Team Number :	apmcmlz2501509

2025 APMCM Wuyue Cup summary sheet

As the most prevalent malignant tumor among women globally, breast cancer is characterized by metastasis as the core factor leading to deteriorated prognosis and reduced survival rates. Accurate identification of biomarkers associated with metastatic risk is crucial for early clinical intervention and treatment decision-making.

Multi-omics technologies and multidimensional data have provided abundant resources for elucidating the molecular mechanisms of breast cancer metastasis. Our team collected and integrated multi-omics data, including transcriptome, epigenome, genome, with clinical data from the TCGA database, obtaining 118403 high-dimensional features. Traditional models like ordinary logistic regression and Lasso regression not only require prolonged computation but also struggle to balance feature relevance and redundancy, resulting in predicted F1 score often below 0.75.

This study utilized 946 breast cancer samples from the TCGA database as the research object. After preprocessing including missing value removal, normalization, and sample intersection screening, an integrated dataset was constructed. A quadratic unconstrained binary optimization (QUBO) model was developed, aiming to maximize feature-label correlation and minimize feature redundancy.

The results demonstrate that the QUBO model achieves stable solutions in just 1.1202ms on real-world data, with a mere 28-point variation in QUBO values across 10 solution sets. After screening 20 core feature subsets, the model achieves an F1 score of 0.8235 in performance validation, significantly outperforming traditional models (Lasso regression: F1 score 0.7103; Naive Bayes: F1 score 0.5833).

This study demonstrates quantum computing's efficiency and superiority in optimizing high-dimensional bioinformatics data, providing a clinically applicable and biologically rational multi-omics approach for breast cancer metastasis prediction.

Keywords: breast cancer, quantum computing, QUBO model, multi-omics data, tumor metastasis

Contents

I Background

Breast cancer metastasis is **a critical turning point** leading to **a sharp deterioration in patient prognosis**.[1] Its occurrence signifies the progression of the disease from a locally controllable stage to a systemic and refractory phase, significantly reducing the five-year survival rate and making treatment decisions extremely complex.

However, current clinical predictions primarily rely on **static pathological indicators** such as tumor size, lymph node status, and histological classification.[2] Although these indicators possess certain prognostic value, they essentially represent descriptions of established facts or rough estimates of population risks, **failing to capture the core dynamic processes driving metastasis**: namely, the highly heterogeneous clonal evolution within tumors, the intricate molecular network regulation, and the real-time interactions of the tumor microenvironment.[2]

Therefore, the existing system **struggles to provide precise early warning** for individual patients before the critical transition point, often resulting in **delayed and passive clinical interventions**.

How to overcome the limitations of static pathology, systematically analyze and predict this dynamic biological process in advance**,** thereby **achieving proactive prevention and control**, constitutes **a scientific challenge that urgently needs to be addressed** from the molecular mechanism to the clinical translation level.

II Problem analysis

Metastasis is a complex, multi-step biological process involving the interaction and spatiotemporal dynamic evolution of multi-level molecular mechanisms, including genomic variations, transcriptional regulatory abnormalities, epigenetic modifications, and alterations in protein functions. Single-omics data (e.g., transcriptome alone) can only reveal partial information at this level, making it

difficult to comprehensively capture the complex molecular networks driving metastasis, with inherent biases and limitations. Therefore, **integrating multi-omics data such as genomics, transcriptomics, and epigenomics** has become an inevitable trend for systematically elucidating metastasis mechanisms and identifying robust biomarkers.[1] However, the integrated analysis of multi-omics data faces severe challenges such as high data dimensions, significant heterogeneity, substantial noise, and relatively limited sample sizes, with its processing and mining processes typically being time-consuming and labor-intensive.[3] When constructing predictive models, identifying the most biologically significant and predictive key feature combinations from massive datasets constitutes a **NP-hard combinatorial optimization problem**. Conventional statistical or machine learning models often exhibit low computational efficiency, are prone to local optima, and consume excessive computational resources.[3] In recent years, the **QUBO model** based on quantum computing principles has provided a **promising solution** to this bottleneck problem.[4]

The core advantage of the QUBO model in solving nonlinear problems lies in its **objective function**.

$$H(q) = -\sum_i a_i q_i + \sum_{i<j} \beta_{ij} q_i q_j \tag{1}$$

quadratic term in $\sum_{i<j} \beta_{ij} q_i q_j$ it **directly encodes pairwise interactions between features**. Unlike traditional linear models that require manual construction of interaction terms, this framework achieves this through coefficients β_{ij} . This approach inherently quantifies nonlinear synergistic (negative values) or antagonistic (positive values) effects between features, thereby automatically mining and leveraging complex nonlinear patterns in multi-omics data – such as gene pathway synergy and cross-omics regulation – while optimizing the feature subset. Consequently, the selected biomarker set more accurately reflects the true biological network logic.[4] The core computational advantage of the QUBO model lies in its ability to **map combinatorial optimization problems into energy-minimization form**, enabling **parallel solution through quantum annealing**. This process simultaneously explores an exponential solution space, with computation time

primarily determined by the physical systems eigen-time scale. This effectively avoids the **computational explosion** characteristic of traditional algorithms as problem sizes increase.[5] This enables the framework to efficiently obtain high-quality feature subsets through single or few iterations when processing high-dimensional multi-omics data, significantly reducing the substantial computational costs required by traditional iterative screening methods.

III Problem hypothesis

1. The integration of multi-omics data can comprehensively capture molecular information associated with breast cancer metastasis, effectively predicting metastatic risk.

2. Assuming the QUBO model of quantum computing can overcome the bottleneck of multi-omics computation, it can rapidly construct a high-precision breast cancer metastasis prediction model.

3. The synergistic application of multi-omics and quantum computing can accurately predict the critical point of breast cancer metastasis, supporting early clinical intervention.

IV Our work

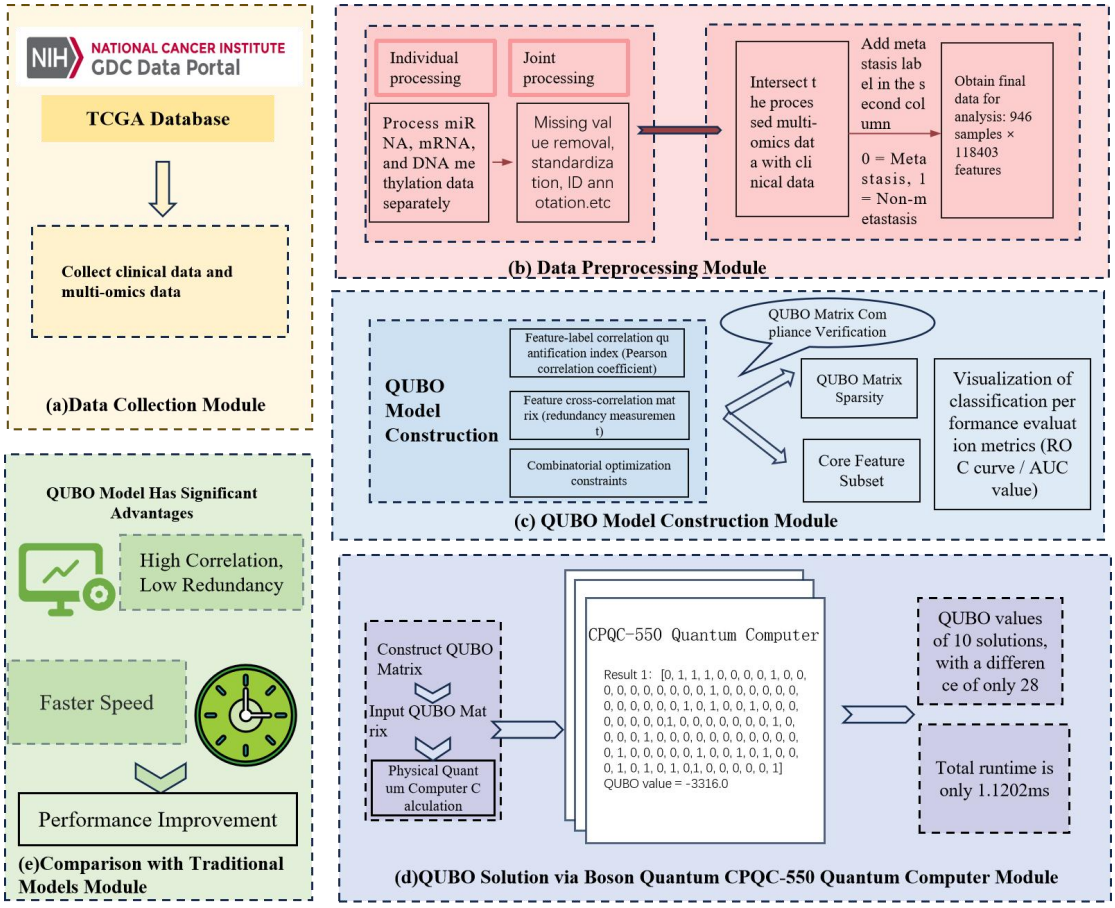

Figure 1: Workflow Diagram

V Data Collection and Preprocessing

5.1 Data collection

This study aims to construct a breast cancer (BRCA) metastasis prediction model based on multi-omics data. To achieve this goal, we collected clinical information from the PanCancer Atlas database of the International Cancer Genome Consortium (TCGA), along with multi-omics data including DNA methylation data, mRNA expression data, miRNA expression data, mutation data, and copy number data. The data preprocessing workflow of this study referred to the multi-omics data integration standards proposed by Ell Rott et al. (2025) [9], and their data processing approach provided direct reference for this study.

5.2 Data preprocessing

5.2.1 Overall process

All raw data were downloaded from the TCGA Pancancer Atlas (https://gdc.cancer.gov/about-data/publications/pancanatlas), with the initial clinical dataset comprising **10956** samples. Based on the collected data, we first performed independent processing and normalization for each omics dataset. Subsequently, we screened samples of the target cancer type (breast cancer BRCA) from the clinical master table. Finally, through the intersection of sample IDs, the processed multi-omics data were matched and integrated with the target samples.

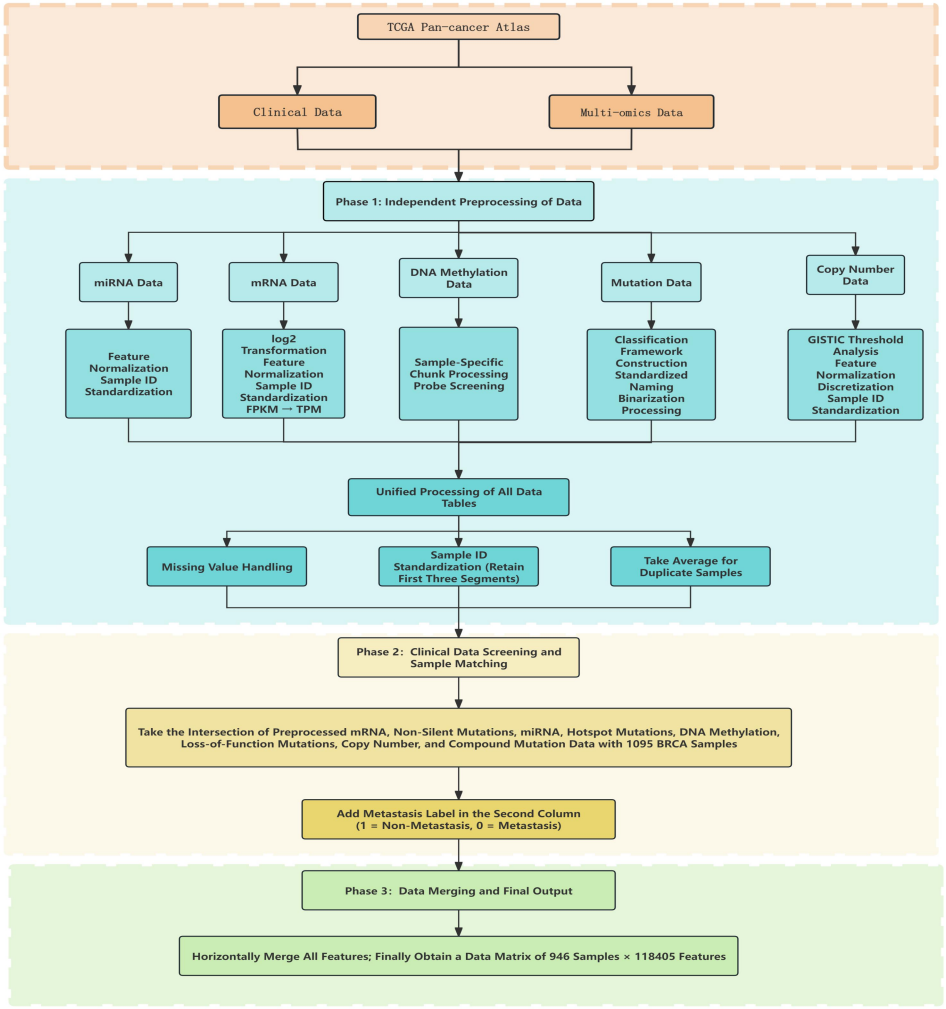

Figure 2: Overall Workflow of Data Preprocessing

5.2.2 Independent preprocessing of multi-omics data

Prior to sample screening, we performed standardization and feature construction on the downloaded omics data files of various types.

mRNA expression data: The original FPKM values were log2-transformed to correct for distribution, and standardized gene features were labeled with the unified prefix "C:GEXP:" where "C" indicates continuous data. Subsequently, expression levels were converted to TPM, and records of unknown genes were removed.

miRNA expression data: Using a similar workflow, features were normalized and labeled with the prefix "C:MIR:".

DNA methylation data: First, distinguish tumor samples from normal samples (ID containing "11A"). To control computational load, a block processing strategy was adopted: first, screen for low-methylation, non-tissue-specific probes in normal samples, then identify high-frequency, high-methylation probes in tumor samples. Finally, map probes to genes using probe gene annotation files, labeled in the format "C:METH:gene name".

Somatic mutation data: Based on the functional impact of mutations, four types of binary features were constructed: non-silent mutations, hotspot mutations, loss-of-function mutations, and compound mutations, each labeled with standardized prefixes (e.g., "B:MUTA:"). The "B" represents binary values, and different mutation types are distinguished by unique coding. For example, non-silent mutations are labeled as "B:MUTA:nons:gene name".

Copy number variation (CNV) data: Based on GISTIC analysis results, cancer-associated genes were screened and standardized with the notation "C:CNVR:". The continuous copy number status was discretized into three categories: deletion (1), normal (0), and amplification (+1).

During data processing, all sample IDs were standardized to a uniform format (retaining the first three parts), and the mean value was calculated for duplicate samples.

5.2.3 Target Sample Screening and Data Integration

After independent preprocessing of omics data in each group, sample-level integration was performed:

1. Target cohort extraction: From the clinical master list containing **10956** samples, breast cancer patients with the cancer type code "BRCA" were extracted, yielding a total of **1099** samples.

2. Clinical label processing: In this cohort, samples were labeled according to metastatic status: **733** metastatic samples and **362** non-metastatic samples. Four samples with unknown metastatic status were excluded, resulting in a final target sample pool of 1095 cases.

3. Matching and Intersection Screening: The IDs of the aforementioned 1095 BRCA samples were matched with eight pre-processed omics data tables to identify the overlapping samples present in all data tables. This step ensures that each ultimately selected sample has complete data across all omics dimensions.

4. Construction of the final dataset: Based on the obtained **946** overlapping samples, eight corresponding single-omics data subsets were generated through clipping. Subsequently, these subsets were horizontally merged in the feature dimension to form a unified multi-omics integrated data matrix for subsequent modeling analysis.

5.2.4 Missing Value Handling

To enhance data quality, missing values were processed both before and after integration. An iterative dual-screening strategy was employed: two approaches were compare – "eliminate high-missing samples first, then high-missing features" versus "eliminate high-missing features first, then high-missing samples" – and the approach with the highest final retained information (features × samples) was selected. Subsequently, all remaining missing values were removed.

5.3 Final Data Overview

Through the aforementioned workflow, we constructed a high-quality, multi-dimensional, and sample-aligned multi-omics dataset for breast cancer research. This dataset encompasses multiple biological dimensions, including gene expression, methylation, mutations, and copy number variations, comprising a total of 946 samples with definitive metastatic labels. All data processing and analysis were performed using Python scripts, ensuring automation and reproducibility of the workflow. This provides a reliable data foundation for subsequent feature selection and quantum-optimized modeling.

VI QUBO model

6.1 Model principle

First, set the binary variable:

$$x_i \in \{0,1\}, i = 1,2,...,n \tag{2}$$

among $x_i = 1$ indicates the selection of the i-th feature $x_i = 0$ It indicates that the i-th feature is not selected, where n denotes the size of the considered feature subset.

During feature selection, we should maximize the correlation between features and labels (breast cancer metastasis status):

$$f_{corr}(x) = \sum_{i=1}^{n} r_i x_i \tag{3}$$

Transform it into a minimization problem based on the characteristics of the Qubo model:

$$-\lambda_1 \sum_{i=1}^{n} r_i x_i \tag{4}$$

The correlation coefficient is r^i:

$$r_i = corr(f_i, y) \tag{5}$$

here f_i The i-th feature is represented, and y indicates the transition state (0 for transition, 1 for no transition).

When two or more features are highly correlated, they provide similar information. To reduce multicollinearity, we should minimize the redundancy among selected features by setting a redundancy penalty term:

$$f_{red}(x) = \lambda_2 \sum_{i=1}^{n} \sum_{j>i}^{n} R_{ij} x_i x_j \tag{6}$$

R_{ij}For the redundancy matrix:

$$R_{ij} = \left| \rho(f_i, f_j) \right|, i \neq j \tag{7}$$

among $\rho(f_i, f_j)$ as a feature f_i with features f_j The correlation coefficient between them.

Set feature quantity constraints to encourage selection first k_{target} feature:

$$f_{target}(x) = \alpha_{target} (\sum_{i=1}^{n} x_i - k_{target})^2 \tag{8}$$

When expanded:

$$f_{target}(x) = \alpha_{target} \left[\sum_{i=1}^{n} x_i^2 + 2 \sum_{i=1}^{n} \sum_{j>i}^{n} x_i x_j - 2k_{target} \sum_{i=1}^{n} x_i + k_{target}^2 \right] \tag{9}$$

also because $x_i = x_i^2$ Binary variable), so:

$$f_{target}(x) = \alpha_{target} \left[(1 - 2k_{target}) \sum_{i=1}^{n} x_i + 2 \sum_{i=1}^{n} \sum_{j>i}^{n} x_i x_j + k_{target}^2 \right] \tag{10}$$

Punish the selection of exceeding k_{target} with a characteristic;

$$f_{\max}(x) = \alpha_{\max} \left[(1 - 2k_{\max}) \sum_{i=1}^{n} x_i + 2 \sum_{i=1}^{n} \sum_{j>i}^{n} x_i x_j + k_{\max}^2 \right] \tag{11}$$

Finally, the complete objective function of the QUBO model is established:

$$\begin{aligned}
&\min E(x) = E_1(x) + E_2(x) + E_3(x) + E_4(x) \\
&E_1(x) = -\lambda_1 \sum_{i=1}^{n} r_i x_i \\
&E_2(x) = \lambda_2 \sum_{i=1}^{n} \sum_{j>i}^{n} R_{ij} x_i x_j \\
&E_3(x) = \alpha_{target} \left[(1 - 2k_{target}) \sum_{i=1}^{n} x_i + 2 \sum_{i=1}^{n} \sum_{j>i}^{n} x_i x_j \right] \\
&E_4(x) = \alpha_{\max} \left[(1 - 2k_{\max}) \sum_{i=1}^{n} x_i + 2 \sum_{i=1}^{n} \sum_{j>i}^{n} x_i x_j \right]
\end{aligned} \tag{12}$$

constant term k_{target} And $k_{\max}$ It can be ignored in optimization.

You can directly fill the QUBO matrix with the objective function. Diagonal elements (linear terms):

$$Q_{ii} = -\lambda_1 r_i + \alpha_{target}(1 - 2k_{target}) + \alpha_{\max}(1 - 2k_{\max}) \tag{13}$$

Non-diagonal elements (quadratic terms):

$$Q_{ij} = Q_{ji} = \lambda_2 R_{ij} + 2\alpha_{target} + 2\alpha_{\max}, i < j \tag{14}$$

The parameter values depend on the data scale.

Table 1: Parameter Values Table

Feature Number	λ_1	λ_2	α_{target}	$\alpha_{\max}$	k_{target}	$k_{\max}$
>1000	5	2	1	5	20	30
500-1000	3	1.5	1	5	15	25
200-500	2	1	1	5	12	20
<=200	1.5	0.8	1	5	10	15

The solution of this model will give the optimal feature selection.

6.2 Model results

We constructed a sparse matrix using the top 100 most correlated features, generated the QUBO matrix, and uploaded it to the **CPQC-550** hardware for

computation. The 20 core features were selected and analyzed using a logistic regression classifier in machine learning, yielding an F1 score of 0.8235, precision of 0.8090, recall rate of 0.8385, and AUC value of 0.8099. Among the selected 20 core features, the gene *hsa-miR-30e-5p* exhibited the highest positive correlation coefficient of 0.2323, while the gene *hsa-miR-330-5p* showed the highest negative correlation coefficient of -0.1962.

Given the large scale of our feature data, 100 features were insufficient to represent the entire dataset. We therefore screened the **top 10,000** most correlated features to build the QUBO model, which was solved using the open-source SA solver. The results were analyzed with a logistic regression classifier, ultimately identifying 544 core features. Post-solution analysis yielded an F1 score of 0.8619, precision of 0.7939, recall rate of 0.9427, AUC value of 0.8645, and a cross-validation F1 score of 0.8537.

6.3 Interpretation of result

The SA solver comprehensively captures complex molecular network interactions associated with breast cancer metastasis through its 10000-dimensional high-dimensional features, which is the key reason for its leading performance in core metrics such as accuracy, recall rate, and AUC. Notably, the recall rate improvement exceeds 10%, indicating that high-dimensional features can more accurately identify metastasis-related molecular signals, aligning with the clinical core requirement of "reducing missed diagnoses."

The quantum computer achieves over 80% of the comprehensive performance with just 100 features, while its accuracy marginally surpasses the SA solver. This demonstrates the efficiency of quantum annealing in filtering core features through quantum tunneling effects. The feature set exhibits low redundancy, enabling more focused regulation of the "main effect factor" – a key objective of the QUBO models redundant reduction design.

The SA solver employs a probabilistic search mechanism based on thermal

fluctuations, requiring iterative optimization in high-dimensional data to approach the optimal solution. In contrast, quantum computing leverages quantum superposition and tunneling effects to rapidly explore the solution space. However, current hardware limitations in feature dimensions prevent it from covering all transfer-related molecular patterns.

Table 2: Comparison Table of 10,000 Features vs. 100 Features

Metric	10000 feature（SA Optimizer）	100 feature（CPQC-550）	Diff	Diff%	Better
Accuracy	0.7958	0.7570	-0.0388	-4.9%	10000 feature
Precision	0.7939	0.8090	+0.0151	+1.9%	100 feature
Recall	0.9427	0.8385	-0.1042	-11.1%	10000 feature
F1-Score	0.8619	0.8235	-0.0384	-4.5%	10000 feature
AUC	0.8645	0.8099	-0.0546	-6.3%	10000 feature
OVERALL	0.8518	0.8076	-0.0442	-5.2%	10000 feature

VII Comparison of QUBO Model and Traditional Model

7.1 Conventional model

7.1.1 Basic Principles of Ordinary Logistic Regression

Logistic regression is a binary classification algorithm derived from linear

models. It works by using the Sigmoid activation function to transform linear feature combinations into probability values within the [0,1] range, enabling probabilistic modeling of sample categories. The algorithm optimizes model parameters through maximum likelihood estimation, ensuring the predicted probabilities best match the true labels (transferred=1, untransferred=0).

7.1.2 Analysis of the Ordinary Logistic Regression Result

The macro F1 score was 0.6452, and the weighted F1 score was 0.6879. The category 1 F1 score was 0.7664, indicating acceptable performance in positive sample recognition; however, the category 0 F1 score was only 0.5241, with both precision and recall rates for negative samples being relatively low. The absence of regularization constraints led to model overfitting due to multicollinearity among features, which compromised classification performance.

7.1.3 Basic Principles of Lasso Logistic Regression

Lasso logistic regression is a regularized enhancement of standard logistic regression, with its core innovation being the introduction of an L1 regularization term (L1 norm) in the loss function. By imposing sparse penalties on feature weights, it forces low-contribution features to have their weights compressed to zero, thereby achieving dual objectives of feature selection and model simplification. Essentially, it seeks an optimal balance between model fit and complexity, effectively mitigating overfitting issues in high-dimensional datasets.

7.1.4 Analysis of Lasso Logistic Regression Results

The macro F1 score is 0.7103 and the weighted F1 score is 0.7427, indicating good classification balance. By category, Category 1 (positive samples) achieves an F1 score of 0.8021, balancing precision and recall; Category 0 (negative samples) has an F1 score of 0.6186, slightly lower than positive samples but still maintaining

effective recognition capability. L1 regularization effectively reduces feature redundancy interference, which is the core reason for its excellent performance.

7.1.5 Basic Principles of naive Bayes

Simple Bayesian is a probabilistic classification model based on Bayes theorem and the assumption of "conditional independence of features." Given the category labels, all features are assumed to be independent, which significantly simplifies the computational complexity of joint probability calculations. The model first calculates the prior probabilities of each category and the conditional probabilities of features from the training set. It then applies Bayes theorem to compute the posterior probabilities of new samples belonging to different categories, ultimately selecting the category with the highest posterior probability as the prediction.

7.1.6 Analysis of Simple Bayesian Results

The macro F1 score is 0.5833, and the weighted F1 score is 0.6525, indicating a significant imbalance in classification performance. The F1 score for Category 1 is 0.7799, with a recall rate of 0.8490 (the highest among the three), but the precision rate is only 0.7212. The F1 score for Category 0 is merely 0.3867, and the recall rate is as low as 0.3152, posing an extremely high risk of missing negative samples. The core cause of its performance shortcomings lies in the mismatch between the assumption of feature condition independence and the actual feature correlations.

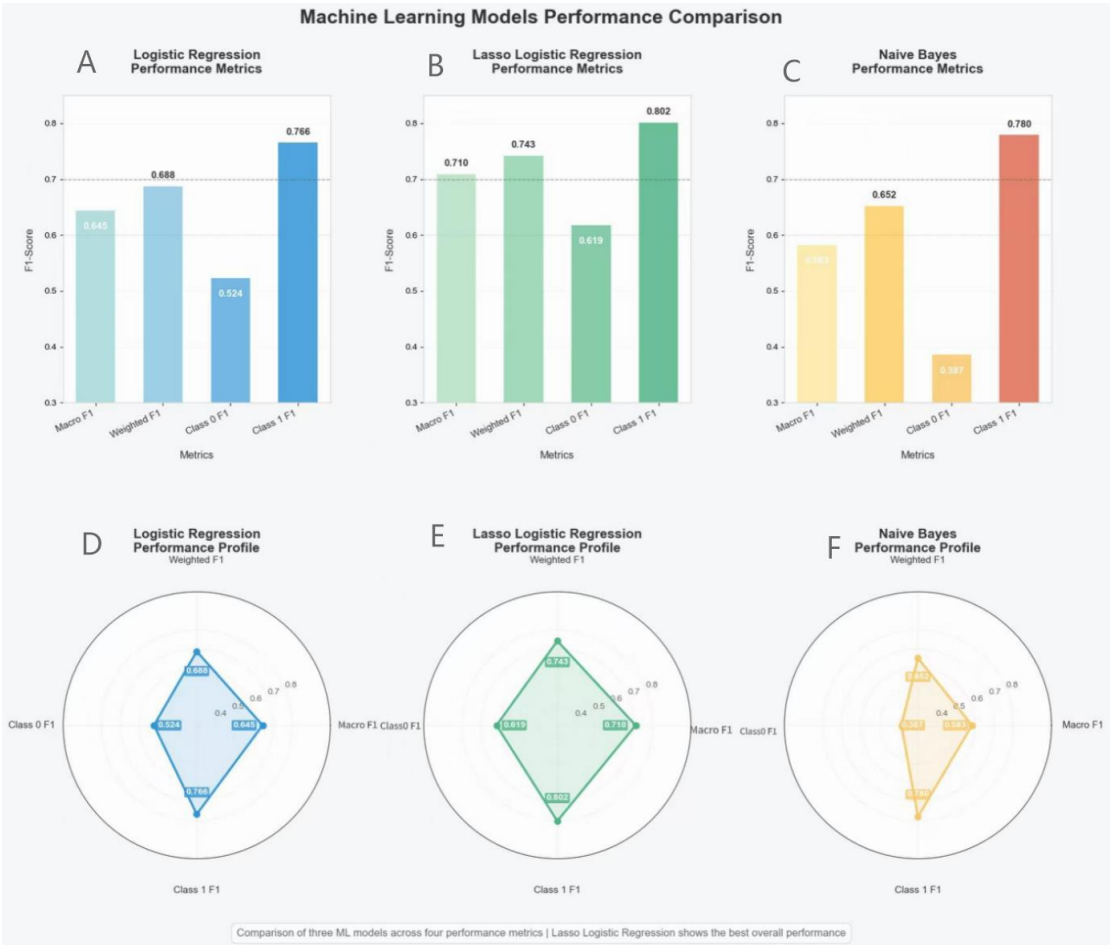

Figure 3: Performance of Three Traditional Models(Panels A–C (Bar Charts): Show scores of F1 metrics (Macro F1, Weighted F1, etc.) for each model. Panels D–F (Radar Charts): Display the performance profiles of the three models across the metrics.)

7.2 Advantages of QUBO Model

The Qubo model demonstrated superior performance across all core metrics: it achieved an accuracy of 0.796, an F1 score of 0.862, and an AUC of 0.865, all significantly higher than traditional models. The AUC value (0.865) approached the "excellent" threshold (0.9), indicating enhanced discrimination between positive and negative samples. The bubble representing Qubos performance (AUC=0.865) was positioned in the "high accuracy + high F1 score" region of the graph, whereas traditional models bubbles shifted toward the "low performance" area, demonstrating Qubos more balanced classification performance and generalization capability. Qubos cumulative score (2.52) was significantly higher than other models (Lasso Logistic:

2.25; Regular Logistic: 2.04; Naive Bayes: 1.85), further validating its comprehensive performance advantage.

Figure 4: Performance Comparison Between QUBO Model and Traditional Models（Panel A (Parallel Coordinates Plot): Shows trends of core metrics (Accuracy, F1-Score, AUC) across models to reflect relative performance. Panel B (Performance Heatmap): Uses color gradients to display metric scores for quick performance distinction. Panel C (Bubble Chart): Plots Accuracy vs. F1-Score, highlighting the QUBO model's "high dual-metric" advantage. Panel D (Stacked Bar Chart): Compares cumulative metric scores, showing the QUBO model's superior overall performance.)

Functional enrichment analysis of genes screened from the Qubo model revealed that selected genes such as ***SETX, MED22, GTF2A1, MED14, JUN, ZNHIT1, MED19, and MED18*** were significantly enriched in the process of **"Positive Regulation of DNA-templated Transcription Initiation (GO: 2000144)."** This indicates that these genes may synergistically enhance the transcriptional initiation efficiency of key genes, collectively driving the metastatic progression of cancer. Specifically, the core of this process involves activating or improving the efficiency of the first step in gene transcription from DNA to RNA. In the context of cancer metastasis, this often implies the abnormal activation of pro-metastatic gene programs.

The screened genes are precisely the key executors of this process.

Among them, ***JUN*** is a classic proto-oncogene, whose encoded c-Jun protein is a **core component of the transcription factor *AP-1***, capable of directly binding to DNA and initiating the transcription of a series of genes related to cell proliferation, invasion, and metastasis.[1] More importantly, multiple genes in the list belong to the subunits of the Mediator complex, including ***MED14, MED19, MED22, and MED18***. This complex serves as a critical bridge connecting gene-specific transcription factors (such as *AP-1*) with the basal transcription machinery (RNA polymerase II). Its hyperactivity significantly amplifies transcriptional signals. Studies have demonstrated that overexpression of *MED14* in breast cancer leads to hyperactivity of the Mediator complex, enhancing tumor initiation and metastasis capabilities.[2] ***MED19*** has been demonstrated to promote the growth and metastasis of osteosarcoma, and drives bone metastasis and invasion in bladder urothelial carcinoma through bone morphogenetic protein 2 (*BMP-2*).[3][4] The high expression of *MED22* in hepatocellular carcinoma (HCC) is associated with poor prognosis in patients, and it also holds prognostic value in renal cell carcinoma (RCC) and pancreatic cancer.[5][6] This highlights its critical role in these cancer progression processes. Recent studies have also revealed the pivotal role of the mediator complex itself in cancer metastasis, such as the discovery that its subunit *Med4* acts as a gatekeeper in the metastatic relapse of breast cancer.[7]

The remaining genes are also involved in the positive regulation of transcription initiation at various levels. ***SETX (senataxin)*** is a DNA/RNA helicase whose dysfunction can **affect genomic integrity and inflammatory responses**, thereby creating a microenvironment conducive to tumor development and metastasis.[8] ***GTF2A1*** is a subunit of the **universal transcription factor *TFIIA***, directly involved in the assembly and stabilization of the pre-transcriptional initiation complex. ***ZNHIT1***, on the other hand, i**s a chromatin remodeling factor that may assist transcription initiation by altering chromatin accessibility.**

These genes do not operate in isolation; they form a functional network. The core advantage of the QUBO model lies in its ability to more effectively capture complex,

nonlinear gene interaction patterns in multi-omics data, which are critical for complex biological processes such as cancer metastasis.

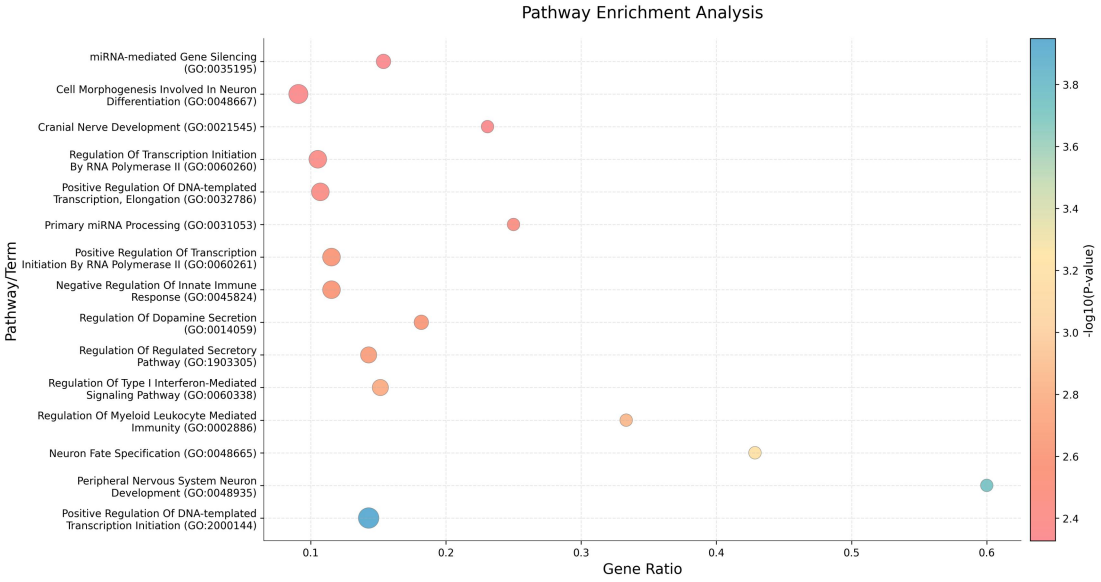

Figure 5: Pathway Enrichment Analysis

The horizontal axis represents gene enrichment ratio (Gene Ratio), while the vertical axis indicates pathway entries. The dot colors correspond to significance levels (-log10 (P-value)), with red indicating higher significance than blue.

VIII Conclusion

This study constructs an innovative and **translational-oriented systematic framework** integrating high-dimensional multi-omics data integration and fusion, quantum computing-driven optimization, and in-depth biological mechanism decipherment. Addressing the long-standing bottleneck issues in multi-omics data analysis within the field of cancer research – including high-dimensional redundancy, high heterogeneity interference, and low computational efficiency – empirical studies have confirmed that the quantum computing-based **QUBO model** not only exhibits **exceptional computational performance (millisecond-level convergence to stable solutions)** and advantages in high-dimensional feature screening but also breaks through the inherent limitations of traditional models in deciphering nonlinear gene

interaction networks. This framework not only provides a quantitative prediction tool with interpretability, operability, and high precision for the early accurate prediction of breast cancer metastasis – effectively bridging the **cognitive gap between molecular omics profiles and clinical decision-making practices** – but also offers a potential direction for paradigm innovation in precision oncology research. Its prominent **clinical translational value** is reflected in its ability to screen **core biomarkers** with clinical application potential (such as *MED* family members, *JUN* gene, etc.) and their synergistic regulatory molecular networks, while its favorable generalizability provides a transferable and reusable technical framework for multi-omics translational research in other cancer types. Furthermore, this study further demonstrates the **transformative application potential of quantum computing in the field of bioinformatics high-dimensional data processing**, pioneers new research pathways for multi-omics data mining in complex diseases, and accelerates the translational implementation process from omics data resources to personalized therapeutic strategies.

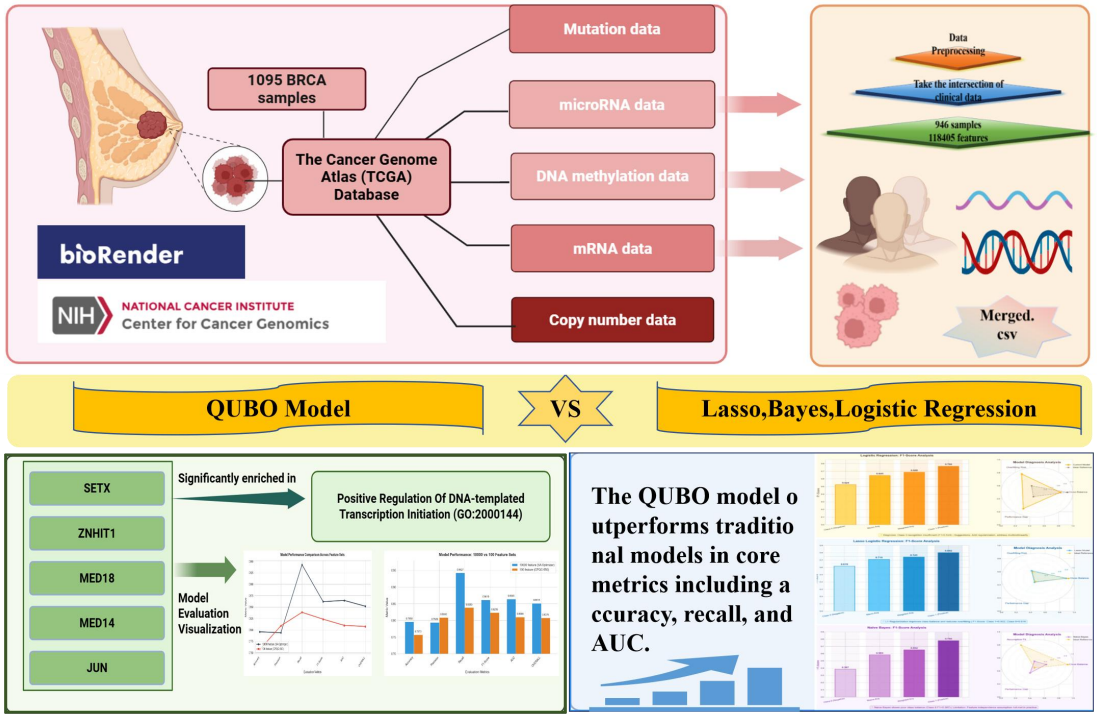

Figure 6:Work Summary

IX Reference

[1] Jafri Z, Li Y, Zhang J, OMeara CH, Khachigian LM. Jun, an Oncological Foe or Friend? Int J Mol Sci. 2025 Jan 10;26(2):555. doi: 10.3390/ijms26020555. PMID: 39859271; PMCID: PMC11766113.

[2] Richart L , Picod-Chedotel M L , Wassef M ,Richart L , Picod-Chedotel M L , Wassef M , et al.XIST loss impairs mammary stem cell differentiation and increases tumorigenicity through Mediator hyperactivation[J].Cell, 2022(12):185.DOI:10.2139/ssrn.3809998.

[3] Kent, W. J., Sugnet, C. W., Furey, T. S., Roskin, K. M., Pringle, T. H., Zahler, A. M., & Haussler, D. Kent, W. J., Sugnet, C. W., Furey, T. S., Roskin, K. M., Pringle, T. H., Zahler, A. M., & Haussler, D. (2002). The human genome browser at UCSC. Genome Research, 12(6), 996-1006.https://doi.org/10.1101/gr.229102

[4] UCSC Genome Browser. (n.d.). Human MED19 (transcript: uc001nlb.3) [GRCh38/hg38].http://genome.ucsc.edu . Accessed December 25, 2025.

[5] Uhlén, M., Fagerberg, L., Hallström, B. M., et al. (2015). Tissue - based map of the human proteome. Science, 347 (6220), 1260419.https://doi.org/10.1126/science.1260419

[6] The Human Protein Atlas. (n.d.). Expression of MED22 in cancer - Summary.https://www.proteinatlas.org/ENSG00000134057-MED22/pathology . Accessed December 25, 2025.

[7] Bae SY, Ling HH, Chen Y, Chen H, Kumar D, Zhang J, Viny AD, DePinho RA, Giancotti FG. Mediator Subunit Med4 Enforces Metastatic Dormancy in Breast Cancer. bioRxiv [Preprint]. 2024 Nov 1:2023.11.18.566087. doi: 10.1101/2023.11.18.566087. PMID: 38014033; PMCID: PMC10680920.

[8] Xu TN, Guo YJ, Pan WW. Impact of R-loops on Genomic Stability[J]. Chinese Journal of Biochemistry and Molecular Biology, 2023(09):1238-1246.

[9] Ellrott K, Wong CK, Yau C, Castro MAA, Lee JA, Karlberg BJ, Grewal JK, Lagani V, Tercan B, Friedl V, Hinoue T, Uzunangelov V, Westlake L, Loinaz X, Felau

I, Wang PI, Kemal A, Caesar-Johnson SJ, Shmulevich I, Lazar AJ, Tsamardinos I, Hoadley KA; Cancer Genome Atlas Analysis Network; Robertson AG, Knijnenburg TA, Benz CC, Stuart JM, Zenklusen JC, Cherniack AD, Laird PW. Classification of non-TCGA cancer samples to TCGA molecular subtypes using compact feature sets. Cancer Cell. 2025 Feb 10;43(2):195-212.e11. doi: 10.1016/j.ccell.2024.12.002. Epub 2025 Jan 2. PMID: 39753139; PMCID: PMC11949768.

X Appendix

Qubo model

```
import kaiwu.license as kw_license
# Obtain from: visit the Kaiwu platform or contact technical support
user_id = "128639423083503618"
sdk_code = "a5VtdXcWN04XihYj7KVc7l9qCa55fv"
kw_license.init(user_id, sdk_code)
import pandas as pd
import numpy as np
import kaiwu as kw
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score, roc_auc_score
from sklearn.linear_model import LogisticRegression
import warnings
import time
warnings.filterwarnings('ignore')

# Record start time
start_time = time.time()

# 1. Load data
print("="*80)
print("Step 1: Loading data...")
print("="*80)

# Ensure the file path is correct
try:
    df = pd.read_csv('~/brca_merged_final.csv')  # replace with the
actual filename
    print(f" Data loaded successfully! Shape: {df.shape}")
except FileNotFoundError:
    # If the file does not exist, create a sample dataset for testing
    print(" File not found, generating synthetic data for
demonstration...")
    np.random.seed(42)
    n_samples = 200
    n_features = 500  # Increase feature count to simulate realistic
scenario
```

```
    # Create discriminative features: metastatic and non-metastatic
samples have different distributions
    data = {
        'Sample_ID': [f'Sample_{i:03d}' for i in range(n_samples)],
        'Metastasis_Status': np.random.choice([0, 1], size=n_samples,
p=[0.4, 0.6])
    }

    # Add discriminative features
    labels = data['Metastasis_Status']
    for i in range(n_features):
        if i < 50:  # First 50 features have strong discriminative power
            # Metastatic and non-metastatic have different means
            mean_0 = np.random.uniform(-1, 0)
            mean_1 = np.random.uniform(0, 1)
            feature_values = np.where(labels == 0,
                                      np.random.normal(mean_0, 0.8,
n_samples),
                                      np.random.normal(mean_1, 0.8,
n_samples))
        elif i < 100:  # Next 50 features have moderate discriminative
power
            mean_0 = np.random.uniform(-0.5, 0)
            mean_1 = np.random.uniform(0, 0.5)
            feature_values = np.where(labels == 0,
                                      np.random.normal(mean_0, 1,
n_samples),
                                      np.random.normal(mean_1, 1,
n_samples))
        else:  # Other features are random noise
            feature_values = np.random.randn(n_samples)

        data[f'Gene_{i:04d}'] = feature_values

    df = pd.DataFrame(data)
    print(f" Synthetic data creation complete: {df.shape}")
    print("  First 50 features strong signal, 50-100 moderate signal,
rest random noise")

print(f" Data preview - first 5 rows:")
print(df.head())
print(f"\n First 10 columns: {df.columns.tolist()[:10]}...")
print(f"Total columns: {len(df.columns)}")
```

```
# 2. Prepare data
print("\n" + "="*80)
print("Step 2: Data preparation...")
print("="*80)

# Check required columns exist
required_columns = ['Sample_ID', 'Metastasis_Status']
if 'Sample_ID' not in df.columns or 'Metastasis_Status' not in
df.columns:
    print(" File not found, attempting auto-detection...")
    # Assume first column is ID and second is the label
    df.columns = ['Sample_ID', 'Metastasis_Status'] + [f'Feature_{i}'
for i in range(len(df.columns)-2)]
    print(f" Columns renamed")

# Separate labels and features
sample_ids = df['Sample_ID']
labels = df['Metastasis_Status']  # 0: metastatic, 1: non-metastatic
features = df.drop(['Sample_ID', 'Metastasis_Status'], axis=1)

print(f"  Data statistics:")
print(f"  Samples: {len(sample_ids)}")
print(f"  Features: {len(features.columns)}")
print(f"  Label distribution:")
label_counts = labels.value_counts()
for value, count in label_counts.items():
    status = "Metastatic" if value == 0 else "Non-metastatic"
    percentage = count / len(labels) * 100
    print(f"    {status} ({value}): {count} samples
({percentage:.1f}%)")

# 3. Use all features
print("\n" + "="*80)
print("Step 3: Using all features for selection...")
print("="*80)

feature_names = features.columns.tolist()
n_features_total = len(feature_names)
print(f" Using all {n_features_total} features")
print(f" Example feature names:")
for i, name in enumerate(feature_names[:5]):
    print(f"  {i}. {name}")
print(f"  ... total {len(feature_names)} features")
```

```
# 4. Data standardization
print("\n" + "="*80)
print("Step 4: Data standardization...")
print("="*80)

scaler = StandardScaler()
features_scaled = pd.DataFrame(
    scaler.fit_transform(features),
    columns=feature_names
)
print(" Data standardization complete")

# 5. Compute feature-label correlations
print("\n" + "="*80)
print("Step 5: Compute feature-label correlations...")
print("="*80)

print("Computing correlations for all features and labels, this may
take a while...")
start_corr = time.time()

# Compute correlations in batches to avoid memory issues
batch_size = 1000
correlations = []
n_batches = (n_features_total + batch_size - 1) // batch_size

for batch in range(n_batches):
    start_idx = batch * batch_size
    end_idx = min((batch + 1) * batch_size, n_features_total)
    batch_features = features_scaled.iloc[:, start_idx:end_idx]

    for col in batch_features.columns:
        corr = np.corrcoef(features_scaled[col], labels)[0, 1]
        correlations.append(corr)

    progress = (batch + 1) / n_batches * 100
    print(f"  Progress: {progress:.1f}%
({end_idx}/{n_features_total})")

# Convert to a correlation coefficient matrix
correlation_series = pd.Series(correlations,
index=features_scaled.columns)
correlation_abs_series = correlation_series.abs()
```

```
corr_time = time.time() - start_corr
print(f"  Correlation computation complete, time: {corr_time:.2f}s")

print(f"  Correlation statistics:")
print(f"  Mean correlation: {correlation_series.mean():.4f}")
print(f"  Mean absolute correlation:
{correlation_abs_series.mean():.4f}")
print(f"  Max positive correlation: {correlation_series.max():.4f}
({correlation_series.idxmax()})")
print(f"  Max negative correlation: {correlation_series.min():.4f}
({correlation_series.idxmin()})")
# 5. Compute feature-label correlations (additional code)

# Count features in different correlation intervals
print("\n Correlation distribution stats:")
print("-" * 60)

# Define correlation thresholds
thresholds = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

for i in range(len(thresholds)-1):
    lower = thresholds[i]
    upper = thresholds[i+1]

    # Count features with absolute correlation in this interval
    count = np.sum((correlation_abs_series >= lower) &
(correlation_abs_series < upper))
    percentage = count / n_features_total * 100

    # Only print intervals with lower bound >= 0.1
    if lower >= 0.1:
        print(f"  Correlation {lower:.1f}-{upper:.1f}: {count:5d}
features ({percentage:5.1f}%)")
    else:
        print(f"  Correlation {lower:.1f}-{upper:.1f}: {count:5d}
features ({percentage:5.1f}%)")
# Individually count the features with an absolute correlation greater
than 0.1
corr_gt_01 = np.sum(correlation_abs_series > 0.1)
corr_gt_01_percent = corr_gt_01 / n_features_total * 100

print("\n" + "="*60)
print(f" Key statistics:")
```

```
print(f"  Features with |correlation| > 0.1: {corr_gt_01:,}
({corr_gt_01_percent:.1f}% of all features)")
print(f"  Features with |correlation| > 0.2:
{np.sum(correlation_abs_series > 0.2):,}")
print(f"  Features with |correlation| > 0.3:
{np.sum(correlation_abs_series > 0.3):,}")
print(f"  Features with |correlation| > 0.4:
{np.sum(correlation_abs_series > 0.4):,}")
df = pd.DataFrame(Q_adjust_matrix)
# Save to CSV file
df.to_csv('Qubo_matrix.csv', index=False, header=False)
# 6. Build QUBO matrix using kaiwu.qubo.QuboModel
print("\n" + "="*80)
print("Step 6: Build QUBO matrix using kaiwu.qubo.QuboModel...")
print("="*80)

# Set target feature counts (adjust based on total features)
if n_features_total > 1000:
    k_target = 20
    k_max = 30
    top_features =10000
elif n_features_total > 500:
    k_target = 15
    k_max = 25
    top_features = 150
elif n_features_total > 200:
    k_target = 12
    k_max = 20
    top_features = 100
else:
    k_target = 10
    k_max = 15
    top_features = 50

# Set bit width parameter
bit_width = 8  # target bit width

print(f"Adjust parameters based on total features
({n_features_total}):")
print(f"  Target features (k_target): {k_target}")
print(f"  Max features (k_max): {k_max}")
print(f"  Top N correlated features to consider: {top_features}")
print(f"  Target bit width: {bit_width} bits")
```

```
# Compute redundancy among features (pairwise correlations) - only
compute for top_features
print(f"\nComputing redundancy among top {top_features} correlated
features...")
start_redundancy = time.time()

# Get indices of top correlated features
top_corr_indices = correlation_abs_series.nlargest(min(top_features,
len(correlation_abs_series))).index
top_features_scaled = features_scaled[top_corr_indices]

# Compute correlation matrix
feature_correlation_subset = top_features_scaled.corr().values
feature_labels_correlation_subset =
correlation_series[top_corr_indices].values

# Build redundancy matrix (absolute values)
redundancy_matrix_subset = np.abs(feature_correlation_subset)
np.fill_diagonal(redundancy_matrix_subset, 0)  # set diagonal to 0

redundancy_time = time.time() - start_redundancy
print(f"✓ Redundancy matrix computed, time: {redundancy_time:.2f}s")

# Set QUBO weight parameters
# Adjust weights based on data scale
if  n_features_total > 1000:
    lambda1 = 5.0  # correlation weight
    lambda2 = 2.0
elif n_features_total > 500:
    lambda1 = 3.0
    lambda2 = 1.5
elif n_features_total > 200:
    lambda1 = 2.0
    lambda2 = 1.0
else:
    lambda1 = 1.5
    lambda2 = 0.8

print(f"  QUBO parameters:")
print(f"  Correlation weight (lambda1): {lambda1}")
print(f"  Redundancy penalty weight (lambda2): {lambda2}")
print(f"  Target selected features (k_target): {k_target}")
print(f"  Max allowed features (k_max): {k_max}")
```

```
# 7. Directly construct QUBO matrix
print("\n" + "="*80)
print("Step 7: Directly construct QUBO matrix...")
print("="*80)

start_qubo = time.time()

# Get number of subset features
n_subset = len(top_corr_indices)
print(f"Constructing {n_subset}x{n_subset} QUBO matrix...")

# ==================== Direct QUBO build ====================
print("  Building QUBO matrix...")

# Initialize QUBO matrix
Q_matrix = np.zeros((n_subset, n_subset))

# 1. Add linear terms: maximize correlation (minimize negative
correlation)
for i in range(n_subset):
    # Maximizing correlation is equivalent to minimizing negative
correlation
    Q_matrix[i, i] = -lambda1 * feature_labels_correlation_subset[i]

# 2. Add quadratic terms: redundancy penalty
for i in range(n_subset):
    for j in range(i+1, n_subset):
        redundancy = redundancy_matrix_subset[i, j]
        weight = lambda2 * redundancy
        Q_matrix[i, j] = weight
        Q_matrix[j, i] = weight

# 3. Add feature count constraint
# Constraint 1: encourage selecting k_target features, penalty: 1.0
* (Σ x_i - k_target)^2
# (Σ x_i)^2 - 2*k_target*Σ x_i + k_target^2
# (Σ x_i)^2 = Σ x_i^2 + 2*Σ_{i<j} x_i*x_j

# Diagonal term: (1 - 2*k_target)
# Off-diagonal term: 2
penalty_target = 1.0
for i in range(n_subset):
    Q_matrix[i, i] += penalty_target * (1 - 2*k_target)
```

```
for i in range(n_subset):
    for j in range(i+1, n_subset):
        Q_matrix[i, j] += 2 * penalty_target
        Q_matrix[j, i] += 2 * penalty_target

# 4. Add max feature count constraint (high penalty)
# Constraint 2: penalize exceeding k_max features: 5.0 * (Σ x_i -
k_max)^2 (when exceeded)
# Note: this is a soft constraint; use a high penalty coefficient
penalty_max = 5.0
for i in range(n_subset):
    Q_matrix[i, i] += penalty_max * (1 - 2*k_max)

for i in range(n_subset):
    for j in range(i+1, n_subset):
        Q_matrix[i, j] += 2 * penalty_max
        Q_matrix[j, i] += 2 * penalty_max

print(f"  QUBO matrix construction complete")
print(f"  QUBO matrix properties:")
print(f"  Shape: {Q_matrix.shape}")
print(f"  Diagonal range: [{np.min(np.diag(Q_matrix)):.2f},
{np.max(np.diag(Q_matrix)):.2f}]")
print(f"  Mean diagonal: {np.mean(np.diag(Q_matrix)):.2f}")
print(f"  Sparsity: {(np.count_nonzero(Q_matrix) /
(n_subset*n_subset) * 100):.2f}%")
Q_matrix=kw.qubo.adjust_qubo_matrix_precision(Q_matrix,
bit_width=8)
# ==================== Create QUBO model ====================
print("\n  Creating QUBO model...")

# 1. Create QuboModel instance
model = kw.qubo.QuboModel()

# 2. Define binary variables
variables = [kw.core.Binary(f'x{i}') for i in range(n_subset)]
print(f"  Created {len(variables)} binary variables")

# 3. Create list of expression terms
terms = []

# 4. Add linear terms (diagonal)
print("  Adding linear terms (diagonal)...")
for i in range(n_subset):
```

```
    weight = Q_matrix[i, i]
    if abs(weight) > 1e-10:  # only add significant non-zero terms
        terms.append(weight * variables[i])

# 5. Add quadratic terms (off-diagonal elements)
print("  Adding quadratic terms (off-diagonal)...")
for i in range(n_subset):
    for j in range(i+1, n_subset):
        weight = Q_matrix[i, j]
        if abs(weight) > 1e-10:  # only add significant non-zero terms
            terms.append(weight * variables[i] * variables[j])

# 6. Set objective function
print("  Setting objective...")
objective = kw.core.quicksum(terms)
model.set_objective(objective)
print("✓ Objective set")

# 7. Add constraints (if necessary)
# Note: Since we have encoded the constraints into the Q matrix, there
is no need to add additional constraints.
print("  Checking constraints...")
print("   All constraints encoded into Q matrix; no additional
constraints needed")

# Get original feature indices
original_indices = [feature_names.index(feature) for feature in
top_corr_indices]

qubo_time = time.time() - start_qubo
print(f" QUBO matrix construction complete, time: {qubo_time:.2f}s")
# 8. Solve QUBO problem using kaiwu simulated annealing
print("\n" + "="*80)
print("Step 8: Solve QUBO problem using kaiwu simulated annealing...")
print("="*80)

start_solve = time.time()

print("Using kaiwu SimulatedAnnealingOptimizer solver...")

try:
    # Create simulated annealing optimizer
   # Configure real-device solver (call CIM hardware)
    kw.common.CheckpointManager.save_dir = '~/result/'
```

```
    optimizer =
kw.classical.SimulatedAnnealingOptimizer(initial_temperature=1e8,
                                                    alpha=0.9,
                                                    cutoff_temperatur
e=0.01,
                                                    iterations_per_t=
200,
                                                    size_limit=100)

    optimizer = kw.cim.PrecisionReducer(optimizer, 8)
    # Create the solver
    solver = kw.solver.SimpleSolver(optimizer)
    kw.common.CheckpointManager.save_dir = None
    num_solutions = 1
    all_solutions = []

    for run in range(num_solutions):
        try:
            sol_dict, qubo_value = solver.solve_qubo(model)

                # Convert solution dict to binary vector
            solution_vector = np.zeros(n_subset, dtype=int)
            for var_name, value in sol_dict.items():
                if var_name.startswith('x'):
                    idx = int(var_name[1:])
                    solution_vector[idx] = int(value)

            all_solutions.append(solution_vector)

        except Exception as inner_e:
            print(f"  Run {run+1} failed: {inner_e}")
                # Add random solution as fallback
            random_solution = np.random.choice([0, 1], size=n_subset)
            all_solutions.append(random_solution)

    solutions_array = np.array(all_solutions)

    solve_time = time.time() - start_solve
    print(f"   Solve completed! Obtained {len(solutions_array)}
solutions")
    print(f"   Solve time: {solve_time:.2f}s")

except Exception as e:
    print(f"Main solving method failed: {e}")
```

```
    # Fallback to simpler optimizer
    print("Trying fallback optimizer...")
    try:
        optimizer = kw.classical.TabuSearchOptimizer(
            max_iter=100,
            size_limit=5
        )
        solver = kw.solver.SimpleSolver(optimizer)

        num_solutions = 30
        all_solutions = []

        for run in range(num_solutions):
            try:
                sol_dict, qubo_value = solver.solve_qubo(model)

# Convert solution dict to binary vector
                solution_vector = np.zeros(n_subset, dtype=int)
                for var_name, value in sol_dict.items():
                    if var_name.startswith('x'):
                        idx = int(var_name[1:])
                        solution_vector[idx] = int(value)

                all_solutions.append(solution_vector)

            except Exception as inner_e:
                print(f"  Run {run+1} failed: {inner_e}")
                # Add random solution as fallback
                random_solution = np.random.choice([0, 1],
size=n_subset)
                all_solutions.append(random_solution)

        solutions_array = np.array(all_solutions)

        solve_time = time.time() - start_solve
        print(f"   Fallback solve completed! Obtained
{len(solutions_array)} solutions")
        print(f"   Solve time: {solve_time:.2f}s")

    except Exception as final_e:
        print(f"All solving methods failed: {final_e}")
        print("Using random solutions as fallback...")
```

```
        # Generate a random solution as a last resort
        solutions_array = np.random.choice([0, 1], size=(30,
n_subset))
        solve_time = time.time() - start_solve
        print(f"   Solve time: {solve_time:.2f}s")

# 9. Analyze results
print("\n" + "="*80)
print("Step 9: Analyze solutions...")
print("="*80)

def analyze_solutions(solutions, feature_names_subset,
original_indices,
                      k_target, k_max, correlation_series):
    """Analyze solutions from simulated annealing"""
    results = []

    for i, solution in enumerate(solutions):
        # Solution is already in 0/1 format
        qubo_solution = solution.astype(int)

        # Compute number of selected features
        selected_count = np.sum(qubo_solution)

        # Compute energy (objective value)
        energy = 0
        if len(solution) > 0:
            # Compute QUBO energy: x^T Q x

        # Compute penalty score for feature count (to assess constraint
satisfaction)
            penalty_score = 0
        if selected_count > k_max:
            penalty_score = 5.0 * (selected_count - k_max) ** 2
        elif selected_count > k_target:
            penalty_score = 2.0 * (selected_count - k_target) ** 2

        # Get selected feature names and original indices
        selected_subset_indices = np.where(qubo_solution == 1)[0]
        selected_features = [feature_names_subset[idx] for idx in
selected_subset_indices]
        selected_original_indices = [original_indices[idx] for idx in
selected_subset_indices]
```

```
        # Compute total correlation score
        correlation_score = 0
        if selected_features:
            for feature in selected_features:
                if feature in correlation_series.index:
                    correlation_score +=
abs(correlation_series[feature])

        results.append({
            'solution_id': i,
            'selected_count': selected_count,
            'energy': energy,
            'penalty_score': penalty_score,
            'total_score': energy + penalty_score,  # total score
(lower is better)
            'selected_features': selected_features,
            'selected_original_indices': selected_original_indices,
            'correlation_score': correlation_score,
            'avg_correlation': correlation_score / max(1,
len(selected_features))
        })

    return pd.DataFrame(results)

# Analyze all solutions
feature_names_subset = list(top_corr_indices)
results_df = analyze_solutions(solutions_array,
feature_names_subset, original_indices,
                               k_target, k_max, correlation_series)

print(f"  Solution analysis results:")
print(f"  Number of solutions: {len(results_df)}")
if len(results_df) > 0:
    print(f"  Selected features range:
{results_df['selected_count'].min()} -
{results_df['selected_count'].max()}")
    print(f"  Average selected features:
{results_df['selected_count'].mean():.1f} ±
{results_df['selected_count'].std():.1f}")
    print(f"  Target features: {k_target}, Max features: {k_max}")


# 10. Select best solution
print("\n" + "="*80)
print("Step 10: Select best solution...")
```

```
print("="*80)

# Select the optimal solution (based on the total score)
if not results_df.empty and 'total_score' in results_df.columns:
    # Prioritize solutions that fall within the constraints
    valid_df = results_df[results_df['selected_count'] <= k_max]

    if not valid_df.empty:
        # Select the solution with the lowest total score among those
that fall within the constraints
        best_idx = valid_df['total_score'].idxmin()
        best_solution = valid_df.loc[best_idx]
        selection_criteria = "The lowest total score within the
constraints"
    else:
        # If no solution is within the constraints, use the one with
the lowest total score
        best_idx = results_df['total_score'].idxmin()
        best_solution = results_df.loc[best_idx]
        selection_criteria = "The lowest total score among all
solutions"

    print(f"  Best solution selection criterion:
{selection_criteria}")
    print(f"  Solution index: {best_idx}")
    print(f"  Total score: {best_solution['total_score']:.4f}")
    print(f"  QUBO energy: {best_solution['energy']:.4f}")
    print(f"  Constraint penalty:
{best_solution['penalty_score']:.4f}")
    print(f"  Selected features: {best_solution['selected_count']}
(target: {k_target}, max: {k_max})")

    selected_indices =
best_solution.get('selected_original_indices', [])
    selected_features = best_solution.get('selected_features', [])
else:
    print(" No valid solution found")
    selected_indices = []
    selected_features = []

# 11. Show selected features
print("\n" + "="*80)
print("Step 11: Show selected features and their correlation with the
label...")
```

```
print("="*80)

if len(selected_features) > 0:
    print(f" Selected {len(selected_features)} features (target:
{k_target}, max: {k_max}):")

    # Collect selected features' correlations and sort
    selected_correlations = []
    for feature in selected_features:
        if feature in correlation_series.index:
            corr = correlation_series[feature]
            selected_correlations.append((feature, corr))

    if len(selected_correlations) > 0:
        selected_correlations.sort(key=lambda x: abs(x[1]),
reverse=True)

        print(f"\n Selected features sorted by correlation:")
        display_count = min(10, len(selected_correlations))
        for i, (feature, corr) in
enumerate(selected_correlations[:display_count], 1):
            corr_type = "Positive" if corr > 0 else "Negative"
            significance = ""
            if abs(corr) > 0.5:
                significance = "  Strong correlation"
            elif abs(corr) > 0.3:
                significance = "  Moderate correlation"
            print(f"  {i:2d}. {feature:30s} ({corr_type}:
{abs(corr):.4f}){significance}")

        # Statistical information
        selected_corr_values = [corr for _, corr in
selected_correlations]
        print(f"\n Selected features statistics:")
        print(f"  Mean absolute correlation:
{np.mean(np.abs(selected_corr_values)):.4f}")
        print(f"  Median absolute correlation:
{np.median(np.abs(selected_corr_values)):.4f}")
        print(f"  Strong correlated features (abs>0.5): {sum(1 for
corr in selected_corr_values if abs(corr) > 0.5)}")
        print(f"  Moderate correlated features (0.3<abs<=0.5): {sum(1
for corr in selected_corr_values if 0.3 < abs(corr) <= 0.5)}")
else:
    print(" Warning: No features selected!")
```

```
# 12. Machine learning evaluation
print("\n" + "="*80)
print("Step 12: Machine learning model evaluation...")
print("="*80)

# Prepare evaluation data
X = features_scaled.values
y = labels.values

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

print(f" Data split:")
print(f"  Train: {X_train.shape[0]} samples, {X_train.shape[1]}
features")
print(f"  Test: {X_test.shape[0]} samples, {X_test.shape[1]}
features")
import pandas as pd
import numpy as np
import kaiwu as kw
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV,
StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score, roc_auc_score, confusion_matrix,
classification_report
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
import warnings
import time
import matplotlib.pyplot as plt
import seaborn as sns
warnings.filterwarnings('ignore')

# Set fonts and plotting style (include Chinese fonts if available)
plt.rcParams['font.sans-serif'] = ['SimHei', 'DejaVu Sans']
plt.rcParams['axes.unicode_minus'] = False
sns.set_style("whitegrid")

# Define evaluation function with grid search and cross-validation
```

```
def evaluate_features_with_gridsearch(X_train, X_test, y_train,
y_test, feature_indices,
                                      method_name="Feature selection
method"):
    """Evaluate selected features using grid search and
cross-validation"""

    print(f"\n Evaluation method: {method_name}")
    print("-" * 60)

    if feature_indices is None or len(feature_indices) == 0:
        print(" Warning: No valid feature indices, returning default
metrics")
        return {
            'method': method_name,
            'selected_features': 0,
            'best_params': None,
            'cv_score': 0,
            'cv_std': 0,
            'accuracy': 0,
            'precision': 0,
            'recall': 0,
            'f1_score': 0,
            'f1_score_0': 0,  # F1-score for the metastasis category
            'f1_score_1': 0,  # F1-score for the non-metastasis
category
            'roc_auc': 0,
            'specificity': 0,
            'training_time': 0
        }

    # Ensure feature_indices are valid
    valid_indices = []
    for idx in feature_indices:
        try:
            idx_int = int(idx)
            if 0 <= idx_int < X_train.shape[1]:
                valid_indices.append(idx_int)
        except:
            continue

    if len(valid_indices) == 0:
        print(f"If no valid feature indices exist, return the default
metrics")
```

```
        return {
            'method': method_name,
            'selected_features': 0,
            'best_params': None,
            'cv_score': 0,
            'cv_std': 0,
            'accuracy': 0,
            'precision': 0,
            'recall': 0,
            'f1_score': 0,
            'f1_score_0': 0,  # F1-score for the metastasis category
            'f1_score_1': 0,  # F1-score for the non-metastasis
category
            'roc_auc': 0,
            'specificity': 0,
            'training_time': 0
        }

    print(f"✓ Selected {len(valid_indices)} features")

    # Select features
    X_train_selected = X_train[:, valid_indices]
    X_test_selected = X_test[:, valid_indices]

    # Record start time
    start_time = time.time()

    try:
        # 1. Create a preprocessing and modeling pipeline
        pipeline = Pipeline([
            ('scaler', StandardScaler()),
            ('classifier', LogisticRegression(random_state=42,
max_iter=1000))
        ])

        # 2. Define parameter grid
        param_grid = {
            'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100],  #
regularization strength
            'classifier__penalty': ['l1', 'l2'],  # penalty type
            'classifier__solver': ['liblinear', 'saga'],  # solver
            'classifier__class_weight': [None, 'balanced']  #
class_weight
        }
```

```
        # 3. Set cross-validation strategy (5-fold stratified
cross-validation)
        cv_strategy = StratifiedKFold(n_splits=5, shuffle=True,
random_state=42)

        # 4. Perform grid search with cross-validation
        print("  Running grid search and cross-validation...")
        grid_search = GridSearchCV(
            pipeline,
            param_grid,
            cv=cv_strategy,
            scoring='f1',  # optimize F1 score
            n_jobs=-1,  # use all CPU cores
            verbose=0,
            refit=True  # refit on full training data with best params
        )

        # Train the model
        grid_search.fit(X_train_selected, y_train)

        training_time = time.time() - start_time
        print(f"  ✓ Grid search complete! Time:
{training_time:.2f}s")

        # 5. Print best parameters
        print(f"  Best parameters:")
        for param, value in grid_search.best_params_.items():
            param_name = param.replace('classifier__', '')
            print(f"    {param_name}: {value}")
        print(f"  Best cross-validation F1 score:
{grid_search.best_score_:.4f}")

        # 6. Use the best model for prediction
        best_model = grid_search.best_estimator_
        y_pred = best_model.predict(X_test_selected)
        y_pred_proba = best_model.predict_proba(X_test_selected)[:,
1]

        # 7. Calculate evaluation metrics
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average='binary',
zero_division=0)
```

```
        recall = recall_score(y_test, y_pred, average='binary',
zero_division=0)
        f1 = f1_score(y_test, y_pred, average='binary',
zero_division=0)

        # Calculate F1 score for each class
        f1_per_class = f1_score(y_test, y_pred, average=None,
zero_division=0)
        if len(f1_per_class) == 2:
            f1_0 = f1_per_class[0]  # F1-score for the metastasis
category
            f1_1 = f1_per_class[1]  # F1-score for the non-metastasis
category
        else:
            f1_0 = 0
            f1_1 = f1

        roc_auc = roc_auc_score(y_test, y_pred_proba)

        # Calculate specificity
        cm = confusion_matrix(y_test, y_pred)
        tn, fp, fn, tp = cm.ravel()
        specificity = tn / (tn + fp) if (tn + fp) > 0 else 0

        # Obtain the standard deviation of cross-validation scores
        cv_results = grid_search.cv_results_
        best_index = grid_search.best_index_
        cv_std = cv_results['std_test_score'][best_index]

        # Print test set performance
        print(f"  Test set performance:")
        print(f"    Accuracy: {accuracy:.4f}")
        print(f"    Precision: {precision:.4f}")
        print(f"    Recall: {recall:.4f}")
        print(f"    F1 score (overall): {f1:.4f}")
        print(f"    F1 score (Metastatic/0): {f1_0:.4f}")
        print(f"    F1 score (Non-metastatic/1): {f1_1:.4f}")
        print(f"    Specificity: {specificity:.4f}")
        print(f"    AUC: {roc_auc:.4f}")

        # Print classification report
        print(f"\n  Detailed classification report:")
        print(classification_report(y_test, y_pred,
target_names=['Metastatic (0)', 'Non-metastatic (1)']))
```

```
        # 8. Create a results dictionary
        metrics = {
            'method': method_name,
            'selected_features': len(valid_indices),
            'best_params': str(grid_search.best_params_),
            'cv_score': grid_search.best_score_,  # cross-validation
F1 score
            'cv_std': cv_std,  # cross-validation std
            'accuracy': accuracy,
            'precision': precision,
            'recall': recall,
            'f1_score': f1,
            'f1_score_0': f1_0,  # F1-score for the metastasis
category
            'f1_score_1': f1_1,  # F1-score for the non-metastasis
category
            'roc_auc': roc_auc,
            'specificity': specificity,
            'training_time': training_time,
            'best_model': best_model,
            'grid_search': grid_search
        }

        return metrics

    except Exception as e:
        print(f"  Evaluation error: {e}")
        import traceback
        traceback.print_exc()

        # If grid search fails, fall back to simple logistic regression
        print("  Grid search failed, falling back to simple logistic
regression...")
        try:
            # Fall back to simple logistic regression
            simple_model = LogisticRegression(
                C=1.0,
                penalty='l2',
                solver='liblinear',
                max_iter=1000,
                random_state=42
            )
```

```
            # Standardize data
            scaler = StandardScaler()
            X_train_scaled = scaler.fit_transform(X_train_selected)
            X_test_scaled = scaler.transform(X_test_selected)

            # Train model
            simple_model.fit(X_train_scaled, y_train)

                    # Predict
            y_pred = simple_model.predict(X_test_scaled)
            y_pred_proba =
simple_model.predict_proba(X_test_scaled)[:, 1]

            # Compute metrics
            accuracy = accuracy_score(y_test, y_pred)
            precision = precision_score(y_test, y_pred,
average='binary', zero_division=0)
            recall = recall_score(y_test, y_pred, average='binary',
zero_division=0)
            f1 = f1_score(y_test, y_pred, average='binary',
zero_division=0)

            # Compute per-class F1 scores
            f1_per_class = f1_score(y_test, y_pred, average=None,
zero_division=0)
            if len(f1_per_class) == 2:
                f1_0 = f1_per_class[0]  # F1-score for the metastasis
category
                f1_1 = f1_per_class[1]  # F1-score for the
non-metastasis category
            else:
                f1_0 = 0
                f1_1 = f1

            roc_auc = roc_auc_score(y_test, y_pred_proba) if
len(np.unique(y_test)) > 1 else 0

            # Calculate specificity
            cm = confusion_matrix(y_test, y_pred)
            tn, fp, fn, tp = cm.ravel()
            specificity = tn / (tn + fp) if (tn + fp) > 0 else 0

            training_time = time.time() - start_time
```

```
            metrics = {
                'method': method_name + " (simple model)",
                'selected_features': len(valid_indices),
                'best_params': "C=1.0, penalty='l2',
solver='liblinear'",
                'cv_score': 0,
                'cv_std': 0,
                'accuracy': accuracy,
                'precision': precision,
                'recall': recall,
                'f1_score': f1,
                'f1_score_0': f1_0,  # F1 for metastatic class
                'f1_score_1': f1_1,  # F1 for non-metastatic class
                'roc_auc': roc_auc,
                'specificity': specificity,
                'training_time': training_time,
                'best_model': simple_model,
                'grid_search': None
            }

            return metrics

        except Exception as e2:
            print(f"  Simple model failed too: {e2}")
            return {
                'method': method_name + " (failed)",
                'selected_features': len(valid_indices),
                'best_params': None,
                'cv_score': 0,
                'cv_std': 0,
                'accuracy': 0,
                'precision': 0,
                'recall': 0,
                'f1_score': 0,
                'f1_score_0': 0,  # F1 for metastatic class
                'f1_score_1': 0,  # F1 for non-metastatic class
                'roc_auc': 0,
                'specificity': 0,
                'training_time': time.time() - start_time,
                'best_model': None,
                'grid_search': None
            }

# Save selected features
```

```
print("\n" + "="*80)
print(" Saving results...")
print("="*80)
qubo_metrics = evaluate_features_with_gridsearch(X_train, X_test,
y_train, y_test, selected_indices,
                                    method_name="Feature selection
method")
if selected_features and len(selected_features) > 0:
    print("Saving QUBO-selected features...")
    selected_features_df = pd.DataFrame({
        'Rank': range(1, len(selected_features) + 1),
        'Feature_Name': selected_features,
        'Correlation': [correlation_series[f] if f in
correlation_series.index else 0 for f in selected_features],
        'Absolute_Correlation': [abs(correlation_series[f]) if f in
correlation_series.index else 0 for f in selected_features],
        'Feature_Index': selected_indices,
        'Importance_Score': [abs(correlation_series[f]) * 100 if f in
correlation_series.index else 0 for f in selected_features]
    })

    # Sort by correlation
    if 'Absolute_Correlation' in selected_features_df.columns:
        selected_features_df =
selected_features_df.sort_values('Absolute_Correlation',
ascending=False)

    # Save to CSV
    selected_features_df.to_csv('selected_features_kaiwu_qubomodel.
csv', index=False)
    print(" QUBO selected features saved to
'selected_features_kaiwu_qubomodel.csv'")

    # Save all feature correlations
    print("Saving all feature correlation information...")
    all_features_corr_df = pd.DataFrame({
        'Feature_Name': feature_names,
        'Correlation': correlation_series.values,
        'Absolute_Correlation': correlation_abs_series.values
    }).sort_values('Absolute_Correlation', ascending=False)

    all_features_corr_df.to_csv('all_features_correlations.csv',
index=False)
```

```
    print(" All features correlation info saved to
'all_features_correlations.csv'")

# Save evaluation results
print("Saving evaluation results...")
try:
    # Save detailed results
    results_comparison.to_csv('ml_evaluation_with_gridsearch.csv',
index=False, encoding='utf-8-sig')
    print(" Detailed evaluation results saved to
'ml_evaluation_with_gridsearch.csv'")

    # Save simplified results (for reporting)
    results_comparison_display.to_csv('ml_evaluation_simple.csv',
index=False, encoding='utf-8-sig')
    print(" Simplified evaluation results saved to
'ml_evaluation_simple.csv'")
except Exception as e:
    print(f" Could not save evaluation results: {e}")

# Summary report
print("\n" + "="*80)
print(" Project completed!")
print("="*80)

total_time = time.time() - start_time

print(f"\n Key results summary:")
print(f"  • Total features: {n_features_total:,}")
print(f"  • Total samples: {len(sample_ids)} (Train:
{X_train.shape[0]}, Test: {X_test.shape[0]})")
print(f"  • QUBO selected features: {len(selected_features)} (Target:
{k_target}, Max: {k_max})")
print(f"  • Selection ratio:
{len(selected_features)/n_features_total*100:.2f}%")
print(f"  • QUBO cross-validation F1 score:
{qubo_metrics.get('cv_score', 0):.4f} ± {qubo_metrics.get('cv_std',
0):.4f}")
print(f"  • QUBO test F1 score: {qubo_metrics.get('f1_score',
0):.4f}")
print(f"  • QUBO test AUC: {qubo_metrics.get('roc_auc', 0):.4f}")
print(f"  • QUBO F1 (Metastatic): {qubo_metrics.get('f1_score_0',
0):.4f}")
```

```
print(f"  • QUBO F1 (Non-metastatic): {qubo_metrics.get('f1_score_1',
0):.4f}")

# Performance analysis
print(f"\n QUBO feature selection performance analysis:")
print(f"  • Accuracy: {qubo_metrics.get('accuracy', 0):.4f}")
print(f"  • Precision: {qubo_metrics.get('precision', 0):.4f}")
print(f"  • Recall: {qubo_metrics.get('recall', 0):.4f}")
print(f"  • Specificity: {qubo_metrics.get('specificity', 0):.4f}")

# Clinical significance analysis
print(f"\n Clinical significance analysis:")
f1_transfer = qubo_metrics.get('f1_score_0', 0)  # F1 for metastatic
class
f1_no_transfer = qubo_metrics.get('f1_score_1', 0)  # F1 for
non-metastatic class

if f1_transfer > 0.7:
    print(f"  Metastatic prediction performance excellent (F1:
{f1_transfer:.4f})")
elif f1_transfer > 0.5:
    print(f"  Metastatic prediction performance good (F1:
{f1_transfer:.4f})")
else:
    print(f"  Metastatic prediction performance needs improvement
(F1: {f1_transfer:.4f})")
if f1_no_transfer > 0.7:
    print(f"  Non-metastatic prediction performance excellent (F1:
{f1_no_transfer:.4f})")
elif f1_no_transfer > 0.5:
    print(f"  Non-metastatic prediction performance good (F1:
{f1_no_transfer:.4f})")
else:
    print(f"  Non-metastatic prediction performance needs
improvement (F1: {f1_no_transfer:.4f})")
print(f"\n Top QUBO-selected features (top 10):")
if 'selected_correlations' in locals() and
len(selected_correlations) > 0:
    display_count = min(10, len(selected_correlations))
    for i, (feature, corr) in
enumerate(selected_correlations[:display_count], 1):
        corr_type = "Positive" if corr > 0 else "Negative"
        significance = "Strong" if abs(corr) > 0.5 else "Moderate" if
abs(corr) > 0.3 else "Weak"
```

```
        print(f"  {i:2d}. {feature:35s} ({corr_type}:
{abs(corr):.4f}) - {significance}")
results_list=[]
print(f"\n Timing per step:")
print(f"  Data loading and preprocessing: {corr_time +
redundancy_time:.2f}s")
print(f"  QUBO matrix construction: {qubo_time:.2f}s")
print(f"  Solver run: {solve_time:.2f}s")
print(f"  Model training and evaluation: {sum(r.get('training_time',
0) for r in results_list):.2f}s")
print(f"  Total processing time: {total_time:.2f}s")

print(f"\n All tasks completed!")
print(f" Generated files list:")
print("  1. selected_features_kaiwu_qubomodel.csv - QUBO selected
features list")
print("  2. all_features_correlations.csv - All features correlation
info")
print("  3. ml_evaluation_with_gridsearch.csv - Detailed evaluation
results")
print("  4. ml_evaluation_simple.csv - Simplified evaluation
results")
```

Traditional model

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, RobustScaler
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.metrics import (
    accuracy_score, f1_score, roc_auc_score,
    precision_score, recall_score, balanced_accuracy_score,
    classification_report, confusion_matrix
)
from imblearn.over_sampling import SMOTE, RandomOverSampler
import warnings
warnings.filterwarnings('ignore')
import os
from datetime import datetime

# ==============================
```

```
# 1. Data Loading and Preprocessing
# ==============================
def load_and_prepare_data(filepath):
    """Load and preprocess data from CSV file."""

    try:
        # Try different encoding methods
        try:
            data = pd.read_csv(filepath)
        except UnicodeDecodeError:
            data = pd.read_csv(filepath, encoding='gbk')
        except:
            data = pd.read_csv(filepath, encoding='latin1')

        print(f"Data shape: {data.shape}")
        print(f"Data column names: {data.columns.tolist()}")

        # Display the first few rows of data for verification
        print(f"\nData first 5 rows:")
        print(data.head())

        # Check the column names to see if there is an obvious sample
ID column.
        sample_id_col = None
        label_col = None

        # Find the potential sample ID column (usually the first column,
and it typically contains terms like 'Sample' or 'ID').
        first_col = data.columns[0]
        if any(keyword in first_col.lower() for keyword in ['sample',
'id', 'patient', 'case']):
            sample_id_col = first_col
            print(f"The sample ID column has been identified:
'{sample_id_col}'")

        # Find the potential label column (has 2 unique values, and
is likely the second column)
        potential_label_cols = []
        for col in data.columns:
            unique_vals = data[col].nunique()
            if unique_vals == 2:
                potential_label_cols.append(col)

        if not potential_label_cols:
```

```
            # If no column with only 2 unique values is found, try the
second column.
            if len(data.columns) > 1:
                label_col = data.columns[1]
                print(f"No obvious label column was found, so the
second column is used as the label: '{label_col}'")
        else:
            # If there are multiple potential label columns, prioritize
the second column
            if len(data.columns) > 1 and data.columns[1] in
potential_label_cols:
                label_col = data.columns[1]
            else:
                label_col = potential_label_cols[0]
            print(f"The label column has been identified:
'{label_col}'")

        # If no sample ID column is identified, the first column is
assumed to be the sample ID.
        if sample_id_col is None and len(data.columns) > 0:
            sample_id_col = data.columns[0]
            print(f"Assuming the first column is the sample ID:
'{sample_id_col}'")

        # If no label column was found, assume the second column is
the label
        if label_col is None and len(data.columns) > 1:
            label_col = data.columns[1]
            print(f"Assuming the second column is the label:
'{label_col}'")
        elif len(data.columns) <= 1:
            print("Error: Not enough columns to find a label")
            return None, None, None, None

        print(f"\nSample ID column: '{sample_id_col}'")
        print(f"Label column: '{label_col}'")

        # Get sample IDs
        sample_ids = data[sample_id_col] if sample_id_col in
data.columns else None

        # Separate features and labels
        # Feature columns: all columns except sample ID and label
```

```
        feature_cols = [col for col in data.columns if col not in
[sample_id_col, label_col]]

        if not feature_cols:
            print("Error: No feature columns found")
            return None, None, None, None

        print(f"Number of feature columns: {len(feature_cols)}")

        X = data[feature_cols]
        y = data[label_col]

        # Show label distribution
        print(f"\nLabel distribution:")
        print(y.value_counts())

        return X, y, sample_ids, label_col

    except Exception as e:
        print(f"Error loading data: {e}")
        import traceback
        traceback.print_exc()
        return None, None, None, None

def clean_data(X, y):
    """Clean data: drop columns and rows with missing values"""
    print("\nCleaning data...")
    original_shape = X.shape

    # Check and handle Inf values
    for col in X.columns:
        if X[col].dtype in ['float32', 'float64', 'float16']:
            inf_mask = np.isinf(X[col])
            if inf_mask.any():
                X[col] = X[col].replace([np.inf, -np.inf], np.nan)

    # Drop columns with missing values
    columns_with_missing = X.columns[X.isna().any()].tolist()
    if columns_with_missing:
        X = X.drop(columns=columns_with_missing)
        print(f"Dropped {len(columns_with_missing)} columns with
missing values")

    # Check for missing labels and drop corresponding rows
```

```
    if y.isna().any():
        non_nan_mask = ~y.isna()
        X = X[non_nan_mask]
        y = y[non_nan_mask]
        print(f"Dropped {sum(y.isna())} rows with missing labels")

    # Ensure labels are integers
    try:
        y = y.astype(int)
        # Ensure labels are 0 and 1
        unique_labels = np.unique(y)
        if set(unique_labels) != {0, 1}:
            print(f"Warning: labels are not 0 and 1, but are:
{unique_labels}")
            # Try mapping labels to 0 and 1
            label_map = {unique_labels[0]: 0, unique_labels[1]: 1}
            y = y.map(label_map)
    except Exception as e:
        print(f"Error converting labels to integers: {e}")
        # Attempt factorization
        y = pd.factorize(y)[0]
        print(f"Factorized labels: {np.unique(y)}")

    print(f"After cleaning data shapes: X={X.shape}, y={y.shape}")
    print(f"Dropped {original_shape[1] - X.shape[1]} features")

    return X, y

def preprocess_features(X):
    """Preprocess features: convert categorical variables to
numeric"""
    categorical_cols =
X.select_dtypes(exclude=[np.number]).columns.tolist()
    if categorical_cols:
        print(f"Converting {len(categorical_cols)} categorical
features to numeric")
        for col in categorical_cols:
            X[col] = pd.factorize(X[col])[0]
    return X

# ==============================
# 2. Model definition and training
# ==============================
class ModelTrainer:
```

```
    """Base class for model trainers"""
    def __init__(self, name):
        self.name = name
        self.model = None
        self.selected_features = None
        self.feature_importance_df = None  # DataFrame to store
feature importance

    def select_features(self, X, y, n_features=300):
        """Feature selection - select up to 300 features"""
        print(f"  Feature selection: selecting {min(n_features,
X.shape[1])} out of {X.shape[1]} features")

        if X.shape[1] <= n_features:
            self.selected_features = X.columns.tolist()
            # Create placeholder importance scores (1.0)
            self.feature_importance_df = pd.DataFrame({
                'feature': X.columns,
                'importance': [1.0] * len(X.columns)
            }).sort_values('importance', ascending=False)
            return X

        try:
            # Use SelectKBest to select features
            selector = SelectKBest(f_classif, k=min(n_features,
X.shape[1]))
            X_selected = selector.fit_transform(X, y)

            # Get the selected feature names
            self.selected_features =
X.columns[selector.get_support()].tolist()

            # Get feature scores as importance
            feature_scores = selector.scores_
            selected_indices = selector.get_support()

            # Create feature importance DataFrame
            self.feature_importance_df = pd.DataFrame({
                'feature': X.columns,
                'importance': feature_scores,
                'selected': selected_indices
            })

            # Keep only selected features and sort by importance
```

```
            selected_features_df =
self.feature_importance_df[self.feature_importance_df['selected']]
.copy()
            selected_features_df =
selected_features_df.sort_values('importance', ascending=False)

            # Reindex
            selected_features_df =
selected_features_df.reset_index(drop=True)
            self.feature_importance_df =
selected_features_df[['feature', 'importance']]

            print(f"  Selected {len(self.selected_features)}
features")

            # Return selected feature data
            return pd.DataFrame(X_selected,
columns=self.selected_features)

        except Exception as e:
            print(f"  Feature selection failed: {e}")
            self.selected_features = X.columns.tolist()
            # Create default importance scores
            self.feature_importance_df = pd.DataFrame({
                'feature': X.columns,
                'importance': [1.0] * len(X.columns)
            })
            return X

    def handle_imbalance(self, X, y):
        """Handle class imbalance"""
        class_counts = y.value_counts()
        imbalance_ratio = max(class_counts) / min(class_counts) if
min(class_counts) > 0 else float('inf')

        print(f"  Class distribution: {dict(class_counts)}")
        print(f"  Imbalance ratio: {imbalance_ratio:.2f}")

        if len(class_counts) < 2:
            print("  Warning: only one class present, cannot handle
imbalance")
            return X, y

        if imbalance_ratio > 3:
```

```
            # Ensure enough samples for SMOTE
            if min(class_counts) > 5:
                try:
                    smote = SMOTE(random_state=42, k_neighbors=min(5,
class_counts.min() - 1))
                    X_resampled, y_resampled = smote.fit_resample(X,
y)
                    print(f"  Used SMOTE to handle class imbalance,
resampled shape: {X_resampled.shape}")
                    return X_resampled, y_resampled
                except Exception as e:
                    print(f"  SMOTE failed: {e}")

        # Use RandomOverSampler as fallback
        try:
            ros = RandomOverSampler(random_state=42)
            X_resampled, y_resampled = ros.fit_resample(X, y)
            print(f"  Used RandomOverSampler to handle class
imbalance, resampled shape: {X_resampled.shape}")
            return X_resampled, y_resampled
        except Exception as e:
            print(f"  RandomOverSampler failed: {e}")
            return X, y

    def train(self, X_train, y_train, X_test, y_test):
        """Train model (to be implemented by subclasses)"""
        raise NotImplementedError

    def evaluate(self, X_test, y_test):
        """Evaluate model"""
        y_pred = self.model.predict(X_test)
        y_prob = self.model.predict_proba(X_test) if
hasattr(self.model, 'predict_proba') else None

        # Calculate metrics
        metrics = {}
        metrics['accuracy'] = accuracy_score(y_test, y_pred)
        metrics['balanced_accuracy'] =
balanced_accuracy_score(y_test, y_pred)
        metrics['f1_macro'] = f1_score(y_test, y_pred,
average='macro')
        metrics['f1_weighted'] = f1_score(y_test, y_pred,
average='weighted')
```

```
        # Calculate F1 score for each class
        unique_classes = np.unique(y_test)
        for cls in unique_classes:
            metrics[f'f1_class{cls}'] = f1_score(y_test, y_pred,
labels=[cls], average=None)[0]
            metrics[f'precision_class{cls}'] =
precision_score(y_test, y_pred, labels=[cls], average=None)[0]
            metrics[f'recall_class{cls}'] = recall_score(y_test,
y_pred, labels=[cls], average=None)[0]

        # Calculate AUC (only for binary classification)
        if len(unique_classes) == 2 and y_prob is not None and
y_prob.shape[1] > 1:
            try:
                metrics['auc'] = roc_auc_score(y_test, y_prob[:, 1])
            except:
                metrics['auc'] = np.nan
        else:
            metrics['auc'] = np.nan

        return metrics, y_pred, y_prob

    def get_feature_importance(self):
        """Get feature importance DataFrame"""
        return self.feature_importance_df

class LassoLogisticTrainer(ModelTrainer):
    """Lasso logistic regression trainer"""
    def __init__(self):
        super().__init__("Lasso Logistic Regression")

    def train(self, X_train, y_train, X_test, y_test):
        print(f"\nTraining {self.name}...")

        # Handle class imbalance
        X_train_resampled, y_train_resampled =
self.handle_imbalance(X_train, y_train)

        # Standardize
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train_resampled)
        X_test_scaled = scaler.transform(X_test)

        # Train Lasso logistic regression (L1 regularization)
```

```
        self.model = LogisticRegression(
            penalty='l1',
            C=0.1,
            solver='liblinear',
            max_iter=1000,
            random_state=42
        )
        self.model.fit(X_train_scaled, y_train_resampled)

        # Obtain feature coefficients as importance scores
        if hasattr(self.model, 'coef_'):
            coef = self.model.coef_[0] if
len(self.model.coef_.shape) > 1 else self.model.coef_
            abs_coef = np.abs(coef)

            # Update feature importance DataFrame
            if self.feature_importance_df is not None and
len(self.selected_features) == len(coef):
                self.feature_importance_df['coefficient'] = coef
                self.feature_importance_df['abs_coefficient'] =
abs_coef

                # Use coefficient absolute value as importance, and
re-sort
                self.feature_importance_df['importance'] = abs_coef
                self.feature_importance_df =
self.feature_importance_df.sort_values('importance',
ascending=False)

        metrics, y_pred, y_prob = self.evaluate(X_test_scaled,
y_test)
        return metrics, y_pred, y_prob

class RegularLogisticTrainer(ModelTrainer):
    """Regular logistic regression trainer"""
    def __init__(self):
        super().__init__("Regular Logistic Regression")

    def train(self, X_train, y_train, X_test, y_test):
        print(f"\nTraining {self.name}...")

        # Handle class imbalance
        X_train_resampled, y_train_resampled =
self.handle_imbalance(X_train, y_train)
```

```
        # Standardize
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train_resampled)
        X_test_scaled = scaler.transform(X_test)

        # Train regular logistic regression (L2 regularization)
        self.model = LogisticRegression(
            penalty='l2',
            C=1.0,
            solver='lbfgs',
            max_iter=1000,
            random_state=42
        )
        self.model.fit(X_train_scaled, y_train_resampled)

        # Obtain feature coefficients as importance scores
        if hasattr(self.model, 'coef_'):
            coef = self.model.coef_[0] if
len(self.model.coef_.shape) > 1 else self.model.coef_
            abs_coef = np.abs(coef)

            # Update feature importance DataFrame
            if self.feature_importance_df is not None and
len(self.selected_features) == len(coef):
                self.feature_importance_df['coefficient'] = coef
                self.feature_importance_df['abs_coefficient'] =
abs_coef

                # Use coefficient absolute value as importance, and
re-sort
                self.feature_importance_df['importance'] = abs_coef
                self.feature_importance_df =
self.feature_importance_df.sort_values('importance',
ascending=False)

        metrics, y_pred, y_prob = self.evaluate(X_test_scaled,
y_test)
        return metrics, y_pred, y_prob

class NaiveBayesTrainer(ModelTrainer):
    """Naive Bayes trainer"""
    def __init__(self):
        super().__init__("Naive Bayes")
```

```
    def train(self, X_train, y_train, X_test, y_test):
        print(f"\nTraining {self.name}...")

        # Handle class imbalance
        X_train_resampled, y_train_resampled =
self.handle_imbalance(X_train, y_train)

        # Standardize (Naive Bayes is insensitive to scaling, but keep
for consistency)
        scaler = RobustScaler()
        X_train_scaled = scaler.fit_transform(X_train_resampled)
        X_test_scaled = scaler.transform(X_test)

        # Train Naive Bayes
        self.model = GaussianNB()
        self.model.fit(X_train_scaled, y_train_resampled)

        # Naive Bayes has no direct coefficients; use feature std
deviation differences as importance
        if len(self.selected_features) > 0 and X_train_scaled.shape[1]
== len(self.selected_features):
            # Calculate the standard deviation of each feature for each
category
            std_by_class = []
            unique_classes = np.unique(y_train_resampled)

            for cls in unique_classes:
                X_class = X_train_scaled[y_train_resampled == cls]
                std_by_class.append(np.std(X_class, axis=0))

            # Use class-wise std difference as importance
            if len(std_by_class) >= 2:
                importance = np.abs(std_by_class[0] -
std_by_class[1])
            else:
                importance = np.std(X_train_scaled, axis=0)

            # Update feature importance DataFrame
            if self.feature_importance_df is not None and
len(self.selected_features) == len(importance):
                self.feature_importance_df['importance'] =
importance
```

```
                self.feature_importance_df =
self.feature_importance_df.sort_values('importance',
ascending=False)

        metrics, y_pred, y_prob = self.evaluate(X_test_scaled,
y_test)
        return metrics, y_pred, y_prob

# ==============================
# 3. Main function
# ==============================
def main():
    print("=" * 80)
    print("Comprehensive model training: Lasso Logistic Regression,
Regular Logistic Regression, Naive Bayes")
    print("=" * 80)

    # Set data path
    data_path = r"./brca_merged_final.csv"

    if not os.path.exists(data_path):
        print(f"Error: file does not exist - {data_path}")
        return

    # 1. Load data
    X, y, sample_ids, target_col = load_and_prepare_data(data_path)
    if X is None or y is None:
        print("Data loading failed, exiting")
        return

    print(f"\nTarget column: '{target_col}'")
    print(f"Number of features: {X.shape[1]}")
    print(f"Number of samples: {X.shape[0]}")

    # 2. Preprocessing
    X = preprocess_features(X)
    X, y = clean_data(X, y)

    # Check data
    if X.shape[1] == 0:
        print("Error: No features remaining after cleaning!")
        return

    print(f"\nProcessed feature count: {X.shape[1]}")
```

```
    # 3. Split train and test sets (fixed random seed to ensure
consistent splits across models)
    print("\nSplitting train and test sets...")
    try:
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=0.3, random_state=42, stratify=y
        )
    except:
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=0.3, random_state=42
        )

    print(f"Train set: {X_train.shape[0]} samples")
    print(f"Test set: {X_test.shape[0]} samples")
    print(f"Number of features: {X_train.shape[1]}")

    # 4. Create model trainers
    trainers = {
        'lasso': LassoLogisticTrainer(),
        'regular': RegularLogisticTrainer(),
        'naive_bayes': NaiveBayesTrainer()
    }

    # 5. Train and evaluate all models
    results = {}
    all_predictions = {}
    all_feature_importances = {}

    for model_name, trainer in trainers.items():
        print(f"\n{'='*60}")
        print(f"Processing {trainer.name}")
        print('='*60)

        # Copy data to avoid modifying originals
        X_train_processed = X_train.copy()
        X_test_processed = X_test.copy()

        # Feature selection - select up to 300 features
        X_train_selected =
trainer.select_features(X_train_processed, y_train, n_features=300)
        selected_features = trainer.selected_features

        # Ensure test set has the same features
```

```
        X_test_selected = X_test_processed[selected_features]

        print(f"  Number of selected features:
{len(selected_features)}")

        # Train model
        metrics, y_pred, y_prob = trainer.train(
            X_train_selected, y_train,
            X_test_selected, y_test
        )

        # Save results
        results[model_name] = {
            'trainer': trainer,
            'metrics': metrics,
            'selected_features': selected_features
        }

        all_predictions[model_name] = {
            'y_pred': y_pred,
            'y_prob': y_prob if y_prob is not None else None
        }

        # Save feature importance
        feature_importance_df = trainer.get_feature_importance()
        if feature_importance_df is not None:
            all_feature_importances[model_name] =
feature_importance_df

    # 6. Output results
    print("\n" + "="*80)
    print("Model evaluation results summary")
    print("="*80)

    # Create results DataFrame
    result_rows = []
    for model_name, result in results.items():
        trainer = result['trainer']
        metrics = result['metrics']

        # Extract per-class F1 scores
        f1_scores = {}
        for key, value in metrics.items():
            if key.startswith('f1_class'):
```

```
                f1_scores[key] = value

        row = {
            'Model': trainer.name,
            'Accuracy': f"{metrics['accuracy']:.4f}",
            'Balanced Accuracy':
f"{metrics['balanced_accuracy']:.4f}",
            'Macro F1': f"{metrics['f1_macro']:.4f}",
            'Weighted F1': f"{metrics['f1_weighted']:.4f}",
            'AUC': f"{metrics.get('auc', 'N/A')}",
            'Selected Features': len(result['selected_features'])
        }

        # Add per-class F1 scores
        for key, value in f1_scores.items():
            class_num = key.replace('f1_class', '')
            row[f'Class{class_num}_F1'] = f"{value:.4f}"
            row[f'Class{class_num}_Precision'] =
f"{metrics.get(f'precision_class{class_num}', 'N/A')}"
            row[f'Class{class_num}_Recall'] =
f"{metrics.get(f'recall_class{class_num}', 'N/A')}"

        result_rows.append(row)

    results_df = pd.DataFrame(result_rows)
    print("\nEvaluation metrics summary:")
    print(results_df.to_string(index=False))

    # 7. Output feature importance
    print("\n" + "="*80)
    print("Feature importance summary")
    print("="*80)

    # Create output directory
    output_dir = "E:/model_comparison_results"
    os.makedirs(output_dir, exist_ok=True)
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

    # Save each model's feature importance
    for model_name, importance_df in
all_feature_importances.items():
        if importance_df is not None:
            # Keep only top 300 features (if more than 300)
            if len(importance_df) > 300:
```

```
                importance_df = importance_df.head(300)

            output_path = os.path.join(output_dir,
f"{model_name}_feature_importance_{timestamp}.csv")

            # Add model name column
            importance_df['model'] = trainers[model_name].name

            # Ensure only two columns: feature and importance score
            if 'importance' in importance_df.columns:
                # If there are extra columns, keep only needed ones
                importance_df = importance_df[['feature',
'importance']].copy()
            else:
                # If there is no 'importance' column, use the first
numeric column
                numeric_cols =
importance_df.select_dtypes(include=[np.number]).columns
                if len(numeric_cols) > 0:
                    importance_df = importance_df[['feature',
numeric_cols[0]]].copy()
                    importance_df.columns = ['feature', 'importance']

            # Ensure importance is numeric
            importance_df['importance'] =
pd.to_numeric(importance_df['importance'], errors='coerce')

            # Sort by importance descending
            importance_df = importance_df.sort_values('importance',
ascending=False)

            # Save to CSV
            importance_df.to_csv(output_path, index=False,
encoding='utf-8-sig')
            print(f"\n{trainers[model_name].name} feature importance
saved to: {output_path}")
            print(f"Number of features: {len(importance_df)}")
            print("Top 10 most important features:")
            print(importance_df.head(10).to_string(index=False))
            print("-" * 60)

    # 8. Save all results to CSV
    summary_path = os.path.join(output_dir,
f"model_comparison_summary_{timestamp}.csv")
```

```
    results_df.to_csv(summary_path, index=False,
encoding='utf-8-sig')
    print(f"\nModel comparison summary saved to: {summary_path}")

    # 9. Save predictions
    predictions_df = pd.DataFrame({
        'True Label': y_test.values,
        'Lasso Logistic Regression Prediction':
all_predictions['lasso']['y_pred'],
        'Regular Logistic Regression Prediction':
all_predictions['regular']['y_pred'],
        'Naive Bayes Prediction':
all_predictions['naive_bayes']['y_pred']
    })

    predictions_path = os.path.join(output_dir,
f"predictions_{timestamp}.csv")
    predictions_df.to_csv(predictions_path, index=False,
encoding='utf-8-sig')
    print(f"Predictions saved to: {predictions_path}")

    # 10. Final summary report
    print("\n" + "="*80)
    print("Final summary report")
    print("="*80)

    # Find best model per metric
    metrics_to_compare = ['Accuracy', 'Balanced Accuracy', 'Macro F1',
'Weighted F1']
    best_models = {}

    for metric in metrics_to_compare:
        if metric in results_df.columns:
            # Extract numeric values for comparison
            values = []
            for val in results_df[metric]:
                try:
                    values.append(float(val))
                except:
                    values.append(0)

            if values and len(values) > 0:
                best_idx = np.argmax(values)
```

```
                best_models[metric] =
results_df.iloc[best_idx]['Model']

    print("\nBest model per metric:")
    for metric, model in best_models.items():
        print(f"  {metric}: {model}")

    # F1 score summary
    print("\nF1 scores per class:")
    for model_name, result in results.items():
        trainer = result['trainer']
        metrics = result['metrics']

        f1_scores = []
        for key in metrics.keys():
            if key.startswith('f1_class'):
                class_num = key.replace('f1_class', '')
                f1_scores.append(f"Class{class_num}:
{metrics[key]:.4f}")

        if f1_scores:
            print(f"{trainer.name}: {', '.join(f1_scores)}")

    print("\n" + "="*80)
    print("Training complete!")
    print(f"All results saved to: {output_dir}")
    print("="*80)

if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        print(f"Program error: {e}")
        import traceback
        traceback.print_exc()

        # Save error information
        error_dir = "E:/model_comparison_results"
        os.makedirs(error_dir, exist_ok=True)
        error_path = os.path.join(error_dir,
f"error_{datetime.now().strftime('%Y%m%d_%H%M%S')}.txt")

        with open(error_path, 'w', encoding='utf-8') as f:
```

```
            f.write(f"Error time:
{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")
            f.write(f"Error message: {str(e)}\n")
            f.write("Error trace:\n")
            f.write(traceback.format_exc())

        print(f"Error details saved to: {error_path}")
```