Team Number:	2400232
Problem Chosen:	A

2024 APMCM summary sheet

Classification models are essential in machine learning, particularly in tasks such as **fraud detection** and **image recognition**. However, traditional computational methods exhibit significant limitations when processing high-dimensional and large-scale datasets under constraints of computational resources. This study investigates quantum computing-based approaches to address these challenges, specifically through a **QUBO-based feature selection (QFS)** algorithm and a **QUBO framework** for training quantize convolutional neural networks.

For fraud detection, the **QFS algorithm** was applied to the German Credit dataset comprising 20 features and 1,000 instances. By quantifying feature importance and redundancy, a QUBO model was constructed for feature selection. Experiments using K-Nearest Neighbor **(KNN)**, Naive Bayes **(NB)**, and Support Vector Machine **(SVM)** classifiers demonstrated improvements in **accuracy, precision, and recall** compared to models trained on the full dataset. Additionally, the **adjusted F-value** metric confirmed the effectiveness of QFS in addressing the cost-sensitive nature of fraud detection, presenting a novel method for credit scoring.

For image recognition, the training process of multilayer **feedforward neural networks** was reformulated into a **QUBO problem**, transforming the process into an Ising model. Using the **Kaiwu SDK** for simulated annealing, experiments on the MNIST dataset revealed that the quantum approach enhanced efficiency and accuracy, demonstrating its applicability to deep learning tasks.

These findings highlight the utility of quantum computing in feature selection, fraud detection, and image recognition, underscoring its potential as a transformative tool for overcoming computational limitations in machine learning.

Keywords: QUBO   Quantum Feature Selection   Classification   Discrete   CNN

Contents

I. Introduction

1.1 Problem Context

Regarding Question 1,in contemporary data analysis, machine learning models have demonstrated exceptional performance across various tasks such as classification, regression, and data prediction. However, as input data dimensionality continues to escalate, models' memory and computational power requirements grow exponentially. Consequently, dimensionality reduction becomes crucial in those scenarios. Within machine learning, reduction of dimensionality can effectively reduce model complexity and mitigate overfitting, making precise evaluation of input features' impact on output labels.

Feature selection effectively addresses challenges in machine learning arising from high-dimensional data, including model performance limitations, elevated computational costs, and overfitting problems. By removing redundant and irrelevant features and selecting only a subset of available features, models can focus on critical information while maintaining or even improving predictive accuracy and generalization capabilities. This approach simultaneously reduces computational resource consumption during training, ensuring better adaptability and stability when confronting new data.

Figure 1 Elaboration of Credit Scoring

In the financial domain, credit scoring plays a pivotal role in assessing customer credit risk. Accurate credit scoring models help financial institutions make rational lending decisions and minimize the risk of non-performing loans. However, developing effective credit scoring models presents numerous challenges, with a key issue being the identification of the most valuable feature subset from numerous feature attributes to enhance model accuracy and efficiency.

This paper proposes a feature selection algorithm based on the **Quadratic Unconstrained Binary Optimization** (QUBO) problem for extracting the optimal feature subset from the German credit scoring dataset. The dataset comprises 20 feature attributes, including 13 categorical and 7 numerical attributes, with 1,000 instances. The model's objective is to determine weight parameters β and extract the most valuable feature subset for credit score prediction.

Regarding Question 2, in today's technological field, computer vision is at a rapid development stage. With the increasing demand for image classification applications, people expect higher accuracy in image classification to ensure the precision and reliability of various applications. Deep learning models, based on deep neural network architectures, achieve outstanding accuracy in image classification tasks through extensive training on large datasets. Classic convolutional neural networks (CNNs) and their numerous variants have shown excellent performance in practical applications of image classification. However, as models become larger and more complex, the computational resources required grow exponentially.

Quantum computing is a rapidly emerging and highly promising area of computation technology, using qubits as basic information units to construct a computational paradigm fundamentally different from traditional classical computing. It has demonstrated significant speed advantages in handling certain complex combinatorial optimization problems. This paper employs a multi-layer feedforward neural network training algorithm suitable for quantum computers, mapping the training process of neural networks onto the self-evolving system model of quantum computers, achieving the transformation from Convolutional Neural Network (CNN) to QUBO model, ultimately decoding the optimal network parameters.

The paper uses the MNIST handwritten digit dataset for experiments, transforming the training of neural networks into a QUBO model, and employing the simulated annealing algorithm of Kaiwu SDK to solve this problem.

1.2 Problem Reformulation

For the first problem, to extract the most valuable feature subset for credit scoring prediction, we separately set the mutual information between feature pairs and between feature attributes and label values to represent their correlation, and their linear combination can be expressed as the objective function. The objective function can be understood as a function for measuring the value of a feature subset. Therefore, the problem can be transformed into: How to obtain the optimal model weight parameters β and the optimal feature subset by minimizing the objective function.

For Problem 2, to address the issue of high computational requirements in existing models for image classification, we combine the training of multilayer feedforward neural networks with quantum computers by encoding variables in binary form and representing constraints, transforming the neural network CNN into QCBO model, and further into QUBO model. We use the MNIST handwritten digits dataset to verify the model.

II. Model and Algorithm of Problem 1

2.1 Construction of the QUBO Model for Feature Selection

Assume that it is necessary to classify the dataset $D = \{(x^i, y^i)\}_{i\in[N]}$, where for $i \in [N]$, the n-dimensional feature value $x^i \in X \subseteq \mathbb{R}^n$, and the label $y^i \in Y \subseteq \mathbb{N}$. Here, $[N]$ denotes the set $\{1, \ldots, n\}$. The FS problem corresponds to finding a subset $S \subset [n]$ of these n-dimensional features such that $D_S = \{(x^i_S, y^i)\}_{i\in[N]}$ and $x^i_S = (x^i_j)_{j\in S}$ are combined.

The proposed formula is:

$$x^* = \arg \min_{x\in\{0,1\}^n} Q(x, \beta)$$

where x is represented as a binary indicator vector $x = (x_1, \ldots, x_n) \in \{0, 1\}^n$.

In the objective function, $\beta \in [0, 1]$, we define:

1. $I_i = I(x_i; y) \geq 0$ as the importance item, representing the mutual information between a single feature data point x_i and the class label y, measuring the importance of each feature. In the objective function, the importance item is set to be maximized;
2. $R_{ij} = I(x_i; x_j) \geq 0$ as the redundancy item, representing the mutual information between feature pairs, measuring the correlation between each feature pair. In the objective function, the redundancy item is set to be minimized.

Formally,the equation represents the feature selection problem discussed in this paper, and we refer to our method as QUBO Feature Selection, abbreviated as QFS. The QUBO objective function is typically written in quadratic form:

$$Q(x, \beta) = x^T Q(\beta) x$$

The elements of this matrix are given by:

$$Q_{ij}(\beta) = R_{ij} - \beta(R_{ij} + \delta_{ij} I_i)$$

Here, $\delta_{ij} := 1_{\{i=j\}}$ denotes the Kronecker delta function, used to transform I into a diagonal matrix. The solution of the equation represents the optimal feature subset. Based on the model framework proposed by us, the implementation flow of QFS is shown in Figure 1.

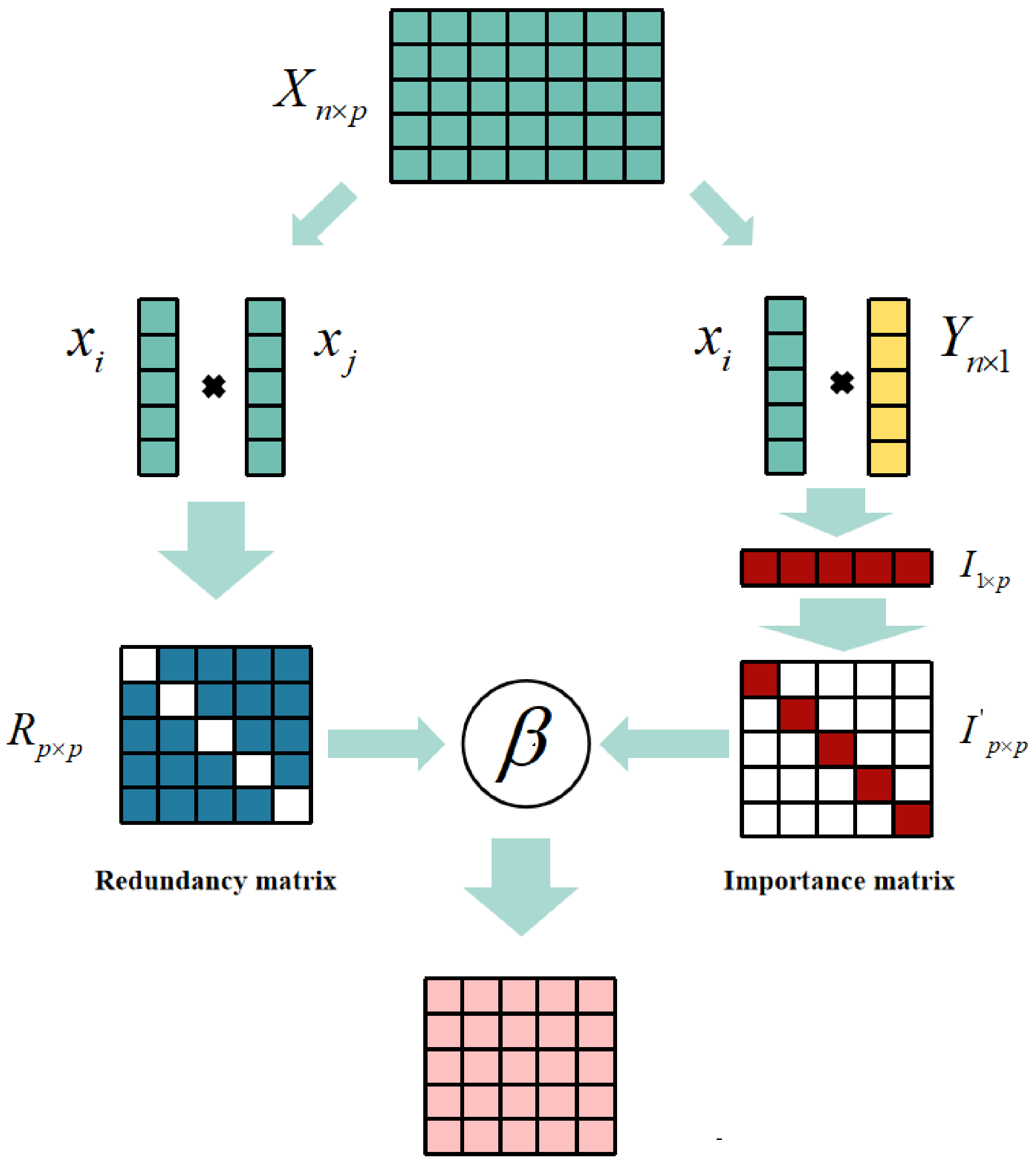


Figure 2 Flowchart for the construction of the QUBO matrix

2.2 Calculation of Mutual Information

For two random variables, mutual information (mutual information, MI) is the amount of information that one random variable reduces due to knowing another random variable. The calculation of mutual information usually requires estimating the joint distribution and marginal distributions of random variables, while the distribution of real-valued data involves probability density functions, integral operations, and complex parameter estimation issues, leading to difficulties in estimation. Therefore, we perform dispersion processing on the data set D to simplify calculations.

2.2.1 *Determine the number of bins and bin boundaries*

We first set the number of bins B to control the degree of dispersion. For each feature dimension i in the data set, calculate $\frac{l}{B+1}$ bits and record them as q_i^l, where $l \in \{0, ..., B\}$. Then, for $l \in [B-1]$, the bin B_i^l is defined as the interval $[q_i^{l-1}, q_i^l]$, and the last bin $B_i^B = [q_i^{B-1}, q_i^B]$.

2.2.2 *Data dispersion processing*

For each data point $x_j^i (j \in [N])$ in the data set, based on its value at each feature dimension i, determine the corresponding bin. If $x_j^i \in B_i^l$, let $b_j^i = l$, This achieves the transformation from the original real-valued feature into a discretized form.

The data x_i^j has been transformed into a discretized bin number b_i^j, resulting in a discretized dataset $D = \left\{(b_i^j, y_i^j)\right\}_{i \in [N]}$, where $b_i^j \in [B]$.

2.2.3 *Calculate Mutual Information*

After achieving the discretization of the dataset, we obtain the probability mass function of the discretized data:

$$\hat{p}(b, y) := \frac{1}{N} \sum_{(b', y') \in D} 1_{\{b=b' \wedge y=y'\}}$$

And the indicator function:

$$1_P := \begin{cases} 1 & \text{if } P \text{ evaluates to true} \\ 0 & \text{otherwise} \end{cases}$$

Therefore, the mutual information can be approximated as follows:

$$I(x_i, y) \approx \sum_{b \in [B]} \sum_{y \in Y} \hat{p}_{X_i,Y}(b, y) \log \left(\frac{\hat{p}_{X_i,Y}(b, y)}{\hat{p}_{X_i}(b) \hat{p}_Y(y)} \right)$$

$$I(x_i, x_j) \approx \sum_{b \in [B]} \sum_{b' \in [B]} \hat{p}_{X_i,X_j}(b, b') \log \left(\frac{\hat{p}_{X_i,X_j}(b, b')}{\hat{p}_{X_i}(b) \hat{p}_{X_j}(b')} \right)$$

Among them, the marginal distributions of the feature subset and label are as follows:

$$\hat{p}_{X_i X_j}(b_i, b_j) := \sum_{y \in Y, b_k \in B | k \neq i,j} \hat{p}(b, y)$$

$$\hat{p}_{X_iY}(b_i, y) := \sum_{b_k \in B | k \neq i} \hat{p}(b, y)$$

$$\hat{p}_{X_i}(b_i) := \sum_{y \in Y, b_k \in B | k \neq i} \hat{p}(b, y)$$

$$\hat{p}_{Y}(y) := \sum_{b_k \in B | k} \hat{p}(b, y)$$

Discretization can approximate mutual information calculation, while avoiding the assumption of real-valued data distribution, greatly simplifying the estimation process.

For Feature Selection (FS), it is known from existing literature that: For all $Q_k^*(\cdot, \beta)$ defined by the equation and $k \in \{0, 1, ..., n\}$, there must exist an $\beta \in [0, 1]$ such that $x^* \in \arg\min_x Q(x, \beta)$ and $\|x^*\|_1 = k$.

This means that by adjusting β, any desired number of features k can be achieved, demonstrating the inevitability of model solutions.

Based on this conclusion, this paper designs the QFS algorithm. When provided with the importance vector I of the residual matrix R and the expected number of selected features k, finding a suitable β^* so that solving the feature selection problem constructed based on β^* yields the optimal feature vector $x \in Q^*(\beta^*)$ with exactly k non-zero entries. To do this, the paper introduces:

$$Q_k^*(\beta) := \min_{x \in \{0,1\}^n} Q(x, \beta) \quad s.t. \|x^*\|_1 = k$$

where $0 \leq k \leq n$.

III. Model and Algorithm of Problem 2

Most researches uses the mean squared error (MSE) loss function to construct supervised learning tasks for Convolutional neural network (CNN). By using the constraint representation of network topology and the binary representation of variables, this paper transforms part of the training section of CNN into a quadratic constrained binary optimization (QCBO) problem, then introduces penalty functions to eliminate the constraints in QCBO, obtaining the QUBO model, which can be solved through quantum solvers.

3.1 Training of CNN

Assume that CNN is a multilayer neural network with quantization parameters θ. The training problem is:

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} (y_i - f(x_i))^2$$

where N is the size of the data set D, (x_i, y_i) is the ith data sample, and the prediction output of the network is:

$$f(x) = f^{(L)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(x)$$

where $f^{(k)}(x) = g(W^{(k)}x+b^{(k)})$ is the activation function of the kth layer, $\circ$ denotes the composition operation, $W^{(k)}$ are weight parameters, $b^{(k)}$ are bias parameters, and g is a nonlinear activation function such as Sigmoid, ReLU, etc.

In QNN, we define the loss function as:

$$L_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \tilde{y}_i)^2$$

where $\tilde{y}_i$ represents the predicted value.

The quantized neural network consists of linear layers, convolutional layers, pooling layers, and normalization layers, followed by an activation function for output. We represent its structure as follows:

For the linear layer transformation, we have:

$$W^{(k)} a^{(k-1)} + b^{(k)} = s^{(k)}$$

For the convolutional layer change, we have:

$$x^{(k)} * a^{(k-1)} = s^{(k)}$$

where $*$ denotes convolution. For the MNIST image recognition task, we set up a 3x3 2D convolution kernel, which can be represented as:

$$\sum_{r=0}^{2}\sum_{c=0}^{2} x^{(k)}(i,j)a^{(k-1)(r+i,c+j)} = s^{(k)}(r,c), \forall r,c$$

r and c are the row and column indices of the two-dimensional feature map.

We establish a filter of size 3x3 for the average pooling layer to perform two-dimensional average pooling operation, whose equivalent form is given by:

$$\frac{1}{9}\sum_{i=0}^{2}\sum_{j=0}^{2} a^{(k)}(r+i,c+j) = a^{(k)}_{after-pooling}(r,c), \quad \forall r,c$$

For the maximum pooling layer, the equivalent form is given by:

$$\sum_{i=0}^{2}\sum_{j=0}^{2} \chi^{(k)}(r,c,i,j) = a^{(k)}_{after-pooling}(r,c), \quad \forall r,c$$

$$\sum_{i=0}^{2}\sum_{j=0}^{2} \chi^{(k)}(r,c,i,j) = 1, \quad \forall r,c$$

$$\chi^{(k)}(r,c,i,j)a^{(k)}(r+i,c+j)+(1-\chi^{(k)}(r,c,i,j))M_1-a^{(k)}(r+i,c+j)M_2 = 0\forall r,c \quad \forall i,j,i',j' \in \{0,1,2\}$$

$$\chi^{(k)}(r,c,i,j) \in \{0,1\}, \quad \forall r,c \quad \forall i,j \in \{0,1,2\}$$

Where M_1 and M_2 are sufficiently large positive values.

Each neuron in CNN contains two operations: linear transformation and activation function.

For the linear transformation, its form is:

$$W^{(k)}a^{(k-1)} + b^{(k)} = s^{(k)}$$

The activation function used is:

$$\text{sign}(x) = \begin{cases} +1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$

Due to the complex structure of CNN, it is not suitable for optimization on classical computers. To address this issue, we transform the CNN problem into QCBO through two steps.

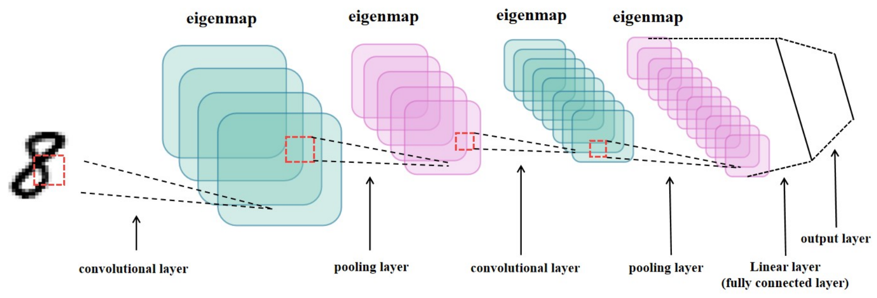


Figure 3 Convolutional Network Ntructure Diagram

3.2 Transforming CNN into QCBO

This section describes how to express the training of CNN as a combinatorial optimization problem involved in QCBO.

We convert the equation into:

$$a^{(k)} \odot s^{(k)} = r^{(k)}$$

$$a^{(k)} + 2r^{(k)} \geq 1$$

where $a^{(k)} \in \{-1, +1\}$, $s^{(k)} \in \mathbb{Z}^H$, $\odot$ denotes matrix dot product, ensuring the same sign between a and s.

The equation ensures that when $s = 0$, $a = \pm 1$. Subsequently, by introducing auxiliary variables, the inequality constraint is transformed into an equality constraint:

$$a^{(k)} + 2r^{(k)} = 1 + t^{(k)}$$

For the linear transformation constraint , we transform the linear transformation constraint of layer k into:

$$W_i^{(k)} a_i^{(k-1)} + b^{(k)} = s_i^{(k)}, \forall i$$

The decision variable set is $\{W^{(k)}, b^{(k)}, s_i^{(k)}, r_i^{(k)}, t_i^{(k)}, a_i^{(k)}, y_i | k = 1, 2, ..., L$ and $i = 1, 2, ..., N\}$.

We need to optimize these decision variables under the above constraints to minimize the mean square error.

To enable computation on quantum computers, we encode the decision variables based on binary coding agreements, thereby establishing a connection with Ising models.

According to the type and range of variables, we determine the coding rules. For example, in the first-layer neural network, we need to satisfy the binarization of all variables in the linear transformation constraint of the neural network. For the bias term $b^{(1)}$, we set it as $b^{(1)} = \sum_j 2^j \sigma_b^{(1)}(j)$, where $\sigma_b^{(1)}(j) \in \{0, 1\}$, σ represents the collection of all binary variables used in the coding, H is the binary variable, through multiplying each binary position by the corresponding power of 2 and summing them up to represent the value of the bias term. Similarly, we also encode $W^{(1)}$, $b^{(1)}$, and $s_i^{(1)}$ as binary variables (this can be omitted in the appendix if not needed). For feature values x_i, we quantize them into integers, where $x_i \in X^n$ and $X^n = [-2^B, 2^B]$, B represents the bit width of the input.

Thus, we obtain the QCBO form of the QNN training problem.

3.3 Conversion of QCBO to QUBO

If we need to perform calculations on a quantum computer, we need to remove the constraints in QUBO and convert it into an unconstrained QUBO format. Therefore, we need to introduce penalty functions to eliminate all constraints and use the Rosenberg polynomial to reduce the degree of the loss function to two.

After using penalty functions to eliminate constraints, we obtain the new loss function:

$$
\begin{aligned}
L_{\text{penalty}} &= \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y}_i)^2 \\
&+ \lambda_1 \sum_{i=1}^{N} \|W^{(1)}x_i + b^{(1)} - s_i^{(1)}\|^2 \\
&+ \lambda_2 \sum_{i=1}^{N}\sum_{k=2}^{L-1} \|W^{(k)}a_i^{(k-1)} + b^{(k)} - s_i^{(k)}\|^2 \\
&+ \lambda_3 \sum_{i=1}^{N} \|W^{(L)}a_i^{(L-1)} + b^{(L)} - y_i\|^2 \\
&+ \lambda_4 \sum_{i=1}^{N}\sum_{k=1}^{L-1} \|a_i^{(k)} \odot s_i^{(k)} - r_i^{(k)}\|^2 \\
&+ \lambda_5 \sum_{i=1}^{N}\sum_{k=1}^{L-1} \|a_i^{(k)} + 2r_i^{(k)} - 1 - t_i^{(k)}\|^2.
\end{aligned}
\tag{1}
$$

The second to fourth lines are linear transformation penalty functions, where λ is the penalty coefficient, thus forming a high-order unconstrained binary optimization problem. Then, by introducing auxiliary variables through the Rosenberg polynomial, the high-order binary optimization problem is gradually transformed into a quadratic binary optimization problem. Any high-order binary optimization problem can be solved by selecting the Rosenberg reduction method, replacing the product of two binary variables with one auxiliary binary variable, constructing the Rosenberg polynomial and adding it to the loss function, thereby transforming it into a quadratic binary optimization problem. Finally, we obtain the mathematical form of the QUBO problem.

IV. QUBO Model Setting

4.1 Question 1: QUBO solution

The components of the QFS algorithm proposed in this paper are solutions to the QUBO model:

$$\min_{x\in\{0,1\}^n} x^{\mathrm{T}} Q(\beta) x$$

This paper converts the established QUBO model into an Ising model and solves it locally in Python using the simulated annealing solver provided by Kaiwu. The weight parameter and the optimal feature subset were obtained as follows.

Simulated Annealing Optimizer Parameters:

- Initial temperature: 1000
- Cooling rate (): 0.99
- Cutoff temperature: 0.0001
- Iterations per temperature: 100

Solving time on CPQC-1 (s):2.199ms

4.2 Question 2: QUBO solution

Simulated Annealing Optimizer Parameters:

- Initial temperature: 1000
- Cooling rate (): 0.99
- Cutoff temperature: 0.0001
- Iterations per temperature: 100

Solving time on CPQC-1 (s):3.154ms

V. Experiment and Analysis

5 . 1 Experiment of problem 1

5 . 1 . 1 *Experimental methods and evaluation indicators*

Under the research framework outlined in Question 1, a series of rigorous experimental procedures were implemented to comprehensively and thoroughly evaluate the effectiveness of the proposed method. Specifically, the optimal feature subsets identified using the QFS algorithm were input into three models: k-Nearest Neighbors (KNN), Naïve Bayes (NB), and Support Vector Machines (SVM). For each model, metrics such as accuracy, precision, recall, and Adjusted-F score were calculated based on predictions using the optimal feature subsets. For comparison, the same metrics were also computed using the original feature dataset with the aforementioned models.

It is important to note that when assessing the performance of the models, we not only considered their mathematical significance and theoretical value but also incorporated practical implications relevant to the specific problem context. For this study, the German Credit Scoring dataset from the UCI Machine Learning Repository was utilized. According to its documentation, in the context of credit scoring, misclassifying customers with bad credit as good credit has five times the cost of misclassifying customers with good credit as bad. Consequently, the Adjusted-F score was chosen as the fourth evaluation metric. This score is a weighted harmonic mean of precision and recall, and its calculation formula is as follows:

$$F(\beta) - Score = \frac{(\beta^2 + 1) \times (precision \times recall)}{\beta^2 \times (precision + recall)}$$

We set the weight ($\beta = \sqrt{5}$) to adjust the balance between precision and recall. The complete set of evaluation metrics is outlined as follows:

Table 1 Explanation of the indicators

Evaluation Indicator	Formula
$Accuracy$	$\frac{TP+TN}{TP+TN+FP+FN}$
$Precision$	$\frac{TP}{TP+FP}$
$Recall$	$\frac{TP}{TP+FN}$
$AdjustedF - Value$	$\frac{(\beta^2+1)\times(precision\times recall)}{\beta^2\times(precision+recall)}$

5.1.2 *Experiment Result*

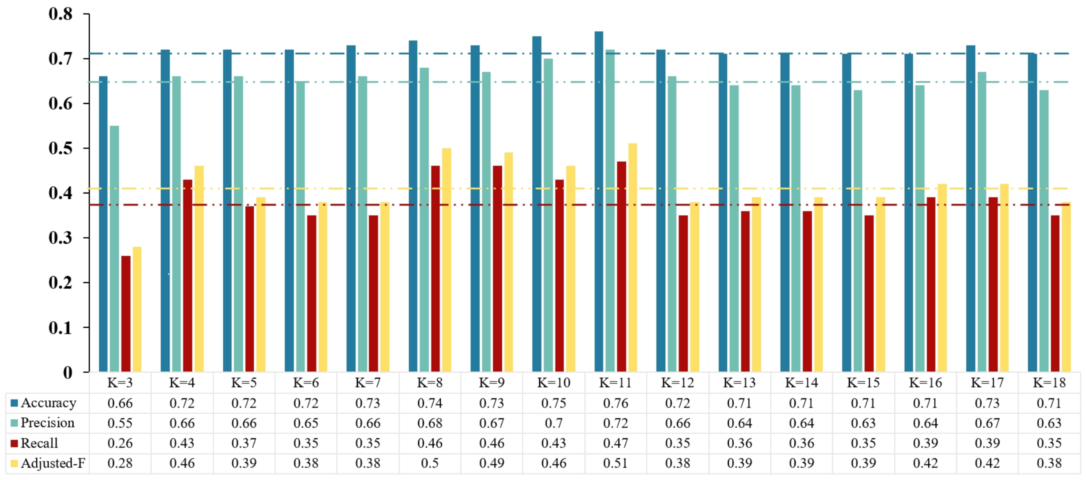
In the experiment, three models were utilized: the k-Nearest Neighbors (KNN) model, the Naïve Bayes (NB) classifier, and the Support Vector Machine (SVM). Each model was evaluated using both the original feature dataset and a subset of selected features. The evaluation metrics included Accuracy, Precision, Recall, and the Adjusted F-Value.

Table 2 Explanation of the indicators

$Model$	Accuracy	$Precision$	$Recall$	$Adjusted-F-Value$
KNN	0.72	0.65	0.38	0.41
KNN^*	0.76(+5.5%)	0.72(+10.7%)	0.47(+23.7%)	0.51(+24.4%)
NB	0.72	0.53	0.62	0.61
NB^*	0.76(+5.5%)	0.59(+11.3%)	0.62(+0.0%)	0.62(+1.6%)
SVM	0.74	0.68	0.45	0.47
SVM^*	0.75(+1.4%)	0.69(+1.5%)	0.51(+13.3%)	0.54(+14.9%)

Overall, the performance of all models showed partial improvement after feature selection. However, the extent of improvement varied among different classification models depending on the number of selected features. Notably, feature selection had a significant positive impact on the performance of the KNN and SVM models, highlighting the critical role of quantum computing in feature engineering. Furthermore, the enhancement in Recall was particularly pronounced, demonstrating substantial potential for applications in fraud detection and intrusion detection domains.

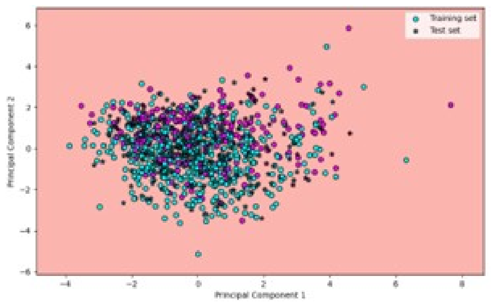
When feature extraction is applied to the KNN model for classification, we observe that, except when the number of extracted variables K is particularly small or particularly large, the improvement across all performance metrics is significant. This indicates that our model effectively eliminates redundant information among numerous variables through quantum computing, thereby enhancing classification accuracy.



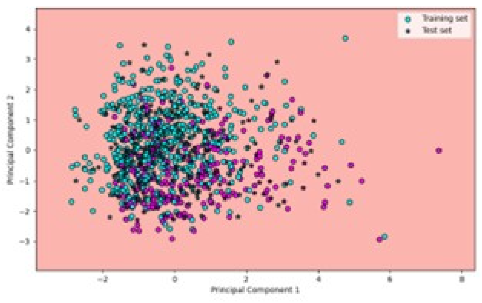
	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10	K=11	K=12	K=13	K=14	K=15	K=16	K=17	K=18
🟦 Accuracy	0.66	0.72	0.72	0.72	0.73	0.74	0.73	0.75	0.76	0.72	0.71	0.71	0.71	0.71	0.73	0.71
🟩 Precision	0.55	0.66	0.66	0.65	0.66	0.68	0.67	0.7	0.72	0.66	0.64	0.64	0.63	0.64	0.67	0.63
🟥 Recall	0.26	0.43	0.37	0.35	0.35	0.46	0.46	0.43	0.47	0.35	0.36	0.36	0.35	0.39	0.39	0.35
🟨 Adjusted-F	0.28	0.46	0.39	0.38	0.38	0.5	0.49	0.46	0.51	0.38	0.39	0.39	0.39	0.42	0.42	0.38

Figure 4 Results of classification task done by KKN.

By analyzing the scatter plots, confusion matrices, and precision-recall ($P - R$) curves before and after feature extraction, several key improvements can be observed. The scatter plots demonstrate that the data points are more distinctly clustered, indicating that the key features become more separable after extraction. This enhanced separability reduces overlap between classes, thereby minimizing classification ambiguity. Correspondingly, the confusion matrices show a notable decrease in misclassifications, with more predictions aligning correctly with true labels. Furthermore, the P-R curves illustrate an increase in precision and recall, signifying a higher probability of correct predictions and reduced false positives. Collectively, these results confirm that feature extraction significantly enhances the model's predictive accuracy and overall performance.

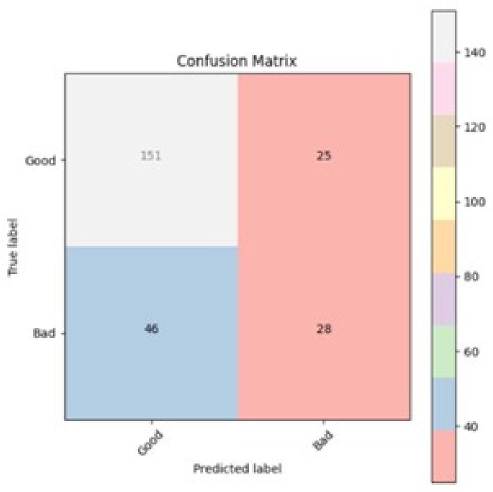


((a)) Before

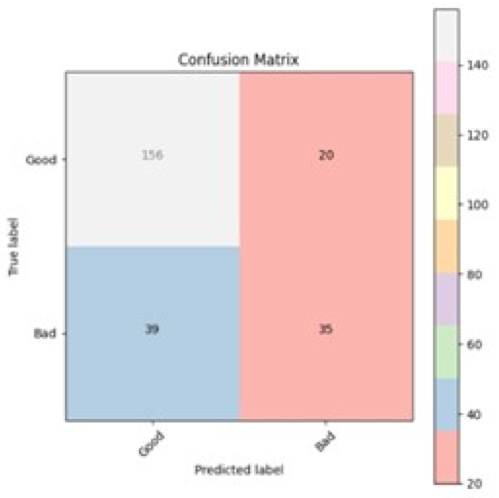


((b)) After

Figure 5 Scatter plot of data before and after feature selection

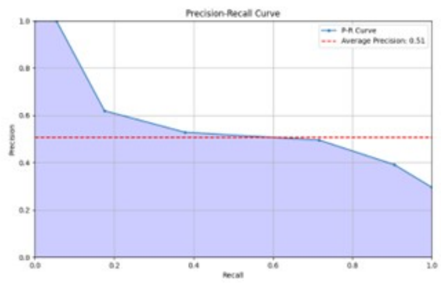


((a)) Before

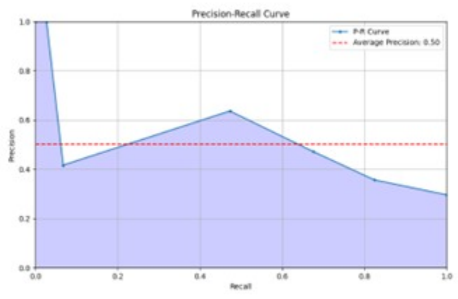


((b)) After

Figure 6 Confusion Matrix before and after feature selection



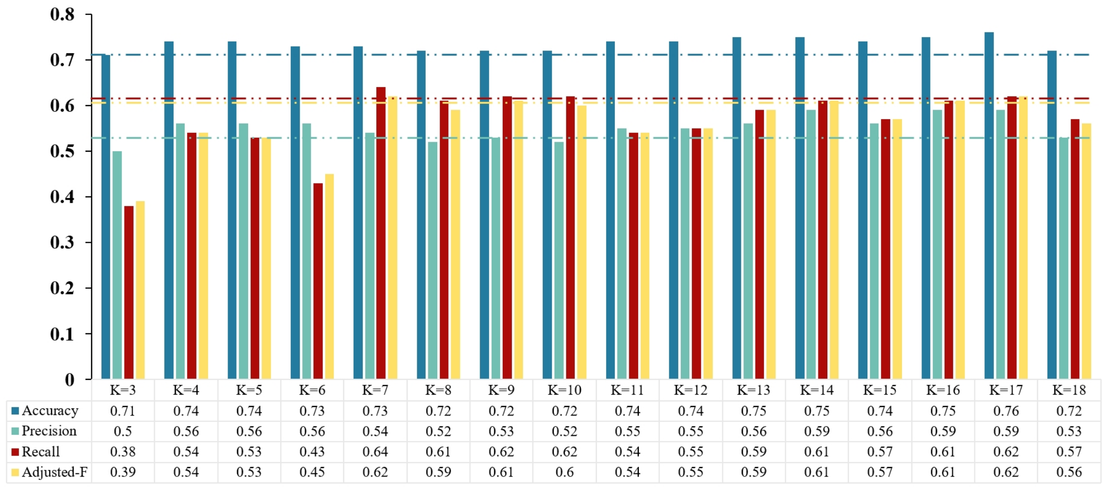
((a)) Before



((b)) After

Figure 7 P-R curve before and after feature selection

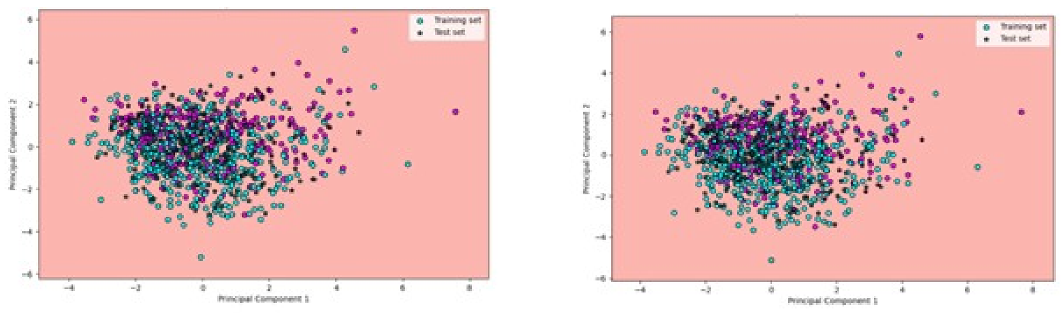
In the Naive Bayes classifier, regardless of the number of extracted features, feature extraction does not lead to a decrease in the model's prediction accuracy. Moreover, a significant improvement in prediction accuracy is observed when the value of (K) is relatively large. At the same time, the enhancement in Precision is particularly notable. Although the improvement in Recall is relatively modest, the overall contribution to the model's accuracy remains significant in the Naive Bayes classification model.



	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10	K=11	K=12	K=13	K=14	K=15	K=16	K=17	K=18
🟦 Accuracy	0.71	0.74	0.74	0.73	0.73	0.72	0.72	0.72	0.74	0.74	0.75	0.75	0.74	0.75	0.76	0.72
🟩 Precision	0.5	0.56	0.56	0.56	0.54	0.52	0.53	0.52	0.55	0.55	0.56	0.59	0.56	0.59	0.59	0.53
🟥 Recall	0.38	0.54	0.53	0.43	0.64	0.61	0.62	0.62	0.54	0.55	0.59	0.61	0.57	0.61	0.62	0.57
🟨 Adjusted-F	0.39	0.54	0.53	0.45	0.62	0.59	0.61	0.6	0.54	0.55	0.59	0.61	0.57	0.61	0.62	0.56

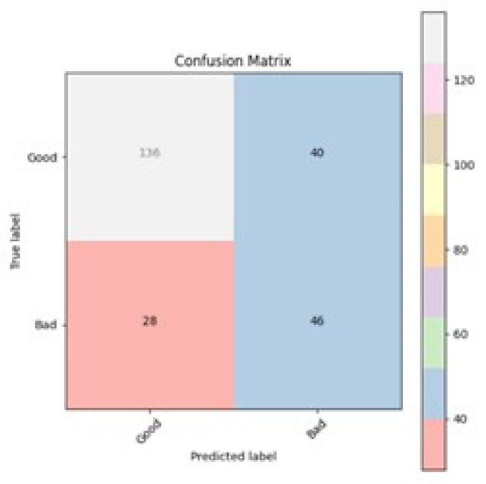
Figure 8 Results of classification task done by naive bayes.

By comparing scatter plots, confusion matrices, and precision-recall $(P-R)$ curves before and after feature extraction, it can be observed that classification accuracy has improved significantly. The data becomes more distinguishable, highlighting a clearer separation between classes. Additionally, the prediction correctness has shown a remarkable enhancement. These results demonstrate the effectiveness of feature extraction in optimizing classification performance.The following graph demonstrate the result of our experiment.

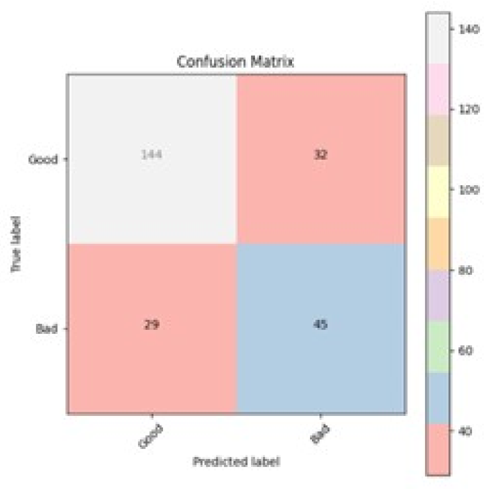


((a)) Before **((b)) After**

Figure 9 Scatter plot of data before and after feature selection

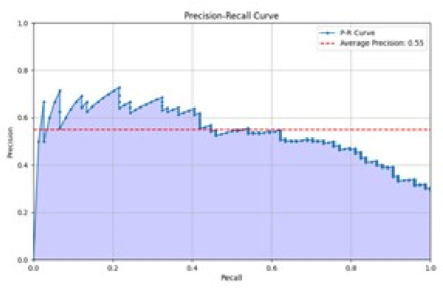


((a)) Before

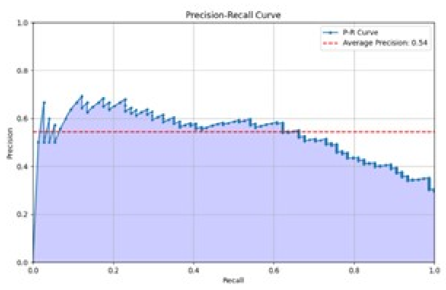


((b)) After

Figure 10 Confusion matrix before and after feature selection



((a)) Before



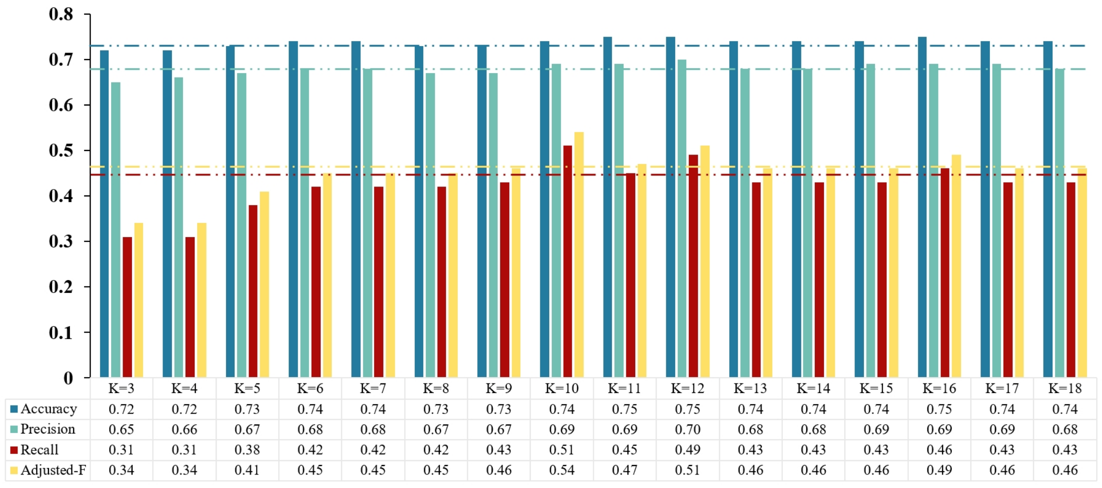
((b)) After

Figure 11 P-R curve before and after feature selection

In the context of support vector machines (SVM), the process of feature extraction often plays a crucial role in shaping the model's performance metrics, particularly when the number of extracted features falls within a moderate range. Under these circumstances, it has been observed that while feature extraction may lead to a reduction in the overall prediction accuracy of the model, it simultaneously contributes to a significant improvement in the recall rate. This trade-off can be attributed to the model's increased capacity to identify true positive instances, even if it occasionally compromises on precision.

Notably, this shift in performance dynamics also reflects in the Adjusted-P value of the model, which demonstrates a marked improvement following feature extraction. The Adjusted-P value, often used as a refined indicator of model fit, captures the balance between explanatory power and parsimony. Its improvement suggests that the feature extraction process is enhancing the model's ability to generalize while mitigating overfitting risks.

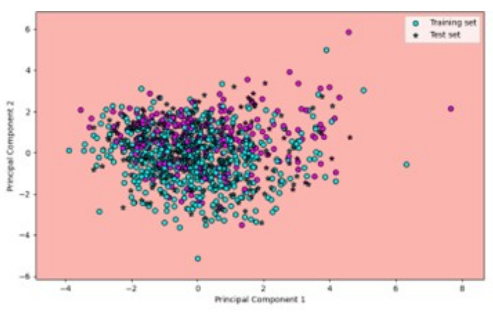
Such changes are particularly relevant in specialized domains where enhanced recall is prioritized, for instance, in medical diagnostics, fraud detection, or rare event classification. In these fields, the enhancement of recall and the subsequent improvement in the Adjusted-P value can lead to a substantial increase in the model's practical utility. This improvement underscores the value of feature extraction not only in refining model interpretability but also in tailoring predictive performance to domain-specific requirements, thereby demonstrating its pivotal role in advancing SVM applications.



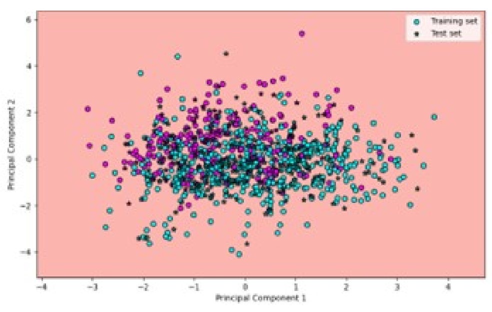
	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10	K=11	K=12	K=13	K=14	K=15	K=16	K=17	K=18
Accuracy	0.72	0.72	0.73	0.74	0.74	0.73	0.73	0.74	0.75	0.75	0.74	0.74	0.74	0.75	0.74	0.74
Precision	0.65	0.66	0.67	0.68	0.68	0.67	0.67	0.69	0.69	0.70	0.68	0.68	0.69	0.69	0.69	0.68
Recall	0.31	0.31	0.38	0.42	0.42	0.42	0.43	0.51	0.45	0.49	0.43	0.43	0.43	0.46	0.43	0.43
Adjusted-F	0.34	0.34	0.41	0.45	0.45	0.45	0.46	0.54	0.47	0.51	0.46	0.46	0.46	0.49	0.46	0.46

Figure 12 Results of classification task done by SVM.

By comparing scatter plots, confusion matrices, and precision-recall (P-R) curves before and after feature extraction, we observe that the data features become more distinct, leading to a significant improvement in prediction accuracy.

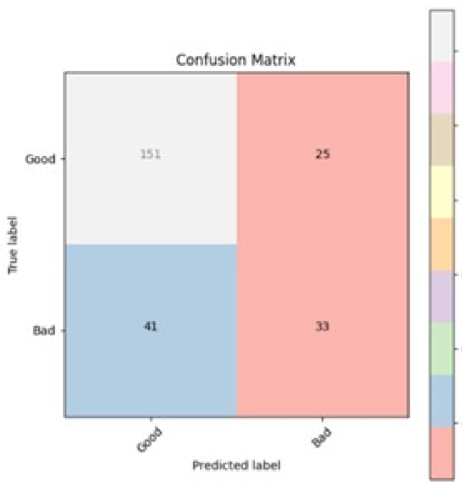


((a)) Before

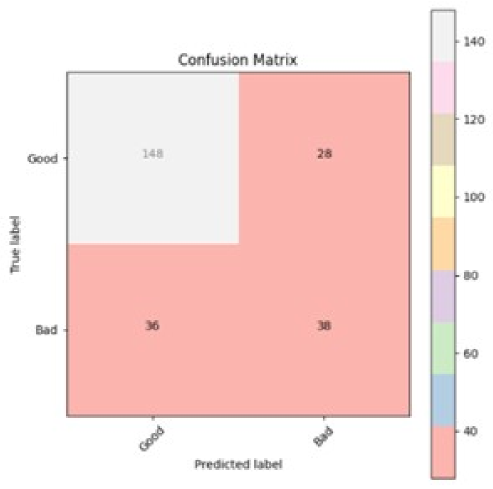


((b)) After

Figure 13 Scatter plot of data before and after feature selection

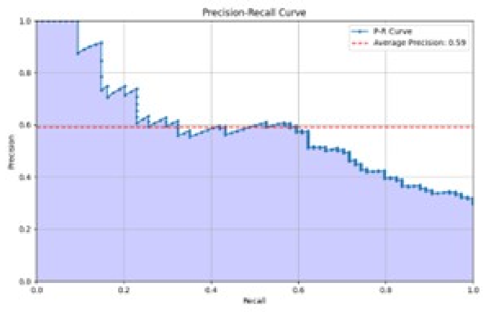


((a)) Before

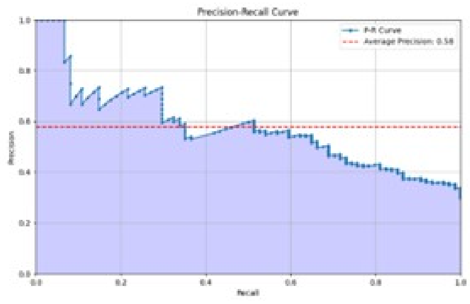


((b)) After

Figure 14 Confusion matrix of data before and after feature selection



((a)) Before



((b)) After

Figure 15 Confusion matrix of data before and after feature selection

VI. Conclusion and Future Work

This study, through a series of rigorous experiments, conducted a comparative analysis of quantum computing-based methods (such as the QFS algorithm) and traditional machine learning models (including KNN, NB, and SVM) in the context of credit evaluation in the financial domain. The findings provide significant insights, summarized as follows:

6 . 1 The Superiority of Quantum Computing in Feature Selection

Feature selection performed using the QFS algorithm significantly enhances the performance of traditional machine learning models. By comparing models using the original feature dataset with those employing the optimal feature subsets selected by the QFS algorithm, it was observed that KNN and SVM models achieved marked improvements in key evaluation metrics such as accuracy, precision, recall, and the adjusted F-score. This indicates that the QFS algorithm is adept at identifying features critical to credit evaluation while eliminating redundant or noisy features, thereby optimizing the predictive capabilities of the models.

Compared to traditional feature selection methods, the QFS algorithm demonstrates unique advantages. It efficiently searches complex, high-dimensional feature spaces and rapidly identifies optimal feature subsets. This capability is particularly valuable when handling financial credit data, which often involves a large number of features with intricate interrelationships. Traditional methods frequently succumb to the challenges of the ”curse of dimensionality,” resulting in diminished model performance, whereas quantum computing-based algorithms effectively overcome this limitation.

6 . 2 The Importance of Quantum Computing in Credit Evaluation

Accurate assessment of customer credit risk is crucial in the financial sector. The performance improvements exhibited by the integration of the QFS algorithm with traditional machine learning models suggest that financial institutions can make more precise judgments about customers' creditworthiness, reducing the occurrence of misjudgments. For instance, in scenarios where customers with poor credit are incorrectly classified as having good credit (a high-cost error) or customers with good credit are erroneously deemed as having poor credit (a relatively lower-cost error), models aided by quantum computing in feature selection achieve a better balance between these two types of risks, enabling more rational credit decision-making.

In summary, quantum computing demonstrates significant advantages and critical application value in the domain of credit evaluation within the financial sector. In the future, as quantum computing technology becomes more deeply integrated with financial practices, it is expected to drive further innovation and breakthroughs, addressing complex challenges that traditional methods struggle to resolve.

VII. References

[1] Nembrini R, Ferrari Dacrema M, Cremonesi P. Feature selection for recommender systems with quantum computing[J]. Entropy, 2021, 23(8): 970.

[2] Mücke S, Heese R, Müller S, et al. Feature selection on quantum computers[J]. Quantum Machine Intelligence, 2023, 5(1): 11.

[3] Chakraborty S, Shaikh S H, Chakrabarti A, et al. A hybrid quantum feature selection algorithm using a quantum inspired graph theoretic approach[J]. Applied Intelligence, 2020, 50(6): 1775-1793.

[4] Agrawal R K, Kaur B, Sharma S. Quantum based whale optimization algorithm for wrapper feature selection[J]. Applied Soft Computing, 2020, 89: 106092.

[5] Hur T, Kim L, Park D K. Quantum convolutional neural network for classical data classification[J]. Quantum Machine Intelligence, 2022, 4(1): 3.

[6] Song, X., Liu, T., Li, S. E., et al. (2023). Training Multi-layer Neural Networks on Ising Machine. arXiv preprint, arXiv:2311.03408.

[7] Lu, T. C. (2021). CNN Convolutional Layer Optimisation Based on Quantum Evolutionary Algorithm. Connection Science, 33(3), 482–494.

[8] Henderson, M., Shakya, S., Pradhan, S., et al. (2020). Quanvolutional Neural Networks: Powering Image Recognition with Quantum Circuits. Quantum Machine Intelligence, 2(1), 2.

[9] Lin, C. H. A., Liu, C. Y., Chen, S. Y. C., et al. (2024). Quantum-Trained Convolutional Neural Network for Deepfake Audio Detection. arXiv preprint, arXiv:2410.09250.

[10] Zhang, Z., Mi, X., Yang, J., et al. (2023). Remote Sensing Image Scene Classification in Hybrid Classical–Quantum Transferring CNN with Small Samples. Sensors, 23(18), 8010.

VIII. Appendix

Listing 1: The python Source code of Algorithm of feature selection for problem 1

```
#-----------------------------------------------------------------
#  Filename: FeatureSelection.py                         |
#  Description: Feature Selection For Classification Model |
#  Author: Team Quantum Quenching                        |
#-----------------------------------------------------------------

import numpy as np
from gurobipy import Model, GRB, LinExpr
from scipy.optimize import root_scalar
import kaiwu as kw
import math
from kaiwu.qubo._qubo_expression import QuboExpression
import pandas as pd
import time
from ucimlrepo import fetch_ucirepo
kw.license.init(user_id="678958809208586626",
    sdk_code="mEgNKZCZ1EXd0BBneJOAGKewRKujHA")


# ------------------- Data Fetching and Preparation
    ----------------------------
statlog_german_credit_data = fetch_ucirepo(id=144)

# The dataset is fetched from UCI Machine Learning Repository, 1000
    data in total
X = statlog_german_credit_data.data.features.to_numpy() # Collect
    features in matrix X (20 features in total)
Y = statlog_german_credit_data.data.targets.to_numpy() # Collect
    targets in vector Y (labels)

# Convert X to DataFrame for better manipulation
df = pd.DataFrame(X)
```

```
# Process each column to convert categorical strings like 'A102' to
    integers like '102'
for column in df.columns:
    if df[column].astype(str).str.match(r'[A-Z]\d+').any(): # Check for
        strings like 'A102'
        df[column] =
            df[column].astype(str).str.extract(r'(\d+)').astype(int)

# Convert back to numpy array
X = df.to_numpy()

# Get information of the dataset, N for amount of elements (1000), p
    for amount of features (20)
N,P = X.shape

# --------------------------- Model Preparation
    ---------------------------------
class BinnedInfo():
    def __init__(self, binnedX, edgesX, iscatX, nbinsX):
        self.binned = binnedX
        self.edges = edgesX
        self.iscat = iscatX
        self.nbins = nbinsX

def mquantile(data, prob: float) -> float:
    """
    Defining a function that return the quantile of a given distribution
        or data array
    Args:
        data: Any data array (this would determine the distribution)
        prob: Accumulated probability
    Returns:
        The value of the quantile
    """
    # get the size of the data
    data = np.array(data)
    n = data.size
    if n == 0:
        raise ValueError("Empty input, please try again.")

    # calculate the probabilities
    if prob >= 1 - .5/n:
        return data.max()
    elif prob <= .5 / n:
        return data.min()

    # look for i that satisfies (i + .5) / n <= prob <= (i + 1.5) / n
    t = n * prob - .5
    i = np.floor(t).astype(int)

    # arranging data
    data = np.partition(data, i)

    if i == t: # if it is exactly the integer index
        return float(data[i]) # return in forms of float
    else:
        # perform linear interpolation
        low_val = data[i]
        high_val = data[i + 1:].min() if (i + 1) < n else low_val

        if np.isinf(low_val):
            return low_val

        return float(low_val + (high_val - low_val) * (t - i))

def iComputeMIXIXJ(Xi, Xj, nbinXi, nbinXj):
    """
    Defining a function that return the mutual information (MI) between
        Xi and Xj
    Args:
        Xi: A column of a certain feature or a label
        Xi: A column of a certain feature or a label
        nbinXi: Number of bins for Xi
        nbinXj: Number of bins for Xj
    Returns:
        The value of the mutual information (MI)
    """
    # Attain numbers of elements
    N = len(Xi)

    # Create an all-zero matrix for counting frequencies
    NXY = np.zeros((nbinXi,nbinXj))

    # Plug in the statistical frequency
    for n in range(len(Xi)):
        a = int(Xi[n-1])
        b = int(Xj[n-1])
        if 0 <= a <= nbinXi and 0 <= b <= nbinXj:
            NXY[a-1, b-1] += 1

    # Compute edge distribution
    NX = np.sum(NXY,axis=1)
    NY = np.sum(NXY,axis=0)

    # Calculate the expectation of the joint distribution
    NXYDivN = NX[:,None]*NY[None,:]/N

    # Extract non-zero element indexes and corresponding values
    nozeroID = NXY != 0
    nonzeroNXY = NXY[nozeroID]
    nonzeroNXNYDivN = NXYDivN[nozeroID]

    # Computational Mutual Information
    epsilon = 1e-10 # make sure the denominator
    joint_distribution_probability = nonzeroNXY / N # Calculate the
        actual joint distribution probability
    log_prob_ratio = np.log(nonzeroNXY / nonzeroNXNYDivN) # Calculate
        the log-likelihood ratio
    mi = np.sum(log_prob_ratio * joint_distribution_probability)
    return mi

def iComputeMIXX(binnedX, nbinsX):
    """
    Defining a function that return the mutual information (MI) for X
    Args:
        binnedX: the feature matrix after being discretized into bins
        nbinX: Number of bins for all features in X
    Returns:
        The value of the mutual information (MI) matrix for x
    """
    p = binnedX.shape[1]
    Q = np.zeros((p, p))
    for i in range(p):
        for j in range(i+1, p):
            Xi = binnedX[:, i-1]
            Xj = binnedX[:, j-1]
            nbinsXi = nbinsX[i-1]
            nbinsXj = nbinsX[j-1]
            Q[i-1,j-1] = iComputeMIXIXJ(Xi, Xj, nbinsXi, nbinsXj)

    Q=Q+Q.T
    return Q

def iComputeMIXY(binnedX, binnedY, nbinsX, nbinsY):
    """
    Defining a function that return the mutual information (MI) between
        X and Y
    Args:
        binnedX: the feature matrix after being discretized into bins
        binnedY: the label vector after being discretized into bins
        nbinX: Number of bins for X
        nbinY: Number of bins for Y
    Returns:
        The value of the mutual information (MI) matrix between X and Y
    """
    p = binnedX.shape[1]
    f = np.zeros((p, 1))
    for i in range(p):
        Xi = binnedX[:, i-1]
        nbinsXi = nbinsX[i-1]
        f[i-1] = iComputeMIXIXJ(Xi, binnedY, nbinsXi, nbinsY[0])
    return f

def iDiscretize(x, nbins):
    """
    Defining a function that discretize the data into bins
    Args:
        x: an input data array
        nbins: number of bins
    Returns:
        binnedx: the discretized data array
        edges: the edges of bins
        iscat: a label that shows whether the data is categorical
        nbins: number of bins
    """
    if pd.api.types.is_categorical_dtype(x) or isinstance(x,
        pd.Categorical):
        binnedx = x.codes.astype(float) # Convert categories to
            numerical codes
        edges = None
        iscat = True
        nbins = len(pd.unique(x))
    else:
        probs = np.linspace(0, 1, nbins+1 ) # Compute probabilities for
            quantiles
        edges = [mquantile(x, p) for p in probs ] # Get quantile-based
            bin edge

        # Ensure edges are numeric
        edges = np.array(edges, dtype=float)
        # edges=edges[1:nbins-1]
        binnedx = np.searchsorted(edges, x, side="left")
        binnedx = np.clip(binnedx, 1, nbins)
        iscat = False

    return binnedx, edges, iscat, nbins

def iBinPredictors(X, nbins):
    """
    Defining a function that initialize the bins and preserve
        information of discretized data
    Args:
        X: the input data matrix
        nbins: number of bins
    Returns:
        binned_info: the binned information object
    """
    # Determine size of input
    if isinstance(X, pd.DataFrame):
        N,p = X.shape

    else:
        X = np.array(X)
        N,p = X.shape

    binnedX = np.zeros((N, p))
    edgesX = [None] * p
    iscatX = [False] * p
    nbinsX = [0] * p

    # Iterate over predictors
    for i in range(p):
        oneX = X.iloc[:, i] if isinstance(X, pd.DataFrame) else X[:, i]

        # Determine the number of bins
        unique_values = np.unique(oneX)
        nbinsi = min(nbins, len(unique_values))

        # Discretize the column
        binned_col, edges, iscat, nbins_used = iDiscretize(oneX, nbinsi)
        binnedX[:, i] = binned_col
        edgesX[i] = edges
        iscatX[i] = iscat
        nbinsX[i] = nbins_used

    # Return results as a dictionary
    binned_info = BinnedInfo(binnedX, edgesX, iscatX, nbinsX)

    return binned_info


def iBinXY(X, Y, nbins):
    """
    Defining a function that complete the discretizing procedure of X
        and Y
    Args:
        X: the input data matrix X
        Y: the input data matrix Y
        nbins: number of bins
    Returns:
        binnedXinfo: the binned information object for X
        binnedYinfo: the binned information object for Y
    """
    binnedXInfo = iBinPredictors(X, nbins)
    binnedYInfo = iBinPredictors(Y, nbins)

    return binnedXInfo, binnedYInfo

# -------------------------- Parameter Estimation
    -------------------------------
# Solve function using classical annealing (Gurobi optimization)
def solve(Q):
    """
    Defining a function that solve the equation to estimate beta
    Args:
        Q: a QUBO function
    Returns:
        result: a vector of binary variables that are used to estimate
            beta
    """
    n = Q.shape[0]
    model = Model()
    model.setParam('OutputFlag', 0) # Suppress output

    # Add binary variables
    x = model.addVars(n, vtype=GRB.BINARY, name="x")

    # Define the QUBO objective
    objective = LinExpr()
    for i in range(n):
        for j in range(n):
            if Q[i, j] != 0: # Avoid unnecessary calculations
                objective += Q[i, j] * x[i] * x[j]

    model.setObjective(objective, GRB.MINIMIZE)

    # Solve the QUBO problem
    model.optimize()

    # Extract solution
    result = {
        'BestX': np.array([x[i].X for i in range(n)])
    }
    return result

# Function to compute ‘howmany‘
def howmany(beta, R, J):
    """
    Defining a function that solve the equation to estimate beta
    Args:
        beta: a parameter to ensure the number of feature selected is
            equal to K (users could define K)
        R: matrix that reflect mutual information between each features
            within X
        J: matrix that reflect mutual information between Y and all
            features within X
    Returns:
        n: the number of features selected
        result: a vector of binary variables that are used to estimate
            beta
    """
    # Compute the QUBO matrix
    Q = (1 - beta) * R - beta * np.diag(J.flatten())
    # Get dimensions of QUBO matrix
    r, c = Q.shape
    # print(f"Matrix dimensions: r={r}, c={c}")

    # Solve QUBO problem
    result=solve(Q)
    # print(result)
    # Count the number of "1"s in the solution
    n = int(np.sum(result['BestX']))
    return n, result


# Function to find ‘beta‘ such that ‘howmany(beta, R, J) - K = 0‘
def find_beta(R, J, K, beta_min=0, beta_max=1, tolerance=1e-6):
    """
    Defining a function to find 'beta' such that satisfies
        'howmany(beta, R, J) - K = 0'
    Args:
        R: matrix that reflect mutual information between each features
            within X
        J: matrix that reflect mutual information between Y and all
            features within X
        K: number of features selected
        beta_min: minimum value of beta
        beta_max: maximum value of beta
        tolerance: tolerance value to check for convergence
    Returns:
        the estimated value of beta
    """
    from scipy.optimize import root_scalar

    # Objective function for root finding
    def objective(beta):
        n, _ = howmany(beta, R, J)
        return n - K

    # Use root_scalar to find the root
    solution = root_scalar(objective, bracket=[beta_min, beta_max],
        xtol=tolerance)

    if solution.converged:
        print(f"Found beta: {solution.root}")
        return solution.root
    else:
        raise ValueError("Failed to find a suitable beta within the
            given range")

# ---------------------- Setup of QUBO Model and Solver
    -------------------------
def sa(R,J,beta):
    """
    Defining a function to solve the final QUBO
    Args:
        R: matrix that reflect mutual information between each features
            within X
        J: matrix that reflect mutual information between Y and all
            features within X
        beta: a parameter to ensure the number of feature selected is
            equal to K (users could define K)
    Returns:
        the result of features selected
    """
    x = kw.qubo.ndarray(20, 'x', kw.qubo.Binary)
    Q = (1 - beta) * R - beta * np.diag(J.flatten())
    np.savetxt('matrix.csv', Q, delimiter=',', fmt='%.3f')
    q_model = 0;
    for i in range(20):
        for j in range(20):
            if Q[i, j] != 0:
                q_model += Q[i, j]*x[i]*x[j]
    q_model = kw.qubo.make(q_model)
    ci = kw.qubo.qubo_model_to_ising_model(q_model)
    obj_ising = ci;
    ising =ci.get_ising()
    matrix = ising["ising"] # Store the Ising model matrix
    bias = ising["bias"] # Store the Ising model bias
    worker = kw.classical.SimulatedAnnealingOptimizer(
        initial_temperature=1000,
        beta=0.99,
        cutoff_temperature=0.0001,
        iterations_per_t=1000)
    output = worker.solve(matrix)
    opt = kw.sampler.optimal_sampler(matrix, output, bias=0,
        negtail_ff=True)
    best = opt[0][0]
    vars = obj_ising.get_variables()
    sol_dict = kw.qubo.get_sol_dict(best, vars)
    print(sol_dict)

    return

# ------------------------------ Main Function
    ----------------------------------
if __name__ == "__main__":
    # Define R (20x20 matrix), J (20-element vector), K (target value)
    print('------------------------ Start Setting up the QUBO Matrix
        ------------------------')
    nbins = 20
    binnedXInfo, binnedYInfo = iBinXY(X, Y, nbins)
    binnedX = binnedXInfo.binned
    nbinsX = binnedXInfo.nbins
    binnedY = binnedYInfo.binned
    nbinsY = binnedYInfo.nbins
    # print(nbinsX,
    #      nbinsY)
    R0 = iComputeMIXX(binnedX, nbinsX)
    # print(000)
    # print(R0)
    # Y is already in binned form, so binnedY is equal to Y
    J = iComputeMIXY(binnedX, binnedY, nbinsX, nbinsY)
    K = 7 # Target number of "1"s in the solution, in the experiment, we
        shifted this value
    R = R0/(K-1)
    print('------------------------ Start looking for optimal beta
        ------------------------')
    # Find beta
    beta = find_beta(R, J, K)
    print('---------------------- beta is calculated, start solving QUBO
        ----------------------')
    start_time = time.time()

    sa(R, J, beta)
    end_time = time.time()
    execution_time = end_time - start_time
    print('problem solved, the execution time of pyhton program when K=7
        is:', execution_time)
    print('On quantum computer in the Kaiwu Developer platform is ')
```

Listing 2: The python code for classification

```
#-----------------------------------------------------------------
#  Filename: Experiment.py                                  |
#  Description: Testing feature selection in three models |
#  Author: Team Quantum Quenching                           |
#-----------------------------------------------------------------


import numpy as np
import matplotlib.pyplot as plt
from knn import KNN
from naiv_bay import naivBayes
from svm import SVM
import pandas as pd
from ucimlrepo import fetch_ucirepo
# ------------------------ Result of Feature Selected
    ----------------------------
All = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
    19, 20]
Best_3 = [1, 3, 12]
Best_4 = [1, 2, 3, 6]
Best_5 = [1, 2, 3, 6, 7]
Best_6 = [1, 2, 3, 4, 6, 7]
Best_7 = [1, 2, 3, 4, 6, 7, 20]
Best_8 = [1, 2, 3, 4, 6, 7, 12, 20]
Best_9 = [1, 2, 3, 4, 6, 7, 12, 13, 20]
Best_10 = [1, 2, 3, 4, 6, 7, 8, 12, 13, 20]
Best_11 = [1, 2, 3, 4, 5, 6, 7, 9, 12, 13, 20]
Best_12 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 20]
Best_13 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 15, 20]
Best_14 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 15, 17, 20]
Best_15 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 15, 16, 17, 20]
Best_16 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 15, 16, 17, 19, 20]
Best_17 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 15, 16, 17, 19, 20]
Best_18 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 15, 16, 17, 18, 19,
    20]

# ---------------------------- Start Experiment
    --------------------------------
# the experiment could choose any list of selected feature above
# we show some of the result here, the full result is shown in the paper
# print('Results of KNN classification before feature selection:')
# KNN(All)
# print('Results of KNN classification after feature selection:')
# KNN(Best_11)
# print('Results of naive bayes classification before feature
    selection:')
# naivBayes(All)
# print('Results of naive bayes classification after feature
    selection:')
# naivBayes(Best_17)
print('Results of support vector machine classification before feature
    selection:')
SVM(All)
print('Results of support vector machine classification after feature
    selection:')
SVM(Best_10)
```

Listing 3: The python code for knn classification

```
 #--------------------------------------------------------------
#  Filename: knn.py                                       |
#  Description: Classification task done by knn           |
#  Author: Team Quantum Quenching                         |
#--------------------------------------------------------------


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score,
    confusion_matrix,precision_recall_curve,average_precision_score,recall_score
from matplotlib.colors import ListedColormap
from itertools import product

def KNN(selected_features):
    """
    Defining a function to do classification task and visualization by
        KNN algorithm
    Args:
        selected_features: list of selected features:
    Returns:
        none
    """
    # Importing the dataset
```

```
dataset = pd.read_csv('german_credit_data_processed.csv')
X = dataset.iloc[:, selected_features].values
y = dataset.iloc[:, 21].values
y_binary = np.where(y == 2, 1, 0)

# Split data into train set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y_binary,
    test_size=0.25, random_state=0)

# Feature scaling
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Proceed KNN regression
classifier = KNeighborsClassifier(n_neighbors=5, metric='minkowski',
    p=2)
classifier.fit(X_train, y_train)

# Proceed prediction on test data
y_pred = classifier.predict(X_test)
y_proba = classifier.predict_proba(X_test)[:, 1] # Probability of
    predicting 'True'

# Calculation of Accuracy and Precision
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, labels=[0, 1],
    pos_label=1, average='macro')

# Print result
print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")

# Calculate confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=[0, 1])
print("Confusion matrix:\n", cm)
```

```
# Calculation of recall
recall = recall_score(y_test, y_pred, labels=[0, 1], pos_label=1)
print(f"Recall: {recall:.2f}")

# Calculation of F1 score
beta = 5 ** (1 / 2)
Weight_F_Score = (1 + beta ** 2) * (precision * recall) / (beta ** 2
    * precision + recall)
print(f"F1-Score: {Weight_F_Score:.2f}")

# Dimensionality reduction
from sklearn.decomposition import PCA
pca = PCA(n_components=2) # dim -> 2
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Visualization
x_min, x_max = X_train_pca[:, 0].min() - 1, X_train_pca[:, 0].max()
    + 1
y_min, y_max = X_train_pca[:, 1].min() - 1, X_train_pca[:, 1].max()
    + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))

# Using the training dataset model to predict the class of each
    point in the grid
z =
    classifier.predict(sc.transform(pca.inverse_transform(np.c_[xx.ravel(),
    yy.ravel()])))
z = z.reshape(xx.shape)

plt.figure(figsize=(10, 6))
plt.pcolormesh(xx, yy, z, cmap=plt.cm.Pastel1, shading='auto')

# Scatter Plot
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1], c=y_train,
    cmap=plt.cm.cool, edgecolors='k',
          label='Training set')
plt.scatter(X_test_pca[:, 0], X_test_pca[:, 1], c=y_test,
    cmap=plt.cm.cool, marker='*', edgecolor='k',
          label='Test set')

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title('K-NN on German Credit Data')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()

plt.figure(figsize=(6, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Pastel1)
plt.title('Confusion Matrix')
plt.colorbar()
tick_marks = np.arange(2)
plt.xticks(tick_marks, ['Good', 'Bad'], rotation=45)
plt.yticks(tick_marks, ['Good', 'Bad'])

fmt = 'd'
thresh = cm.max() / 2.
for i, j in product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
            horizontalalignment="center",
            color="gray" if cm[i, j] > thresh else "black")

plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.tight_layout()
plt.show()

# Calculate p-r curve
precision, recall, _ = precision_recall_curve(y_test, y_proba)
avg_precision = average_precision_score(y_test, y_proba)
```

```
    # Visualize p-r curve
    plt.figure(figsize=(10, 6))
    plt.plot(recall, precision, marker='.', label='P-R Curve')
    plt.fill_between(recall, precision, beta=0.2, color='b')
    plt.title('Precision-Recall Curve')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.axhline(y=avg_precision, color='red', linestyle='--',
        label='Average Precision: {:.2f}'.format(avg_precision))
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.0])
    plt.legend()
    plt.grid()
    plt.show()
```

Listing 4: The python code for naive bayes classification

```
 # ----------------------------------------------------------------
#  Filename: naiv_bay.py                                    |
#  Description: Classification task done by naive bayes |
#  Author: Team Quantum Quenching                           |
# ----------------------------------------------------------------

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from itertools import product
from sklearn.decomposition import PCA
from sklearn.metrics import confusion_matrix
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix,
    precision_score, precision_recall_curve, \
    average_precision_score, recall_score

def naivBayes(features_selected):
    """
```

```
Defining a function to do classification task and visualization by
    Naive Bayes
Args:
    features_selected: list of selected features:
Returns:
    none
"""
# Importing the dataset
dataset = pd.read_csv('german_credit_data_processed.csv')
X = dataset.iloc[:, features_selected].values
y = dataset.iloc[:, 21].values
y_binary = np.where(y == 2, 1, 0)

# Splitting the dataset into the Training set and Test set
X_train, X_test, y_train, y_test = train_test_split(X, y_binary,
    test_size = 0.25, random_state = 0)

# Feature Scaling
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Fitting Naive Bayes to the Training set
classifier = GaussianNB()
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)
y_proba = classifier.predict_proba(X_test)[:, 1]

# Making the Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

# Calculate accuracy and precision
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
precision = precision_score(y_test, y_pred, labels=[0, 1],
    pos_label=1)
print(f"Precision: {precision:.2f}")

# Calculate confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion:\n", cm)

# Calculation of Recall
recall = recall_score(y_test, y_pred, labels=[0, 1], pos_label=1)
print(f"Recall: {recall:.2f}")

# Calculation of F1 score
beta = 5 ** (1 / 2)
Weight_F_Score = (1 + beta ** 2) * (precision * recall) / (beta ** 2
    * precision + recall)
print(f"F1-Score: {Weight_F_Score:.2f}")

# Dimensionality reduction
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Visualization
x_min, x_max = X_train_pca[:, 0].min() - 1, X_train_pca[:, 0].max()
    + 1
y_min, y_max = X_train_pca[:, 1].min() - 1, X_train_pca[:, 1].max()
    + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))

# Using the training dataset model to predict the class of each
    point in the grid
z =
    classifier.predict(sc.transform(pca.inverse_transform(np.c_[xx.ravel(),
    yy.ravel()])))
z = z.reshape(xx.shape)
plt.figure(figsize=(10, 6))
plt.pcolormesh(xx, yy, z, cmap=plt.cm.Pastel1, shading='auto')

# Scatter Plot
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1], c=y_train,
    cmap=plt.cm.cool, edgecolors='k',
           label='Training set')
plt.scatter(X_test_pca[:, 0], X_test_pca[:, 1], c=y_test,
    cmap=plt.cm.cool, marker='*', edgecolor='k',
           label='Test set')

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title('Gaussian Naive Bayes Classification Results with PCA')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()

plt.figure(figsize=(6, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Pastel1)
plt.title('Confusion Matrix')
plt.colorbar()
tick_marks = np.arange(2)
plt.xticks(tick_marks, ['Good', 'Bad'], rotation=45)
plt.yticks(tick_marks, ['Good', 'Bad'])

fmt = 'd'
thresh = cm.max() / 2.
for i, j in product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
             horizontalalignment="center",
             color="gray" if cm[i, j] > thresh else "black")

plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.tight_layout()
plt.show()
```

```
    # Calculate p-r curve
    precision, recall, _ = precision_recall_curve(y_test, y_proba)
    avg_precision = average_precision_score(y_test, y_proba)

    # Visualize p-r curve
    plt.figure(figsize=(10, 6))
    plt.plot(recall, precision, marker='.', label='P-R Curve')
    plt.fill_between(recall, precision, beta=0.2, color='b')
    plt.title('Precision-Recall Curve')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.axhline(y=avg_precision, color='red', linestyle='--',
        label='Average Precision: {:.2f}'.format(avg_precision))
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.0])
    plt.legend()
    plt.grid()
    plt.show()
```

Listing 5: The python code for SVM classification experiment

```
# ------------------------------------------------------------------
#  Filename: naiv_bay.py                                      |
#  Description: Classification task done by support vector machine |
#  Author: Team Quantum Quenching                             |
# ------------------------------------------------------------------

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score,
    confusion_matrix,recall_score,precision_recall_curve,average_precision_score
from itertools import product
```

```python
def SVM(features_selected):
    """
    Defining a function to do classification task and visualization by
        SVM algorithm
    Args:
        features_selected: list of selected features:
    Returns:
        none
    """
    # Input features and labels
    dataset = pd.read_csv('german_credit_data_processed.csv')
    X = dataset.iloc[:, features_selected].values
    y = dataset.iloc[:, 21].values
    y_binary = np.where(y == 2, 1, 0)

    # Split the dataset into train and test data
    X_train, X_test, y_train, y_test = train_test_split(X, y_binary,
        test_size=0.25, random_state=0)

    # Feature Scaling
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)

    # Fitting Naive Bayes to the Training set
    classifier = SVC(kernel='linear',probability=True) #

    classifier.fit(X_train, y_train) #

    # Predicting the Test set results
    y_pred = classifier.predict(X_test)
    y_proba = classifier.predict_proba(X_test)[:, 1] #



    # Calculate accuracy and precision
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, labels=[0, 1],
        pos_label=1, average='macro')
    print(f"Accuracy: {accuracy:.2f}")
    print(f"Precision: {precision:.2f}")

    # Making the Confusion Matrix
    cm = confusion_matrix(y_test, y_pred)
    print("Confusion matrix:\n", cm)

    # Calculation of Recall
    recall = recall_score(y_test, y_pred, labels=[0, 1], pos_label=1,)
    print(f"Recall: {recall:.2f}")

    # Calculation of F1 score
    beta = 5 ** (1 / 2)
    Weight_F_Score = (1 + beta ** 2) * (precision * recall) / (beta ** 2
        * precision + recall)
    print(f"F1-Score: {Weight_F_Score:.2f}")

    # Dimensionality reduction
    from sklearn.decomposition import PCA
    pca = PCA(n_components=2) #      2
    X_train_pca = pca.fit_transform(X_train)
    X_test_pca = pca.transform(X_test)

    # Visualization
    x_min, x_max = X_train_pca[:, 0].min() - 1, X_train_pca[:, 0].max()
        + 1
    y_min, y_max = X_train_pca[:, 1].min() - 1, X_train_pca[:, 1].max()
        + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                      np.arange(y_min, y_max, 0.02))

    # Using the training dataset model to predict the class of each
        point in the grid
    z =
        classifier.predict(sc.transform(pca.inverse_transform(np.c_[xx.ravel(),
        yy.ravel()])))
```

```
z = z.reshape(xx.shape)
plt.figure(figsize=(10, 6))
plt.pcolormesh(xx, yy, z, cmap=plt.cm.Pastel1, shading='auto')

# Scatter Plot
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1], c=y_train,
    cmap=plt.cm.cool, edgecolors='k',
           label='Training set')
plt.scatter(X_test_pca[:, 0], X_test_pca[:, 1], c=y_test,
    cmap=plt.cm.cool, marker='*', edgecolor='k',
           label='Test set')

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title('SVM Classification Result')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()

plt.figure(figsize=(6, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Pastel1)
plt.title('Confusion Matrix')
plt.colorbar()
tick_marks = np.arange(2)
plt.xticks(tick_marks, ['Good', 'Bad'], rotation=45)
plt.yticks(tick_marks, ['Good', 'Bad'])

fmt = 'd'
thresh = cm.max() / 2.
for i, j in product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
            horizontalalignment="center",
            color="gray" if cm[i, j] > thresh else "black")

plt.ylabel('True label')
plt.xlabel('Predicted label')
```

```
    plt.tight_layout()
    plt.show()

    # Calculate p-r curve
    precision, recall, _ = precision_recall_curve(y_test, y_proba)
    avg_precision = average_precision_score(y_test, y_proba) #



    # Visualize p-r curve
    plt.figure(figsize=(10, 6))
    plt.plot(recall, precision, marker='.', label='P-R Curve')
    plt.fill_between(recall, precision, beta=0.2, color='b')
    plt.title('Precision-Recall Curve')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.axhline(y=avg_precision, color='red', linestyle='--',
        label='Average Precision: {:.2f}'.format(avg_precision))
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.0])
    plt.legend()
    plt.grid()
    plt.show()
```

Listing 6: The python code for Convolution neural network for image classification task

```
   #-----------------------------------------------------------------
#  Filename: CNN_Classical.py                              |
#  Description: CNN for image classification               |
#  Author: Team Quantum Quenching                          |
#-----------------------------------------------------------------
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# ---------------------------- Model Initialization
    --------------------------------
```

```
plt.rcParams['font.sans-serif'] = ['SimHei']

# ------------------------ Data Fetching and Preparation
    ---------------------------
mnist = tf.keras.datasets.mnist
(train_x,train_y),(test_x,test_y) = mnist.load_data()
print('\n train_x:%s, train_y:%s, test_x:%s,
    test_y:%s'%(train_x.shape,train_y.shape,test_x.shape,test_y.shape))

# Normalization
X_train,X_test =
    tf.cast(train_x/255.0,tf.float32),tf.cast(test_x/255.0,tf.float32)
y_train,y_test = tf.cast(train_y,tf.int16),tf.cast(test_y,tf.int16)

# ------------------------------- Model Setup
    -----------------------------------
# Set up the frame of a model

model = tf.keras.Sequential()
# First, convolution layer
model.add(Conv2D(16, kernel_size=(3, 3), activation='relu',
    padding='same', input_shape=(28,28,1)))
# Second, max-pooling layer
model.add(MaxPooling2D(pool_size=(2, 2)))
# Third, flatten layer
model.add(Flatten(input_shape=(28,28)))
# Fourth, full-connected layer
model.add(Dense(128,activation='relu'))
# Last, output layer
model.add(Dense(10,activation='softmax'))
print('\n',model.summary())

# Compile the model and set the optimizer
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['sparse_c

# ------------------------------- Model Training
    -----------------------------------
```

```
fit_model =
    model.fit(X_train,y_train,batch_size=64,epochs=5,validation_split=0.2)

# ------------------------------- Model Evaluation
    -----------------------------------
print('--------------')
model.evaluate(X_test,y_test,verbose=2)

# ------------------------------- Model Evaluation
    -----------------------------------
plt.figure(figsize=(10,3))

plt.figure()
for i in range(3):
    num = np.random.randint(1,10000)

    plt.subplot(1,3,i+1)
    plt.axis('off')
    plt.imshow(test_x[num],cmap='gray')
    demo = tf.reshape(X_test[num],(1,28,28))
    y_pred = np.argmax(model.predict(demo))
    plt.title('Label'+str(test_y[num])+'\nPred'+str(y_pred))

plt.show()
```

Listing 7: The python code for discretize weights and training CNN

```
#------------------------------------------------------------------
#  Filename: ImageClassification_Quantum.py              |
#  Description: Discretizing and encoding the weights of a neural |
#            network into a QUBO model and using the simulated |
#             annealing algorithm to solve it.           |
#  Author: Team Quantum Quenching                         |
#------------------------------------------------------------------

import numpy as np
import cv2
import os
```

```python
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt
import kaiwu as kw
from dimod import BinaryQuadraticModel
from dimod import SimulatedAnnealingSampler

kw.license.init(user_id="678958809208586 26",
    sdk_code="mEgNKZCZ1EXd0BBneJOAGKewRKujHA")

def load_and_preprocess_images(dataset_dir, image_size=(28, 28),
    flatten=True):
    """
    Defining a function that load the image dataset
    Args:
        dataset_dir: the file path of the dataset
        image_size: size of the image to be resized to
        flatten: whether to flatten the image or not
    Returns:
        The images and labels as numpy arrays
    """
    images, labels = [], []
    supported_extensions = ('.jpg', '.jpeg', '.png', '.bmp', '.tiff')

    print(f"Processing images from directory: {dataset_dir}")
    if not os.path.isdir(dataset_dir):
        print(f"Directory not found: {dataset_dir}")
        return np.array(images), np.array(labels)

    files = os.listdir(dataset_dir)
    print(f"Finding {len(files)} documents")

    for filename in files:
        if filename.lower().endswith(supported_extensions):
            filepath = os.path.join(dataset_dir, filename)
            img = cv2.imread(filepath, cv2.IMREAD_GRAYSCALE)
            if img is not None:
                try:
                    # cut out the center square
                    coords = cv2.findNonZero(img)
                    if coords is not None:
                        x, y, w, h = cv2.boundingRect(coords)
                        img = img[y:y + h, x:x + w]
                    # adjust size to desired size
                    img = cv2.resize(img, image_size)
                    # split image into 4 patches
                    h_split = np.array_split(img, 2, axis=0)
                    patches = [np.array_split(h_part, 2, axis=1) for
                        h_part in h_split]
                    patches = [item for sublist in patches for item in
                        sublist]
                    patch_values = []
                    for patch in patches:
                        white_pixels = cv2.countNonZero(patch)
                        if white_pixels > 50:
                            value = 1
                        elif white_pixels < 20:
                            value = -1
                        else:
                            value = 0
                        patch_values.append(value)
                    images.append(patch_values)
                    # Decoding the label from the filename
                    try:
                        label_str = filename.split('_')[2]
                        label = int(
                            label_str.replace('.jpg', '').replace('.jpeg',
                                '').replace('.png',
                                '').replace('.bmp','').replace(
                                '.tiff', ''))
                        labels.append(label)
                        print(f"Successfully loaded image: {filename},
                            label: {label}")
                    except (IndexError, ValueError) as e:
                        print(f"Unable to decode label from filename:
```

```
                        {filename}, error: {e}")
              except Exception as e:
                 print(f"Failed to process image: {filepath}, error:
                    {e}")
           else:
              print(f"Unable to read image: {filepath}")
        else:
           print(f"Skipppping unsupported file type: {filename}")

     if not images:
        print("Unable to load any images.")
     else:
        print(f"Successfully loaded {len(images)} images")
     return np.array(images), np.array(labels)

def quantize_and_encode(weights, bits=3):
    """
    Defining a function that encodes the weights
    Args:
       weights: input weights to be encoded
       bits: number of bits to use for binary encoding
    Returns:
       Encoded weights and the discretized weights
    """
    # Discretize the weights into 7 levels
    quant_levels = np.array([-3, -2, -1, 0, 1, 2, 3])
    quantized = np.zeros_like(weights, dtype=int)
    for i in range(weights.shape[0]):
       for j in range(weights.shape[1]):
          idx = np.argmin(np.abs(quant_levels - weights[i, j]))
          quantized[i, j] = quant_levels[idx]

    # Binary encoding of the weights
    encoded = {}
    for i in range(weights.shape[0]):
       for j in range(weights.shape[1]):
          w = quantized[i, j]
```

```
            sign = 1 if w < 0 else 0
            magnitude = abs(w)
            binary = [int(x) for x in format(magnitude, f'0{bits}b')]

            encoded[f'W_{i}_{j}_s'] = sign
            for b in range(bits):
                encoded[f'W_{i}_{j}_b{b}'] = binary[bits - 1 - b] # LSB
                    first
    return encoded, quantized

def sa(Q, T_init=1000, beta=0.99, T_min=0.0001, iterations_per_T=10):
    """
    Defining a function that constructs the QUBO model and uses the
        simulated annealing algorithm to solve it.
    Args:
        T_init: initial temperature
        beta: cooling rate
        T_min: minimum temperature
        iterations_per_T: number of iterations per temperature
    Returns:
        Encoded weights and the discretized weights
    """
    worker = kw.classical.SimulatedAnnealingOptimizer(
            initial_temperature=T_init,
            beta=beta,
            cutoff_temperature=T_min,
            iterations_per_t=iterations_per_T)
    qubo_model = 0
    x = kw.qubo.ndarray(200, 'x', kw.qubo.Binary)
    for i in range(200):
        for j in range(200):
            qubo_model += Q[i][j]*x[i]*x[j]
    qubo_model = kw.qubo.make(qubo_model)
    ci = kw.qubo.qubo_model_to_ising_model(qubo_model)
    obj_ising = ci
    ising = ci.get_ising()
    matrix = ising["ising"]
    output = worker.solve(matrix)
    opt = kw.sampler.optimal_sampler(matrix, output, bias=0,
        negtail_ff=True)
    best = opt[0][0]
    vars = obj_ising.get_variables()
    # Convert the optimal solution into a dictionary of variable values
    sol_dict = kw.qubo.get_sol_dict(best, vars)
    obj_val = kw.qubo.get_val(qubo_model, sol_dict)
    return obj_val, sol_dict

def build_quubo(X, y, quant_levels, bits=3, rho=10.0,
    lambda_penalty=10.0):
    """
    Defining a function that constructs the QUBO model for finding the
        optimal solution
    Args:
        X: input data
        y: output data
        quant_levels: discretized weights
        bits: number of bits to use for binary encoding
        rho: penalty term for weights
        lambda_penalty: penalty term for binary variables
    Returns:
        The QUBO model
    """
    bqm = BinaryQuadraticModel({}, {}, 0.0, 'BINARY')

    num_samples, num_features = X.shape
    num_outputs = y.shape[1] # Outputs are one-hot encoded
    num_bits = bits

    W_vars = {}
    b_vars = {}
    for i in range(num_outputs):
        for j in range(num_features):
            var_name = f'W_{i}_{j}_s'
            W_vars[(i, j, 's')] = var_name
            bqm.add_variable(var_name, 0.0)
            for b in range(num_bits):
                var_name = f'W_{i}_{j}_b{b}'
                W_vars[(i, j, b)] = var_name
                bqm.add_variable(var_name, 0.0)
    for i in range(num_outputs):
        # Bias variables
        var_name = f'b_{i}_s'
        b_vars[(i, 's')] = var_name
        bqm.add_variable(var_name, 0.0)
        for b in range(num_bits):
            var_name = f'b_{i}_{b}'
            b_vars[(i, b)] = var_name
            bqm.add_variable(var_name, 0.0)

    # defining the objective function
    for idx in range(num_samples):
        x_i = X[idx]
        y_i = y[idx]
        for k in range(num_outputs):
            for j in range(num_features):
                # contributing to the linear term
                var_s = W_vars[(k, j, 's')]
                # (1 - 2 * s) * sum_b W_k_j_b * 2^b
                bqm.add_linear(var_s, -2 * y_i[k] * (-1) * x_i[j]) # s=1
                    (1-2*1)=-1
                print(f"Adding linear item: {var_s} -> {-2 * y_i[k] *
                    (-1) * x_i[j]}")

                # contributing to the quadratic term
                for b in range(num_bits):
                    var_b = W_vars[(k, j, b)]
                    coeff = -2 * y_i[k] * (2 ** b) * x_i[j]
                    bqm.add_linear(var_b, coeff)
                    print(f"Adding linear item: {var_b} -> {coeff}")
            # contributing to the bias term
            var_b_s = b_vars[(k, 's')]
            bqm.add_linear(var_b_s, -2 * y_i[k] * (-1))
            print(f"Adding linear item: {var_b_s} -> {-2 * y_i[k] *
                (-1)}")
            # contributing to the quadratic term
            for b in range(num_bits):
                var_b = b_vars[(k, b)]
                coeff = -2 * y_i[k] * (2 ** b)
                bqm.add_linear(var_b, coeff)
                print(f"Adding linear item: {var_b} -> {coeff}")

            # binary term
            for j in range(num_features):
                var_s = W_vars[(k, j, 's')]
                for b in range(num_bits):
                    var_b = W_vars[(k, j, b)]
                    bqm.add_quadratic(var_s, var_b, lambda_penalty)
                    print(f"Adding quadratic penalty: {var_s} * {var_b} ->
                        {lambda_penalty}")

            # penalty binary term
            var_b_s = b_vars[(k, 's')]
            for b in range(num_bits):
                var_b = b_vars[(k, b)]
                # add quadratic penalty
                bqm.add_quadratic(var_b_s, var_b, lambda_penalty)
                print(f"Adding quadratic penalty: {var_b_s} * {var_b} ->
                    {lambda_penalty}")

    # penalty term
    for i in range(num_outputs):
        for j in range(num_features):
            # adding penalty term to the weights
            for b in range(bits):
                var_b = W_vars[(i, j, b)]
                bqm.add_linear(var_b, rho)
                print(f"Adding linear penalty: {var_b} -> {rho}")
    for i in range(num_outputs):
        for b in range(bits):
            var_b = b_vars[(i, b)]
            bqm.add_linear(var_b, rho)
            print(f"Adding linear penalty: {var_b} -> {rho}")

    return bqm

def decode_solution(solution, num_outputs, num_features, bits=3):
    """
    Defining a function that decodes the solution
    Args:
        solution: the solution obtained from the QUBO model
        num_outputs: number of outputs
        num_features: number of features
        bits: number of bits to use for binary encoding
    Returns:
        The decoded weights and biases
    """
    W_decoded = np.zeros((num_outputs, num_features))
    b_decoded = np.zeros(num_outputs)
    for i in range(num_outputs):
        for j in range(num_features):
        # decoding weights
            var_s = f'W_{i}_{j}_s'
            var_bs = [f'W_{i}_{j}_b{b}' for b in range(bits)]
            sign = solution.get(var_s, 0)
            magnitude = 0
            for b in range(bits):
                magnitude += solution.get(var_bs[b], 0) * (2 ** b)
            W_decoded[i, j] = magnitude if sign == 0 else -magnitude
            print(f"decode weights W[{i}][{j}]: sign={sign},
                magnitude={magnitude}, value={W_decoded[i, j]}")
        # decoding bias
        var_b_s = f'b_{i}_s'
        var_b_bs = [f'b_{i}_{b}' for b in range(bits)]
        sign = solution.get(var_b_s, 0)
        magnitude = 0
        for b in range(bits):
            magnitude += solution.get(var_b_bs[b], 0) * (2 ** b)
        b_decoded[i] = magnitude if sign == 0 else -magnitude
        print(f"decode bias b[{i}]: sign={sign}, magnitude={magnitude},
            value={b_decoded[i]}")
    return W_decoded, b_decoded

def compute_loss(X, y, W, b):
    """
    Defining a function that calculates the loss function
    Args:
        X: input data
        y: output data
        W: decoded weights
        b: decoded biases
    Returns:
        The total loss and the loss values for each sample
    """
    num_samples = X.shape[0]
    loss_values = []
    for idx in range(num_samples):
        input_vector = X[idx]
        true_label = y[idx]
        linear_output = np.dot(W, input_vector) + b
        # sign as activation function
        predicted = np.sign(linear_output)
        predicted[predicted == 0] = 1
        # MSE as Loss function
        loss = np.mean((true_label - predicted) ** 2)
        loss_values.append(loss)
    total_loss = np.sum(loss_values)
    return total_loss, loss_values

def convert_bqm_to_qubo_matrix(bqm):
    """
    Defining a function that converts the BQM to a QUBO matrix
    Args:
        bqm: the binary quadratic model
    Returns:
        The QUBO matrix and the variable order
    """
    # Use methods to access linear and quadratic terms
    linear = bqm.linear
    quadratic = bqm.quadratic
    variables = sorted(set(linear.keys()) | set(i for pair in
        quadratic.keys() for i in pair))
    num_vars = len(variables)

    # Create a mapping from variable to index
    var_to_index = {var: idx for idx, var in enumerate(variables)}
    Q = np.zeros((num_vars, num_vars))
    for var, coeff in linear.items():
        Q[var_to_index[var], var_to_index[var]] = coeff

    # Fill in the off-diagonal with quadratic terms
    for (var1, var2), coeff in quadratic.items():
        idx1, idx2 = var_to_index[var1], var_to_index[var2]
        Q[idx1, idx2] += coeff
        if idx1 != idx2:
            Q[idx2, idx1] += coeff # Symmetric matrix

    return Q, variables # Return QUBO matrix and variable order



if __name__ == '__main__':
# ---------------------- Data Fetching and Preparation
    ----------------------
    dataset_dir =
        r'C:\code\code-py\-NumPy-CNN-Implementation-for-Image-Classification-main\ImageCl
    X, y = load_and_preprocess_images(dataset_dir)

    if X.size == 0:
        print("Empty dataset, please check the directory and file
        format.")
      exit(1)

    # normalize the pixel values to [-1, 1]
    X = X.astype(float)
    X = X / np.max(np.abs(X))

# ------------------- Discretization and Encoding
    -----------------------
    # label encoding
    y_encoder = LabelEncoder()
    y_encoded = y_encoder.fit_transform(y)
    num_classes = len(np.unique(y_encoded))
    y_one_hot = np.eye(num_classes)[y_encoded]

    X_reduced = X

    # quantization parameters
    bits = 3

    # initial weights and biases
    initial_W = np.zeros((num_classes, X_reduced.shape[1]))
    initial_b = np.zeros(num_classes)

    # quantize and encode the weights and biases
    encoded_W, quantized_W = quantize_and_encode(initial_W, bits=bits)
    encoded_b, quantized_b = quantize_and_encode(initial_b.reshape(-1,
        1), bits=bits)

    # defining the quantization levels
    quant_levels = np.array([-3, -2, -1, 0, 1, 2, 3])

# ----------------- Building and Solving the QUBO Model
    --------------------
    # constructing the QUBO model
    qubo = build_quubo(X_reduced, y_one_hot, quant_levels, bits=bits)
    qubo_matrix, var_order = convert_bqm_to_qubo_matrix(qubo)
    np.savetxt('qubo_matrix.csv', qubo_matrix, delimiter=',')

    # Output the QUBO model
    print("QUBO model constructed")




    # Solve the QUBO model using Simulated Annealing
    best_energy,_ = sa(qubo_matrix, T_init=1000, beta=0.99,
        T_min=0.0001, iterations_per_T=10)
    sampler = SimulatedAnnealingSampler()
    qubo_dict, offset = qubo.to_qubo()
    response = sampler.sample_qubo(qubo_dict, num_reads=100)
    best_solution = response.first.sample

    print("Optimal energy:", best_energy)

# ------------------ Decoding and Evaluating the Solution
    --------------------
    # Decode the solution
    decoded_W, decoded_b = decode_solution(best_solution, num_classes,
        X_reduced.shape[1], bits)

    print("Decoded weights:", decoded_W)
    print("Decoded biases:", decoded_b)

    # Compute the loss function
    total_loss, loss_values = compute_loss(X_reduced, y_one_hot,
        decoded_W, decoded_b)
    print(f"Total loss: {total_loss/10}")
```