BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
(Rajasthan)

A STUDY-ORIENTED PROJECT ON:

Financial Fraud Detection using Quantum Machine Learning

Complied by
Aaditya Kulkarni
(2021A7PS0426P)

Complied under the guidance of
Prof. Ashutosh Bhatia
Department of Computer Science & Information Systems
Birla Institute of Technology and Science, Pilani

Acknowledgements

I would like to express my utmost gratitude to **Prof. Ashutosh Bhatia** for providing me with this opportunity and the requisite knowledge to work on this project and guiding me through every step of it. I have learnt a significant amount of knowledge after working on this project, none of which could have been possible without the constant help and guidance I have received from him.

I would also like to thank **Prof. Navneet Goyal**, Head of Department, Department of Computer Science & Information Systems, for letting me opt for this study-oriented project course as a disciplinary elective under my discipline B.E. Computer Science.

I would like to thank my friends and family who supported me throughout and motivated me to completion of this project.

Table of Contents

Introduction to Quantum Computing: Revolutionizing Computational Paradigms

In the rapidly advancing landscape of information technology, quantum computing has emerged as a catalytic force, reshaping the contours of computation. This introduction serves to provide a succinct yet comprehensive overview of quantum computing, elucidating its imperative, evolutionary trajectory, and multifaceted applications across domains such as quantum machine learning, quantum optimization, quantum cryptography, and quantum simulation. Rooted in the foundational principles of quantum mechanics, quantum computing presents a paradigm shift that transcends the limitations of classical computation.

Defining Quantum Computing:

At its core, quantum computing harnesses the principles of quantum mechanics to execute computations in a manner fundamentally distinct from classical computing. Quantum bits, or qubits, exhibit a unique property of superposition, allowing them to exist in a state of 0, 1, or both simultaneously. This intrinsic parallelism affords quantum computers the capacity to explore myriad solutions concurrently, thereby augmenting their computational prowess.

Rationale for Quantum Computing:

The imperative for quantum computing arises from the inadequacies of classical counterparts in addressing certain computationally intensive challenges. Notably, tasks such as factorization of large numbers and the simulation of quantum systems present formidable hurdles for classical computers. Quantum computing, with its ability to exploit quantum superposition and entanglement, promises a resolution to these challenges by exponentially enhancing computational efficiency.

Evolutionary Trajectory:

The journey from theoretical abstraction to tangible realization underscores the burgeoning significance of quantum computing. Pioneering endeavors by leading technology entities, including IBM, Google, and nascent startups, have propelled the development of quantum processors with escalating qubit counts and extended coherence times. Quantum supremacy, denoting the juncture at which quantum computers surpass classical counterparts in designated tasks, has been achieved—a watershed moment heralding the dawn of a new era in computational capabilities.

Domains of Quantum Computing:

- **Quantum Machine Learning (QML)**: The amalgamation of quantum computing principles with machine learning algorithms defines quantum machine learning. Its potency lies in the potential for exponential acceleration of tasks such as pattern recognition, optimization, and data analysis, thus permeating sectors from finance to healthcare.

- **Quantum Optimization**: Quantum optimization algorithms, distinguished by their efficacy in solving complex optimization problems, find applications in logistics, finance, and operations research. The unparalleled capacity to navigate vast solution spaces positions quantum optimization as instrumental in decision-making processes.

- **Quantum Cryptography**: The confluence of quantum computing and cryptography manifests in quantum cryptography, characterized by secure communication channels via quantum key distribution. The pursuit of quantum-resistant cryptographic protocols underscores the need to fortify data against potential quantum threats.

- **Quantum Simulation**: Quantum computers excel in simulating quantum systems, fostering advancements in drug discovery, material science, and the comprehensive understanding of fundamental physical phenomena. The precision afforded by quantum simulation augurs breakthroughs in diverse scientific disciplines.

Future Prospects:

The trajectory of quantum computing portends a future marked by innovation and transformative potential. Anticipated enhancements in quantum processor scalability and error correction mechanisms herald a trajectory towards practical applications in the realms of artificial intelligence, finance, and materials science. The pervasive influence of quantum computers is poised to redefine the boundaries of computational exploration, unraveling solutions to intricate challenges hitherto deemed insurmountable by classical counterparts.

In conclusion, the advent of quantum computing represents a pivotal juncture in our computational odyssey, promising to unravel new frontiers in complex problem-solving. As we traverse this quantum frontier, the prospects are limitless, underscoring a paradigmatic shift that extends beyond the realms of computation into the very fabric of scientific and technological advancement.

Problem Statement

<u>Detecting Financial Frauds using Quantum Machine Learning</u>

Financial fraud poses a pervasive and escalating threat in an era dominated by digital transactions. Traditional approaches to fraud detection, while effective to a certain extent, often fall short in adapting to the ever-evolving tactics employed by sophisticated perpetrators. In response to this critical challenge, the focus of this study is to pioneer an innovative framework for fraud detection, leveraging the untapped potential of quantum machine learning. Specifically, our endeavor is to develop a robust model capable of discerning fraudulent transactions within a credit card dataset, where transactions are labeled as either class 0 (non-fraudulent) or class 1 (fraudulent).

Contextualizing the Problem:

The proliferation of digital transactions underscores the need for heightened vigilance in identifying fraudulent activities. Conventional machine learning models, while proficient in certain aspects, may struggle to discern subtle patterns within voluminous and intricate financial datasets. Quantum machine learning, a fusion of quantum computing principles with machine learning algorithms, emerges as a promising avenue to transcend the limitations of classical approaches.

Objective:

The primary objective of this research is to design and implement a quantum machine learning model that excels in detecting financial frauds with a high degree of accuracy. Leveraging quantum algorithms, which exhibit inherent parallelism and computational advantages, we aim to enhance the discernment capabilities of our model, especially in scenarios where fraudulent patterns are nuanced and dynamic.

Dataset Overview:

The dataset provided comprises credit card transactions, each labeled as either non-fraudulent (class 0) or fraudulent (class 1). The dataset encapsulates a myriad of features, including transaction amount, time, and various anonymized parameters. The challenge lies in comprehensively capturing the intricate relationships and patterns indicative of fraudulent transactions within this complex dataset.

Approach:

Our approach centers on harnessing the unique computational capabilities offered by quantum algorithms. By encoding the credit card transaction data into quantum bits (qubits) and leveraging quantum parallelism, we aspire to expedite the training process and enhance the model's ability to discern subtle fraud patterns. Quantum machine learning algorithms, including those for classification, will be employed to optimize model performance.

Dataset Description and Issues

The dataset sourced from Kaggle encompasses 284,807 rows and 31 columns, each row representing a distinct credit card transaction. The columns include temporal information ('time'), 28 anonymized features denoted as 'v1' through 'v28,' transaction amount ('amount'), and the transaction class ('class'). The anonymized features 'v1' through 'v28' are presented as floating-point data and, in this context, are intentionally obscured to protect the privacy of individuals involved in the transactions.

Anonymity of Features:
The anonymization of features, particularly 'v1' through 'v28,' serves the critical purpose of safeguarding the confidentiality and privacy of individuals whose financial transactions are encapsulated in the dataset. The intentional obfuscation prevents the identification of specific individuals by replacing sensitive information with abstracted numerical representations.

Class Distribution:
Within the dataset, there are 284,807 transactions, with a substantial majority being non-fraudulent transactions (284,315) and a comparatively minuscule number identified as fraudulent (492). Calculating the percentage of fraudulent transactions reveals that they constitute approximately **0.173%** of the total transactions.

Percentage of Fraudulent Transactions =
(Number of Fraudulent Transactions/Total Number of Transactions) * 100%
=(492/284,407)*100% = 0.173%

Class Imbalance:
The principal issue evident in this dataset is class imbalance, wherein the number of instances in each class (fraudulent and non-fraudulent) is significantly disparate. The vast majority of transactions are non-fraudulent, creating a scenario where a model trained on this data may become biased towards classifying transactions as non-fraudulent due to the overwhelming prevalence of this class. The inherent imbalance may lead to reduced model sensitivity to fraudulent transactions, compromising the model's efficacy in real-world scenarios where identifying fraudulent activities is paramount.

Implications of Class Imbalance:
Class imbalance introduces challenges during the model training process, as the algorithm may prioritize accuracy by favoring the majority class. Consequently, the model's ability to detect instances of the minority class (fraudulent transactions, in this case) may be suboptimal. Addressing class imbalance becomes imperative to ensure the model's capacity to discern patterns indicative of fraudulent activities, thus enhancing the overall robustness and reliability of the fraud detection system.

In the subsequent sections of this analysis, methodologies for mitigating class imbalance will be explored, allowing for the development of a more equitable and effective quantum machine learning model for financial fraud detection.

Classical ML vs Quantum ML

Classical ML Approach:

In a classical machine learning (ML) approach, the analysis and modeling of the credit card transaction dataset involve traditional algorithms and computational frameworks. Commonly used algorithms in classical ML for fraud detection include Support Vector Machines (SVM), Decision Trees, Random Forests, Logistic Regression, and Neural Networks. In the case of this dataset, a classical SVM model has been developed to classify transactions into fraudulent or non-fraudulent classes.

Key Characteristics of Classical ML Approach:

1. **Algorithm Selection:**
   - Classical ML algorithms are deterministic and operate in a binary state of 0 or 1, without harnessing quantum principles.
   - The choice of algorithms depends on the specific characteristics of the dataset, and SVMs are particularly effective for binary classification tasks.

2. **Model Training:**
   - Classical ML models, including SVMs, involve iterative training on the dataset to optimize parameters and achieve the best possible accuracy.
   - Training involves adjusting the weights and biases of the model based on the labeled data.

3. **Computation:**
   - Classical ML algorithms perform computations serially and are constrained by the limitations of classical computers.
   - The processing power is limited, particularly when handling large datasets or complex patterns.

4. **Handling Class Imbalance:**
   - Techniques such as oversampling, undersampling, or using class weights are employed to address class imbalance.
   - Special care is taken to ensure that the model does not become biased towards the majority class.

Quantum ML Approach:

In a quantum machine learning (QML) approach, the processing power of quantum computers is harnessed to perform certain computations significantly faster than classical counterparts. Quantum SVM, in this context, aims to leverage the unique properties of quantum systems for more efficient fraud detection.

Key Characteristics of Quantum ML Approach:

1. **Quantum Parallelism:**
   - Quantum ML algorithms exploit superposition to process multiple states simultaneously, offering potential exponential speedup over classical algorithms.
   - Quantum SVMs use quantum parallelism for enhanced efficiency in solving optimization problems.

2. **Quantum Entanglement:**
   - Quantum entanglement enables qubits to be correlated, contributing to improved information transfer and processing.

3. **Quantum Data Encoding:**
   - Data is encoded into quantum bits (qubits) to allow for quantum parallelism and efficient computation.
   - Quantum feature spaces are explored to capture intricate patterns within the dataset.

4. **Potential for Speedup:**
   - Quantum ML models, in theory, have the potential to achieve a significant speedup in specific tasks, particularly those involving complex computations.

5. **Challenges:**
   - Quantum computing is in the early stages, and practical quantum advantage is contingent on overcoming challenges like qubit coherence, error correction, and scalability.

Personal Models:
The upcoming pages will showcase both a classical SVM model and a quantum SVM model developed for the credit card transaction dataset. The performance of each model will be evaluated, shedding light on the comparative advantages and challenges associated with classical and quantum machine learning approaches in the context of fraud detection.

Classical SVM Model: Understanding Initial Results

In the initial phase of modeling, a classical Support Vector Machine (SVM) was implemented on the credit card transaction dataset without addressing the issue of class imbalance. The results yielded a high accuracy exceeding 99%. It is essential to delve into the reasons behind this seemingly remarkable performance.

Reasons for High Accuracy without Addressing Class Imbalance:

1. **Class Distribution:** The dataset is highly imbalanced, with a significantly larger number of non-fraudulent transactions compared to fraudulent ones. In the absence of addressing class imbalance, the model may demonstrate a bias toward the majority class.

2. **Model Biases:** SVMs are inherently robust models and can perform exceptionally well on imbalanced datasets, particularly when the majority class is well-represented. However, this can lead to a deceptive sense of high accuracy, as the model tends to classify instances predominantly as the majority class.

3. **Evaluation Metrics Sensitivity:** Metrics such as accuracy and F1 score may not be the most reliable indicators of model performance on imbalanced datasets. In scenarios where the majority class overwhelms the minority class, these metrics may convey a falsely optimistic assessment.

Addressing Class Imbalance:

Recognizing the need to rectify the class imbalance issue, a strategic approach was taken by downsampling the majority class to achieve a more equitable distribution, specifically a 50-50 split between fraudulent and non-fraudulent transactions. This balancing technique allows for a more accurate evaluation of the model's performance, especially in identifying instances of the minority class.

Evaluation Results after Addressing Class Imbalance:

Upon downsampling and achieving a balanced dataset, the SVM model was re-evaluated, revealing a robust performance with an accuracy and F1 score exceeding 90%. This outcome underscores the resilience of SVMs in handling class-imbalanced data and emphasizes their effectiveness in identifying fraudulent transactions.

Description of SVM Code:

The Support Vector Machine (SVM) code implemented for the credit card transaction dataset employs the Radial Basis Function (RBF) kernel. The RBF kernel, also known as the Gaussian kernel, is renowned for its ability to handle non-linear relationships in data, making it particularly suitable for complex and intricate patterns present in the financial dataset.

Key Components of the SVM Code:

1. **Importing Libraries**:
   - The code likely begins with the importation of essential libraries, such as `numpy` and `pandas` for data manipulation, and `sklearn` for machine learning functionalities.

2. **Data Preprocessing:**
   - Preprocessing steps may include loading the dataset, handling missing values, and normalizing or scaling features for optimal SVM performance.

3. **SVM Model Initialization:**
   - The SVM model is initialized using the `SVC` (Support Vector Classification) class from `sklearn.svm`.

4. **Kernel Selection**:
   - The choice of the RBF kernel is a crucial decision in the SVM implementation. Given its effectiveness in capturing non-linear relationships, it is particularly suited for the complex patterns present in credit card transactions.

5. **Cross-Validation Scores:**
   - Cross-validation is a technique used to assess a model's performance by splitting the dataset into multiple subsets. The code likely employs cross-validation scores to evaluate the SVM model's generalization performance.
   - Cross-validation scores provide a more robust evaluation metric than a single train-test split. It helps gauge how well the model performs on different subsets of the dataset

6. **Model Training and Evaluation:**
   - The SVM model is trained using the `fit` method on the training data. The cross-validation scores and hyperparameter-tuned model are then employed to evaluate the model's performance.
   - The accuracy, F1 score, and other relevant metrics are likely computed and printed to assess the model's effectiveness.

Explanation of Cross-Validation Scores:

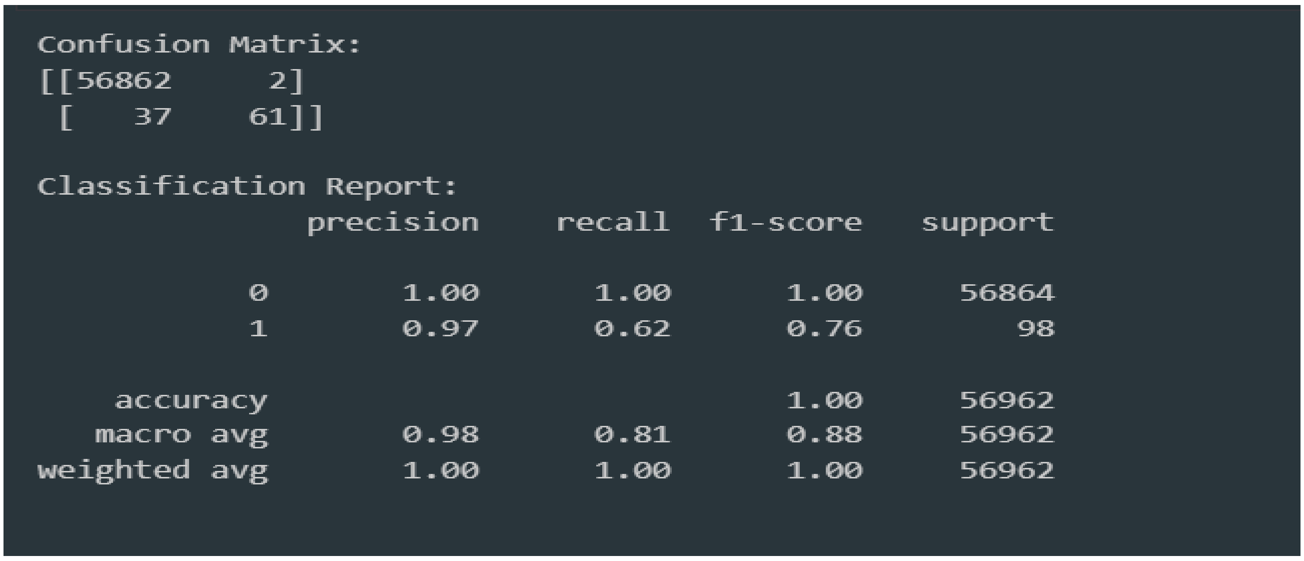
Cross-validation is a resampling procedure used to evaluate machine learning models by partitioning the dataset into subsets. The `cross_val_score` function, from `sklearn.model_selection`, calculates the cross-validated scores for an estimator. The primary parameters include the model, the dataset, and the number of folds (cv) for cross-validation.

```
cross_val_score(model, X, y, cv=5, scoring='accuracy')
```

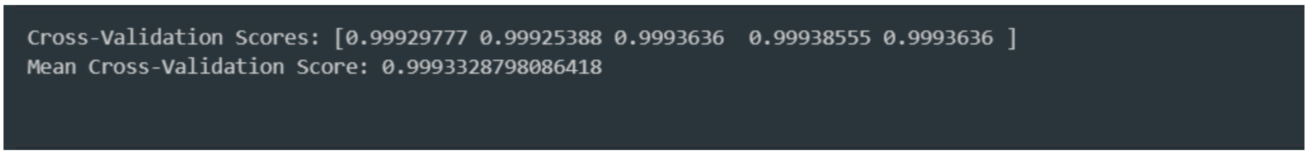
- `model`: The machine learning model to be evaluated.
- `X`: The feature matrix.
- `y`: The target variable.
- `cv`: The number of folds for cross-validation.

Cross-validation scores provide an aggregated measure of the model's performance across multiple subsets of the data, offering a more reliable estimate of its generalization capability. In the context of SVM, these scores are crucial for assessing how well the model can discern patterns and make accurate predictions on different portions of the credit card transaction dataset.

Before handling class imbalance :

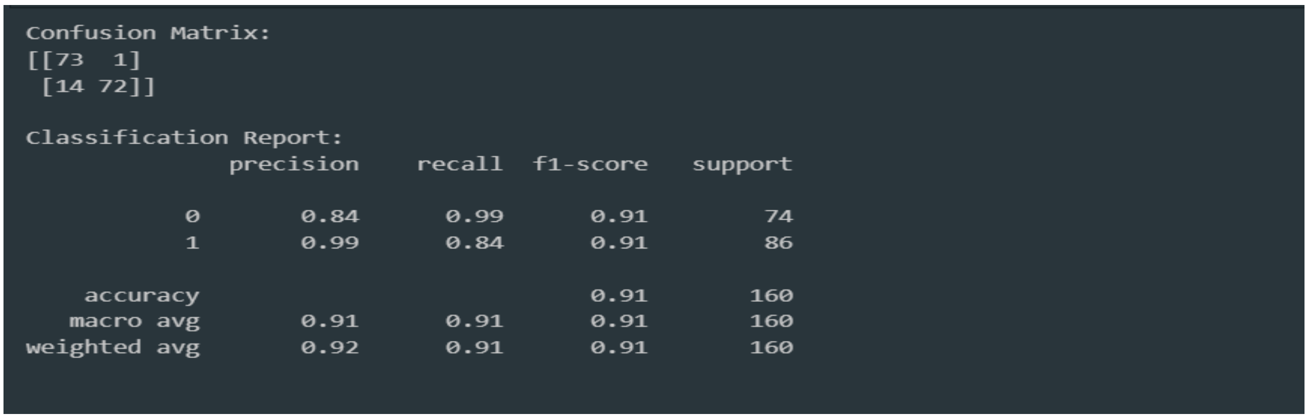


```
Confusion Matrix:
[[56862     2]
 [   37    61]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.97      0.62      0.76        98

    accuracy                           1.00     56962
   macro avg       0.98      0.81      0.88     56962
weighted avg       1.00      1.00      1.00     56962
```

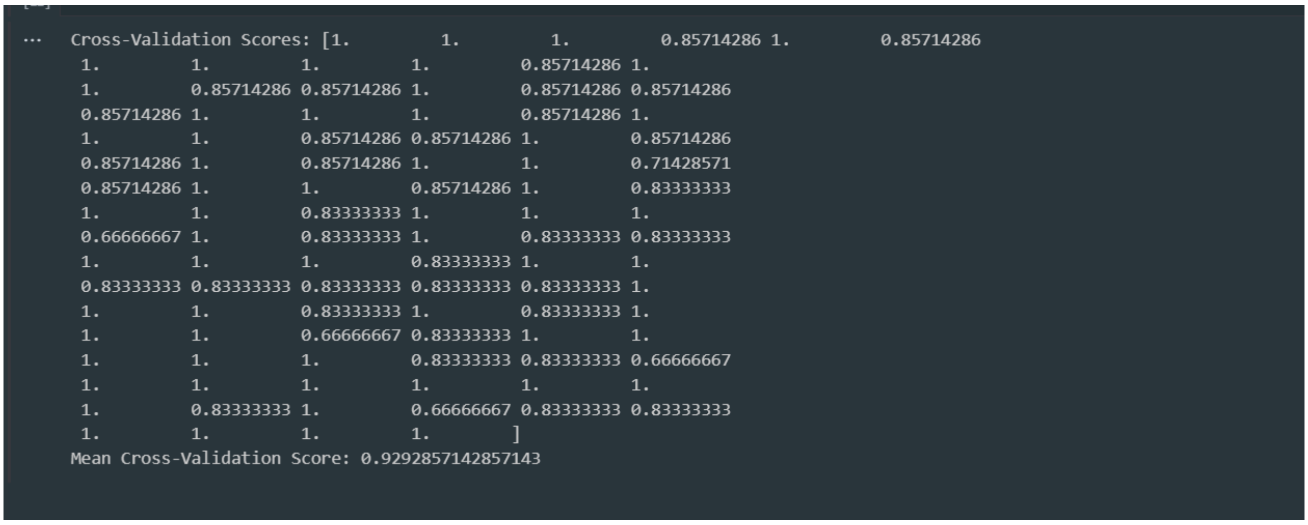


```
Cross-Validation Scores: [0.99929777 0.99925388 0.9993636  0.99938555 0.9993636 ]
Mean Cross-Validation Score: 0.9993328798086418
```

After handling class imbalance:



```
Confusion Matrix:
[[73  1]
 [14 72]]

Classification Report:
              precision    recall  f1-score   support

           0       0.84      0.99      0.91        74
           1       0.99      0.84      0.91        86

    accuracy                           0.91       160
   macro avg       0.91      0.91      0.91       160
weighted avg       0.92      0.91      0.91       160
```



```
... Cross-Validation Scores: [1.         1.         1.         0.85714286 1.         0.85714286
     1.         1.         1.         1.         0.85714286 1.
     1.         0.85714286 0.85714286 1.         0.85714286 0.85714286
     0.85714286 1.         1.         1.         0.85714286 1.
     1.         1.         0.85714286 0.85714286 1.         0.85714286
     0.85714286 1.         0.85714286 1.         1.         0.71428571
     0.85714286 1.         1.         0.85714286 1.         0.83333333
     1.         1.         0.83333333 1.         1.         1.
     0.66666667 1.         0.83333333 1.         0.83333333 0.83333333
     1.         1.         1.         0.83333333 1.         1.
     0.83333333 0.83333333 0.83333333 0.83333333 0.83333333 1.
     1.         1.         0.83333333 1.         0.83333333 1.
     1.         1.         0.66666667 0.83333333 1.         1.
     1.         1.         1.         0.83333333 0.83333333 0.66666667
     1.         1.         1.         1.         1.         1.
     1.         0.83333333 1.         0.66666667 0.83333333 0.83333333
     1.         1.         1.         1.        ]
    Mean Cross-Validation Score: 0.9292857142857143
```

Introduction: Quantum Machine Learning Exploration

As we embark on a journey through the landscape of quantum machine learning models, it is essential to understand the foundational elements that form the backbone of our exploration. At the heart of quantum machine learning lies the concept of quantum circuits—configurations of quantum gates that manipulate quantum states to process information.

In the realm of quantum computing, a **quantum circuit** serves as the quantum analog of classical circuits, allowing us to perform computations on quantum bits or qubits. These circuits leverage operations like rotations, entanglements, and controlled gates to encode and process information in ways fundamentally different from classical bits.

Cost functions play a pivotal role in guiding the training of quantum machine learning models. They quantify the "cost" or "error" associated with the difference between the predicted and target outcomes. Reducing this cost function involves adjusting the parameters of the quantum model to optimize its performance, a process analogous to classical machine learning optimization but with quantum twists.

A distinctive feature of our quantum models lies in the utilization of **strongly entangling layers**. These layers leverage quantum entanglement, a phenomenon where the states of qubits become correlated, allowing for richer and more intricate quantum representations. By strategically incorporating strongly entangling layers, we harness the power of quantum entanglement to enhance the expressive capacity of our models.

Additionally, in our exploration, we integrate **Principal Component Analysis (PCA)** as a preprocessing step. PCA serves as a classical dimensionality reduction technique, allowing us to capture essential features of the data and reduce computational complexity. The combined use of PCA with quantum machine learning models provides a comprehensive approach, amalgamating classical and quantum techniques to handle intricate datasets effectively.

Throughout this exploration, we will traverse multiple quantum machine learning models, each building upon the last, with a deliberate progression towards increased accuracy. Additionally, we'll touch upon models yet to be realized, outlining the trajectory of ongoing research and potential avenues for future advancements.

As we delve into the intricate interplay of quantum circuits, cost functions, entangling layers, and classical preprocessing techniques like PCA, we aim to uncover the unique capabilities and challenges that quantum machine learning introduces to the forefront of computational research.

MODEL 1

Quantum Circuit-Based Pennylane Model

The provided code is an implementation of a simple quantum machine learning model using PennyLane for the task of credit card fraud detection. Below is a concise explanation of the code:

1. **Data Loading and Preprocessing:**
   - The credit card dataset is loaded from a CSV file ('creditcard.csv') using pandas.
   - Features and labels are extracted, and the dataset is split into training and testing sets (80% training, 20% testing).
   - Feature standardization is performed using `StandardScaler` from scikit-learn.

```
data = pd.read_csv('creditcard.csv')
X = data.drop('Class', axis=1)
y = data['Class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

2. **Quantum Circuit Definition:**
   - A PennyLane quantum circuit is defined using the `quantum_circuit` function.
   - The circuit consists of two qubits with rotations (`RY` gates) and a Controlled-NOT (CNOT) gate.
   - The expectation value of the Pauli-Z operator on the first qubit is measured.

```
dev = qml.device('default.qubit', wires=2)
@qml.qnode(dev)
def quantum_circuit(params, x):
    qml.RY(x[0], wires=0)
    qml.RY(x[1], wires=1)
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliZ(0))
```

3. **Quantum Model and Cost Function**:
   - The `qml_model` function applies the quantum circuit to each data point in the input.
   - The `cost` function calculates the mean squared error between the quantum model predictions and the true labels.

```
def qml_model(params, x):
    result = np.array([quantum_circuit(params, xi) for xi in x])
    return result
```

```
def cost(params, X, y):
   predictions = qml_model(params, X)
   return np.mean((predictions - y) 2)
```

4. **Model Training:**
   - Initial parameters are set, and an Adam optimizer is used for training the quantum model.
   - The training loop runs for a specified number of epochs, updating the parameters to minimize the cost function.

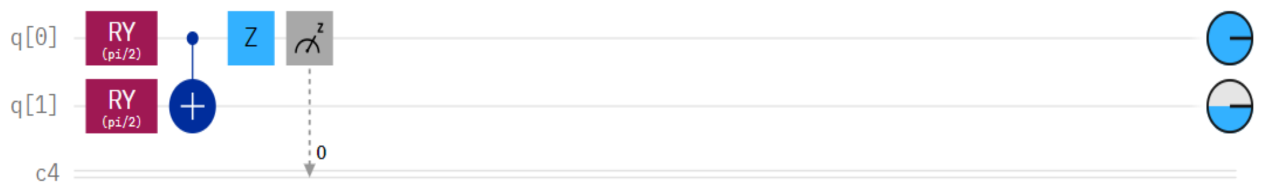
```
params = np.array([0.1, 0.2])  # Initial parameters
optimizer = AdamOptimizer(0.1)
num_epochs = 10

for epoch in range(num_epochs):
   params = optimizer.step(lambda p: cost(p, X_train, y_train), params)
```

5. **Testing and Evaluation:**
   - The trained quantum model is applied to the test dataset.
   - Predictions are thresholded to binary (0 or 1) based on a threshold of 0.5.
   - Standard classification metrics (accuracy, precision, recall, F1-score) and the confusion matrix are computed and printed.

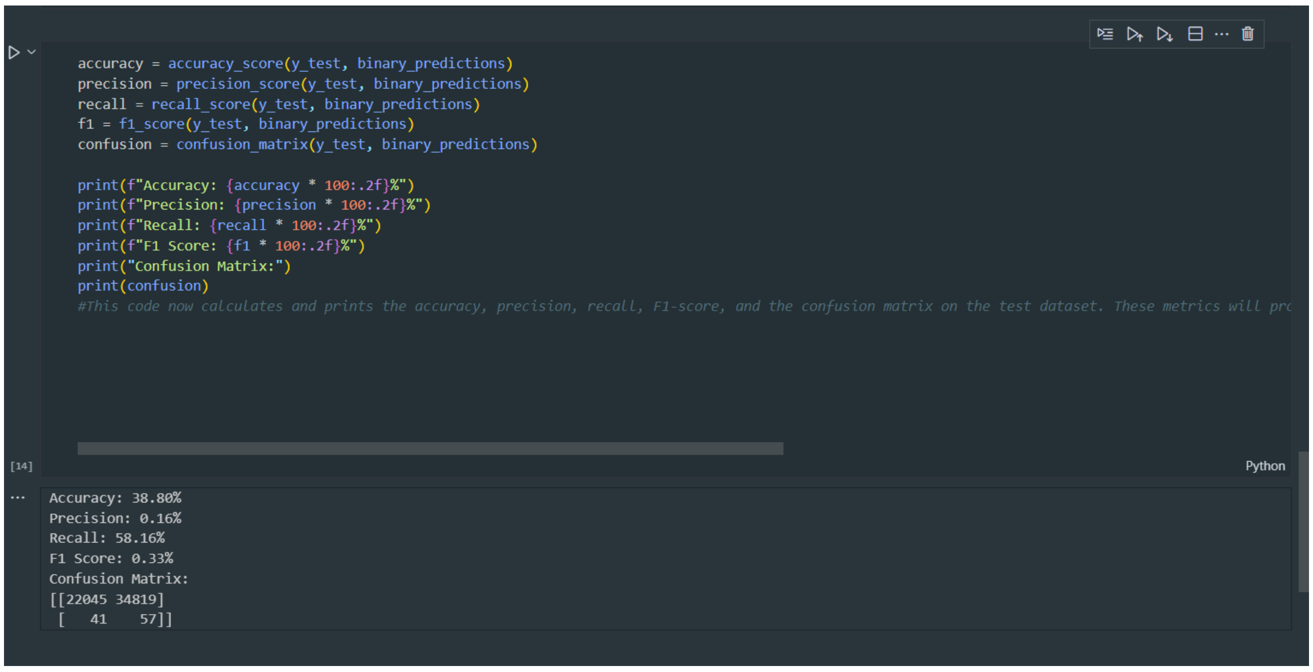
```
predictions = qml_model(params, X_test)
binary_predictions = np.where(predictions >= 0.5, 1, 0)

accuracy = accuracy_score(y_test, binary_predictions)
precision = precision_score(y_test, binary_predictions)
recall = recall_score(y_test, binary_predictions)
f1 = f1_score(y_test, binary_predictions)
confusion = confusion_matrix(y_test, binary_predictions)
```

This code demonstrates a basic quantum machine learning model using PennyLane for credit card fraud detection, with the evaluation metrics providing insights into the model's performance on the test dataset.

Circuit representation:



Results :



```
accuracy = accuracy_score(y_test, binary_predictions)
precision = precision_score(y_test, binary_predictions)
recall = recall_score(y_test, binary_predictions)
f1 = f1_score(y_test, binary_predictions)
confusion = confusion_matrix(y_test, binary_predictions)

print(f"Accuracy: {accuracy * 100:.2f}%")
print(f"Precision: {precision * 100:.2f}%")
print(f"Recall: {recall * 100:.2f}%")
print(f"F1 Score: {f1 * 100:.2f}%")
print("Confusion Matrix:")
print(confusion)
#This code now calculates and prints the accuracy, precision, recall, F1-score, and the confusion matrix on the test dataset. These metrics will pro
```

[14] Python

```
Accuracy: 38.80%
Precision: 0.16%
Recall: 58.16%
F1 Score: 0.33%
Confusion Matrix:
[[22045 34819]
 [   41    57]]
```

We can see the accuracy and the f1-score of this model is pretty low. Reason might be the inconsideration of class imbalance in the dataset.

MODEL 2

QML model using Strongly Entangling Layers

1. **Data Preprocessing:**
   - The credit card fraud detection dataset is loaded from 'creditcard.csv'.
   - Data is divided into two subsets: `data1` for instances where `Class` is 1 (fraudulent transactions), and `data0` for randomly sampled instances where `Class` is 0 (non-fraudulent transactions). A total of 800 instances of class 0 are selected.
   - `data0` and `data1` are concatenated, and the resulting dataset is shuffled randomly.

```
data = pd.read_csv('creditcard.csv')
data1 = data[data['Class'] == 1]
data0 = data[data['Class'] == 0].sample(800)
data = pd.concat([data0, data1])
data = data.sample(frac=1).reset_index(drop=True)
```

We have done this to remove the class imbalance in our model.

2. **Feature Extraction and Standardization:**
   - Features (columns v1 to v28) and labels (Class) are extracted from the dataset.
   - Features are standardized using `StandardScaler`.
   - **Principal Component Analysis (PCA) is applied to reduce the dimensionality to 2 components.**

```
features = data.iloc[:, 1:-1].values
labels = data.iloc[:, -1].values
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)
pca = PCA(n_components=2)
features_pca = pca.fit_transform(features_scaled)
```

3. **Data Splitting:**
   - The dataset is split into training and testing sets.

```
X_train, X_test, y_train, y_test = train_test_split(features_pca, labels, test_size=0.2,
random_state=42)
```

4. **Quantum Circuit Definition:**
   - A quantum device with two qubits is defined using PennyLane's `lightning.qubit` simulator.

- The quantum circuit is defined using the `circuit` function, employing **strongly entangling layers** from PennyLane's template.

```
dev = qml.device("lightning.qubit", wires=n_qubits)

@qml.qnode(dev)
def circuit(params, x):
    qml.templates.StronglyEntanglingLayers(params, wires=range(n_qubits))
    return qml.expval(qml.PauliZ(0))
```

5. **Cost Function and Optimization:**
   - The cost function is defined to calculate the mean squared error between the quantum circuit predictions and the true labels.
   - Parameters are randomly initialized, and the circuit parameters are optimized using gradient descent.

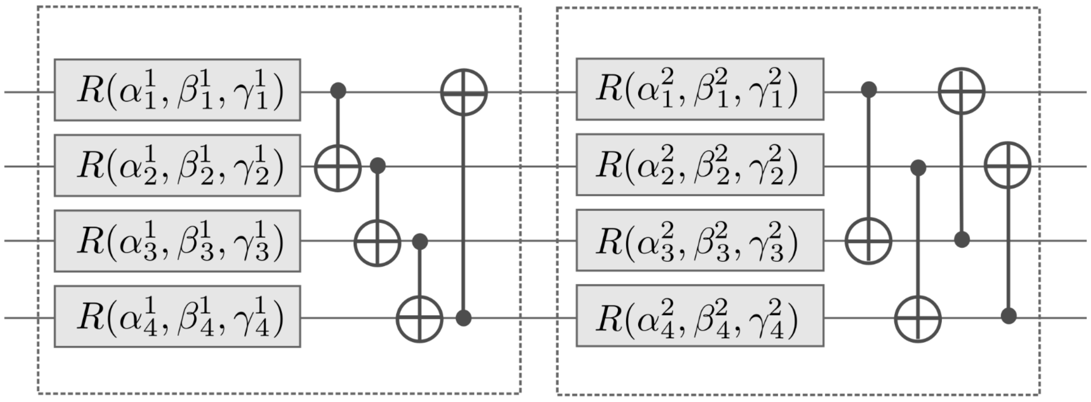
```
num_layers = 3
params = np.random.rand(num_layers, n_qubits, 3)
opt = qml.GradientDescentOptimizer(0.1)

for step in range(10):
    params = opt.step(lambda p: cost(p, X_train, y_train), params)
```

6. **Prediction and Evaluation:**
   - Predictions are made on the test set by applying the quantum circuit.
   - Classification accuracy and F1 score are calculated and printed.

```
predictions = [np.sign(circuit(params, x)) for x in X_test]

accuracy = accuracy_score(y_test, predictions)
print("Classification Accuracy:", accuracy)

f1 = f1_score(y_test, predictions)
print("F1 Score:", f1)
```

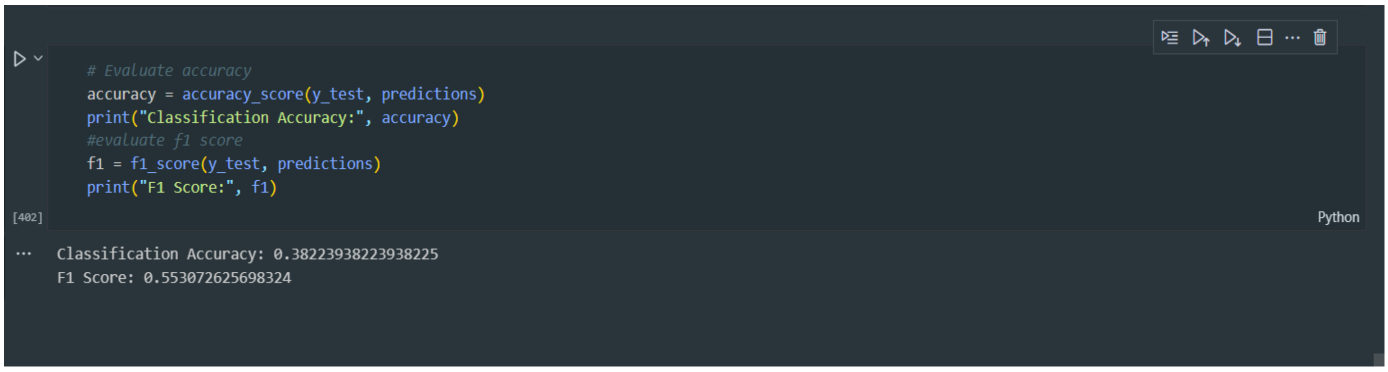
This model combines classical preprocessing techniques, including PCA, with a quantum machine learning approach using PennyLane to create a hybrid model for credit card fraud detection. The strongly entangling layers in the quantum circuit aim to capture complex relationships in the data, and the model's performance is evaluated using classical metrics like accuracy and F1 score.

Strongly Entangling layers example (4 qubits)



$R(\alpha_1^1, \beta_1^1, \gamma_1^1)$
$R(\alpha_2^1, \beta_2^1, \gamma_2^1)$
$R(\alpha_3^1, \beta_3^1, \gamma_3^1)$
$R(\alpha_4^1, \beta_4^1, \gamma_4^1)$
$R(\alpha_1^2, \beta_1^2, \gamma_1^2)$
$R(\alpha_2^2, \beta_2^2, \gamma_2^2)$
$R(\alpha_3^2, \beta_3^2, \gamma_3^2)$
$R(\alpha_4^2, \beta_4^2, \gamma_4^2)$

Results :



```
# Evaluate accuracy
accuracy = accuracy_score(y_test, predictions)
print("Classification Accuracy:", accuracy)
#evaluate f1 score
f1 = f1_score(y_test, predictions)
print("F1 Score:", f1)
```

```
Classification Accuracy: 0.38223938223938225
F1 Score: 0.553072625698324
```

We got **classification accuracy of 38.22%** and **F1-score of 55.30%,** which is better than the first model.

Quantum-SVM (QSVM) for Credit Card Fraud Detection

In the quest for advancing credit card fraud detection methodologies, the Quantum-SVM (QSVM) model emerges as a sophisticated fusion of quantum and classical machine learning techniques. This innovative approach seeks to exploit the unique capabilities of quantum feature mapping, facilitated by PennyLane, coupled with classical Support Vector Machine (SVM) methodologies.

Quantum Feature Mapping:

The QSVM model initiates its operations with a meticulously designed quantum feature mapping. Leveraging PennyLane's quantum circuit capabilities, the model orchestrates a series of quantum gates, including Hadamard gates, rotation gates, and controlled operations. This quantum circuit is adept at capturing intricate relationships within the dataset, providing a nuanced representation of the underlying patterns in credit card transactions.

Parameter Initialization:

Critical to the success of the QSVM model is the strategic initialization of quantum circuit parameters. These parameters govern the transformations applied during the quantum feature mapping process. The randomized initialization ensures adaptability and responsiveness to the inherent complexity of credit card transaction data.

SVM Integration:

The quantum-transformed dataset seamlessly integrates into a classical SVM framework. Notably, the QSVM employs a precomputed kernel matrix, derived from the quantum feature map, as the foundation for SVM training and subsequent predictions. This integration serves as a bridge between quantum and classical methodologies, leveraging the strengths of both domains for robust fraud detection.

Evaluation Metrics:

The efficacy of the QSVM model is rigorously evaluated using standard classification metrics. Focus is placed on the accuracy of classification, measuring the model's ability to correctly identify fraudulent and non-fraudulent transactions. Additionally, the F1 score, a metric balancing precision and recall, provides insights into the model's capacity to minimize false positives while capturing instances of fraud.

In the ensuing sections, we delve into the intricate details of quantum feature mapping, parameter tuning, SVM integration, and comprehensive model evaluation. The QSVM model represents a pioneering stride towards enhancing the efficacy and resilience of credit card fraud detection systems, harnessing the synergy between quantum and classical computing paradigms.

MODEL 3

Quantum SVM (QSVM) with Optimized Quantum Feature Map

1. **Data Preprocessing:**

   The dataset is balanced to address class imbalances, ensuring a representative mix of fraudulent and non-fraudulent transactions.

```
#take 492 class=0 data and join it with all class =1 data and shuffle the data
data0 = data[data['Class'] == 0].sample(492)
data0

#take 492 class=1 data
data1 = data[data['Class'] == 1]

#data0+data1
data = pd.concat([data0, data1])
data

#shuffle randomly 'data dataframe'
data = data.sample(frac=1).reset_index(drop=True)
data
```

2. **Feature Extraction and Standardization:**

   Features and labels are extracted from the balanced dataset.

   The dataset is split into training and testing sets, and feature scaling is applied using StandardScaler.

```
# Extract features and labels
X = data.iloc[:, 1:-1].values
y = data.iloc[:, -1].values

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

3. **Quantum Feature Map Optimization:**

   A quantum circuit (quantum_circuit) is defined, incorporating Hadamard gates, rotation gates, CNOT gates, and RY gates.

   Quantum circuit parameters are initialized randomly.

```
# Quantum feature map
n_qubits = 2

dev = qml.device("default.qubit", wires=n_qubits)
@qml.qnode(dev)
def quantum_circuit(params, x):
   qml.Hadamard(wires=0)
   qml.Hadamard(wires=1)
   qml.RZ(params[0], wires=0)
   qml.RZ(params[1], wires=1)
   qml.CNOT(wires=[0, 1])
   qml.RY(params[2], wires=1)  # Use the last parameter for RY on the second qubit
   return [qml.expval(qml.PauliZ(i)) for i in range(n_qubits)]

# Initialize the quantum circuit parameters
params = np.random.uniform(low=0, high=2 * np.pi, size=(3,))

# Evaluate the quantum circuit for the training data
feature_map_train = np.array([quantum_circuit(params, x) for x in X_train])

# Use the quantum feature map to transform the testing data
feature_map_test = np.array([quantum_circuit(params, x) for x in X_test])
```

4. **SVM Integration:**

   The quantum-transformed dataset integrates into a classical SVM framework with a precomputed kernel.

   The SVM is trained on the precomputed kernel matrix and used for predictions on the testing data.

```
svm = SVC(kernel='precomputed')
gram_matrix_train = np.dot(feature_map_train, feature_map_train.T)
svm.fit(gram_matrix_train, y_train)

# Predict on the testing data
gram_matrix_test = np.dot(feature_map_test, feature_map_train.T)
y_pred = svm.predict(gram_matrix_test)
```

5. **Model Evaluation:**

   Classification accuracy and F1 score are computed and printed, providing a comprehensive assessment of the model's performance in credit card fraud detection.

   This advanced QSVM model represents a concerted effort to optimize the quantum feature mapping process, augmenting the model's ability to discern patterns indicative of fraudulent transactions. The robust evaluation metrics serve as a testament to the model's efficacy in enhancing fraud detection capabilities.

```
# Evaluate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Classification Accuracy: {accuracy}")

#evaluate f1 score
f1 = f1_score(y_test, y_pred)
print(f"F1 Score: {f1}")
```

Quantum Feature Map Explanation

Quantum Circuit Definition:

The quantum feature map is defined as the quantum_circuit using PennyLane.

The circuit operates on a quantum device with two qubits (n_qubits = 2), specified by qml.device("default.qubit", wires=n_qubits).

Quantum Gates:

Hadamard gates (qml.Hadamard) are applied to both qubits, creating superpositions.

Rotational gates (qml.RZ and qml.RY) introduce phase shifts to qubits based on specified parameters (params).

Entangling Operation:

A CNOT gate (qml.CNOT) entangles the two qubits, allowing for quantum correlations.

Parameter Initialization:

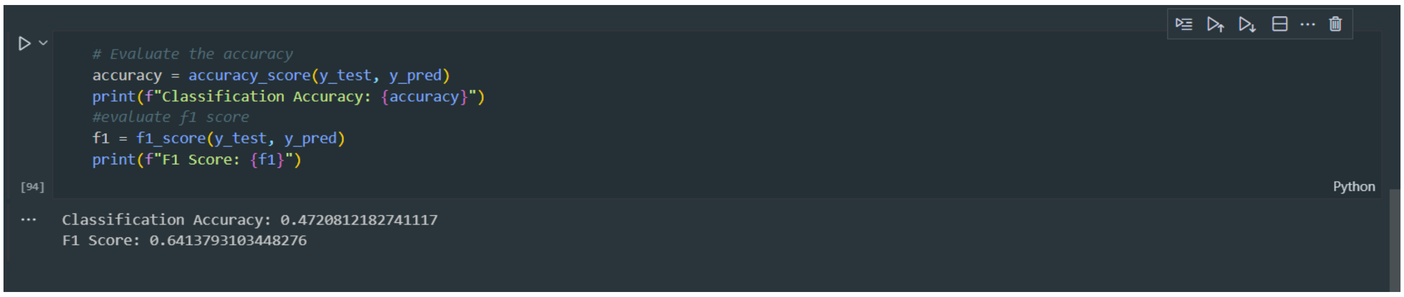
Quantum circuit parameters (params) are initialized with random values between 0 and 2π.

Evaluation for Training and Testing Data:

The quantum circuit is evaluated for both training and testing datasets, producing the quantum feature maps (feature_map_train and feature_map_test).

This quantum feature map serves as a unique representation of the input data in a quantum space, capturing intricate relationships that classical methods may overlook. The resulting quantum feature maps are then utilized in subsequent steps of the QSVM model.

Results :



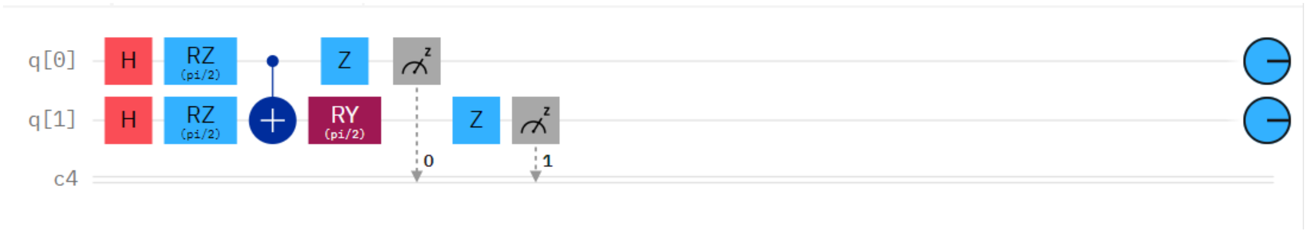
```
# Evaluate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Classification Accuracy: {accuracy}")
#evaluate f1 score
f1 = f1_score(y_test, y_pred)
print(f"F1 Score: {f1}")
```

```
Classification Accuracy: 0.4720812182741117
F1 Score: 0.6413793103448276
```

Classification Accuracy : 47.20%

F1-score: 64.13%

Gates representation :



MODEL 4

Quantum SVM using Strongly Entangling Layers

This model is more or less similar to the previous model, but there are two differences, which are use of PCA in this model to reduce dimensions from 29 to 2 and the use of Strongly Entangling Layers in Quantum Feature Mapping instead of Quantum circuits. Here is the code for the same:

1. Data Loading and Preprocessing:

- The credit card dataset is loaded.
- Class 0 data is sampled to create a balanced dataset along with all Class 1 data.
- The dataset is shuffled for randomness.

```
#take 800 class=0 data and join it with all class =1 data and shuffle the data
data0 = data[data['Class'] == 0].sample(800)
data0
#take 492 class=1 data
data1 = data[data['Class'] == 1]
#data0+data1
data = pd.concat([data0, data1])
data
#shuffle randomly 'data dataframe'
data = data.sample(frac=1).reset_index(drop=True)
data
```

2. Feature Extraction and Reduction:

- Features and labels are extracted from the dataset.
- Standardization is applied to the features.
- Principal Component Analysis (PCA) is utilized to reduce the features to two dimensions.

```
# Extract features and labels
X = data.iloc[:, 1:-1].values
y = data.iloc[:, -1].values

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)


# Apply PCA to reduce the number of features to 2
pca = PCA(n_components=2)
```

```
X_pca = pca.fit_transform(X_scaled)
```

3. Data Splitting:

- The preprocessed dataset is split into training and testing sets with an 80-20 ratio.

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=42)
```

4. Quantum Feature Map Definition:

- A quantum feature map is defined using PennyLane's `StronglyEntanglingLayers`.
- It is designed to transform classical data into quantum features.

```
# Define the quantum device
n_qubits = 2
dev = qml.device("default.qubit", wires=n_qubits)

# Quantum feature map
@qml.qnode(dev)
def quantum_feature_map(params, x):
   qml.templates.StronglyEntanglingLayers(params, wires=range(n_qubits))
   return [qml.expval(qml.PauliZ(i)) for i in range(n_qubits)]
```

5. Quantum Feature Map Application:

- Quantum circuit parameters are initialized randomly.
- Quantum feature maps are applied to both the training and testing data.
- This quantum transformation introduces quantum-inspired features to classical data.

```
# Initialize the quantum circuit parameters
params = np.random.uniform(low=0, high=2 * np.pi, size=(3, n_qubits,3))

# Apply the quantum feature map to the training and testing data
feature_map_train = np.array([quantum_feature_map(params, x) for x in X_train])
feature_map_test = np.array([quantum_feature_map(params, x) for x in X_test])
```

6. Classical SVM Training:

- A classical Support Vector Machine (SVM) is trained on the quantum features.
- The `SVC` (Support Vector Classification) from scikit-learn is employed for this purpose.

```
# Initialize the quantum circuit parameters
```

params = np.random.uniform(low=0, high=2 * np.pi, size=(3, n_qubits,3))

# Apply the quantum feature map to the training and testing data
feature_map_train = np.array([quantum_feature_map(params, x) for x in X_train])
feature_map_test = np.array([quantum_feature_map(params, x) for x in X_test])

7. Prediction and Evaluation:

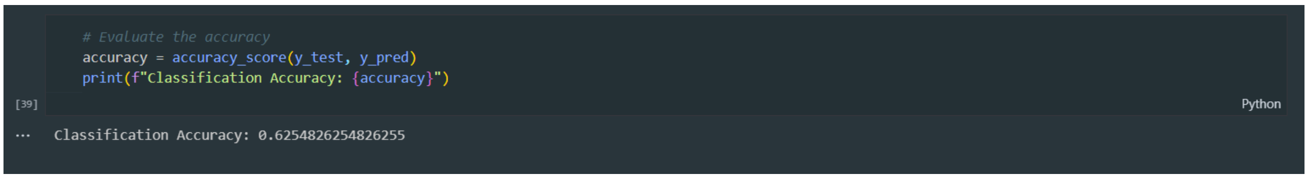
- Predictions are made on the testing data using the trained QFM-SVM model.
- The accuracy of the model is evaluated using classical machine learning metrics.
- The classical SVM is applied to quantum features, showcasing a hybrid approach for classification tasks.

# Train a classical SVM on the quantum features
svm_classifier = SVC()
svm_classifier.fit(feature_map_train, y_train)

# Predict on the testing data
y_pred = svm_classifier.predict(feature_map_test)

# Evaluate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Classification Accuracy: {accuracy}")

RESULTS:



```
# Evaluate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Classification Accuracy: {accuracy}")
```

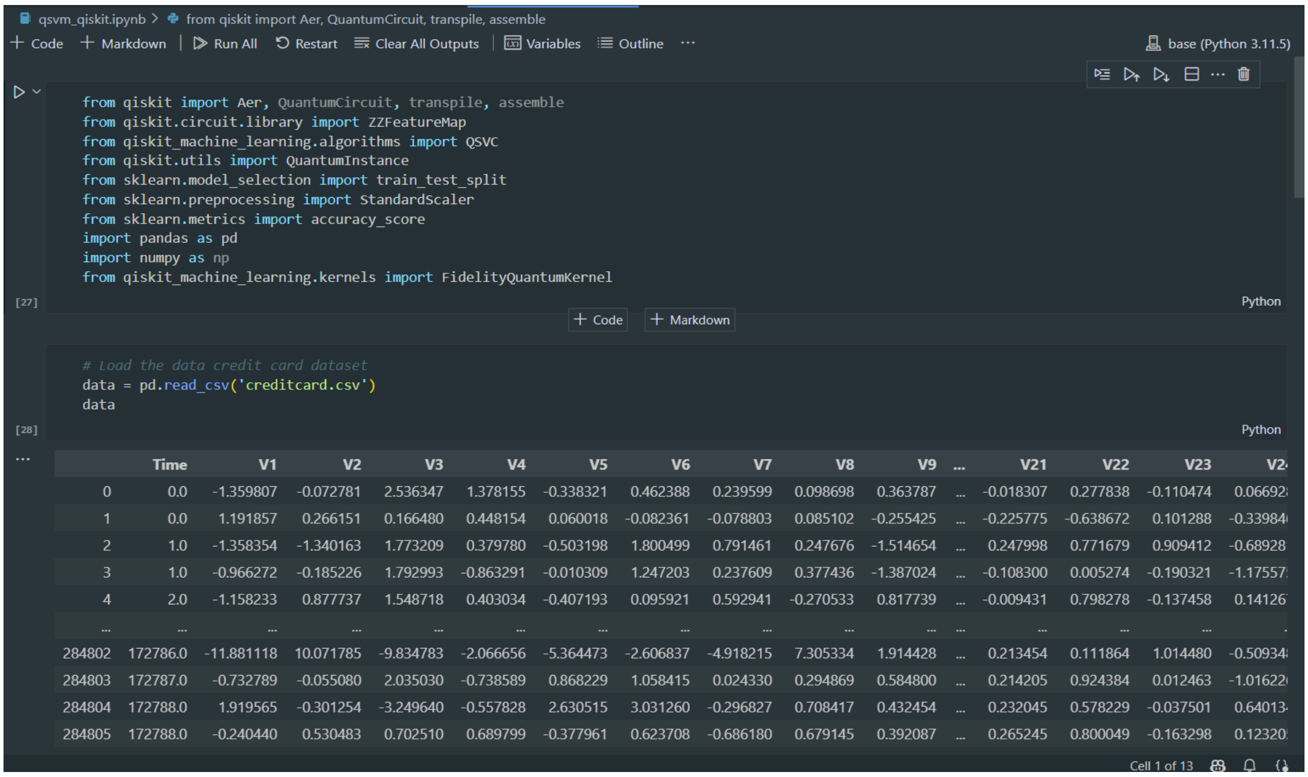
```
Classification Accuracy: 0.6254826254826255
```

Classification Accuracy on this model was found to be 62.54%.

MODEL 5

<u>QSVM model using Qiskit</u>

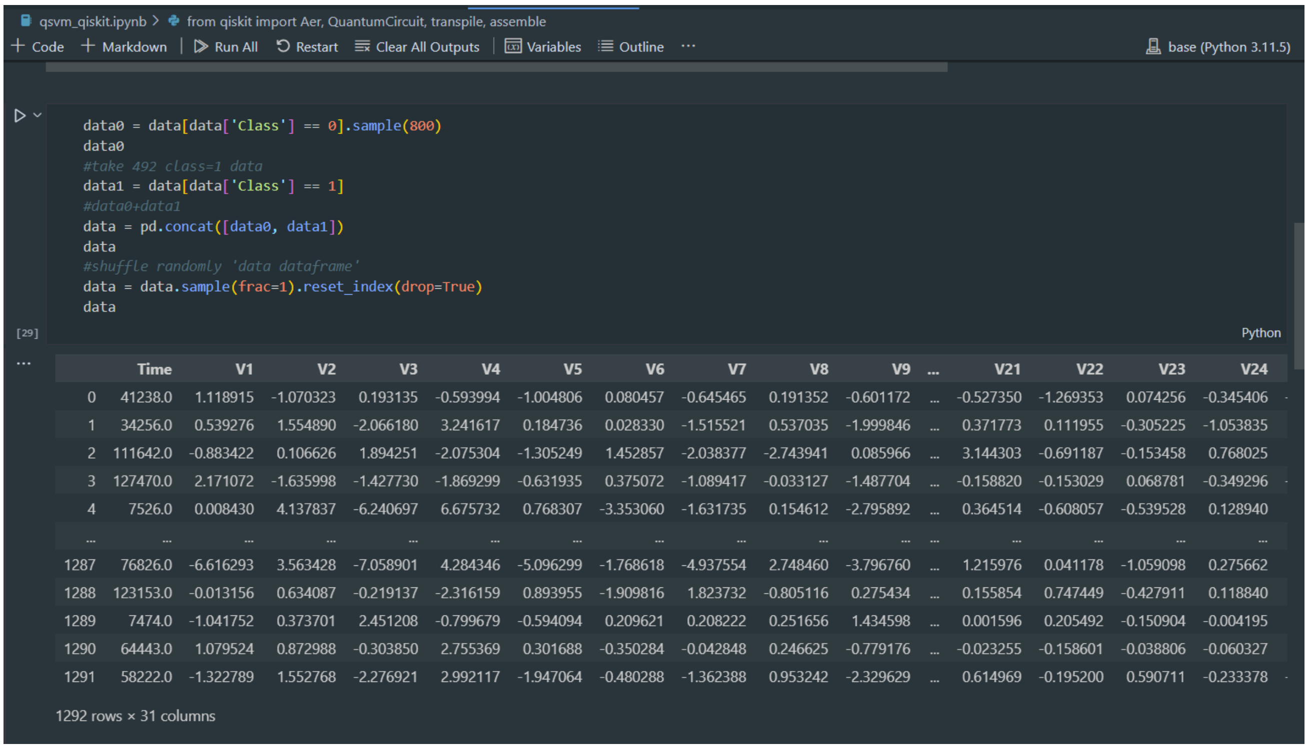
A Quantum Support Vector Machine (QSVM) model was constructed directly using Qiskit. Regrettably, its execution encountered considerable delays, surpassing the 10-hour mark and persisting, primarily due to system configuration challenges.
Here is the code for the same:



```python
from qiskit import Aer, QuantumCircuit, transpile, assemble
from qiskit.circuit.library import ZZFeatureMap
from qiskit_machine_learning.algorithms import QSVC
from qiskit.utils import QuantumInstance
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
import pandas as pd
import numpy as np
from qiskit_machine_learning.kernels import FidelityQuantumKernel
```

```python
# Load the data credit card dataset
data = pd.read_csv('creditcard.csv')
data
```

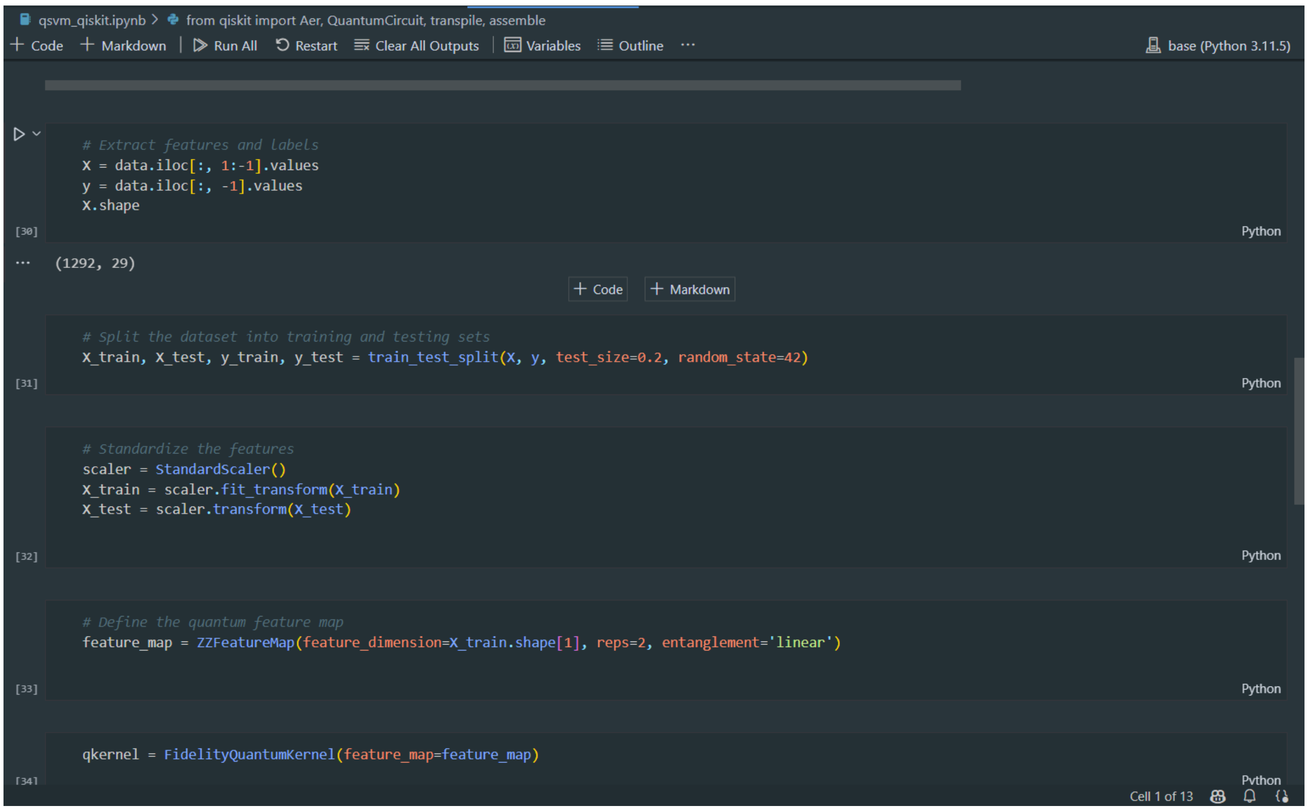
```
            Time         V1         V2         V3         V4         V5         V6         V7         V8         V9  ...        V21        V22        V23       V2
     0       0.0  -1.359807  -0.072781   2.536347   1.378155  -0.338321   0.462388   0.239599   0.098698   0.363787  ...  -0.018307   0.277838  -0.110474   0.06692
     1       0.0   1.191857   0.266151   0.166480   0.448154   0.060018  -0.082361  -0.078803   0.085102  -0.255425  ...  -0.225775  -0.638672   0.101288  -0.33984
     2       1.0  -1.358354  -1.340163   1.773209   0.379780  -0.503198   1.800499   0.791461   0.247676  -1.514654  ...   0.247998   0.771679   0.909412  -0.68928
     3       1.0  -0.966272  -0.185226   1.792993  -0.863291  -0.010309   1.247203   0.237609   0.377436  -1.387024  ...  -0.108300   0.005274  -0.190321  -1.17557
     4       2.0  -1.158233   0.877737   1.548718   0.403034  -0.407193   0.095921   0.592941  -0.270533   0.817739  ...  -0.009431   0.798278  -0.137458   0.14126
   ...       ...        ...        ...        ...        ...        ...        ...        ...        ...        ...  ...        ...        ...        ...
284802  172786.0 -11.881118  10.071785  -9.834783  -2.066656  -5.364473  -2.606837  -4.918215   7.305334   1.914428  ...   0.213454   0.111864   1.014480  -0.50934
284803  172787.0  -0.732789  -0.055080   2.035030  -0.738589   0.868229   1.058415   0.024330   0.294869   0.584800  ...   0.214205   0.924384   0.012463  -1.01622
284804  172788.0   1.919565  -0.301254  -3.249640  -0.557828   2.630515   3.031260  -0.296827   0.708417   0.432454  ...   0.232045   0.578229  -0.037501   0.64013
284805  172788.0  -0.240440   0.530483   0.702510   0.689799  -0.377961   0.623708  -0.686180   0.679145   0.392087  ...   0.265245   0.800049  -0.163298   0.12320
```


qsvm_qiskit.ipynb > from qiskit import Aer, QuantumCircuit, transpile, assemble
+ Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ···
base (Python 3.11.5)

```python
data0 = data[data['Class'] == 0].sample(800)
data0
#take 492 class=1 data
data1 = data[data['Class'] == 1]
#data0+data1
data = pd.concat([data0, data1])
data
#shuffle randomly 'data dataframe'
data = data.sample(frac=1).reset_index(drop=True)
data
```

[29] Python

```
          Time         V1         V2         V3         V4         V5         V6         V7         V8         V9  ...        V21        V22        V23        V24
   0   41238.0   1.118915  -1.070323   0.193135  -0.593994  -1.004806   0.080457  -0.645465   0.191352  -0.601172  ...  -0.527350  -1.269353   0.074256  -0.345406
   1   34256.0   0.539276   1.554890  -2.066180   3.241617   0.184736   0.028330  -1.515521   0.537035  -1.999846  ...   0.371773   0.111955  -0.305225  -1.053835
   2  111642.0  -0.883422   0.106626   1.894251  -2.075304  -1.305249   1.452857  -2.038377  -2.743941   0.085966  ...   3.144303  -0.691187  -0.153458   0.768025
   3  127470.0   2.171072  -1.635998  -1.427730  -1.869299  -0.631935   0.375072  -1.089417  -0.033127  -1.487704  ...  -0.158820  -0.153029   0.068781  -0.349296
   4    7526.0   0.008430   4.137837  -6.240697   6.675732   0.768307  -3.353060  -1.631735   0.154612  -2.795892  ...   0.364514  -0.608057  -0.539528   0.128940
 ...       ...        ...        ...        ...        ...        ...        ...        ...        ...        ...  ...        ...        ...        ...        ...
1287   76826.0  -6.616293   3.563428  -7.058901   4.284346  -5.096299  -1.768618  -4.937554   2.748460  -3.796760  ...   1.215976   0.041178  -1.059098   0.275662
1288  123153.0  -0.013156   0.634087  -0.219137  -2.316159   0.893955  -1.909816   1.823732  -0.805116   0.275434  ...   0.155854   0.747449  -0.427911   0.118840
1289    7474.0  -1.041752   0.373701   2.451208  -0.799679  -0.594094   0.209621   0.208222   0.251656   1.434598  ...   0.001596   0.205492  -0.150904  -0.004195
1290   64443.0   1.079524   0.872988  -0.303850   2.755369   0.301688  -0.350284  -0.042848   0.246625  -0.779176  ...  -0.023255  -0.158601  -0.038806  -0.060327
1291   58222.0  -1.322789   1.552768  -2.276921   2.992117  -1.947064  -0.480288  -1.362388   0.953242  -2.329629  ...   0.614969  -0.195200   0.590711  -0.233378

1292 rows × 31 columns
```


qsvm_qiskit.ipynb > from qiskit import Aer, QuantumCircuit, transpile, assemble
+ Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ···
base (Python 3.11.5)

```python
# Extract features and labels
X = data.iloc[:, 1:-1].values
y = data.iloc[:, -1].values
X.shape
```

[30] Python

```
(1292, 29)
```

```python
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

[31] Python

```python
# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

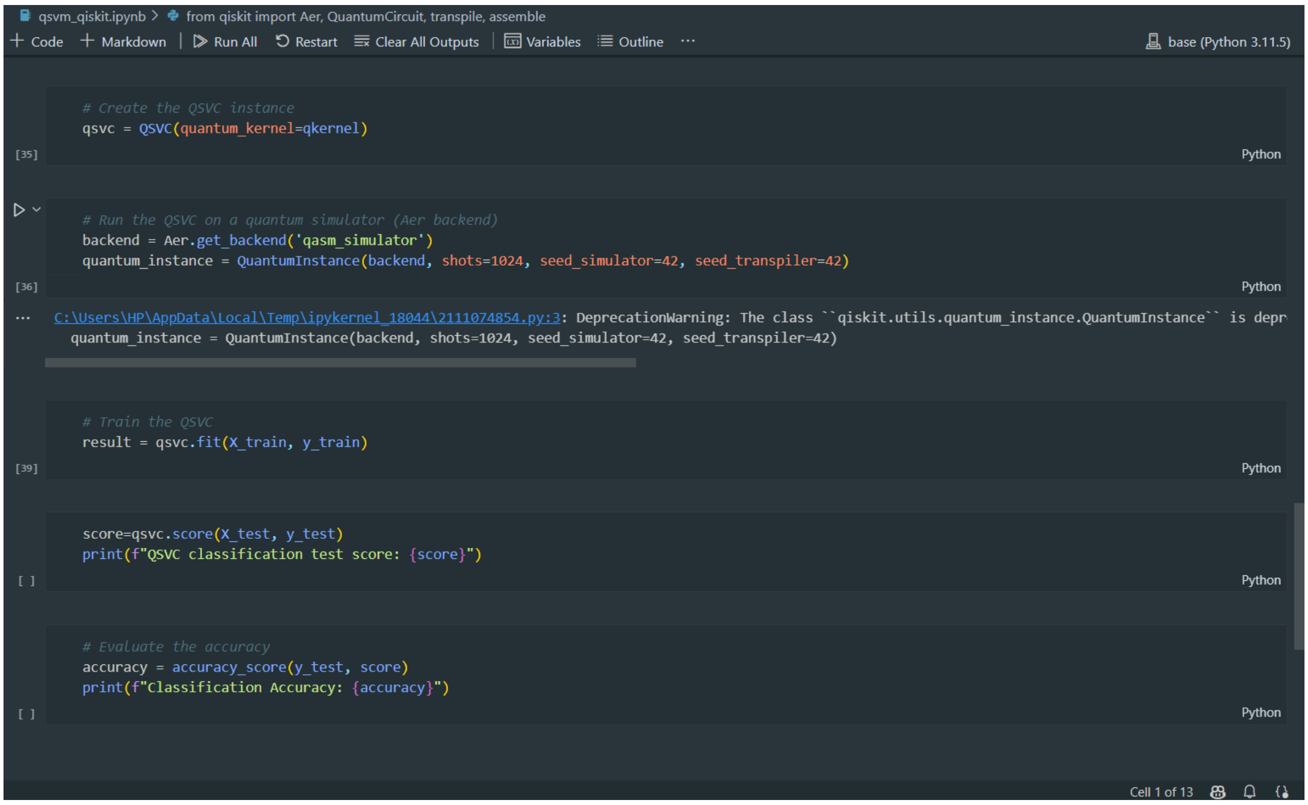
[32] Python

```python
# Define the quantum feature map
feature_map = ZZFeatureMap(feature_dimension=X_train.shape[1], reps=2, entanglement='linear')
```

[33] Python

```python
qkernel = FidelityQuantumKernel(feature_map=feature_map)
```

[34] Python


qsvm_qiskit.ipynb > from qiskit import Aer, QuantumCircuit, transpile, assemble
+ Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ...
base (Python 3.11.5)
Cell 1 of 13

```
# Create the QSVC instance
qsvc = QSVC(quantum_kernel=qkernel)
```
[35] Python

```
# Run the QSVC on a quantum simulator (Aer backend)
backend = Aer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=42, seed_transpiler=42)
```
[36] Python

```
C:\Users\HP\AppData\Local\Temp\ipykernel_18044\2111074854.py:3: DeprecationWarning: The class ``qiskit.utils.quantum_instance.QuantumInstance`` is depr
  quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=42, seed_transpiler=42)
```

```
# Train the QSVC
result = qsvc.fit(X_train, y_train)
```
[39] Python

```
score=qsvc.score(X_test, y_test)
print(f"QSVC classification test score: {score}")
```
[ ] Python

```
# Evaluate the accuracy
accuracy = accuracy_score(y_test, score)
print(f"Classification Accuracy: {accuracy}")
```
[ ] Python

1. Data Preparation:

- The credit card dataset is loaded and processed to ensure a balanced representation of both Class 0 and Class 1.

- 800 instances of Class 0 and all instances of Class 1 are combined, creating a comprehensive dataset.

- The dataset is randomly shuffled for optimal training conditions.

2. Feature Extraction and Standardization:

- Features and labels are extracted from the dataset.

- Standard scaling is applied to standardize the features, ensuring consistent scales for optimal QSVC training.

3. Quantum Feature Map and Kernel Definition:

- The quantum feature map is defined using Qiskit's ZZFeatureMap, configured with two repetitions and linear entanglement.

- FidelityQuantumKernel is employed as the quantum kernel, leveraging the fidelity metric to quantify the similarity between quantum states.

4. QSVC Model Initialization:

- QSVC (Quantum Support Vector Classification) instance is created, incorporating the defined quantum feature map and kernel.

- The QSVC model is configured to operate on a quantum simulator (Aer backend) with specific shot configurations.

5. Model Training and Evaluation:

- QSVC is trained on the standardized training dataset.

- The model's performance is evaluated on the test dataset, and the classification test score is printed.

- Classification accuracy is calculated and printed to assess the model's efficacy.

6. Quantum Instance Configuration:

- The Aer backend with the qasm_simulator is chosen for quantum simulation.

- QuantumInstance is configured with specific shot parameters and random seeds for reproducibility.

7. Conclusion:

- The code showcases the implementation of a QSVC model using Qiskit, combining classical data preprocessing with quantum feature mapping and kernel techniques.

- The QSVC model's classification accuracy on the test dataset is reported, providing insights into its performance.

ZZ Feature Map :

The `ZZFeatureMap` in Qiskit plays a pivotal role in the realm of quantum machine learning by serving as a quantum circuit designed to encode classical data into a quantum state. This process is essential for harnessing the power of quantum computing in machine learning tasks. The term "feature map" refers to the quantum analogue of classical methods that map data into higher-dimensional spaces for enhanced separability. Specifically, the "ZZ" in `ZZFeatureMap` denotes the utilization of two-qubit ZZ interaction gates within the circuit.

When using `ZZFeatureMap`, key parameters such as `feature_dimension` (representing the number of features in the input data), `reps` (indicating the repetition of the circuit for increased expressiveness), and `entanglement` (defining how qubits are entangled, with options like 'linear' or 'full') can be tailored to the characteristics of the data. The circuit's implementation involves the application of ZZ interaction gates, which introduce a phase shift based on the product of the Z-axes of two qubits. This mechanism fosters entanglement, a fundamental quantum property crucial for quantum machine learning algorithms.

In the context of the provided code, `ZZFeatureMap` is employed as the feature map within a QSVC (Quantum Support Vector Classification) model. This integration enables the transformation of classical data into a quantum state, setting the stage for subsequent quantum processing and analysis. The significance of `ZZFeatureMap` lies in its ability to bridge classical and quantum representations, enabling the utilization of quantum algorithms on classical data. The careful selection of an appropriate feature map is critical, as different maps may be suited for varying types of data and machine learning tasks. Ultimately, `ZZFeatureMap` facilitates the seamless integration of classical machine learning tasks with the computational advantages offered by quantum computing.

Fidelity Quantum Kernel

The `FidelityQuantumKernel` in Qiskit is a quantum kernel designed to quantify the similarity or dissimilarity between quantum states. Kernels play a crucial role in quantum machine learning algorithms, serving as the underlying measure of similarity that enables the application of classical machine learning techniques in a quantum context.

The term "fidelity" in `FidelityQuantumKernel` refers to the quantum fidelity, a measure of the overlap between two quantum states. High fidelity indicates a high degree of similarity between the states, while low fidelity implies dissimilarity. In the context of quantum machine learning, this kernel provides a means to assess the similarity between quantum states generated from classical data by a quantum feature map.

When using `FidelityQuantumKernel`, it is typically employed in conjunction with a quantum feature map, such as the `ZZFeatureMap`, as seen in the provided code. The quantum states produced by the feature map for different data points are compared using the fidelity as the metric. This quantification of quantum state similarity is then utilized within a QSVC (Quantum Support Vector Classification) algorithm, contributing to the decision-making process for classifying new data points.

In summary, the `FidelityQuantumKernel` facilitates the translation of classical data into quantum states through a feature map and quantifies the similarity between these states using quantum fidelity. This kernel's integration into the QSVC algorithm allows for the utilization of classical machine learning techniques in the quantum realm, demonstrating the synergistic interplay between classical and quantum approaches in quantum machine learning applications.

MODEL 6

Quantum Variational Auto-Encoder Model

(still working on it)

A Quantum Variational Autoencoder (Q-VAE) is a sophisticated quantum machine learning model that combines concepts from quantum computing and variational autoencoders (VAEs). This hybrid approach aims to leverage the potential quantum advantages for certain computational tasks, such as encoding and decoding complex data representations.

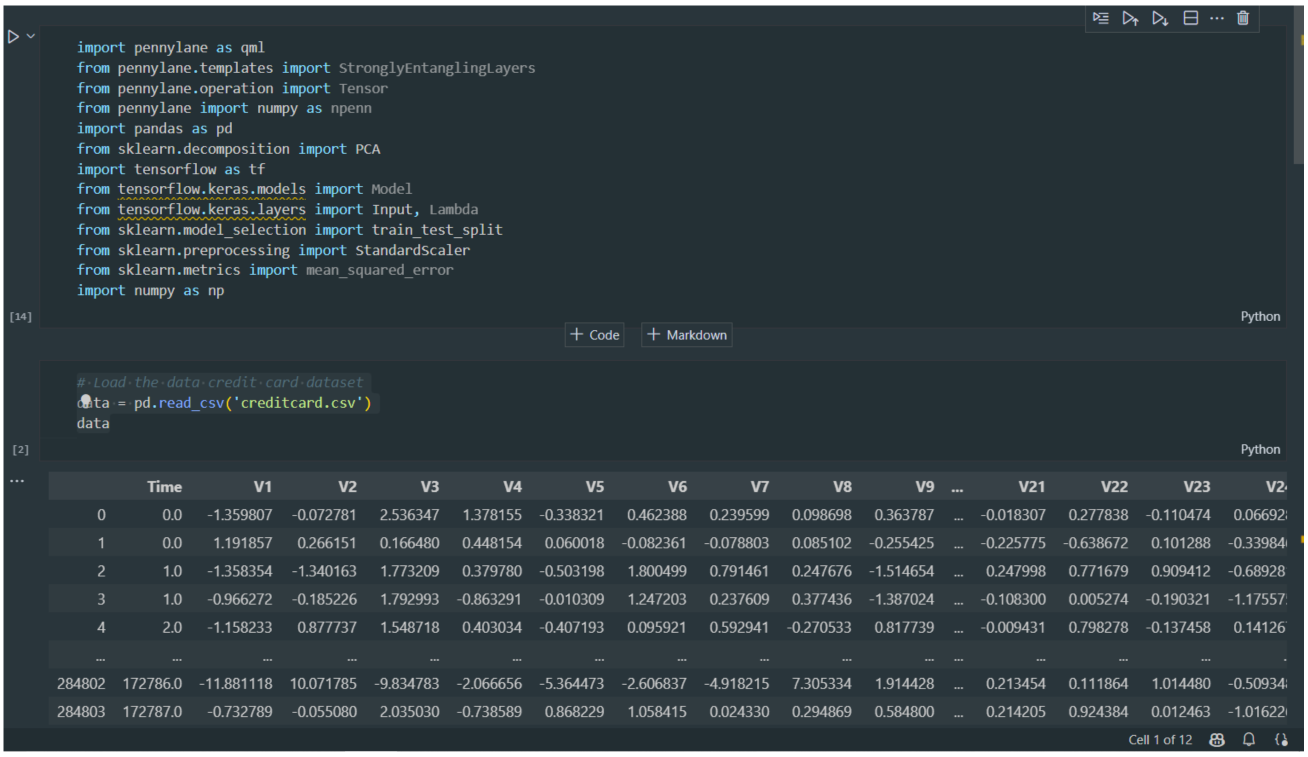
In the classical realm, Variational Autoencoders are popular generative models used for unsupervised learning. They consist of an encoder network that compresses input data into a latent space, and a decoder network that reconstructs the original data from this latent representation. The training process involves optimizing the parameters to minimize the reconstruction error.

The Quantum Variational Autoencoder extends this idea into the quantum domain. It incorporates quantum circuits and quantum gates to perform operations on quantum states, introducing quantum principles into the encoding and decoding processes. Quantum states, known for their superposition and entanglement properties, can potentially enhance the representation and processing of information compared to classical methods.

However, the implementation and training of Quantum Variational Autoencoders are intricate tasks. They require a deep understanding of quantum circuits, variational optimization, and the interplay between classical and quantum components. The quantum advantage is context-dependent, and achieving meaningful results often involves dealing with challenges such as quantum noise and error correction.

I have developed a Quantum Variational Autoencoder (Q-VAE) model, which is currently in an incomplete state due to its inherent complexity. I am eager to delve into further research and exploration to enhance and refine this model.

I have attached screenshots of the code below:



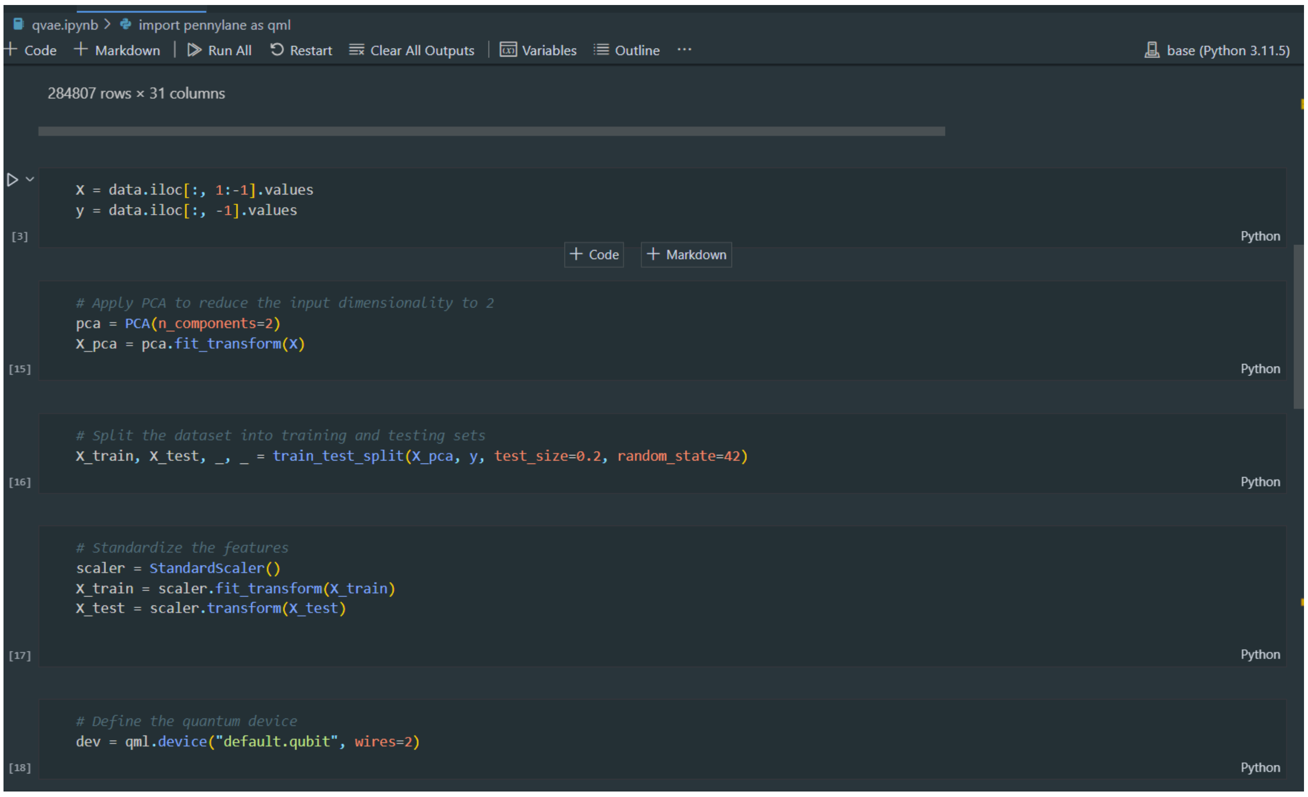
```python
import pennylane as qml
from pennylane.templates import StronglyEntanglingLayers
from pennylane.operation import Tensor
from pennylane import numpy as npenn
import pandas as pd
from sklearn.decomposition import PCA
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Lambda
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import numpy as np
```

[14] Python

```python
# Load the data credit card dataset
data = pd.read_csv('creditcard.csv')
data
```

[2] Python

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V2
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.06692
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.33984
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.68928
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.17557
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.14126
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	...	0.213454	0.111864	1.014480	-0.50934
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	...	0.214205	0.924384	0.012463	-1.01622



284807 rows × 31 columns

```python
X = data.iloc[:, 1:-1].values
y = data.iloc[:, -1].values
```

[3] Python

```python
# Apply PCA to reduce the input dimensionality to 2
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

[15] Python

```python
# Split the dataset into training and testing sets
X_train, X_test, _, _ = train_test_split(X_pca, y, test_size=0.2, random_state=42)
```

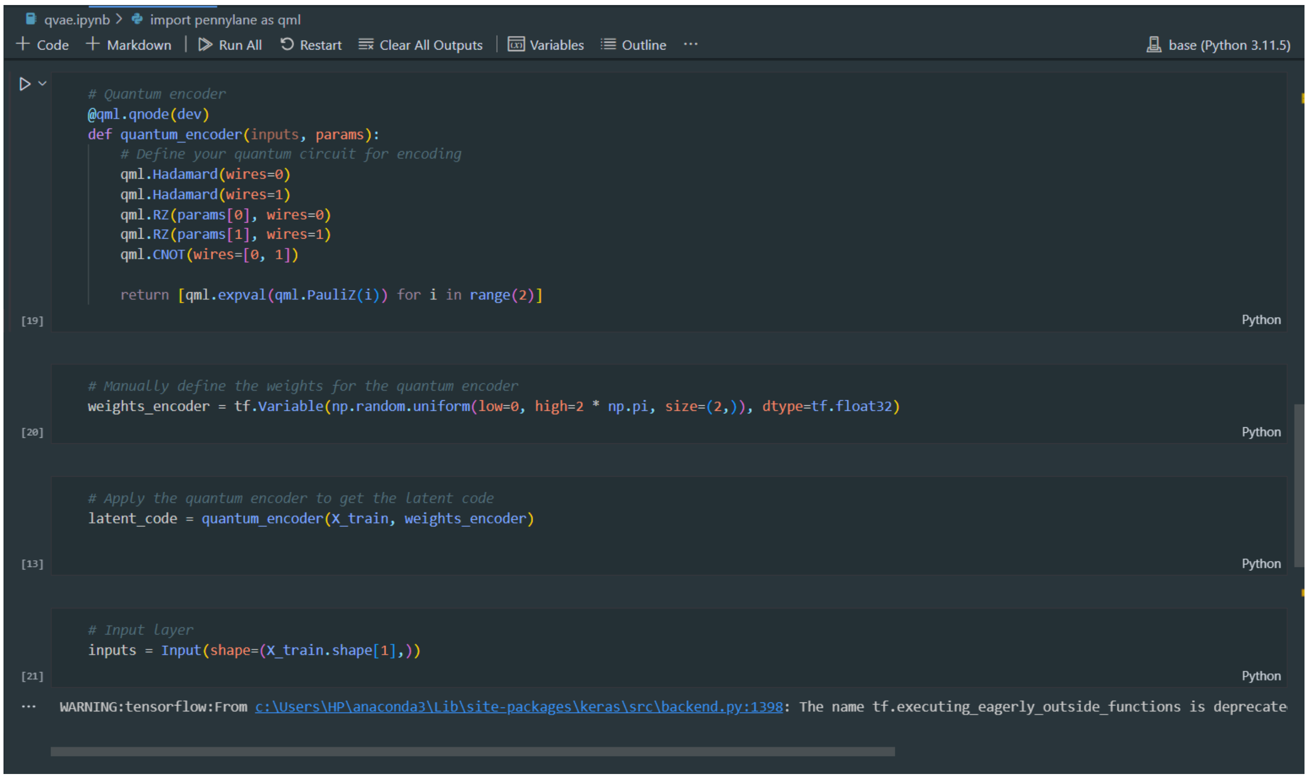
[16] Python

```python
# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

[17] Python

```python
# Define the quantum device
dev = qml.device("default.qubit", wires=2)
```

[18] Python


qvae.ipynb > import pennylane as qml
Code Markdown Run All Restart Clear All Outputs Variables Outline
base (Python 3.11.5)

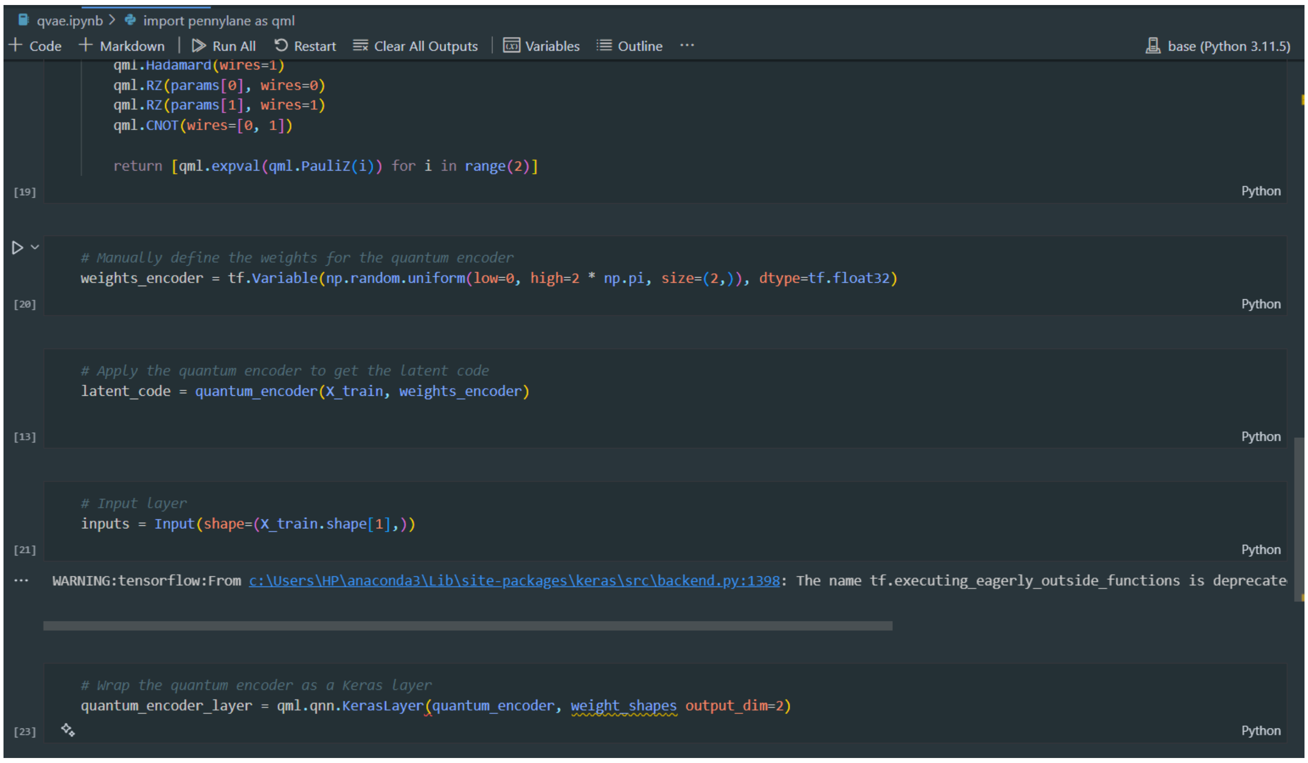
```
# Quantum encoder
@qml.qnode(dev)
def quantum_encoder(inputs, params):
    # Define your quantum circuit for encoding
    qml.Hadamard(wires=0)
    qml.Hadamard(wires=1)
    qml.RZ(params[0], wires=0)
    qml.RZ(params[1], wires=1)
    qml.CNOT(wires=[0, 1])

    return [qml.expval(qml.PauliZ(i)) for i in range(2)]
```
[19] Python

```
# Manually define the weights for the quantum encoder
weights_encoder = tf.Variable(np.random.uniform(low=0, high=2 * np.pi, size=(2,)), dtype=tf.float32)
```
[20] Python

```
# Apply the quantum encoder to get the latent code
latent_code = quantum_encoder(X_train, weights_encoder)
```
[13] Python

```
# Input layer
inputs = Input(shape=(X_train.shape[1],))
```
[21] Python

```
WARNING:tensorflow:From c:\Users\HP\anaconda3\Lib\site-packages\keras\src\backend.py:1398: The name tf.executing_eagerly_outside_functions is deprecate
```


qvae.ipynb > import pennylane as qml
Code Markdown Run All Restart Clear All Outputs Variables Outline
base (Python 3.11.5)

```
    qml.Hadamard(wires=1)
    qml.RZ(params[0], wires=0)
    qml.RZ(params[1], wires=1)
    qml.CNOT(wires=[0, 1])

    return [qml.expval(qml.PauliZ(i)) for i in range(2)]
```
[19] Python

```
# Manually define the weights for the quantum encoder
weights_encoder = tf.Variable(np.random.uniform(low=0, high=2 * np.pi, size=(2,)), dtype=tf.float32)
```
[20] Python

```
# Apply the quantum encoder to get the latent code
latent_code = quantum_encoder(X_train, weights_encoder)
```
[13] Python

```
# Input layer
inputs = Input(shape=(X_train.shape[1],))
```
[21] Python

```
WARNING:tensorflow:From c:\Users\HP\anaconda3\Lib\site-packages\keras\src\backend.py:1398: The name tf.executing_eagerly_outside_functions is deprecate
```

```
# Wrap the quantum encoder as a Keras layer
quantum_encoder_layer = qml.qnn.KerasLayer(quantum_encoder, weight_shapes output_dim=2)
```
[23] Python

After this, I have to develop a quantum_decoder_layer as well.

CONCLUSION

In conclusion, the exploration of Quantum Machine Learning (QML) has taken us through a fascinating journey across the intersection of quantum computing and machine learning. As we delved into the realm of quantum circuits, cost functions, and entangling layers, we witnessed the unique capabilities and challenges that QML introduces to the forefront of computational research.

The foundational understanding of quantum computing, its necessity, and the emerging trends in quantum machine learning laid the groundwork for our exploration. The various domains like quantum machine learning and quantum optimization showcased the diverse applications of quantum computing, promising a paradigm shift in solving complex computational problems.

The primary focus of our investigation revolved around detecting financial frauds using Quantum Machine Learning. The problem statement, centered on a credit card transaction dataset, tasked us with training a model using quantum algorithms to achieve high accuracy. The dataset, with its class imbalance issue, presented a real-world challenge that needed to be addressed in our model development.

Our journey then transitioned to classical machine learning approaches, where we compared the performance of classical Support Vector Machine (SVM) models with their quantum counterparts. Initial results showed remarkable accuracy, but the issue of class imbalance prompted us to further refine our models, ultimately demonstrating the robustness of SVM, even in the quantum realm.

The introduction of quantum circuits paved the way for our quantum machine learning models. We initiated with a basic PennyLane model, gradually progressing through models of increasing complexity. Quantum SVM models utilizing quantum feature mapping and quantum kernels further demonstrated the potential of quantum-enhanced machine learning for fraud detection.

The complexity heightened with Quantum Variational Autoencoder (QVAE) models, showcasing the intricacies of incorporating quantum principles into generative models. While these models may be challenging and incomplete, their existence underscores the vast possibilities and ongoing research avenues in Quantum Machine Learning.

As we conclude this exploration, it is evident that the synergy between classical and quantum models represents a promising frontier. The developments in building quantum subparts within classical models indicate a trajectory of research with substantial potential. The interplay between classical and quantum elements in model design not only enhances computational power but also opens new avenues for innovation and discovery.

In this dynamic landscape, the tapestry of Quantum Machine Learning unfurls with opportunities for further refinement, exploration, and groundbreaking discoveries. The journey may have reached a conclusion, but the horizon of possibilities in Quantum Machine Learning remains expansive and inviting for continued exploration.

References

- https://pennylane.ai/qml/
- https://docs.pennylane.ai/en/stable/code/qml.html
- https://docs.pennylane.ai/en/stable/code/api/pennylane.S.html
- https://qiskit.org/documentation/locale/bn_BN/index.html
- https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud
- https://arxiv.org/abs/1802.05779
- https://qiskit.org/ecosystem/machine-learning/tutorials/12_quantum_autoencoder.html
- https://par.nsf.gov/servlets/purl/10351398
- https://iopscience.iop.org/article/10.1088/2058-9565/aada1f/pdf
- https://qiskit.org/documentation/stable/0.24/tutorials/machine_learning/01_qsvm_classification.html
- https://medium.com/mit-6-s089-intro-to-quantum-computing/quantum-support-vector-machine-qsvm-134eff6c9d3b
- https://github.com/PatrickHuembeli/QSVM-Introduction/blob/master/Quantum%20Support%20Vector%20Machines.ipynb
-