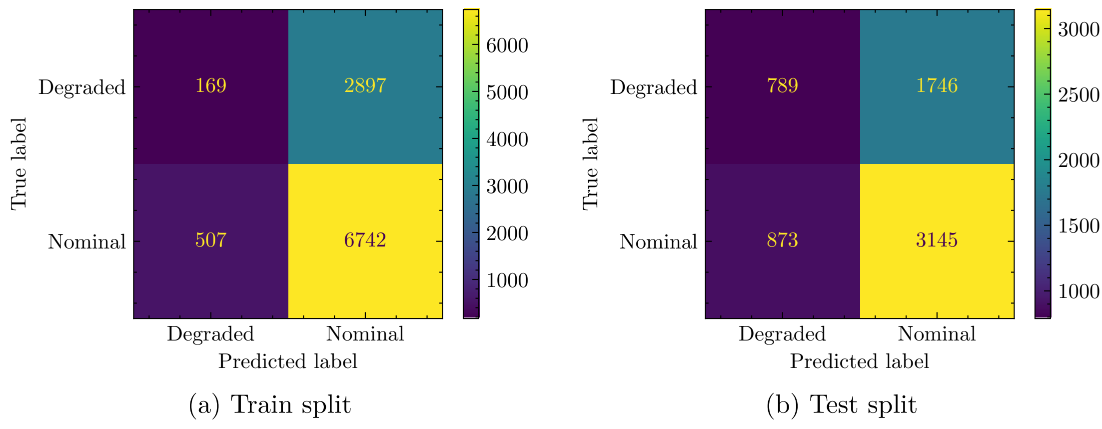
Figure B.2.8: Confussion Matrixes for QNN with ZFeatureMap, RealAmplitudes to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

Figure B.2.9: Confussion Matrixes for QNN with ZZFeatureMap, EfficientSU2 to C-MAPSS FD001 with Sensor Features



(a) Train split

(b) Test split

Figure B.2.10: Confussion Matrixes for QNN with ZFeatureMap, EfficientSU2 to C-MAPSS FD001 with Sensor Features

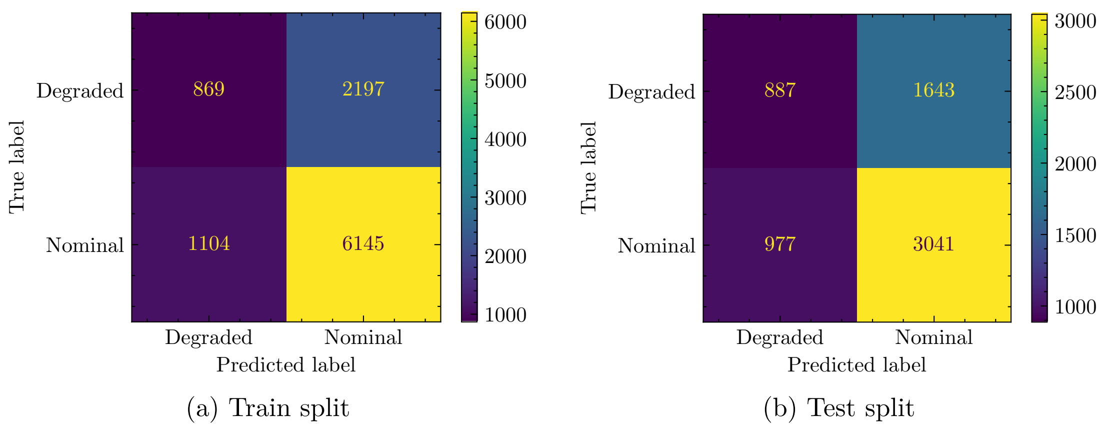


Figure B.2.11: Confussion Matrixes for QNN with RawFeatureVector, EfficientSU2 to C-MAPSS FD001 with Sensor Features

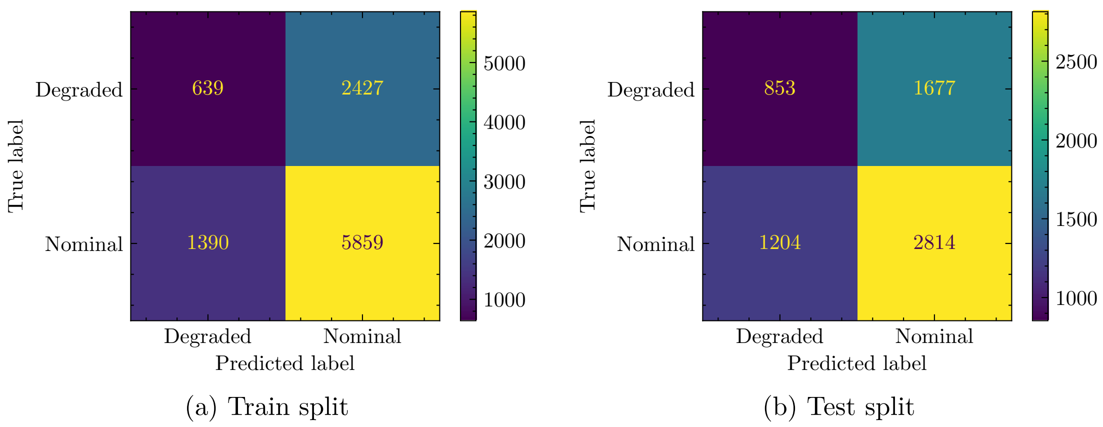


Figure B.2.12: Confussion Matrixes for QNN with RawFeatureVector, RealAmplitudes to C-MAPSS FD001 with Sensor Features

B.3. Pytorch + Quantum Neural Network Classifier

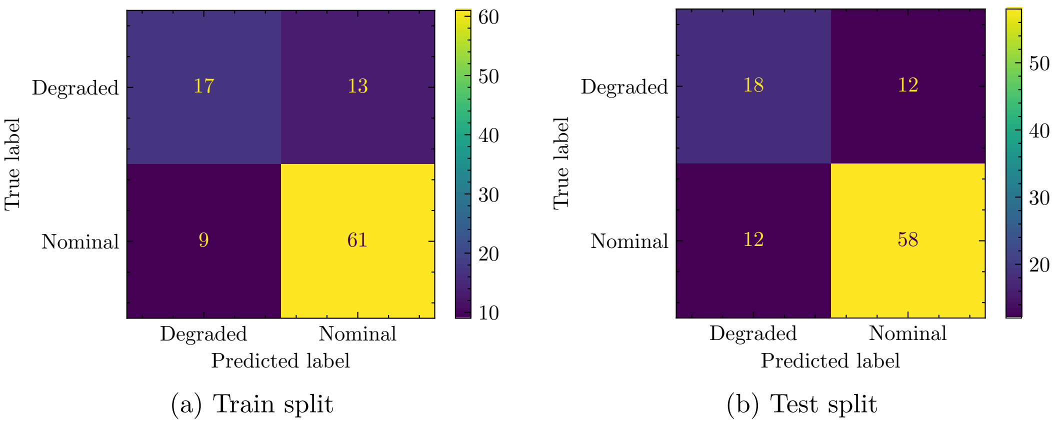


Figure B.3.1: Confussion Matrixes for Pytorch + OpflowQNN with ZZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features

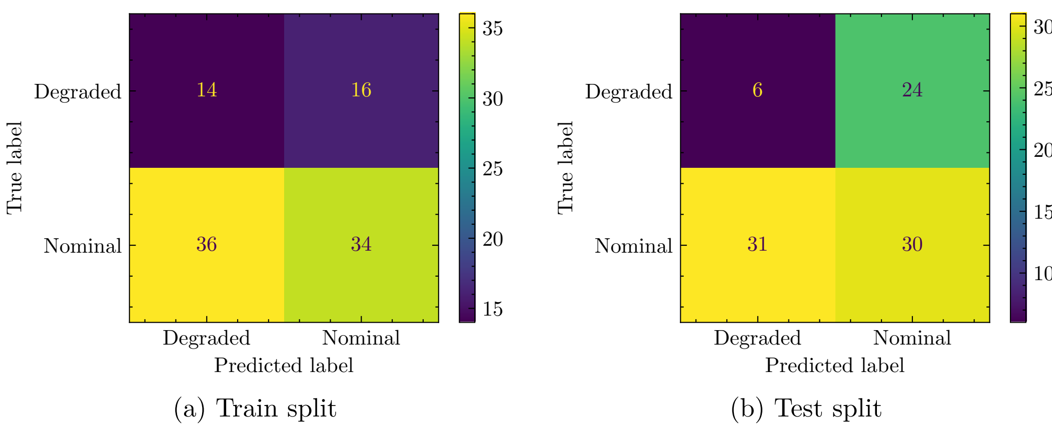
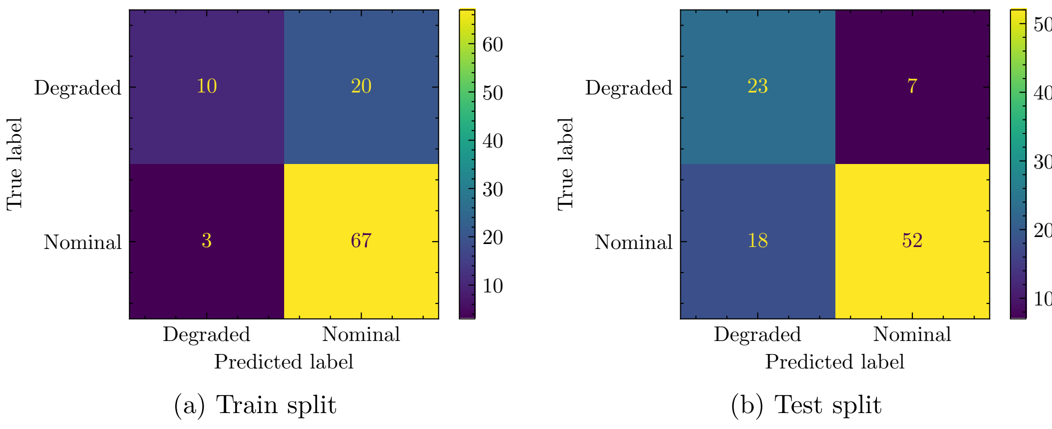
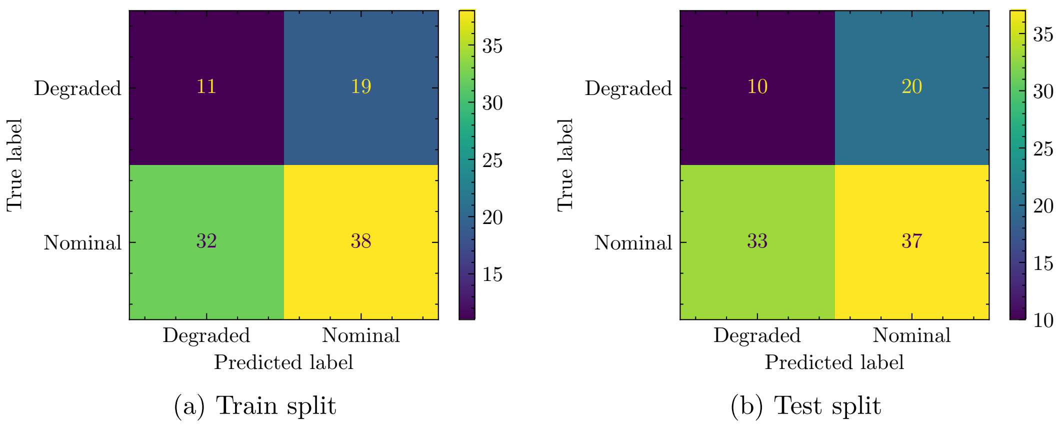


Figure B.3.2: Confussion Matrixes for Pytorch + CircuitQNN with ZZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features



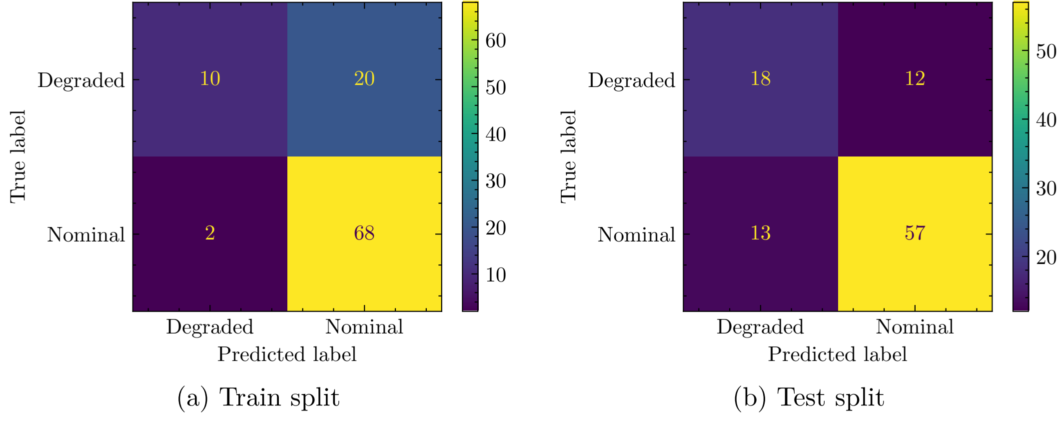
(a) Train split (b) Test split

Figure B.3.3: Confussion Matrixes for Pytorch + OpflowQNN with ZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features



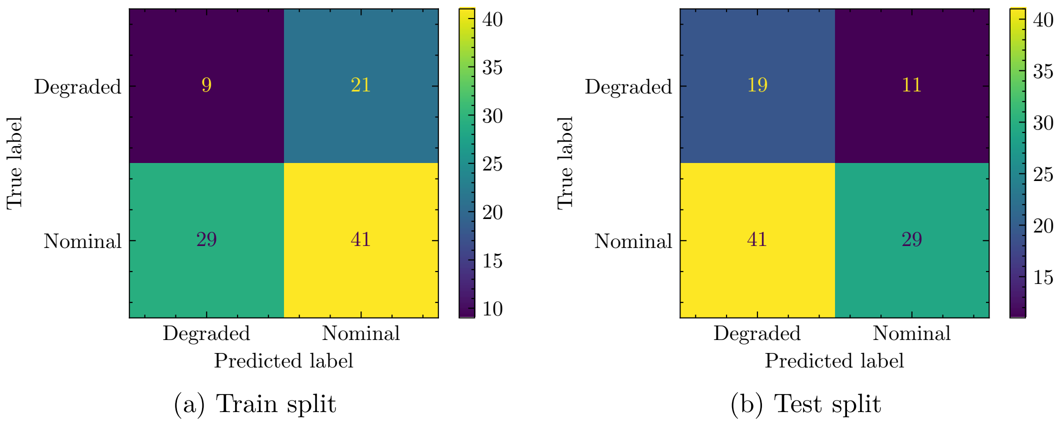
(a) Train split (b) Test split

Figure B.3.4: Confussion Matrixes for Pytorch + CircuitQNN with ZFeatureMap, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features



(a) Train split (b) Test split

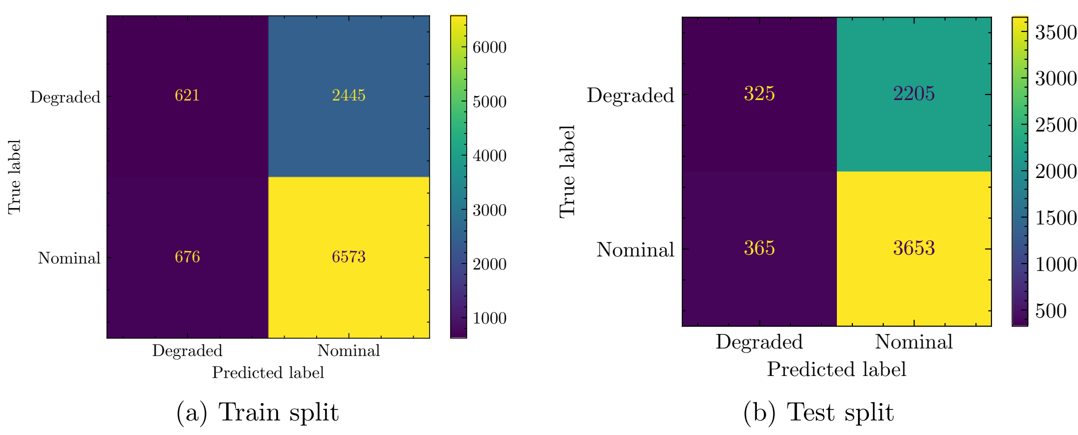
Figure B.3.5: Confussion Matrixes for Pytorch + OpflowQNN with RawFeatureVector, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features



(a) Train split (b) Test split

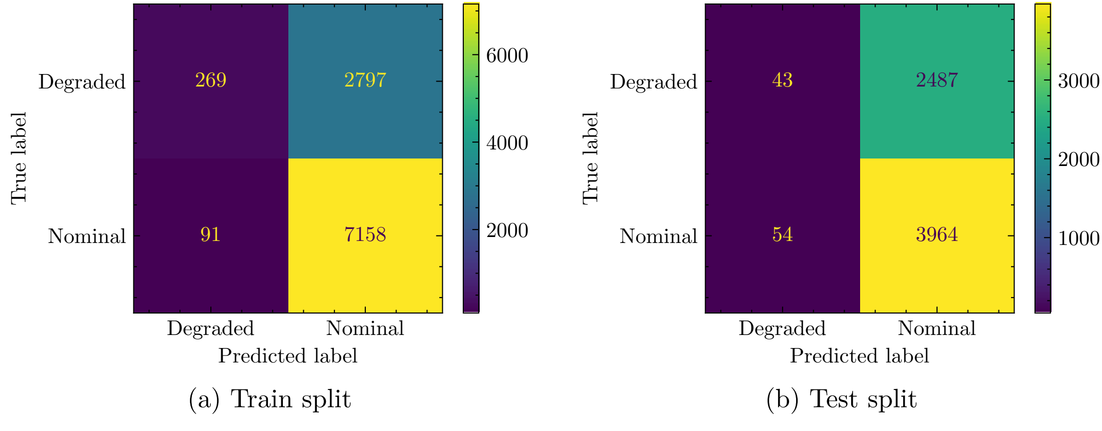
Figure B.3.6: Confussion Matrixes for Pytorch + CircuitQNN with RawFeatureVector, RealAmplitudes and LBFGS Optimizer applied to C-MAPSS FD001 with Statistical Features

B.4. Quantum Support Vector Classifier



(a) Train split (b) Test split

Figure B.4.1: Confussion Matrixes for Quantum SVC with ZFeatureMap applied to C-MAPSS FD001 with Sensors Features



(a) Train split (b) Test split

Figure B.4.2: Confussion Matrixes for Quantum SVC with ZZFeatureMap applied to C-MAPSS FD001 with Sensors Features

Annex C | Codes

This annex shows the Python code structure of the models used, only one case per model is shown. The rest of the codes, including the dataset processing, can be found at GitHub.

C.1. Classical SVC

Code C.1: SVC Code Example.

```
from sklearn.svm import SVC
import time
#...
# Execute Model
svc = SVC()
start = time.time()
_ = svc.fit(X_train, Y_train)
elapsed = time.time() - start
train_score_c4 = svc.score(X_train, Y_train)
test_score_c4 = svc.score(X_test, Y_test)

print(f"Training time: {round(elapsed,3)} seconds")
print(f"Classical SVC on the training dataset: {train_score_c4:.2f}")
print(f"Classical SVC on the test dataset:     {test_score_c4:.2f}")
#...
```

C.2. Variational Quantum Classifier

Code C.2: VQC Code Example.

```
import time
from qiskit_machine_learning.algorithms.classifiers import VQC
from qiskit.circuit.library import ZZFeatureMap, RealAmplitudes
from qiskit.algorithms.optimizers import COBYLA
from qiskit_aer import AerSimulator
from qiskit.utils import QuantumInstance
from matplotlib import pyplot as plt
from IPython.display import clear_output

#...
```

```
# Defining Model Structures
num_features = X_train.shape[1]
feature_map = ZZFeatureMap(feature_dimension=num_features, reps=1)
ansatz = RealAmplitudes(num_qubits=num_features, reps=3)
optimizer = COBYLA(maxiter=100)

objective_func_vals = []
plt.rcParams["figure.figsize"] = (12, 6)
def callback_graph(weights, obj_func_eval):
    clear_output(wait=True)
    objective_func_vals.append(obj_func_eval)
    plt.title("Objective function value against iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Objective function value")
    plt.plot(range(len(objective_func_vals)), objective_func_vals)
    plt.show()

# Initialize Simulator
quantum_instance = QuantumInstance(
    AerSimulator(),
    shots=1024,
    seed_simulator=algorithm_globals.random_seed,
    seed_transpiler=algorithm_globals.random_seed,
    )

# Initialize Model
vqc = VQC(
    feature_map=feature_map,
    ansatz=ansatz,
    optimizer=optimizer,
    quantum_instance=quantum_instance,
    callback=callback_graph,
    )

# Start Training
objective_func_vals = []

start = time.time()
vqc.fit(X_train, Y_train)
elapsed = time.time() - start

print(f"Training time: {round(elapsed)} seconds")

# Evaluate Results
train_score_q4 = vqc.score(X_train, Y_train)
test_score_q4 = vqc.score(X_test, Y_test)

print(f"Quantum VQC on the training dataset: {train_score_q4:.2f}")
print(f"Quantum VQC on the test dataset:     {test_score_q4:.2f}")

#...
```

C.3. Quantum Neural Network Classifier

Code C.3: QNNC Code Example.

```
import time
from qiskit_machine_learning.neural_networks import EstimatorQNN
from qiskit.circuit.library import ZZFeatureMap, RealAmplitudes
from qiskit_aer import AerSimulator
from qiskit.utils import QuantumInstance
from matplotlib import pyplot as plt
from IPython.display import clear_output

#...

# Defining Model Structures
num_features = X_train.shape[1]
qc = QuantumCircuit(num_features)
feature_map = ZZFeatureMap(feature_dimension=num_features)
ansatz = RealAmplitudes(num_qubits=num_features)

objective_func_vals = []
plt.rcParams["figure.figsize"] = (12, 6)
def callback_graph(weights, obj_func_eval):
    clear_output(wait=True)
    objective_func_vals.append(obj_func_eval)
    plt.title("Objective function value against iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Objective function value")
    plt.plot(range(len(objective_func_vals)), objective_func_vals)
    plt.show()

# Construct QNN

estimator_qnn = EstimatorQNN(
    circuit=qc, input_params=feature_map.parameters, weight_params=ansatz.
     ↪ parameters
)
estimator_qnn.forward(X_train[0, :], algorithm_globals.random.random(estimator_qnn
     ↪ .num_weights))

# Construct Neural Network classifier
estimator_classifier = NeuralNetworkClassifier(
    estimator_qnn, optimizer=COBYLA(maxiter=200), callback=callback_graph
)


# Start Training
objective_func_vals = []

start = time.time()
estimator_classifier.fit(X_train, Y_train)
```

```
46 elapsed = time.time() - start
47
48 print(f"Training time: {round(elapsed)} seconds")
49
50 # Evaluate Results
51 train_score = estimator_classifier.score(X_train, Y_train)
52 test_score = estimator_classifier.score(X_test, Y_test)
53
54 print(f"QNNC on the training dataset: {train_score:.2f}")
55 print(f"NNC on the test dataset:     {test_score:.2f}")
56
57 #...
```

C.4. Quantum Neural Network with Pytorch Classifier

Code C.4: QNN Pytorch Classifier Code Example.

```
1 import time
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from torch import Tensor
5 from torch.nn import Linear, CrossEntropyLoss, MSELoss
6 from torch.optim import LBFGS
7 from qiskit_machine_learning.neural_networks import CircuitQNN, TwoLayerQNN
8 from qiskit_machine_learning.connectors import TorchConnector
9 from matplotlib import pyplot as plt
10 from IPython.display import clear_output
11
12 #...
13
14 # Convert Data to torch Tensors
15 X_ = Tensor(X)
16 y01_ = Tensor(y01).reshape(len(y)).long()
17 y_ = Tensor(y).reshape(len(y), 1)
18 X_test_ = Tensor(X_test)
19 y01_test_ = Tensor(y01_test).reshape(len(y_test)).long()
20 y_test_ = Tensor(y_test).reshape(len(y_test), 1)
21
22 # Set up QNN
23 num_features = X_train.shape[1]
24 feature_map = ZZFeatureMap(feature_dimension=num_features)
25 ansatz = RealAmplitudes(num_qubits=num_features)
26 qnn = TwoLayerQNN(num_qubits=nnum_features, feature_map, ansatz,
    ↪ quantum_instance=qi)
27
28 # Set up PyTorch module
29 initial_weights = 0.1 * (2 * algorithm_globals.random.random(qnn1.num_weights) - 1)
30 model1 = TorchConnector(qnn, initial_weights=initial_weights)
31
```

```
32 # Define optimizer and loss
33 optimizer = LBFGS(model1.parameters())
34 f_loss = MSELoss(reduction="sum")
35 
36 # Start training
37 model1.train()
38 
39 def closure():
40     optimizer.zero_grad()
41     loss = f_loss(model1(X_), y_)
42     loss.backward()
43     print(loss.item())
44     return loss
45 
46 # Run optimizer
47 optimizer.step(closure)
48 
49 # Evaluate model and compute accuracy
50 y_predict = []
51 for x, y_target in zip(X, y):
52     output = model1(Tensor(x))
53     y_predict += [np.sign(output.detach().numpy())[0]]
54 
55 print("Accuracy:", sum(y_predict == y) / len(y))
56 
57 #...
```

C.5. Quantum Support Vector Classifier

Code C.5: QSVC Code Example.

```
1 import numpy as np
2 from sklearn.model_selection import train_test_split
3 from sklearn.preprocessing import MinMaxScaler
4 from qiskit import BasicAer
5 from qiskit.circuit.library import ZFeatureMap
6 from qiskit.utils import algorithm_globals
7 from qiskit_machine_learning.kernels import FidelityQuantumKernel
8 from qiskit_machine_learning.algorithms import PegasosQSVC
9 import time
10 
11 #...
12 
13 train_features = MinMaxScaler(feature_range=(0, np.pi)).fit_transform(train_features
     ↪ )
14 test_features = MinMaxScaler(feature_range=(0, np.pi)).fit_transform(test_features)
15 
16 # Define parameters
17 num_qubits = 4
18 tau = 100
19 C = 1000
```

```python

# Set the Kernel from a FeatureMap
algorithm_globals.random_seed = 12345
feature_map = ZFeatureMap(feature_dimension=num_qubits, reps=1)
qkernel = FidelityQuantumKernel(feature_map=feature_map)

# Initialize Model
pegasos_qsvc = PegasosQSVC(quantum_kernel=qkernel, C=C, num_steps=tau)

# Training Model
start = time.time()
pegasos_qsvc.fit(train_features, train_labels)
elapsed = time.time() - start
print(f"Training time: {round(elapsed)} seconds")

# Evaluation
start = time.time()
pegasos_score = pegasos_qsvc.score(test_features, test_labels)
elapsed = time.time() - start

print(f"Testing time: {round(elapsed)} seconds")
print(f"PegasosQSVC classification test score: {pegasos_score}")

#...
```