ENHANCING CREDIT CARD FRAUD DETECTION WITH MACHINE LEARNING
CSE/ISYE 6740 FINAL PROJECT REPORT

Wenxin Jiang GTID 903977667 **Qihao Cheng GTID 904023934**

Yihan Liao GTID 903919945 **Yu Zheng GTID 904043952**

ABSTRACT

This study addresses the challenge of identifying fraudulent transactions in large-scale, imbalanced datasets using machine learning techniques. The IEEE-CIS Fraud Detection dataset, derived from real-world e-commerce transactions, was used as a benchmark. Comprehensive preprocessing steps were applied, including imputation, standardization, and SMOTE-NC, to handle missing data and address the inherent class imbalance. The study evaluated a variety of machine learning models, including Logistic Regression, Support Vector Machines (RBF and Polynomial kernels), Neural Networks, and ensemble methods such as Random Forest, LightGBM, XGBoost, and CatBoost. Model performance was assessed using metrics such as ROC-AUC and PR-AUC, with a focus on balancing precision and recall to optimize fraud detection. Among the evaluated models, LightGBM achieved the highest performance, with a ROC-AUC of 0.98 and PR-AUC of 0.90, demonstrating its superior handling of numerical features and imbalanced data. Our analysis reveals that tree-based models demonstrate superior performance due to their effective handling of both numerical and categorical features. Despite CatBoost's renowned capability in processing categorical data, the low proportion and cardinality of categorical features in this dataset favored LightGBM's efficient numerical processing capabilities. This work provides a scalable, robust framework for fraud detection, addressing critical financial and operational challenges.

1 INTRODUCTION

We live in a world where billions of credit card transactions occur daily. With this massive volume of exchanges comes a critical challenge: fraud detection. Fraudulent schemes are becoming increasingly common, exploiting vulnerabilities in modern payment systems. By 2025, credit card fraud is projected to cause $12.5 billion in losses Rej (2023). As a result, detecting and mitigating fraud has become a top priority for businesses and financial institutions. In this project, we aim to address the challenge of identifying fraudulent transactions within large-scale payment data using machine learning techniques. Our objectives are to identify the best-performing model and enhance the precision of fraud detection while maintaining efficiency in high-throughput environments. The main contribution of our work is the development of a robust and scalable fraud detection model that integrates data preprocessing and advanced machine learning methods. By evaluating our model on past transactions to predict future transactions, we demonstrate its effectiveness in controlling false positives and accurately identifying fraudulent activity.

2 RELATED WORK

Fraud detection is a critical area of research aimed at identifying deceptive practices across various domains, including finance, e-commerce, telecommunications, and healthcare. With the rapid increase in digital transactions, fraud schemes have become more sophisticated, necessitating ad-

vanced detection techniques. This section provides an overview of existing work on fraud detection, focusing on traditional, machine learning-based, and emerging approaches.

2.1 Traditional Fraud Detection Techniques

Early fraud detection systems primarily relied on rule-based approaches, utilizing predefined heuristics to flag suspicious activities. These systems operate by applying specific rules, such as transaction thresholds or patterns, to identify potential fraud. While straightforward and interpretable, rule-based systems often struggle to adapt to evolving fraud patterns and may generate high false-positive rates Kou et al. (2004). Statistical methods, including clustering techniques, have also been employed to detect anomalies indicative of fraud. Clustering algorithms group similar data points, enabling the identification of outliers that may represent fraudulent activities. For instance, k-means clustering has been applied to segment data and highlight unusual patterns Maddila et al. (2020). However, these methods may face challenges in high-dimensional datasets and can be sensitive to parameter selection.

2.2 Machine Learning and Deep Learning in Fraud Detection

Machine learning has become the cornerstone of modern fraud detection systems. Supervised learning techniques, such as logistic regression, decision trees, random forests, and gradient boosting, are widely used to classify transactions as fraudulent or legitimate. For example, Brown and Mues (2012) demonstrated the efficacy of ensemble models for financial fraud detection Brown & Mues (2012). Unsupervised learning techniques, such as auto-encoders, isolation forests, and clustering-based methods, are applied in scenarios with scarce labeled data Schubert et al. (2015). Hybrid models that combine supervised and unsupervised methods have shown promise in addressing the limitations of individual approaches Nguyen et al. (2022).

Deep learning extends the capabilities of traditional ML by analyzing complex and high-dimensional datasets. Recurrent neural networks (RNNs) and long short-term memory networks (LSTMs) excel in modeling temporal dependencies in transaction sequences Chen et al. (2022). Convolutional neural networks (CNNs) have been utilized for image-based fraud detection, such as identifying counterfeit documents Roy et al. (2018). Graph neural networks (GNNs) are particularly effective in modeling relationships between entities, such as user-item interactions in e-commerce or social networks Li et al. (2022). These methods capture structured information, enabling the detection of organized fraud schemes.

2.3 Real-Time and Big Data Approaches

The increasing volume of digital transactions necessitates real-time fraud detection capabilities. Streaming analytics platforms, such as Apache Kafka and Spark, enable the processing of large-scale data streams, ensuring prompt detection Tax et al. (2021). Feature engineering and data fusion from diverse sources enhance model accuracy, as highlighted in fraud detection frameworks combining structured and unstructured data Chen et al. (2019). Fraud detection has applications across financial services (credit card fraud, money laundering), e-commerce (fake reviews, account takeovers), healthcare (insurance fraud), and telecommunications (call spoofing) Jurgovsky et al. (2018). Challenges include imbalanced datasets, evolving fraud patterns, and the high cost of false positives, which can alienate legitimate users.

3 Data Collection and Preparation

The dataset for this analysis originates from the Kaggle competition "IEEE-CIS Fraud Detection", a collaboration between the IEEE Computational Intelligence Society and Vesta Corporation iee . The data is derived from Vesta's real-world e-commerce transactions and includes extensive features ranging from device type to product characteristics. The project goal is to improve fraud detection accuracy, reducing false positives. In business terms, reduce fraud loss and false alarm.

3.1 EXPLORATORY DATA ANALYSIS (EDA)

The dataset contains a total of 432 features, divided into 40 identity features and 392 transaction features. After addressing high missing value columns, the dataset was reduced to 403 features, and further encoding expanded it to 2278 features. The outcome variable, as shown in Figure 1, "isFraud", with 569,877 Negative (non-fraudulent transactions) and 20,663 Positive. Missing value analysis as shown in Figure 2 revealed the following distribution of null percentages across features:

3.2 DATA PREPROCESSING

1. Handling Missing Values: Columns with more than 80% missing values were dropped to avoid introducing noise and excessive imputation. For numerical features, missing values were imputed using the median, assuming it to be a robust measure of central tendency that is less sensitive to outliers. For categorical features, missing values were imputed using the mode, assuming the most frequent category provides a reasonable estimate for the missing values.

2. Standardization of Continuous Variables: Continuous variables, such as `Transaction Amount`, were standardized to rescale them to a standard range, ensuring that features with larger ranges do not disproportionately influence the model.

3. Handling Class Imbalance: SMOTE-NC (Synthetic Minority Oversampling Technique for Nominal and Continuous features) was applied to address the inherent class imbalance in the dataset. This technique generated synthetic samples to improve the model's ability to generalize to rare events, such as fraud detection.

4. Encoding Categorical Variables: Categorical variables were encoded using one-hot encoding, which creates binary columns for each category, preserving interpretability and ensuring compatibility with machine learning algorithms requiring numerical inputs.

4 METHODOLOGY

4.1 ADDRESSING CLASS IMBALANCE WITH SMOTE-NC

To address class imbalance, we employed SMOTE-NC (Synthetic Minority Oversampling Technique for Nominal and Continuous data) Chawla et al. (2002). SMOTE-NC generates synthetic samples for the minority class by interpolating between existing samples, tailored for mixed-type datasets. The synthetic sample $\mathbf{x}'$ is generated as:

$$\mathbf{x}' = \mathbf{x} + \lambda(\mathbf{x}_{nn} - \mathbf{x}),$$

where $\mathbf{x}$ is a minority sample, $\mathbf{x}_{nn}$ is one of its k-nearest neighbors, and $\lambda \sim U(0, 1)$ is a random scalar. This interpolation is applied to continuous features.

For categorical features, SMOTE-NC employs a majority voting mechanism among the nearest neighbors to determine the synthetic value. Let C represent the categorical feature, and $\{c_1, c_2, \ldots, c_k\}$ denote the values of C among the k-nearest neighbors. The synthetic value c' is given by:

$$c' = \text{mode}(c_1, c_2, \ldots, c_k).$$

SMOTE-NC was applied exclusively to the training set to avoid data leakage. The parameters were set to $k\_neighbors = 5$, and categorical features were specified based on domain knowledge. This method ensured a balanced class distribution while preserving the integrity of the validation and test datasets.

4.2 LOGISTIC REGRESSION AND RANDOM FOREST

Logistic Regression can be seen as predicting the probability of a binary outcome by modeling the relationship between input features (X) and the log-odds of the target variable (y). In business applications, logistic regression is highly valued for its interpretability, as the model coefficients provide clear insights into the distribution of contribution among features and outcomes. It is also computationally efficient, making it suitable for analyzing large datasets, and its probabilistic outputs enable flexible decision-making by adjusting thresholds to align with business goals.

Random Forest is a Bagging method that builds a collection of decision trees, the numerous trees the model trained decreased the variance of error. As a by-product. It also provides feature importance scores, helping identify key predictors and enhancing interpretability, which we later find the significance in categorical variables. To align the model with our objectives, several parameters were fine-tuned.

4.3 Ensemble methods and Boosting Based Models

Ensemble models are like a "team" of models working together to improve predictions by combining the strengths of individual models. There are two main techniques for building these teams: Bagging and Boosting. To evaluate the effectiveness and efficiency of boosting algorithms for fraud detection, we implemented and compared three widely used **gradient boosting frameworks: XGBoost, LightGBM, and CatBoost**. These models were chosen due to their ability to capture complex interactions between features, and provide high predictive accuracy.

4.4 Extreme Gradient Boosting(XGBoost)

Introduced in 2016, XGBoost Chen & Guestrin (2016) is a scalable and efficient implementation of gradient boosting, designed to optimize both computational efficiency and predictive accuracy. It combines the strengths of decision trees and boosting techniques, iteratively building models to minimize a custom loss function while applying regularization to avoid overfitting. The core optimization objective in XGBoost is defined as:

$$\mathcal{L}(\phi) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{t=1}^{T} \Omega(f_t),$$

where $l(y_i, \hat{y}_i)$ is the loss function (e.g., logistic loss for classification tasks in this project), and

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \|\mathbf{w}\|^2$$

represents a regularization term penalizing model complexity. Here, T is the number of leaves in the tree, and $\mathbf{w}$ denotes the leaf weights. This regularization term enhances the model's robustness and prevents overfitting, especially in complex datasets.

XGBoost's native support for categorical features, introduced in version 1.5, allows the model to directly handle categorical data without costly preprocessing like one-hot or target encoding. This improves computational efficiency, especially in datasets with many categorical attributes. Additionally, XGBoost employs a level-wise tree growth strategy. This method fully expands all nodes at a given depth before moving to the next level, leading to a more balanced tree structure and reducing overfitting by controlling tree depth. These features—native categorical handling, level-wise growth, and robust regularization—make XGBoost highly effective for imbalanced classification tasks, such as fraud detection, where it is compared to other boosting models.

XGBoost was implemented with key parameters: num_boost_round = 200, max_depth=6, learning_rate=0.1, min_child_weight=3, and sampling ratios of 0.8 for both instances and features to prevent overfitting.

4.5 Light Gradient Boosting Machine (LightGBM)

To evaluate Boosting methods in fraud detection, we used LightGBM, a gradient boosting framework introduced by Microsoft in 2017 Ke et al. (2017). Unlike XGBoost's level-wise growth, LightGBM employs a leaf-wise splitting strategy, which results in deeper trees and higher accuracy, though with a risk of overfitting, mitigated by regularization.

LightGBM introduces histogram-based decision rules, converting continuous features into discrete bins, reducing memory usage and computation time. This makes finding split points more efficient, with complexity reduced to $O(n)$, where n is the number of data bins. For imbalanced datasets, LightGBM uses Gradient-based One-Side Sampling (GOSS), prioritizing samples with larger gradients while downsampling those with smaller gradients, enhancing computational efficiency. To

handle high-dimensional sparse features, LightGBM uses Exclusive Feature Bundling (EFB), reducing dimensionality without sacrificing accuracy. LightGBM demonstrated faster training times than XGBoost, maintaining competitive recall and precision for fraud detection, making it ideal for large-scale, imbalanced datasets.

LightGBM was implemented with n_estimators=200, num_leaves=31, learning_rate=0.1, and feature_fraction=0.8 to balance model complexity and performance.

4.6 CATEGORICAL BOOSTING (CATBOOST)

Given that our dataset contains categorical features, we leveraged CatBoost Prokhorenkova et al. (2018), a gradient boosting framework introduced by Yandex in 2018, to evaluate whether it could improve predictive performance. CatBoost is optimized for handling categorical features without extensive preprocessing, making it ideal for datasets with high-cardinality or complex categorical variables. It uses an advanced "Ordered Target Statistics" approach, automatically encoding categorical variables during training by calculating historical statistics that consider feature importance and temporal dependencies. This eliminates the need for preprocessing techniques like one-hot or label encoding. To prevent prediction shift and target leakage, CatBoost's ordered boosting mechanism ensures that each data point's encoding relies only on prior observations in the training sequence, preserving statistical integrity and enhancing model robustness.

Additionally, CatBoost constructs symmetric decision trees, ensuring balanced splits across all branches, which simplifies optimization and reduces overfitting. This, combined with default parameters optimized through Bayesian optimization, allows CatBoost to deliver strong predictive performance with minimal tuning, making it highly effective for practical applications.

CatBoost was implemented with iterations=500, learning_rate=0.1, and sampling ratios of 0.8 for both observations and features at each tree level.

4.7 SUPPORT VECTOR MACHINES (SVMS)

Support Vector Machines (SVMs) were selected for this project due to their ability to effectively handle high-dimensional data and their robustness in dealing with non-linear relationships through kernel functions. These characteristics make SVMs well-suited for identifying complex patterns in fraud detection, especially when the dataset contains diverse feature interactions. To improve computational efficiency without compromising data representativeness, 20% of the training data was used. This subsampling approach significantly reduced the time and resources required for SVM training while maintaining sufficient data diversity for model generalization.

Radial Basis Function (RBF) SVM: The Radial Basis Function (RBF) kernel is a popular choice for SVM due to its ability to model complex, non-linear decision boundaries. In RBF kernel definition, the squared Euclidean distance between datapoints x_i and x_j, and γ is a kernel parameter that controls the scale. PCA helps reduce training time for RBF SVM by lowering the dimensionality of the feature space from d to k ($k < d$). After PCA, the computation of $\|x_i - x_j\|^2$ involves only k features, reducing the cost of each kernel calculation from $2d + 1$ operations to $2k + 1$. This makes training faster while preserving most of the data variance.

Polynomial SVM: The Polynomial kernel extends SVM's capability by capturing interactions between features through polynomial terms. In Polynomial kernel definition, the dot product of feature vectors x_i and x_j, and coef0, γ, and degree are hyperparameters. PCA reduces the dimensionality of the feature space, lowering the number of terms in the dot product from d to k. This reduces the computational cost of kernel calculations from $2d - 1$ operations to $2k - 1$, accelerating training time while maintaining the model's ability to learn complex patterns in the data.

4.8 NEURON NETWORK

The implemented fully connected neural network serves as a foundational model for fraud detection tasks, designed to handle tabular datasets with mixed feature types. Despite its simplicity, the model incorporates essential techniques for addressing class imbalance, improving generalization, and ensuring robust evaluation.

Model Architecture: The neural network employs a feedforward architecture with four hidden layers [128, 64, 32, 16], using ReLU activation functions throughout. Dropout layers (rate=0.01) are implemented between hidden layers for regularization, and the network concludes with a single output neuron for binary classification. For data preparation, we implemented a custom `FraudDataLoader` to maintain balanced batches with a 50% fraud ratio during training. The preprocessing pipeline includes one-hot encoding for categorical features and standardization for numerical features, with appropriate strategies for handling missing values.

The model is trained using binary cross-entropy loss with logits (BCEWithLogitsLoss), which combines a sigmoid activation with cross-entropy for stable binary classification. The Adam optimizer, with a learning rate of 0.001, adapts learning rates for efficient convergence. Training is conducted over 50 epochs with 200 steps per epoch, ensuring adequate exposure to patterns in the dataset. Balanced batch sampling, with a fraud ratio of 50%, addresses class imbalance by including sufficient minority class instances during training. To enhance generalization, dropout regularization (rate 0.01) mitigates overfitting by randomly deactivating neurons. The training process incorporates periodic model checkpointing for resumability and generates validation metrics and visualizations, such as AUC history and ROC curves, to monitor progress and ensure robust evaluation. These features make the model both effective and straightforward to extend or adapt for fraud detection tasks.

5 Experimental Results

Model	ROC-AUC	PR-AUC	Accuracy
Logistic Regression	0.917	0.7	95.0%
Random Forest	0.947	0.788	96.8%
XGBoost	0.96	0.83	96.5%
LightGBM	**0.98**	**0.90**	**97.6%**
CatBoost	0.97	0.89	97.4%
RBF SVM	0.89	0.67	95.05%
Polynomial SVM	0.88	0.67	95.1%
Neural Network	0.94	0.81	93.2%

Table 1: Model Performance Comparison

In this project, we primarily used ROC-AUC and PR-AUC as the key metrics to evaluate model performance. **ROC-AUC (Receiver Operating Characteristic - Area Under the Curve)** assesses the model's overall ability to discriminate between classes by evaluating the trade-off between true positive (TP) and false positive (FP) rates. **PR-AUC (Precision-Recall - Area Under the Curve)** focuses specifically on the model's precision and recall, making it especially relevant for imbalanced datasets where the minority class is of primary concern (Figure 3).

Logistic Regression: Logistic Regression served as a baseline, achieving ROC-AUC of 0.917 but a low PR-AUC of 0.700, highlighting difficulties with imbalanced datasets.

Random Forest: Random Forest achieved high ROC-AUC (0.947) but struggled with minority class patterns, as indicated by PR-AUC (0.788). Feature selection could improve predictive power.

XGBoost: XGBoost achieved 96.5% accuracy, AUC-ROC of 0.96, and AUC-PR of 0.83 on both datasets, demonstrating robustness to imbalanced data without oversampling.

LightGBM: LightGBM achieved 97.6% accuracy on preprocessed data and 97.3% on oversampled data, with AUC-ROC of 0.98 and AUC-PR of 0.90, slightly outperforming other models.

CatBoost: CatBoost reached 97.4% and 97.1% accuracy on preprocessed and oversampled data, respectively, with AUC-ROC up to 0.972 and AUC-PR up to 0.89, showing strong performance despite noise from oversampling.

RBF SVM: RBF SVM achieved 95.05% accuracy, AUC-ROC of 0.89, and AUC-PR of 0.67 on preprocessed data but degraded on oversampled data due to noise introduced by SMOTE.

Polynomial SVM: Polynomial SVM achieved 95.13% accuracy, AUC-ROC of 0.886, and AUC-PR of 0.67 on preprocessed data. Oversampling improved AUC-ROC to 0.91 but reduced AUC-PR to 0.62 and accuracy to 88%.

Neural Network: Neural Network achieved 94.3% and 93.2% accuracy on training and testing sets, with AUC-ROC up to 0.94 and AUC-PR up to 0.81, validating the architecture and regularization.

6 Discussion and Interpretation

6.1 Importance of Categorical Features

Looking at the results in Table 1, we found that tree-based models performed better than other models in this case. This is likely because tree-based models can naturally handle categorical features, while other models require categorical features to be encoded first, which may lose some information in the process. Furthermore, Based on the experimental results and the importance features (Figure 4), the number of categorical features among the top 20 most important features varied across models: XGBoost (1), Random Forest (2), LightGBM (5), and CatBoost (5). This variation correlates positively with each model's performance on the test set. Additionally, we observed that certain categorical features ranked higher in importance in LightGBM and CatBoost compared to Random Forest and XGBoost, suggesting that these categorical features provide substantial information gain. Therefore, we infer that categorical features play a critical role in this task. The differences in how these models utilize categorical features can be attributed to their distinct approaches to tree construction and growth, as well as their methods for handling categorical data. These differences ultimately result in the observed variations in model performance.

6.2 Impact of Oversampling for Imbalanced Data

Despite the imbalance of original data, the large dataset size allows boosting models to effectively capture the underlying relationships between transactions and fraud without requiring oversampling. However, when oversampling techniques are applied, they introduce synthetic data that may not accurately represent the true distribution of the positive class. This additional noise can lead boosting models to overfit on the artificial patterns, thereby degrading their generalization performance. Furthermore, boosting models are inherently robust to imbalanced data due to their iterative focus on misclassified samples, reducing the necessity of oversampling and highlighting its detrimental impact in this case.

6.3 Performance Comparison and Feature Analysis of LightGBM and CatBoost

CatBoost is well-known for its powerful native algorithms for handling categorical features, which are designed to provide significant performance gains, particularly in datasets where categorical data play a crucial role. However, contrary to expectations, CatBoost slightly underperformed compared to LightGBM in this task. This result necessitated further investigation into the characteristics of the dataset and the models' handling of feature types.

Low Proportion of Categorical Features: The dataset contains a very small proportion (5.2%) of categorical features, accounting for less than 10% of all features. This numerical dominance favors models like LightGBM, which excel at efficiently handling numerical data. In contrast, CatBoost, while highly optimized for categorical features, could not fully leverage its strength due to the sparsity of such features in this dataset. Consequently, its native handling of categorical features provided less of an advantage in this specific scenario.

Impact of Category Cardinality: An analysis of the dataset reveals that most categorical features have low cardinality (Figure 4), with the majority containing only a few unique values. LightGBM, using simple preprocessing techniques, was able to achieve comparable results to CatBoost without requiring its specialized encoding mechanisms. For example, low-cardinality features can be effectively handled with straightforward approaches like one-hot encoding or frequency encoding, reducing the need for CatBoost's sophisticated methods. Only a few categorical features, such as

`Deviceinfo` and `id_33`, exhibit high cardinality, and their impact on model performance appears limited in this dataset.

Implications for Model Selection: While CatBoost's advanced algorithms for encoding categorical data typically outperform other models in datasets rich with categorical features, its performance advantage diminishes in datasets like this one, where categorical features are both sparse and low in cardinality. LightGBM, by efficiently processing numerical data and handling categorical data adequately, emerges as the better-performing model. This result highlights the importance of aligning model selection with dataset characteristics, particularly in terms of the distribution and nature of feature types.

In conclusion, the combination of a low proportion of categorical features and their predominantly low cardinality limited CatBoost's ability to showcase its strengths in this task, allowing LightGBM to outperform it by focusing on its efficiency with numerical data. Future studies could explore whether additional feature engineering or targeted augmentation of categorical data might shift this balance in favor of CatBoost.

7 Error Analysis

One of the most significant challenges in this project was encoding categorical features effectively. Initially, we avoided using one-hot encoding due to concerns about data sparsity and the high dimensionality it would introduce, particularly for logistic regression, SVM models and Neural Network. Instead, we explored frequency encoding and target encoding as alternatives, and even considered implementing ordered target encoding, which simulates the encoding method in CatBoost. Unfortunately, none of these approaches proved satisfactory. Frequency encoding, which assigns the frequency of each category as its numerical representation, was straightforward to implement. However, it has a critical drawback: when two distinct categories share the same frequency, the model treats them as identical, potentially reducing predictive performance. Target encoding, on the other hand, leverages label information, introducing the risk of data leakage. Although k-fold target encoding can mitigate this risk, the added complexity and remaining potential for leakage made us hesitant to adopt it. Finally, ordered target encoding—a refined version of target encoding—addresses data leakage but comes with a computational complexity of $O(n^2)$. Given the large size of our dataset, this method was computationally infeasible.

8 Conclusion

This project ensured balanced class representation and reduced computational complexity. LightGBM demonstrated the highest performance, achieving a ROC-AUC of 0.98 and PR-AUC of 0.90, showcasing its strength in handling numerical features and imbalanced data. SVM models provided insights into non-linear decision boundaries but struggled with imbalanced class recall. Neural networks highlighted the potential for hierarchical feature interaction while maintaining robust performance. The study underscores the importance of aligning model selection with dataset characteristics and emphasizes the trade-offs between precision, recall, and computational efficiency. Remaining challenges include handling noise introduced by oversampling and refining feature engineering to optimize performance further.

REFERENCES

Ieee-cis fraud detection. https://www.kaggle.com/competitions/ieee-fraud-detection.

Iain Brown and Christophe Mues. An experimental comparison of classification algorithms for imbalanced credit scoring data sets. *Expert Systems with Applications*, 39(3):3446–3453, 2012. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2011.09.033. URL https://www.sciencedirect.com/science/article/pii/S095741741101342X.

Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002. URL https://www.jair.org/index.php/jair/article/view/10302.

Liao Chen, Ning Jia, Hongke Zhao, Yanzhe Kang, Jiang Deng, and Shoufeng Ma. Refined analysis and a hierarchical multi-task learning approach for loan fraud detection. *Journal of Management Science and Engineering*, 7(4):589–607, 2022.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794. ACM, 2016. URL https://www.kdd.org/kdd2016/papers/files/rfp0697-chenAemb.pdf.

Yuh-Jen Chen, Wan-Ching Liou, Yuh-Min Chen, and Jyun-Han Wu. Fraud detection for financial statements of business groups. *International Journal of Accounting Information Systems*, 32: 1–23, 2019.

Johannes Jurgovsky, Michael Granitzer, Konstantin Ziegler, Sylvie Calabretto, Pierre-Edouard Portier, Liyun He-Guelton, and Olivier Caelen. Sequence classification for credit-card fraud detection. *Expert systems with applications*, 100:234–245, 2018.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, volume 30, pp. 3146–3154, 2017. URL https://papers.nips.cc/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

Yufeng Kou, Chang-Tien Lu, Sirirat Sirwongwattana, and Yo-Ping Huang. Survey of fraud detection techniques. 2:749–754, 2004.

Ranran Li, Zhaowei Liu, Yuanqing Ma, Dong Yang, and Shuaijie Sun. Internet financial fraud detection based on graph learning. *IEEE Transactions on Computational Social Systems*, 10(3): 1394–1401, 2022.

Santhosh Maddila, Somula Ramasubbareddy, and K. Govinda. *Crime and Fraud Detection Using Clustering Techniques*. Springer Singapore, Singapore, 2020. ISBN 978-981-15-2043-3. doi: 10.1007/978-981-15-2043-3_17. URL https://doi.org/10.1007/978-981-15-2043-3_17.

Nghia Nguyen, Truc Duong, Tram Chau, Van-Ho Nguyen, Trang Trinh, Duy Tran, and Thanh Ho. A proposed model for card fraud detection based on catboost and deep neural network. *IEEE Access*, 10:96852–96861, 2022.

Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: Unbiased boosting with categorical features. 31:6638–6648, 2018. URL https://papers.nips.cc/paper/7898-catboost-unbiased-boosting-with-categorical-features.pdf.

Matt Rej. Credit card fraud statistics (2024), November 7 2023. URL https://merchantcostconsulting.com/lower-credit-card-processing-fees/credit-card-fraud-statistics/.

Abhimanyu Roy, Jingyi Sun, Robert Mahoney, Loreto Alonzi, Stephen Adams, and Peter Beling. Deep learning detecting fraud in credit card transactions. pp. 129–134, 2018.

Erich Schubert, Michael Weiler, and Arthur Zimek. Outlier detection and trend detection: two sides of the same coin. pp. 40–46, 2015.

Niek Tax, Kees Jan de Vries, Mathijs de Jong, Nikoleta Dosoula, Bram van den Akker, Jon Smith, Olivier Thuong, and Lucas Bernardi. Machine learning for fraud detection in e-commerce: A research agenda. pp. 30–54, 2021.

A Appendix

A.1 Figures

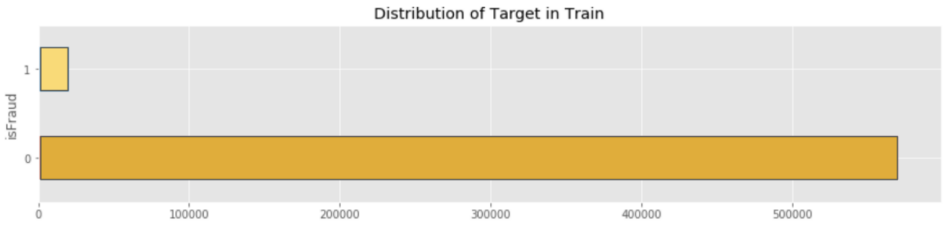
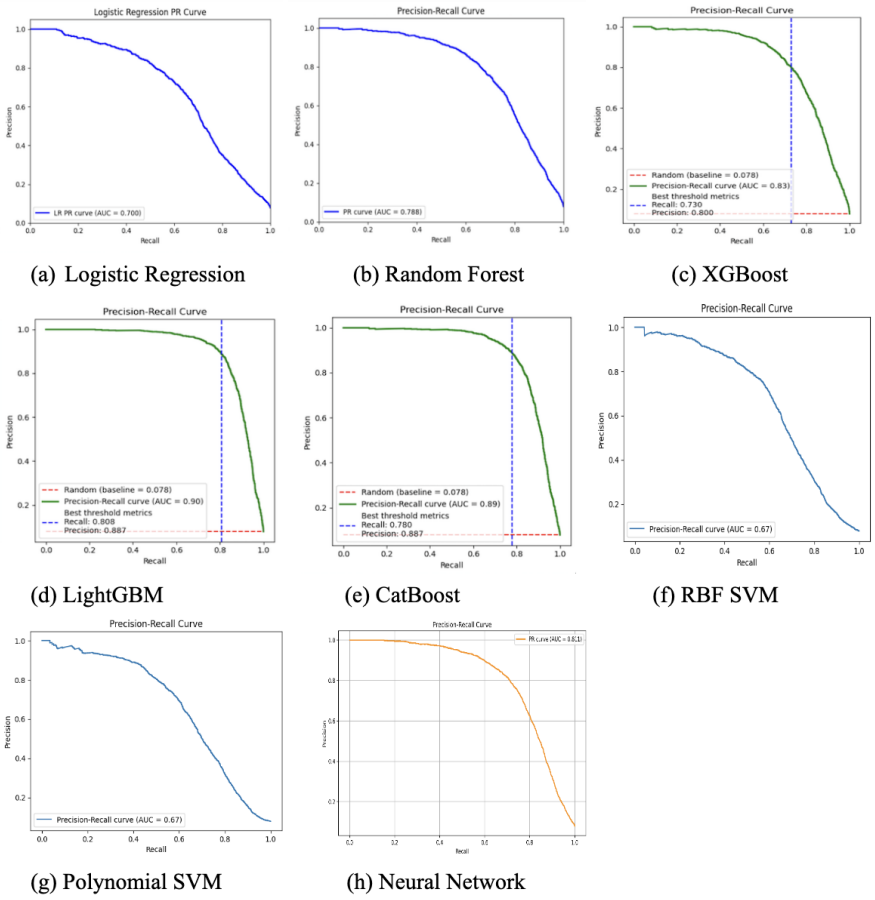


Figure 1: Distribution of Target in Train

Null Percentage	Number of Features
100% Missing	21
96% - 97% Missing	9
68% - 80% Missing	6
40% -60% Missing	151
2% - 18% Missing	133
0% - 1% Missing	92
0% Missing	22 (including the outcome variable)

Figure 2: Missing Features Analysis



(a) Logistic Regression (b) Random Forest (c) XGBoost

(d) LightGBM (e) CatBoost (f) RBF SVM

(g) Polynomial SVM (h) Neural Network

Figure 3: Model Performance Plots for Eight Models. Each subplot represents the PR-AUC curve for a different machine learning model

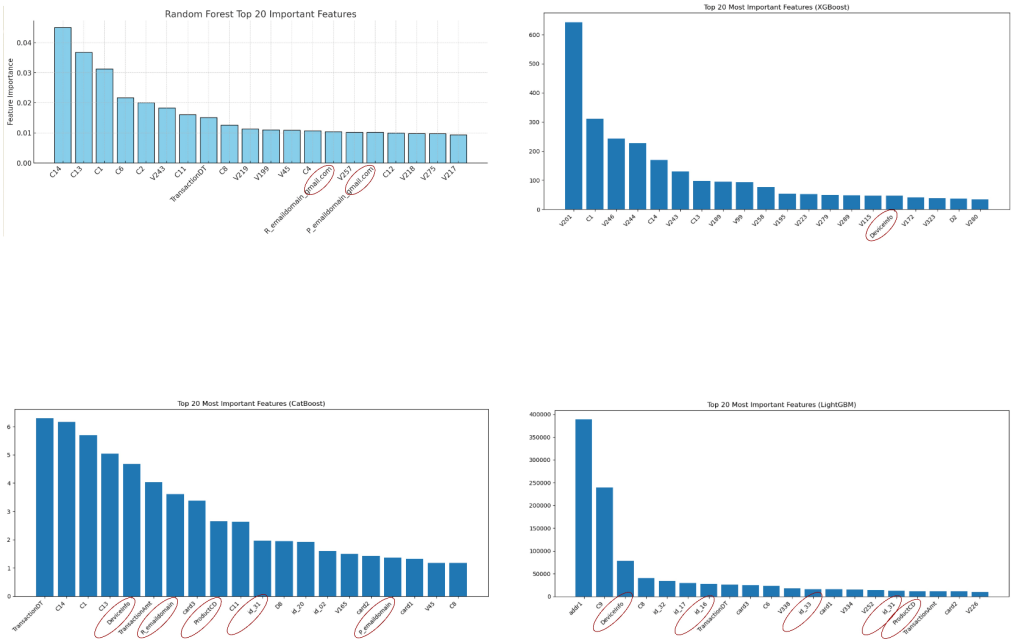


Figure 4: Feature Importance Rankings Derived from Information Gain, with Categorical Features Highlighted

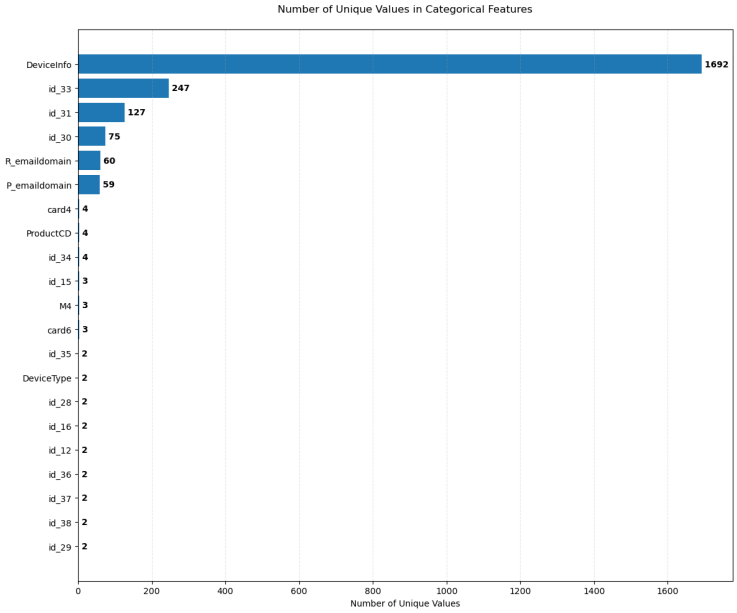


Figure 5: Number of Distinct Values per Categorical Feature (Cardinality Distribution)

```python
"""
Created on FRI Nov 22 2024

Ensemble Methods: Bagging
 -- Random Forests --

@author: Jiang
"""

'''
Ensemble models are like a "team" of models working together to give better predictions.
There are two main ways to build these teams: Bagging and Boosting. In this py, we will focus on bagging.
Bagging (e.g., Random Forest):
    Each model in the team gets a random subset of the data and predicts independently.
    Their outputs are then combined (e.g., by majority voting for classification tasks).

'''

# ================================================================
#                           Step 1: Loading
# ================================================================
print('----------------------------------------------------------')
print('Step 1: Loading')
# ----------------------------------------------------------------
#                           import packages and data
# ----------------------------------------------------------------
# Data Analysis
import pandas as pd
import numpy as np

# Data Visualization
from matplotlib import pyplot as plt
import seaborn as sns

# Machine Learning
from sklearn.metrics import roc_auc_score
from sklearn.ensemble import RandomForestClassifier

# Warnings
import warnings
warnings.filterwarnings('ignore')

# Load data
final_train_df = pd.read_csv('CDA project/final_train_df.csv')
final_test_df = pd.read_csv('CDA project/final_test_df.csv')

# Split X and y
X_train = final_train_df.drop('isFraud', axis=1)
y_train = final_train_df['isFraud']
X_test = final_test_df.drop('isFraud', axis=1)
y_test = final_test_df['isFraud']

# 使用相同的随机森林模型
rfc = RandomForestClassifier(criterion='entropy', max_features='sqrt', max_samples=0.5, min_samples_split=80)
rfc.fit(X_train, y_train)
y_predproba = rfc.predict_proba(X_test)
print(f'Test AUC={roc_auc_score(y_test, y_predproba[:, 1])}')

# # 计算ROC曲线所需的数据
# from sklearn.metrics import roc_curve, auc, precision_recall_curve
# fpr, tpr, _ = roc_curve(y_test, y_predproba[:, 1])
# roc_auc = auc(fpr, tpr)

# # 计算PR曲线所需的数据
# precision, recall, _ = precision_recall_curve(y_test, y_predproba[:, 1])
# pr_auc = auc(recall, precision)

# # 创建子图
# fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# # 绘制ROC曲线
# ax1.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.3f})')
# ax1.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
# ax1.set_xlim([0.0, 1.0])
# ax1.set_ylim([0.0, 1.05])
# ax1.set_xlabel('False Positive Rate')
# ax1.set_ylabel('True Positive Rate')
# ax1.set_title('Receiver Operating Characteristic (ROC) Curve')
# ax1.legend(loc="lower right")

```

```python
# # 绘制PR曲线
# ax2.plot(recall, precision, color='blue', lw=2, label=f'PR curve (AUC = {pr_auc:.3f})')
# ax2.set_xlim([0.0, 1.0])
# ax2.set_ylim([0.0, 1.05])
# ax2.set_xlabel('Recall')
# ax2.set_ylabel('Precision')
# ax2.set_title('Precision-Recall Curve')
# ax2.legend(loc="lower left")

# plt.tight_layout()
# plt.savefig('curves.png')

# 特征重要性分析
plt.figure(figsize=(12, 8))  # 调整为单图大小

try:
    # 计算随机森林的特征重要性
    importances = rfc.feature_importances_
    std = np.std([tree.feature_importances_ for tree in rfc.estimators_], axis=0)

    # 创建特征重要性DataFrame
    feature_importance_rf = pd.DataFrame({
        'feature': X_train.columns,
        'importance': importances,
        'std': std
    })
    feature_importance_rf = feature_importance_rf.sort_values('importance', ascending=False).head(20)

    # 绘制条形图（不使用xerr参数）
    ax = sns.barplot(
        x='importance',
        y='feature',
        data=feature_importance_rf,
        palette='viridis'
    )

    # 手动添加误差条
    for i, row in feature_importance_rf.iterrows():
        ax.errorbar(
            x=row['importance'],
            y=i,
            xerr=row['std'],
            color='black',
            capsize=3,
            capthick=1,
            elinewidth=1,
            linestyle=''
        )

    # 添加标题和标签
    plt.title('Random Forest Feature Importance (Top 20)', pad=20, fontsize=12)
    plt.xlabel('Importance Score', fontsize=10)
    plt.ylabel('Features', fontsize=10)

    # 添加网格线便于阅读
    plt.grid(axis='x', linestyle='--', alpha=0.6)

    # 调整布局
    plt.tight_layout()

    # 保存图片
    plt.savefig('feature_importance.png', dpi=300, bbox_inches='tight')
    plt.show()

    # 打印详细的特征重要性报告
    print("\nRandom Forest Top 20 Important Features:")
    print("----------------------------------------")
    importance_report = feature_importance_rf.copy()
    importance_report['importance'] = importance_report['importance'].round(4)
    importance_report['std'] = importance_report['std'].round(4)
    print(importance_report.to_string(index=False))

    # 计算累积重要性
    cumulative_importance = np.cumsum(feature_importance_rf['importance'])
    print("\n累积重要性:")
    for i, cum_imp in enumerate(cumulative_importance, 1):
        print(f"Top {i} 特征累积重要性: {cum_imp:.4f}")

except Exception as e:
    print(f"生成特征重要性图时发生错误: {str(e)}")
```

```python
"""
Created on Sun Oct 20 2024

Logistic Regression

@author: Jiang
"""
# ----------------------------------------------------------------------------
#                            import packages and data
# ----------------------------------------------------------------------------
# Data Analysis
import pandas as pd
import numpy as np

# Data Visualization
from matplotlib import pyplot as plt

# Machine Learning
from sklearn.metrics import roc_curve, auc, precision_recall_curve
from sklearn.metrics import roc_auc_score


# Warnings
import warnings
warnings.filterwarnings('ignore')

# Load data
final_train_df = pd.read_csv('CDA project/final_train_df.csv')
final_test_df = pd.read_csv('CDA project/final_test_df.csv')

# Split X and y
X_train = final_train_df.drop('isFraud', axis=1)
y_train = final_train_df['isFraud']
X_test = final_test_df.drop('isFraud', axis=1)
y_test = final_test_df['isFraud']
from sklearn.linear_model import LogisticRegression

# 创建和训练逻辑回归模型
lr = LogisticRegression(max_iter=1000)
lr.fit(X_train, y_train)
lr_predproba = lr.predict_proba(X_test)
print(f'Logistic Regression Test AUC={roc_auc_score(y_test, lr_predproba[:, 1])}')

# 计算ROC曲线数据
fpr_lr, tpr_lr, _ = roc_curve(y_test, lr_predproba[:, 1])
roc_auc_lr = auc(fpr_lr, tpr_lr)

# 计算PR曲线数据
precision_lr, recall_lr, _ = precision_recall_curve(y_test, lr_predproba[:, 1])
pr_auc_lr = auc(recall_lr, precision_lr)

# 创建子图
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# 绘制ROC曲线
ax1.plot(fpr_lr, tpr_lr, color='darkorange', lw=2, label=f'LR ROC curve (AUC = {roc_auc_lr:.3f})')
ax1.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
ax1.set_xlim([0.0, 1.0])
ax1.set_ylim([0.0, 1.05])
ax1.set_xlabel('False Positive Rate')
ax1.set_ylabel('True Positive Rate')
ax1.set_title('Logistic Regression ROC Curve')
ax1.legend(loc="lower right")

# 绘制PR曲线
ax2.plot(recall_lr, precision_lr, color='blue', lw=2, label=f'LR PR curve (AUC = {pr_auc_lr:.3f})')
ax2.set_xlim([0.0, 1.0])
ax2.set_ylim([0.0, 1.05])
ax2.set_xlabel('Recall')
ax2.set_ylabel('Precision')
ax2.set_title('Logistic Regression PR Curve')
ax2.legend(loc="lower left")

plt.tight_layout()
plt.savefig('lr_curves.png')
```

```
by: Wenxin Jiang

Since the data is big in size,
we will use function to reduce its memory for fast processing and consuming less storage.
'''
def reduce_mem_usage(df):
    """ iterate through all the columns of a dataframe and modify the data type
        to reduce memory usage.
    """
    start_mem = df.memory_usage().sum() / 1024 ** 2
    print('Memory usage of dataframe is {:.2f} MB'.format(start_mem))

    for col in df.columns:
        col_type = df[col].dtype

        if col_type != object:
            c_min = df[col].min()
            c_max = df[col].max()
            if str(col_type)[:3] == 'int':
                if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
                    df[col] = df[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
                    df[col] = df[col].astype(np.int16)
                elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
                    df[col] = df[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:
                    df[col] = df[col].astype(np.int64)
            else:
                if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.float16).max:
                    df[col] = df[col].astype(np.float16)
                elif c_min > np.finfo(np.float32).min and c_max < np.finfo(np.float32).max:
                    df[col] = df[col].astype(np.float32)
                else:
                    df[col] = df[col].astype(np.float64)
        else:
            df[col] = df[col].astype('category')

    end_mem = df.memory_usage().sum() / 1024 ** 2
    print('Memory usage after optimization is: {:.2f} MB'.format(end_mem))
    print('Decreased by {:.1f}%'.format(100 * (start_mem - end_mem) / start_mem))

    return df
```

In [ ]:
```
import pandas as pd
import numpy as np

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import accuracy_score, roc_curve, auc, precision_recall

import matplotlib.pyplot as plt
```

In [ ]:
```
# RBF SVM before SMOTE
train = pd.read_csv('filled_encoded_data/train_data_filled_encoded.csv')
test = pd.read_csv('filled_encoded_data/test_data_filled_encoded.csv')

X_train, X_val, y_train, y_val = train_test_split(train.drop('isFraud', axis
X_test = test.drop('isFraud', axis=1)
y_test = test['isFraud']
X_sample, _, y_sample, _ = train_test_split(
    X_train, y_train,
    test_size=0.8,  # Retain only 20% of the original data
    stratify=y_train,
    random_state=42
)
scaler = StandardScaler()
X_sample_scaled = scaler.fit_transform(X_sample)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)

pca = PCA(n_components=0.9)
X_sample_pca = pca.fit_transform(X_sample_scaled)
X_val_pca = pca.transform(X_val_scaled)
X_test_pca = pca.transform(X_test_scaled)

grid_search_X = X_sample_pca
grid_search_y = y_sample

param_grid = {
    'C': [0.1, 1, 10],
    'gamma': [1e-3, 1e-2, 0.1]
}

rbf_model = SVC(kernel='rbf', random_state=42)
grid_search = GridSearchCV(rbf_model, param_grid, scoring='pr_auc', cv=3, n_
grid_search.fit(grid_search_X, grid_search_y)

model = SVC(kernel='rbf', C=100, gamma=0.0001, random_state=42)
model.fit(X_sample_pca, y_sample)
y_val_pred = model.predict(X_val_pca)
accuracy_train = accuracy_score(y_val, y_val_pred)
y_val_scores = model.decision_function(X_val_pca)
fpr, tpr, thresholds = roc_curve(y_val, y_val_scores)
```

```
roc_auc = auc(fpr, tpr)
precision, recall, _ = precision_recall_curve(y_val, y_val_scores)
pr_auc = auc(recall, precision)

y_test_pred = model.predict(X_test_pca)
test_accuracy = accuracy_score(y_test, y_test_pred)
y_test_scores = model.decision_function(X_test_pca)
fpr, tpr, _ = roc_curve(y_test, y_test_scores)
roc_auc = auc(fpr, tpr)
precision, recall, _ = precision_recall_curve(y_test, y_test_scores)
pr_auc = auc(recall, precision)

# Precision-Recall Curve
plt.figure()
plt.plot(recall, precision, label=f"Precision-Recall curve (AUC = {pr_auc:.2
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve")
plt.legend(loc="lower left")
plt.show()
```

In [ ]:

```
# RBF SVM after SMOTE
train = pd.read_csv('11.25_SMOTE_onehot/final_train_df.csv')
test = pd.read_csv('11.25_SMOTE_onehot/final_test_df.csv')

rbf_model = SVC(kernel='rbf', random_state=42)
grid_search = GridSearchCV(rbf_model, param_grid, scoring='pr_auc', cv=3, n_
grid_search.fit(grid_search_X, grid_search_y)

model = SVC(kernel='rbf', C=100, gamma=0.0001, random_state=42)
model.fit(X_sample_pca, y_sample)
y_val_pred = model.predict(X_val_pca)
accuracy_train = accuracy_score(y_val, y_val_pred)
y_val_scores = model.decision_function(X_val_pca)
fpr, tpr, thresholds = roc_curve(y_val, y_val_scores)
roc_auc = auc(fpr, tpr)
precision, recall, _ = precision_recall_curve(y_val, y_val_scores)
pr_auc = auc(recall, precision)

y_test_pred = model.predict(X_test_pca)
test_accuracy = accuracy_score(y_test, y_test_pred)
y_test_scores = model.decision_function(X_test_pca)
fpr, tpr, _ = roc_curve(y_test, y_test_scores)
roc_auc = auc(fpr, tpr)
precision, recall, _ = precision_recall_curve(y_test, y_test_scores)
pr_auc = auc(recall, precision)

# Precision-Recall Curve
plt.figure()
plt.plot(recall, precision, label=f"Precision-Recall curve (AUC = {pr_auc:.2
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve")
plt.legend(loc="lower left")
plt.show()
```

In [ ]:

```
# Polynomial SVM before SMOTE
param_grid = {
    'C': [0.1, 1, 10],
    'degree': [2, 3],
    'gamma': ['scale', 0.1],
    'coef0': [0, 1]
}

grid_search = GridSearchCV(
    estimator=SVC(kernel='poly', random_state=42),
    param_grid=param_grid,
    cv=3,  # Stratified k-fold cross-validation
    scoring='pr_auc',
    verbose=2,
    n_jobs=-1
)

grid_search.fit(X_sample_pca, y_sample)

print("Best Parameters:", grid_search.best_params_)
print("Best Cross-Validation Score:", grid_search.best_score_)

model = SVC(
    kernel='poly', C=10,
    degree=3, gamma='scale',
    coef0=1, random_state=42)

model.fit(X_sample_pca, y_sample)

y_val_pred = model.predict(X_val_pca)

accuracy_train = accuracy_score(y_val, y_val_pred)

y_val_scores = model.decision_function(X_val_pca)

# ROC Curve
fpr_train, tpr_train, _ = roc_curve(y_val, y_val_scores)
roc_auc_train = auc(fpr_train, tpr_train)

# Percision Recall Curve
precision_train, recall_train, _ = precision_recall_curve(y_val, y_val_score
pr_auc_train = auc(recall_train, precision_train)

y_test_pred = model.predict(X_test_pca)

test_accuracy = accuracy_score(y_test, y_test_pred)

y_test_scores = model.decision_function(X_test_pca)

# ROC Curve
fpr_test, tpr_test, _ = roc_curve(y_test, y_test_scores)
roc_auc_test = auc(fpr_test, tpr_test)

# Percision Recall Curve
```

```
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_score
pr_auc_test = auc(recall_test, precision_test)
```

In [ ]:

```
# Polynomial SVM after SMOTE
train = pd.read_csv('11.25_SMOTE_onehot/final_train_df.csv')
test = pd.read_csv('11.25_SMOTE_onehot/final_test_df.csv')

model = SVC(
    kernel='poly', C=10,
    degree=3, gamma='scale',
    coef0=1, random_state=42)

model.fit(X_sample_pca, y_sample)
y_val_pred = model.predict(X_val_pca)

accuracy_train = accuracy_score(y_val, y_val_pred)

y_val_scores = model.decision_function(X_val_pca)

# ROC Curve
fpr_train, tpr_train, _ = roc_curve(y_val, y_val_scores)
roc_auc_train = auc(fpr_train, tpr_train)

# Percision Recall Curve
precision_train, recall_train, _ = precision_recall_curve(y_val, y_val_score
pr_auc_train = auc(recall_train, precision_train)

y_test_pred = model.predict(X_test_pca)

test_accuracy = accuracy_score(y_test, y_test_pred)

y_test_scores = model.decision_function(X_test_pca)

# ROC Curve
fpr_test, tpr_test, _ = roc_curve(y_test, y_test_scores)
roc_auc_test = auc(fpr_test, tpr_test)

# Percision Recall Curve
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_score
pr_auc_test = auc(recall_test, precision_test)
```

```python
import os
import pandas as pd
import numpy as np
from tqdm import tqdm
import torch
from torch.utils.data import DataLoader
import argparse
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from utils.reader import DataReader
from utils.loader import FraudDataLoader
from utils.encoder import DataEncoder, get_encoded_data
from utils.validation import calculate_metrics
from sklearn.metrics import precision_recall_curve


# Path configurations
TRAIN_TRANSACTION_PATH = 'data/train_transaction.csv'
TRAIN_IDENTITY_PATH = 'data/train_identity.csv'
TEST_TRANSACTION_PATH = 'data/test_transaction.csv'
TEST_IDENTITY_PATH = 'data/test_identity.csv'
PROCESSED_DATA_PATH = 'data/processed/train_data'
PROCESSED_TEST_PATH = 'data/processed/test_data'
MODEL_SAVE_DIR = 'checkpoints'

# Training configurations
class Config:
    # Model parameters
    model_architecture = {
        'hidden_layers': [128, 64, 32, 16],
        'dropout_rate': 0.01,
    }

    # Training parameters
    num_epochs = 50
    steps_per_epoch = 200
    batch_size = 32
    learning_rate = 0.001
    fraud_ratio = 0.5  # For balanced batches

    # Encoder parameters
    encoder_params = {
        'onehot_threshold': 10,
        'outlier_threshold': -999,
        'std_threshold': 5,
        'cache_dir': 'encoder_cache',
        'force_recompute': False
    }

    # Model saving/loading
    model_save_path = os.path.join(MODEL_SAVE_DIR, 'fraud_detector.pth')
    save_every_n_epochs = 10

    # Visualization parameters
    plot_dir = 'plots'
    plot_metrics_every_n_epochs = 10

    # Runtime options
    load_model = True
    resume_training = False
    test_only = False
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def plot_auc_history(auc_scores, save_path='plots/validation_auc_history.png'):
    """Plot AUC scores over epochs."""
    plt.figure(figsize=(10, 6))
```

```
    plt.plot(range(1, len(auc_scores) + 1), auc_scores, 'b', label='Validation AUC'
)
    plt.xlabel('Epoch')
    plt.ylabel('AUC Score')
    plt.title('Validation AUC over Training')
    plt.grid(True)
    plt.legend()

    # Save plot
    os.makedirs(os.path.dirname(save_path), exist_ok=True)
    plt.savefig(save_path)
    plt.close()

def print_metrics(metrics, phase='Validation'):
    """Print metrics in a consistent format."""
    print(f"\n{phase} Results:")
    print(f"Accuracy: {metrics['Accuracy']*100:.2f}%")
    print("\nConfusion Matrix:")
    print(f"True Positives (TP): {metrics['TP']}")
    print(f"True Negatives (TN): {metrics['TN']}")
    print(f"False Positives (FP): {metrics['FP']}")
    print(f"False Negatives (FN): {metrics['FN']}")
    print(f"Total Samples: {metrics['Total']}")
    print("\nDetailed Metrics:")
    print(f"True Positive Rate (Sensitivity/Recall): {metrics['TPR']:.4f}")
    print(f"True Negative Rate (Specificity): {metrics['TNR']:.4f}")
    print(f"False Positive Rate: {metrics['FPR']:.4f}")
    print(f"False Negative Rate: {metrics['FNR']:.4f}")
    print(f"Precision: {metrics['Precision']:.4f}")
    print(f"F1 Score: {metrics['F1']:.4f}")

def create_model(input_dim, config):
    """Create a neural network model based on configuration."""
    layers = []
    prev_dim = input_dim

    # Create hidden layers
    for hidden_dim in config.model_architecture['hidden_layers']:
        layers.extend([
            torch.nn.Linear(prev_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Dropout(config.model_architecture['dropout_rate'])
        ])
        prev_dim = hidden_dim

    # Add output layer
    layers.append(torch.nn.Linear(prev_dim, 1))

    return torch.nn.Sequential(*layers)

def plot_roc_curve(labels, predictions, phase='Validation'):
    """Plot ROC curve and calculate AUC."""
    if torch.is_tensor(labels):
        labels = labels.cpu().numpy()
    if torch.is_tensor(predictions):
        predictions = predictions.cpu().numpy()

    fpr, tpr, _ = roc_curve(labels, predictions)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2,
             label=f'ROC curve (AUC = {roc_auc:.3f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
```

```
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'{phase} ROC Curve')
    plt.legend(loc="lower right")

    os.makedirs('plots', exist_ok=True)
    plt.savefig(f'plots/{phase.lower()}_roc_curve.png')
    plt.close()

    return roc_auc

def calculate_pr_auc(y_true, y_pred):
    """Calculate Precision-Recall AUC."""
    precision, recall, _ = precision_recall_curve(y_true, y_pred)
    pr_auc = auc(recall, precision)

    # Plot PR curve
    plt.figure(figsize=(10, 6))
    plt.plot(recall, precision, color='darkorange', lw=2,
             label=f'PR curve (AUC = {pr_auc:.3f})')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.grid(True)
    plt.legend()

    # Save plot
    plt.savefig('plots/pr_curve.png', bbox_inches='tight', dpi=300)
    plt.close()

    return pr_auc

def save_checkpoint(model, optimizer, epoch, loss, config):
    """Save model checkpoint."""
    os.makedirs(MODEL_SAVE_DIR, exist_ok=True)
    checkpoint = {
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict() if \
          config.resume_training else None,
       'loss': loss,
    }
    torch.save(checkpoint, config.model_save_path)
    print(f"Checkpoint saved at epoch {epoch}")

def load_checkpoint(model, optimizer, config):
    """Load model checkpoint."""
    if os.path.exists(config.model_save_path):
        checkpoint = torch.load(config.model_save_path, weights_only=True, \
                                map_loction=config.device)
        model.load_state_dict(checkpoint['model_state_dict'])
        if config.resume_training:
            optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
        start_epoch = checkpoint.get('epoch', 0) if config.resume_training else 0
        print(f"Loaded checkpoint from epoch {start_epoch}")
        return start_epoch
    else:
        print("No checkpoint found. Starting from scratch.")
        return 0

if __name__ == "__main__":
    config = Config()

    # Process and split train data
    train_reader = DataReader(
        identity_path=TRAIN_IDENTITY_PATH,
```

```
        transaction_path=TRAIN_TRANSACTION_PATH,
        processed_path=PROCESSED_DATA_PATH
    )

    # Save processed train splits
    if not os.path.exists(PROCESSED_DATA_PATH + "_train_fraud_features.npy"):
        print("\nSaving processed train splits...")
        train_reader.save_splits(PROCESSED_DATA_PATH)

    # Process test data
    test_reader = DataReader(
        identity_path=TEST_IDENTITY_PATH,
        transaction_path=TEST_TRANSACTION_PATH,
        processed_path=PROCESSED_TEST_PATH,
        is_test=True
    )

    # Save processed test data
    if not os.path.exists(PROCESSED_TEST_PATH + "_test_all_features.npy"):
        print("\nSaving processed test data...")
        test_reader.save_splits(PROCESSED_TEST_PATH)

    """ Model setup """
    # Create dataloader and encoder
    dataloader = FraudDataLoader(train_reader=train_reader,
                            test_reader=test_reader, batch_size=config.batch_size)
    encoder = DataEncoder(
        train_reader,
        test_reader,
        **config.encoder_params
    )

    # Get input dimension and create model
    X_sample, _ = dataloader.get_balanced_batch('train', \
                        fraud_ratio=config.fraud_ratio)
    X_encoded = encoder.encode_batch(X_sample)
    input_dim = X_encoded.shape[1]
    print(f"Input dimension: {input_dim}")

    # Create model and optimizer
    model = create_model(input_dim, config).to(config.device)
    optimizer = torch.optim.Adam(model.parameters(), lr=config.learning_rate)
    criterion = torch.nn.BCEWithLogitsLoss()

    # Load checkpoint if requested
    start_epoch = load_checkpoint(model, optimizer, config) if \
                (config.load_model or config.resume_training) else 0

    if not config.test_only:
        # Initialize AUC history
        auc_history = []

        # Training loop
        for epoch in range(start_epoch, config.num_epochs):
            model.train()
            train_loss = 0

            # Training
            for i in (pbar := tqdm(range(config.steps_per_epoch))):
                X_batch, y_batch = dataloader.get_balanced_batch('train', \
                                        fraud_ratio=config.fraud_ratio)
                X_batch_encoded = encoder.encode_batch(X_batch)

                X_batch_tensor = torch.tensor(X_batch_encoded, dtype=torch.float32)
                                            \.to(config.device)
                y_batch_tensor = torch.tensor(y_batch, dtype=torch.float32).\
```

```
                                                to(config.device)

                optimizer.zero_grad()
                outputs = model(X_batch_tensor)
                loss = criterion(outputs.squeeze(), y_batch_tensor)

                loss.backward()
                optimizer.step()

                train_loss += loss.item()
                pbar.set_description(f"Epoch {epoch+1}/{config.num_epochs}, \
                                       loss = {loss.item():.4f}")

            avg_train_loss = train_loss / config.steps_per_epoch

            # Validation
            model.eval()
            val_loss = 0
            all_preds = []
            all_labels = []
            raw_outputs = []

            X_val, y_val = get_encoded_data(train_reader, encoder, split='val')
            val_loader = DataLoader(
                list(zip(X_val, y_val)),
                batch_size=config.batch_size,
                shuffle=False
            )

            with torch.no_grad():
                for X_batch, y_batch in val_loader:
                    X_batch = X_batch.to(config.device)
                    y_batch = y_batch.to(config.device)

                    outputs = model(X_batch)
                    loss = criterion(outputs.squeeze(), y_batch.float())
                    val_loss += loss.item()

                    predicted = (outputs.squeeze() > 0.5).float()
                    all_preds.extend(predicted.cpu())
                    all_labels.extend(y_batch.cpu())
                    raw_outputs.extend(outputs.squeeze().cpu())

            avg_val_loss = val_loss / len(val_loader)
            print(f"Epoch {epoch+1}/{config.num_epochs}, \
                  train loss = {avg_train_loss:.4f}, val loss = {avg_val_loss:.4f})


            # Calculate and store AUC score
            raw_predictions = torch.sigmoid(torch.tensor(raw_outputs))
            auc_score = plot_roc_curve(
                torch.tensor(all_labels),
                raw_predictions,
                f'Validation_epoch_{epoch+1}'
            )
            auc_history.append(auc_score)
            print(f"Validation AUC: {auc_score:.4f}")

            precision, recall, _ = precision_recall_curve(torch.tensor(all_labels).
                                    \cpu(), raw_predictions.cpu())
            pr_auc = auc(recall, precision)
            print(f"Validation PR AUC: {pr_auc:.4f}")

            # Plot AUC history
            if (epoch + 1) % config.plot_metrics_every_n_epochs == 0:
                plot_auc_history(auc_history)
```

```
                val_metrics = calculate_metrics(torch.tensor(all_labels), \
                              torch.tensor(all_preds))
                print_metrics(val_metrics, 'Validation')

            if (epoch + 1) % config.save_every_n_epochs == 0:
                save_checkpoint(model, optimizer, epoch + 1, avg_train_loss, confi)


    # Test evaluation
    print("\nRunning test evaluation...")
    model.eval()
    test_loss = 0
    all_preds = []
    all_labels = []
    raw_outputs = []

    X_test, y_test = get_encoded_data(test_reader, encoder, split='test')
    test_loader = DataLoader(
        list(zip(X_test, y_test)),
        batch_size=config.batch_size,
        shuffle=False
    )
    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            X_batch = X_batch.to(config.device)
            y_batch = y_batch.to(config.device)

            outputs = model(X_batch)
            loss = criterion(outputs.squeeze(), y_batch.float())
            test_loss += loss.item()

            predicted = (outputs.squeeze() > 0.5).float()
            all_preds.extend(predicted.cpu())
            all_labels.extend(y_batch.cpu())
            raw_outputs.extend(outputs.squeeze().cpu())

    # Calculate and print test metrics
test_metrics = calculate_metrics(torch.tensor(all_labels), \
                                 torch.tensor(all_preds))
    print_metrics(test_metrics, 'Test')

    # Calculate and plot ROC curve for test set
    raw_predictions = torch.sigmoid(torch.tensor(raw_outputs))
    auc_score = plot_roc_curve(
        torch.tensor(all_labels),
        raw_predictions,
        'Test'
    )
    print(f"Test AUC: {auc_score:.4f}")
```

```python
import os
import pandas as pd
import numpy as np
from tqdm import tqdm
import torch
from torch.utils.data import DataLoader
import argparse
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from utils.reader import DataReader
from utils.loader import FraudDataLoader
from utils.encoder import DataEncoder, get_encoded_data
from utils.validation import calculate_metrics
from sklearn.metrics import precision_recall_curve


# Path configurations
TRAIN_TRANSACTION_PATH = 'data/train_transaction.csv'
TRAIN_IDENTITY_PATH = 'data/train_identity.csv'
TEST_TRANSACTION_PATH = 'data/test_transaction.csv'
TEST_IDENTITY_PATH = 'data/test_identity.csv'
PROCESSED_DATA_PATH = 'data/processed/train_data'
PROCESSED_TEST_PATH = 'data/processed/test_data'
MODEL_SAVE_DIR = 'checkpoints'

# Training configurations
class Config:
    # Model parameters
    model_architecture = {
        'hidden_layers': [128, 64, 32, 16],
        'dropout_rate': 0.01,
    }

    # Training parameters
    num_epochs = 50
    steps_per_epoch = 200
    batch_size = 32
    learning_rate = 0.001
    fraud_ratio = 0.5  # For balanced batches

    # Encoder parameters
    encoder_params = {
        'onehot_threshold': 10,
        'outlier_threshold': -999,
        'std_threshold': 5,
        'cache_dir': 'encoder_cache',
        'force_recompute': False
    }

    # Model saving/loading
    model_save_path = os.path.join(MODEL_SAVE_DIR, 'fraud_detector.pth')
    save_every_n_epochs = 10

    # Visualization parameters
    plot_dir = 'plots'
    plot_metrics_every_n_epochs = 10

    # Runtime options
    load_model = True
    resume_training = False
    test_only = False
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def plot_auc_history(auc_scores, save_path='plots/validation_auc_history.png'):
    """Plot AUC scores over epochs."""
    plt.figure(figsize=(10, 6))
```

```
    plt.plot(range(1, len(auc_scores) + 1), auc_scores, 'b', label='Validation AUC'
)
    plt.xlabel('Epoch')
    plt.ylabel('AUC Score')
    plt.title('Validation AUC over Training')
    plt.grid(True)
    plt.legend()

    # Save plot
    os.makedirs(os.path.dirname(save_path), exist_ok=True)
    plt.savefig(save_path)
    plt.close()

def print_metrics(metrics, phase='Validation'):
    """Print metrics in a consistent format."""
    print(f"\n{phase} Results:")
    print(f"Accuracy: {metrics['Accuracy']*100:.2f}%")
    print("\nConfusion Matrix:")
    print(f"True Positives (TP): {metrics['TP']}")
    print(f"True Negatives (TN): {metrics['TN']}")
    print(f"False Positives (FP): {metrics['FP']}")
    print(f"False Negatives (FN): {metrics['FN']}")
    print(f"Total Samples: {metrics['Total']}")
    print("\nDetailed Metrics:")
    print(f"True Positive Rate (Sensitivity/Recall): {metrics['TPR']:.4f}")
    print(f"True Negative Rate (Specificity): {metrics['TNR']:.4f}")
    print(f"False Positive Rate: {metrics['FPR']:.4f}")
    print(f"False Negative Rate: {metrics['FNR']:.4f}")
    print(f"Precision: {metrics['Precision']:.4f}")
    print(f"F1 Score: {metrics['F1']:.4f}")

def create_model(input_dim, config):
    """Create a neural network model based on configuration."""
    layers = []
    prev_dim = input_dim

    # Create hidden layers
    for hidden_dim in config.model_architecture['hidden_layers']:
        layers.extend([
            torch.nn.Linear(prev_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Dropout(config.model_architecture['dropout_rate'])
        ])
        prev_dim = hidden_dim

    # Add output layer
    layers.append(torch.nn.Linear(prev_dim, 1))

    return torch.nn.Sequential(*layers)

def plot_roc_curve(labels, predictions, phase='Validation'):
    """Plot ROC curve and calculate AUC."""
    if torch.is_tensor(labels):
        labels = labels.cpu().numpy()
    if torch.is_tensor(predictions):
        predictions = predictions.cpu().numpy()

    fpr, tpr, _ = roc_curve(labels, predictions)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2,
             label=f'ROC curve (AUC = {roc_auc:.3f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
```

```
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'{phase} ROC Curve')
    plt.legend(loc="lower right")

    os.makedirs('plots', exist_ok=True)
    plt.savefig(f'plots/{phase.lower()}_roc_curve.png')
    plt.close()

    return roc_auc

def calculate_pr_auc(y_true, y_pred):
    """Calculate Precision-Recall AUC."""
    precision, recall, _ = precision_recall_curve(y_true, y_pred)
    pr_auc = auc(recall, precision)

    # Plot PR curve
    plt.figure(figsize=(10, 6))
    plt.plot(recall, precision, color='darkorange', lw=2,
             label=f'PR curve (AUC = {pr_auc:.3f})')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.grid(True)
    plt.legend()

    # Save plot
    plt.savefig('plots/pr_curve.png', bbox_inches='tight', dpi=300)
    plt.close()

    return pr_auc

def save_checkpoint(model, optimizer, epoch, loss, config):
    """Save model checkpoint."""
    os.makedirs(MODEL_SAVE_DIR, exist_ok=True)
    checkpoint = {
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict() if \
          config.resume_training else None,
       'loss': loss,
    }
    torch.save(checkpoint, config.model_save_path)
    print(f"Checkpoint saved at epoch {epoch}")

def load_checkpoint(model, optimizer, config):
    """Load model checkpoint."""
    if os.path.exists(config.model_save_path):
        checkpoint = torch.load(config.model_save_path, weights_only=True, \
                                map_loction=config.device)
        model.load_state_dict(checkpoint['model_state_dict'])
        if config.resume_training:
            optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
        start_epoch = checkpoint.get('epoch', 0) if config.resume_training else 0
        print(f"Loaded checkpoint from epoch {start_epoch}")
        return start_epoch
    else:
        print("No checkpoint found. Starting from scratch.")
        return 0

if __name__ == "__main__":
    config = Config()

    # Process and split train data
    train_reader = DataReader(
        identity_path=TRAIN_IDENTITY_PATH,
```

```
        transaction_path=TRAIN_TRANSACTION_PATH,
        processed_path=PROCESSED_DATA_PATH
    )

    # Save processed train splits
    if not os.path.exists(PROCESSED_DATA_PATH + "_train_fraud_features.npy"):
        print("\nSaving processed train splits...")
        train_reader.save_splits(PROCESSED_DATA_PATH)

    # Process test data
    test_reader = DataReader(
        identity_path=TEST_IDENTITY_PATH,
        transaction_path=TEST_TRANSACTION_PATH,
        processed_path=PROCESSED_TEST_PATH,
        is_test=True
    )

    # Save processed test data
    if not os.path.exists(PROCESSED_TEST_PATH + "_test_all_features.npy"):
        print("\nSaving processed test data...")
        test_reader.save_splits(PROCESSED_TEST_PATH)

    """ Model setup """
    # Create dataloader and encoder
    dataloader = FraudDataLoader(train_reader=train_reader,
                            test_reader=test_reader, batch_size=config.batch_size)
    encoder = DataEncoder(
        train_reader,
        test_reader,
        **config.encoder_params
    )

    # Get input dimension and create model
    X_sample, _ = dataloader.get_balanced_batch('train', \
                    fraud_ratio=config.fraud_ratio)
    X_encoded = encoder.encode_batch(X_sample)
    input_dim = X_encoded.shape[1]
    print(f"Input dimension: {input_dim}")

    # Create model and optimizer
    model = create_model(input_dim, config).to(config.device)
    optimizer = torch.optim.Adam(model.parameters(), lr=config.learning_rate)
    criterion = torch.nn.BCEWithLogitsLoss()

    # Load checkpoint if requested
    start_epoch = load_checkpoint(model, optimizer, config) if \
                (config.load_model or config.resume_training) else 0

    if not config.test_only:
        # Initialize AUC history
        auc_history = []

        # Training loop
        for epoch in range(start_epoch, config.num_epochs):
            model.train()
            train_loss = 0

            # Training
            for i in (pbar := tqdm(range(config.steps_per_epoch))):
                X_batch, y_batch = dataloader.get_balanced_batch('train', \
                                        fraud_ratio=config.fraud_ratio)
                X_batch_encoded = encoder.encode_batch(X_batch)

                X_batch_tensor = torch.tensor(X_batch_encoded, dtype=torch.float32)
                                            \.to(config.device)
                y_batch_tensor = torch.tensor(y_batch, dtype=torch.float32).\
```

```
                                                to(config.device)

                optimizer.zero_grad()
                outputs = model(X_batch_tensor)
                loss = criterion(outputs.squeeze(), y_batch_tensor)

                loss.backward()
                optimizer.step()

                train_loss += loss.item()
                pbar.set_description(f"Epoch {epoch+1}/{config.num_epochs}, \
                                       loss = {loss.item():.4f}")

            avg_train_loss = train_loss / config.steps_per_epoch

            # Validation
            model.eval()
            val_loss = 0
            all_preds = []
            all_labels = []
            raw_outputs = []

            X_val, y_val = get_encoded_data(train_reader, encoder, split='val')
            val_loader = DataLoader(
                list(zip(X_val, y_val)),
                batch_size=config.batch_size,
                shuffle=False
            )

            with torch.no_grad():
                for X_batch, y_batch in val_loader:
                    X_batch = X_batch.to(config.device)
                    y_batch = y_batch.to(config.device)

                    outputs = model(X_batch)
                    loss = criterion(outputs.squeeze(), y_batch.float())
                    val_loss += loss.item()

                    predicted = (outputs.squeeze() > 0.5).float()
                    all_preds.extend(predicted.cpu())
                    all_labels.extend(y_batch.cpu())
                    raw_outputs.extend(outputs.squeeze().cpu())

            avg_val_loss = val_loss / len(val_loader)
            print(f"Epoch {epoch+1}/{config.num_epochs}, \
                  train loss = {avg_train_loss:.4f}, val loss = {avg_val_loss:.4f}")


            # Calculate and store AUC score
            raw_predictions = torch.sigmoid(torch.tensor(raw_outputs))
            auc_score = plot_roc_curve(
                torch.tensor(all_labels),
                raw_predictions,
                f'Validation_epoch_{epoch+1}'
            )
            auc_history.append(auc_score)
            print(f"Validation AUC: {auc_score:.4f}")

            precision, recall, _ = precision_recall_curve(torch.tensor(all_labels).
                                   \cpu(), raw_predictions.cpu())
            pr_auc = auc(recall, precision)
            print(f"Validation PR AUC: {pr_auc:.4f}")

            # Plot AUC history
            if (epoch + 1) % config.plot_metrics_every_n_epochs == 0:
                plot_auc_history(auc_history)
```

```
                val_metrics = calculate_metrics(torch.tensor(all_labels), \
                              torch.tensor(all_preds))
                print_metrics(val_metrics, 'Validation')

            if (epoch + 1) % config.save_every_n_epochs == 0:
                save_checkpoint(model, optimizer, epoch + 1, avg_train_loss, confi)


    # Test evaluation
    print("\nRunning test evaluation...")
    model.eval()
    test_loss = 0
    all_preds = []
    all_labels = []
    raw_outputs = []

    X_test, y_test = get_encoded_data(test_reader, encoder, split='test')
    test_loader = DataLoader(
        list(zip(X_test, y_test)),
        batch_size=config.batch_size,
        shuffle=False
    )
    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            X_batch = X_batch.to(config.device)
            y_batch = y_batch.to(config.device)

            outputs = model(X_batch)
            loss = criterion(outputs.squeeze(), y_batch.float())
            test_loss += loss.item()

            predicted = (outputs.squeeze() > 0.5).float()
            all_preds.extend(predicted.cpu())
            all_labels.extend(y_batch.cpu())
            raw_outputs.extend(outputs.squeeze().cpu())

    # Calculate and print test metrics
test_metrics = calculate_metrics(torch.tensor(all_labels), \
                                 torch.tensor(all_preds))
    print_metrics(test_metrics, 'Test')

    # Calculate and plot ROC curve for test set
    raw_predictions = torch.sigmoid(torch.tensor(raw_outputs))
    auc_score = plot_roc_curve(
        torch.tensor(all_labels),
        raw_predictions,
        'Test'
    )
    print(f"Test AUC: {auc_score:.4f}")
```

ISyE 6740 Project - Fraud Detection

In [ ]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import StratifiedKFold
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, roc_auc_score, f1_score, prec
from sklearn.metrics import precision_recall_curve, roc_curve, auc
import xgboost as xgb

import lightgbm as lgb
from lightgbm.callback import early_stopping

from catboost import CatBoostClassifier, Pool
```

1.Data Loading

In [2]:
```
train_df = pd.read_csv('train_split.csv')
test_df = pd.read_csv('test_split.csv')
```

In [5]:
```
oversample_train = pd.read_csv('final_train_df.csv')
oversample_test = pd.read_csv('final_test_df.csv')
```

In [ ]:
```
def reverse_one_hot_encoding(encoded_df, original_categorical_features):
    """
    reverse one-hot encoding for the encoded data to get the original cat

    Args:
        encoded_df: the DataFrame after one-hot encoding
        original_categorical_features: the list of original categorical f

    Returns:
        the DataFrame after reverse one-hot encoding
    """
    decoded_df = encoded_df.copy()

    # process each original categorical feature
    for original_feature in original_categorical_features:
        # find all one-hot columns that start with the original feature n
        one_hot_cols = [col for col in encoded_df.columns if col.startswi

        if one_hot_cols:
            # find the index of the column with value 1
            category_values = [col.split(f"{original_feature}_")[1] for c
            decoded_df[original_feature] = encoded_df[one_hot_cols].idxma
                lambda x: x.split(f"{original_feature}_")[1])
```

```
            # drop one-hot columns
            decoded_df = decoded_df.drop(columns=one_hot_cols)

    return decoded_df

# reverse one-hot encoding for training set
categorical_features = train_df.select_dtypes(include=['object']).columns
decoded_oversample_train_df = reverse_one_hot_encoding(oversample_train,

# reverse one-hot encoding for test set
decoded_oversample_test_df = reverse_one_hot_encoding(oversample_test, ca
```

2. Data Preproccessing

In [8]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 115386 entries, 0 to 115385
Columns: 434 entries, TransactionID to DeviceInfo
dtypes: float64(407), int64(4), object(23)
memory usage: 382.1+ MB
```

In [9]:
```
train_df.nunique()
```

Out[9]:
```
TransactionID     115386
isFraud                2
TransactionDT     114463
TransactionAmt      7220
ProductCD              4
                   ...
id_36                  2
id_37                  2
id_38                  2
DeviceType             2
DeviceInfo          1692
Length: 434, dtype: int64
```

2.1 Fill categorical data by mode and numeric data by median

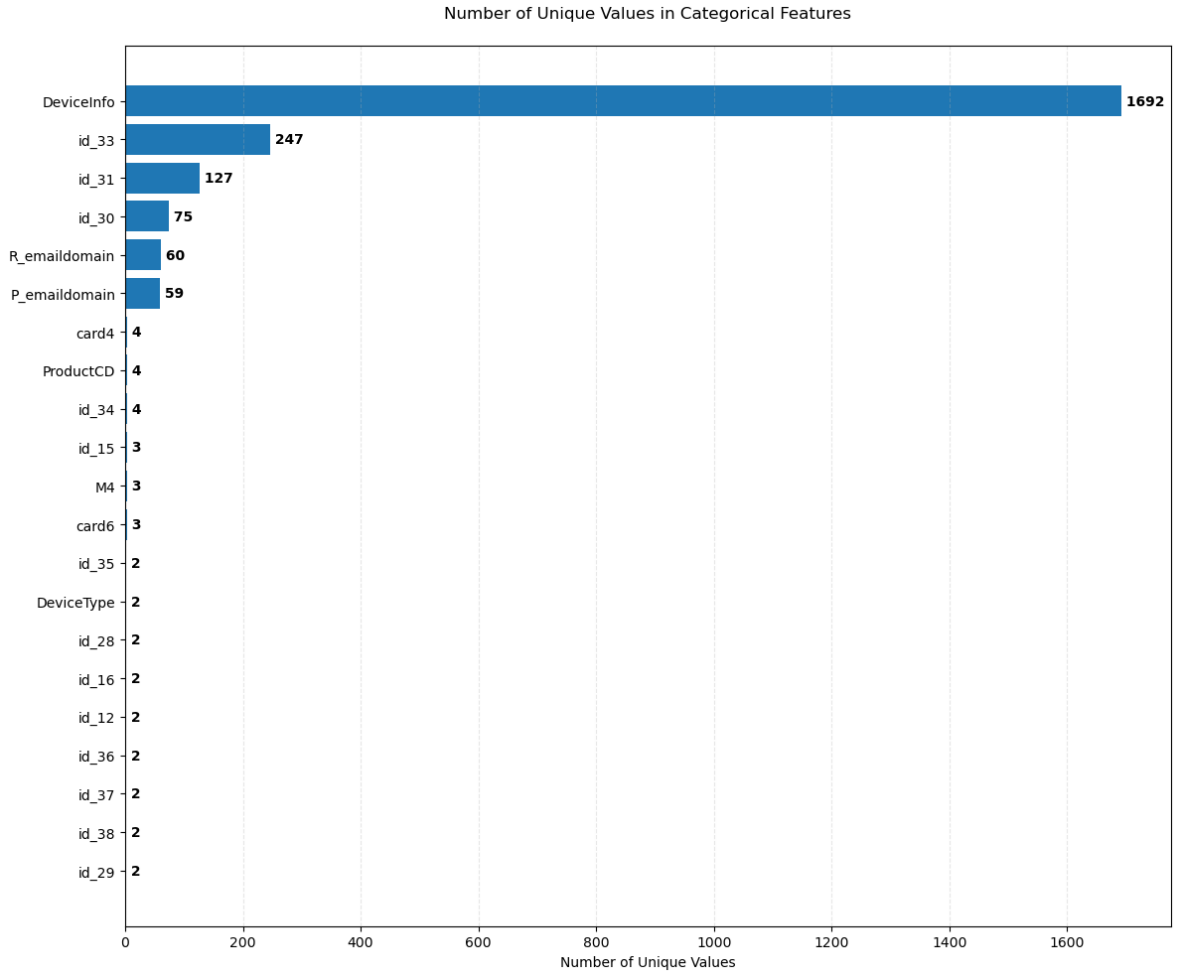
In [ ]:
```
# calculate the missing value ratio for each column
missing_ratio = train_df.isnull().sum() / len(train_df)

# get the column names with missing value ratio less than 80%
columns_to_keep = missing_ratio[missing_ratio < 0.8].index

# keep only these columns in training and test sets
train_df = train_df[columns_to_keep]
test_df = test_df[columns_to_keep]

# separate numeric and categorical features
numeric_features = train_df.select_dtypes(include=['int64', 'float64']).c
categorical_features = train_df.select_dtypes(include=['object']).columns

# fill missing values for numeric features by median
```

```
for feature in numeric_features:
    median_value = train_df[feature].median()
    train_df[feature].fillna(median_value, inplace=True)
    test_df[feature].fillna(median_value, inplace=True)

# fill missing values for categorical features by mode
for feature in categorical_features:
    mode_value = train_df[feature].mode()[0]
    train_df[feature].fillna(mode_value, inplace=True)
    test_df[feature].fillna(mode_value, inplace=True)
```

In [54]:
```
print('categorical features:', list(categorical_features))
```

```
categorical features: ['ProductCD', 'card4', 'card6', 'P_emaildomain', 'R_
emaildomain', 'M4', 'id_12', 'id_15', 'id_16', 'id_28', 'id_29', 'id_30',
'id_31', 'id_33', 'id_34', 'id_35', 'id_36', 'id_37', 'id_38', 'DeviceTyp
e', 'DeviceInfo']
```

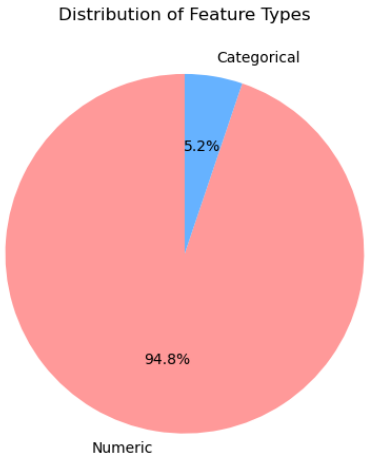
In [68]:
```
def plot_unique_counts(df, categorical_features):
    # calculate the number of unique values for each categorical feature
    unique_counts = pd.Series({feat: df[feat].nunique() for feat in categ
    unique_counts = unique_counts.sort_values(ascending=True)  # sort for

    # create a figure
    plt.figure(figsize=(12, 10))

    # plot horizontal bar chart
    bars = plt.barh(range(len(unique_counts)), unique_counts.values)

    # set y-axis labels
    plt.yticks(range(len(unique_counts)), unique_counts.index)

    # add value labels on the bars
    for i, bar in enumerate(bars):
        width = bar.get_width()
        plt.text(width, i, f' {int(width)}',
                 ha='left', va='center', fontweight='bold')

    # add title and labels
    plt.title('Number of Unique Values in Categorical Features', pad=20)
    plt.xlabel('Number of Unique Values')

    # add grid lines
    plt.grid(axis='x', linestyle='--', alpha=0.3)

    # adjust layout
    plt.tight_layout()
    plt.show()
# get all categorical features
categorical_features = train_df.select_dtypes(include=['object', 'categor

plot_unique_counts(train_df, categorical_features)
```

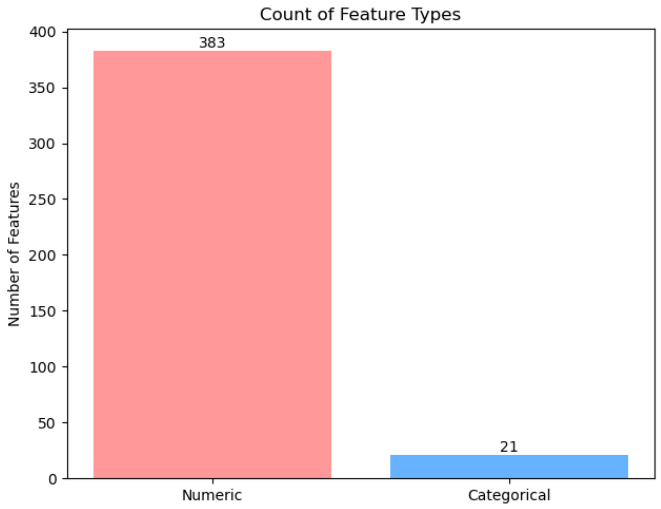


In [67]:

```python
def plot_feature_types_comparison(df):
    # get features of different types
    numeric_features = df.select_dtypes(include=['int64', 'float64']).col
    categorical_features = df.select_dtypes(include=['object', 'category'

    # calculate the number of features
    feature_counts = {
        'Numeric': len(numeric_features),
        'Categorical': len(categorical_features)
    }

    # create a figure
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    # 1. pie chart
    plt.sca(ax1)
    colors = ['#FF9999', '#66B2FF']
    plt.pie(feature_counts.values(),
            labels=feature_counts.keys(),
            autopct='%1.1f%%',
            colors=colors,
            startangle=90)
    plt.title('Distribution of Feature Types')

    # 2. bar chart
    plt.sca(ax2)
    bars = plt.bar(feature_counts.keys(), feature_counts.values(), color=
```

```
    # add value labels on the bars
    for bar in bars:
        height = bar.get_height()
        plt.text(bar.get_x() + bar.get_width()/2., height,
                f'{int(height)}',
                ha='center', va='bottom')

    plt.title('Count of Feature Types')
    plt.ylabel('Number of Features')

    plt.tight_layout()
    plt.show()

    print("\nDetailed Feature Type Information:")
    print(f"\nTotal number of features: {len(df.columns)}")
    print(f"Number of numeric features: {len(numeric_features)} ({len(num
    print(f"Number of categorical features: {len(categorical_features)} (

    print("\nNumeric Features:")
    for i, feat in enumerate(sorted(numeric_features), 1):
        print(f"{i}. {feat}")

    print("\nCategorical Features:")
    for i, feat in enumerate(sorted(categorical_features), 1):
        print(f"{i}. {feat}")

    # create a detailed statistics table
    stats_df = pd.DataFrame({
        'Feature': list(numeric_features) + list(categorical_features),
        'Type': ['Numeric'] * len(numeric_features) + ['Categorical'] * l
        'Unique_Values': [df[col].nunique() for col in numeric_features]
        'Missing_Values': [df[col].isnull().sum() for col in numeric_feat
        'Missing_Percentage': [(df[col].isnull().sum() / len(df) * 100).r
                               [(df[col].isnull().sum() / len(df) * 100).rou
    })

    print("\nDetailed Statistics:")
    print(stats_df.sort_values('Type').to_string(index=False))

plot_feature_types_comparison(train_df)
```





```
          V73     Numeric              7              0
0.0
         V111     Numeric              9              0
0.0
         V109     Numeric              7              0
0.0
         V108     Numeric              7              0
0.0
         V107     Numeric              1              0
0.0
         V106     Numeric             53              0
0.0
         V105     Numeric             82              0
0.0
         V104     Numeric             16              0
0.0
         V103     Numeric            786              0
0.0
         V102     Numeric           1051              0
0.0
         V101     Numeric            748              0
0.0
         V100     Numeric             18              0
0.0
         V110     Numeric              7              0
0.0
         V174     Numeric              9              0
0.0
```

2.1 one-hot encoding

In [11]:

```python
from sklearn.preprocessing import StandardScaler

# Step 1: Create copies of original dataframes
train_data_filled = train_df.copy()
test_data_filled = test_df.copy()

# Step 2: Identify numeric and categorical features
numeric_features = train_data_filled.select_dtypes(include=['int64', 'flo
numeric_features.remove('isFraud')
categorical_features = train_data_filled.select_dtypes(include=['object']

# Step 3: Standardize numeric features
scaler = StandardScaler()
train_data_filled[numeric_features] = scaler.fit_transform(train_data_fil
test_data_filled[numeric_features] = scaler.transform(test_data_filled[nu


# Step 5: Perform one-hot encoding
train_data_dummies = pd.get_dummies(train_data_filled, dtype=int)
test_data_dummies = pd.get_dummies(test_data_filled, dtype=int)

# Step 6: Ensure test set has same columns as train set
missing_cols = set(train_data_dummies.columns) - set(test_data_dummies.co
for col in missing_cols:
    test_data_dummies[col] = 0
```

```
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
```

```
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
```

```
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
/var/folders/my/lj8b6d8x6bndr88sqw0v5xw80000gn/T/ipykernel_59252/407083433
6.py:25: PerformanceWarning: DataFrame is highly fragmented.  This is usua
lly the result of calling `frame.insert` many times, which has poor perfor
mance.  Consider joining all columns at once using pd.concat(axis=1) inste
ad. To get a de-fragmented frame, use `newframe = frame.copy()`
  test_data_dummies[col] = 0
Train shape: (115386, 2682)
Test shape: (28847, 2682)
```

3. Model Training

3.1 XGBoost

3.1.1 Original data

In [42]:

```
X = train_df.drop(columns = ['TransactionID', 'isFraud'])
y = train_df['isFraud']
X[categorical_features] = X[categorical_features].fillna('missing').astyp

X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=0.1,  # 10% as validation set
    random_state=42,
    stratify=y  # keep class ratio
)

X_test = test_df.drop(columns = ['TransactionID', 'isFraud'])
y_test = test_df['isFraud']
X_test[categorical_features] = X_test[categorical_features].fillna('missi

categorical_features = X_train.select_dtypes(include=['object']).columns
for col in categorical_features:
    X_train[col] = X_train[col].astype('category')
    X_val[col] = X_val[col].astype('category')
    X_test[col] = X_test[col].astype('category')
```

```
# XGBoost model parameters
params = {
    'objective': 'binary:logistic',  # Logistic regression for binary cla
    'max_depth': 6,
    'learning_rate': 0.1,
    'min_child_weight': 3,     # prevent overfitting
    'subsample': 0.8,          # prevent overfitting
    'colsample_bytree': 0.8,
    'eval_metric': ['auc', 'logloss']  # Use logloss for better probabili
}

# Create DMatrix for XGBoost
dtrain = xgb.DMatrix(X_train, label=y_train, enable_categorical=True)
dval = xgb.DMatrix(X_val, label=y_val, enable_categorical=True)
dtest = xgb.DMatrix(X_test, label=y_test, enable_categorical=True)

# Train the model
evals = [(dtrain, 'train'), (dval, 'eval')]
xgb_model = xgb.train(params, dtrain, num_boost_round=200, evals=evals, e
```

```
[0]     train-auc:0.85862       train-logloss:0.27473   eval-auc:0.83902
eval-logloss:0.27631
[50]    train-auc:0.96842       train-logloss:0.09686   eval-auc:0.94126
eval-logloss:0.12492
[100]   train-auc:0.97876       train-logloss:0.07938   eval-auc:0.94869
eval-logloss:0.11578
[150]   train-auc:0.98429       train-logloss:0.06854   eval-auc:0.95227
eval-logloss:0.11125
[199]   train-auc:0.98790       train-logloss:0.06068   eval-auc:0.95464
eval-logloss:0.10824
```

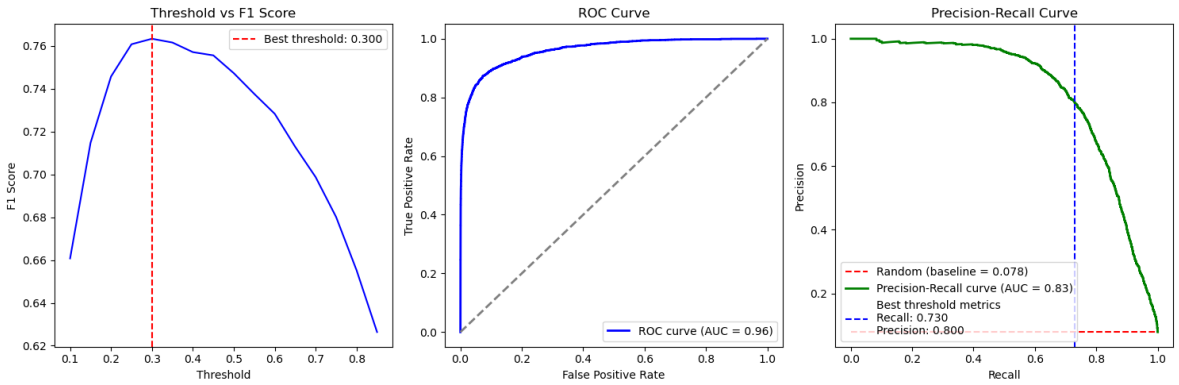
In [13]:

```
def visulize_result_xgb(model, dtest, y_test):
    # Get predicted probabilities
    y_pred_prob = model.predict(dtest)

    # Try different thresholds to find the best F1 score
    thresholds = np.arange(0.1, 0.9, 0.05)
    f1_scores = []
    best_threshold = 0.5
    best_f1 = 0

    for threshold in thresholds:
        y_pred = [1 if prob > threshold else 0 for prob in y_pred_prob]
        f1 = f1_score(y_test, y_pred)
        f1_scores.append(f1)
        if f1 > best_f1:
            best_f1 = f1
            best_threshold = threshold

    print(f"Best threshold: {best_threshold:.3f}")
    print(f"Best F1 Score: {best_f1:.3f}")

    # Use the best threshold for final predictions
    y_pred = [1 if prob > best_threshold else 0 for prob in y_pred_prob]

    # Calculate metrics with the best threshold
    metrics = {
```

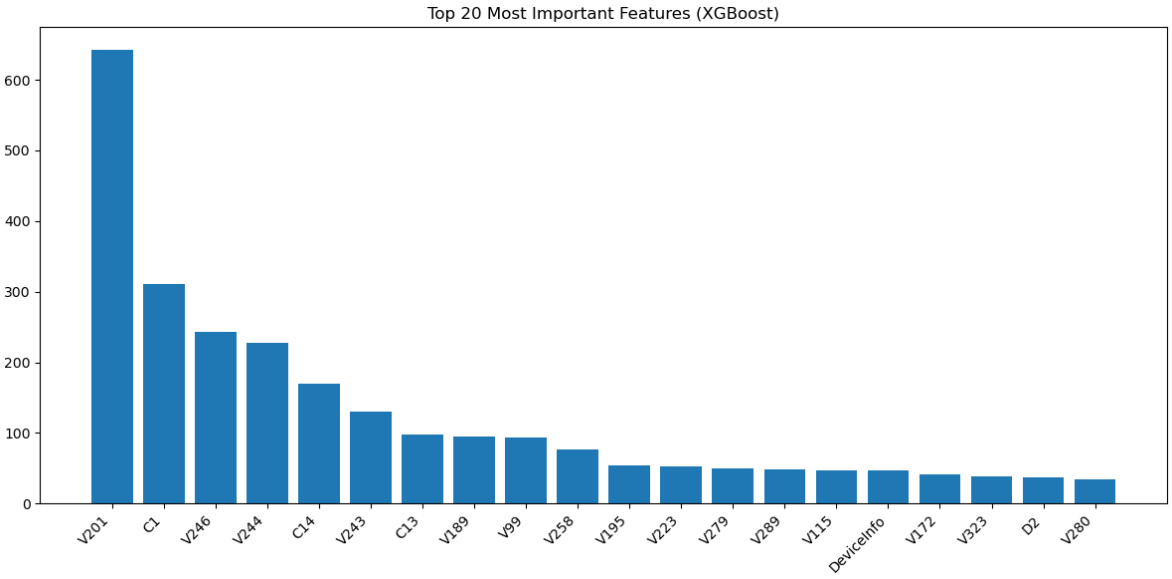
```
        'Accuracy': accuracy_score(y_test, y_pred),
        'AUC': roc_auc_score(y_test, y_pred_prob),
        'F1': f1_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred)
    }

    print("\nMetrics with optimized threshold:")
    print(metrics)
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # 可视化阈值与F1分数的关系
    plt.figure(figsize=(15, 5))

    # Plot threshold vs F1 score
    plt.subplot(1, 3, 1)
    plt.plot(thresholds, f1_scores, 'b-')
    plt.axvline(x=best_threshold, color='r', linestyle='--', label=f'Best
    plt.xlabel('Threshold')
    plt.ylabel('F1 Score')
    plt.title('Threshold vs F1 Score')
    plt.legend()

    # Plot ROC Curve
    plt.subplot(1, 3, 2)
    fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, color='blue', lw=2, label=f"ROC curve (AUC = {roc_
    plt.plot([0, 1], [0, 1], color="gray", linestyle="--", lw=2)
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curve")
    plt.legend(loc="lower right")

    # Plot Precision-Recall Curve
    plt.subplot(1, 3, 3)
    precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
    pr_auc = auc(recall, precision)  # 计算PR曲线下的面积

    no_skill = len(y_test[y_test == 1]) / len(y_test)  # Calculate ratio
    plt.plot([0, 1], [no_skill, no_skill], '--', color='red', label=f'Ran

    # Plot actual model performance
    plt.plot(recall, precision, color='green', lw=2, label=f"Precision-Re
    plt.axvline(x=metrics['Recall'], color='blue', linestyle='--',
                label=f'Best threshold metrics\nRecall: {metrics["Recall"]

    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.title("Precision-Recall Curve")
    plt.legend(loc="lower left")

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

In [14]:
```
visulize_result_xgb(xgb_model, dtest, y_test)
```

```
Best threshold: 0.300
Best F1 Score: 0.763

Metrics with optimized threshold:
{'Accuracy': 0.9645023745970118, 'AUC': 0.961044797154429, 'F1': 0.7634011
090573013, 'Precision': 0.8003875968992248, 'Recall': 0.7296819787985865}

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.98      0.98     26583
           1       0.80      0.73      0.76      2264

    accuracy                           0.96     28847
   macro avg       0.89      0.86      0.87     28847
weighted avg       0.96      0.96      0.96     28847
```



In [48]:
```python
feature_importance = xgb_model.get_score(importance_type='gain')
# convert to DataFrame and sort
importance_df = pd.DataFrame({
    'feature': list(feature_importance.keys()),
    'importance': list(feature_importance.values())
}).sort_values('importance', ascending=False)

# visualize the top 20 most important features
plt.figure(figsize=(12, 6))
plt.bar(importance_df['feature'][:20], importance_df['importance'][:20])
plt.xticks(rotation=45, ha='right')
plt.title('Top 20 Most Important Features (XGBoost)')
plt.tight_layout()
plt.show()
```



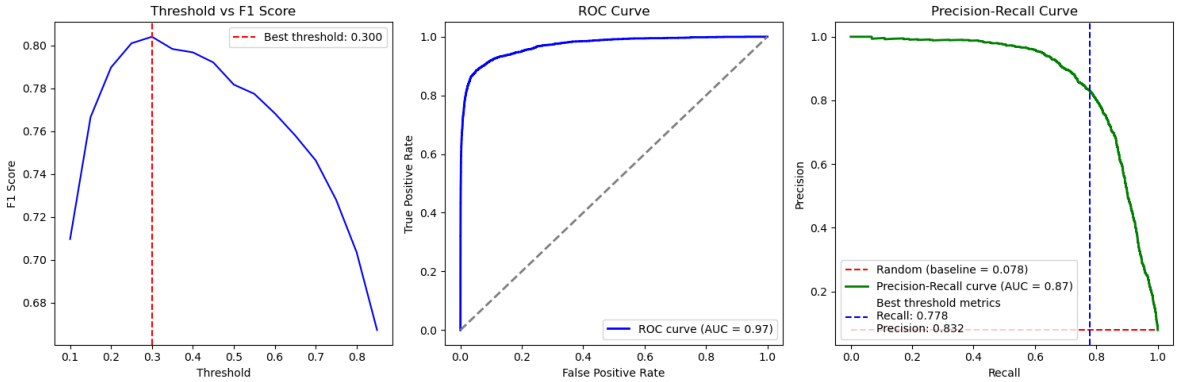
3.1.3 XGBoost one-hot encoding

In [15]:
```
X = train_data_dummies.drop(columns=['isFraud', 'TransactionID'])  # Drop
y = train_data_dummies['isFraud']
X_test = test_data_dummies.drop(columns=['isFraud', 'TransactionID'])
y_test = test_data_dummies['isFraud']
# Split the train_data
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, ra

# XGBoost model parameters
params = {
    'objective': 'binary:logistic',
    'max_depth': 6,
    'learning_rate': 0.1,
    'min_child_weight': 3,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'eval_metric': ['auc', 'logloss']  # Use logloss for better probabili
}

# Create DMatrix for XGBoost
dtrain = xgb.DMatrix(X_train, label=y_train, enable_categorical=True)
dval = xgb.DMatrix(X_val, label=y_val, enable_categorical=True)
dtest = xgb.DMatrix(X_test, label=y_test, enable_categorical=True)

# Train the model
evals = [(dtrain, 'train'), (dval, 'eval')]
xgb_model_one_hot = xgb.train(params, dtrain, num_boost_round=200, evals=
```

```
[0]     train-auc:0.88905       train-logloss:0.27431   eval-auc:0.88303
eval-logloss:0.27493
[50]    train-auc:0.96488       train-logloss:0.10187   eval-auc:0.95366
eval-logloss:0.11225
[100]   train-auc:0.97502       train-logloss:0.08691   eval-auc:0.96184
eval-logloss:0.10120
[150]   train-auc:0.98021       train-logloss:0.07755   eval-auc:0.96568
eval-logloss:0.09500
[199]   train-auc:0.98402       train-logloss:0.06991   eval-auc:0.96836
eval-logloss:0.09047
```

In [16]:
```
visulize_result_xgb(xgb_model_one_hot, dtest, y_test)
```

```
Best threshold: 0.300
Best F1 Score: 0.804

Metrics with optimized threshold:
{'Accuracy': 0.9702222068152667, 'AUC': 0.9705165892173976, 'F1': 0.804015
5144877937, 'Precision': 0.831524303916942, 'Recall': 0.7782685512367491}

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98     26583
           1       0.83      0.78      0.80      2264

    accuracy                           0.97     28847
   macro avg       0.91      0.88      0.89     28847
weighted avg       0.97      0.97      0.97     28847
```



3.1.4 OVERsample data XGB

In [39]:
```
X = decoded_oversample_train_df.drop(columns = ['isFraud'])
y = decoded_oversample_train_df['isFraud']

X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=0.1,  # 10% as validation set
    random_state=42,
    stratify=y  # keep class ratio
)

X_test = decoded_oversample_test_df.drop(columns = ['isFraud'])
```

```
y_test = decoded_oversample_test_df['isFraud']

categorical_features = X_train.select_dtypes(include=['object']).columns
for col in categorical_features:
    X_train[col] = X_train[col].astype('category')
    X_val[col] = X_val[col].astype('category')
    X_test[col] = X_test[col].astype('category')

# XGBoost model parameters
params = {
    'objective': 'binary:logistic',  # Logistic regression for binary cla
    'max_depth': 6,
    'learning_rate': 0.1,
    'min_child_weight': 3,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'eval_metric': ['auc', 'logloss']  # Use logloss for better probabili
}

# Create DMatrix for XGBoost
dtrain = xgb.DMatrix(X_train, label=y_train, enable_categorical=True)
dval = xgb.DMatrix(X_val, label=y_val, enable_categorical=True)
dtest = xgb.DMatrix(X_test, label=y_test, enable_categorical=True)

# Train the model
evals = [(dtrain, 'train'), (dval, 'eval')]
xgb_smote_model = xgb.train(params, dtrain, num_boost_round=200, evals=ev
```

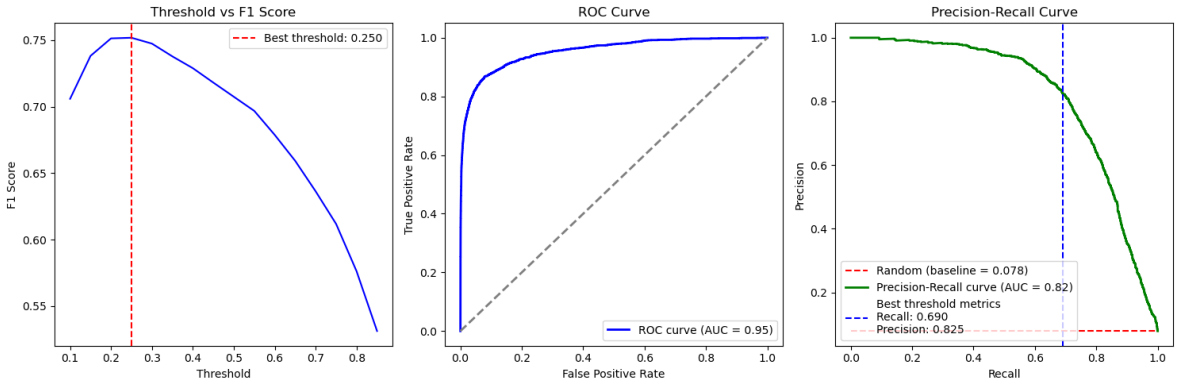
```
[0]     train-auc:0.95108       train-logloss:0.62776   eval-auc:0.93767
eval-logloss:0.63363
[50]    train-auc:0.99245       train-logloss:0.12924   eval-auc:0.98202
eval-logloss:0.21816
[100]   train-auc:0.99640       train-logloss:0.08363   eval-auc:0.99092
eval-logloss:0.17585
[150]   train-auc:0.99758       train-logloss:0.06477   eval-auc:0.99395
eval-logloss:0.15011
[199]   train-auc:0.99823       train-logloss:0.05372   eval-auc:0.99503
eval-logloss:0.13618
```

In [40]:
```
from sklearn.metrics import log_loss

y_pred_prob = xgb_smote_model.predict(dtest)
test_logloss = log_loss(y_test, y_pred_prob)
print(f"Test LogLoss: {test_logloss:.4f}")
```

```
Test LogLoss: 0.1130
```

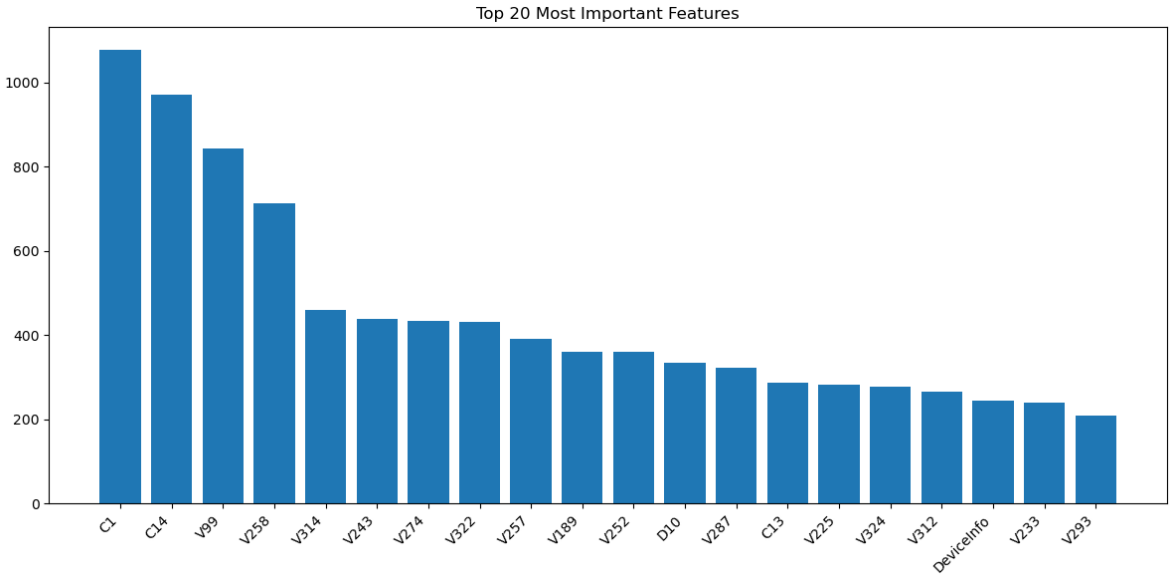
In [41]:
```
visulize_result_xgb(xgb_smote_model, dtest, y_test)
```

```
Best threshold: 0.250
Best F1 Score: 0.752

Metrics with optimized threshold:
{'Accuracy': 0.964225049398551, 'AUC': 0.9540168641745987, 'F1': 0.7518037
518037518, 'Precision': 0.8252375923970433, 'Recall': 0.6903710247349824}

Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98     26583
           1       0.83      0.69      0.75      2264

    accuracy                           0.96     28847
   macro avg       0.90      0.84      0.87     28847
weighted avg       0.96      0.96      0.96     28847
```



In [47]:

```python
feature_importance = xgb_smote_model.get_score(importance_type='gain')
# convert to DataFrame and sort
importance_df = pd.DataFrame({
    'feature': list(feature_importance.keys()),
    'importance': list(feature_importance.values())
}).sort_values('importance', ascending=False)

# visualize the top 20 most important features
plt.figure(figsize=(12, 6))
plt.bar(importance_df['feature'][:20], importance_df['importance'][:20])
plt.xticks(rotation=45, ha='right')
plt.title('Top 20 Most Important Features')
plt.tight_layout()
plt.show()
```



3.2 LightGBM

In [21]:
```
X = train_df.drop(columns = ['TransactionID', 'isFraud'])
y = train_df['isFraud']
# X[categorical_features] = X[categorical_features].fillna('missing').ast

X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=0.1,  # 20% as validation set
    random_state=42,
    stratify=y  # keep class ratio
)

X_test = test_df.drop(columns = ['TransactionID', 'isFraud'])
y_test = test_df['isFraud']
# X_test[categorical_features] = X_test[categorical_features].fillna('mis

categorical_features = X_train.select_dtypes(include=['object']).columns
for col in categorical_features:
    X_train[col] = X_train[col].astype('category')
    X_val[col] = X_val[col].astype('category')
    X_test[col] = X_test[col].astype('category')

dtrain = lgb.Dataset(X_train, label=y_train)
dval = lgb.Dataset(X_val, label=y_val, reference=dtrain)
dtest = lgb.Dataset(X_test, label=y_test)

# set parameters
params = {
    'objective': 'binary',
    'metric': ['auc', 'binary_logloss'],
    'n_estimators': 200,
    'num_leaves': 31,
    'learning_rate': 0.1,
    'feature_fraction': 0.8
}
```

```
# train model
bst = lgb.train(params, dtrain, num_boost_round=200, valid_sets=[dval], c
```

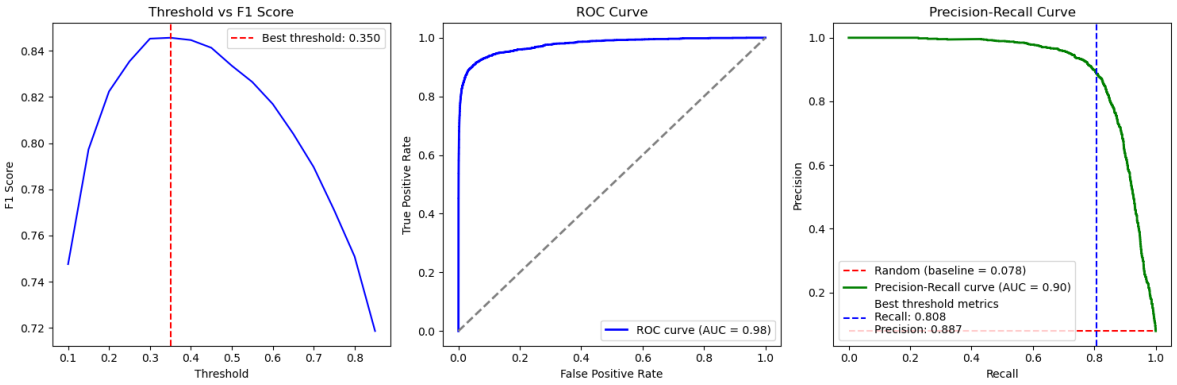
```
/Users/mac/opt/anaconda3/lib/python3.9/site-packages/lightgbm/engine.py:20
4: UserWarning: Found `n_estimators` in params. Will use it instead of arg
ument
  _log_warning(f"Found `{alias}` in params. Will use it instead of argumen
t")
[LightGBM] [Warning] Categorical features with more bins than the configur
ed maximum bin number found.
[LightGBM] [Warning] For categorical features, max_bin and max_bin_by_feat
ure may be ignored with a large number of categories.
[LightGBM] [Info] Number of positive: 8149, number of negative: 95698
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.103919 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 39490
[LightGBM] [Info] Number of data points in the train set: 103847, number o
f used features: 382
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.078471 -> initscore=-2.4
63302
[LightGBM] [Info] Start training from score -2.463302
Training until validation scores don't improve for 10 rounds
Did not meet early stopping. Best iteration is:
[198]   valid_0's auc: 0.972707 valid_0's binary_logloss: 0.0788765
```

In [22]:
```
def visulize_result_lgb(model, X_test, y_test):
        # Get predicted probabilities
    y_pred_prob = model.predict(X_test)

    # Try different thresholds to find the best F1 score
    thresholds = np.arange(0.1, 0.9, 0.05)
    f1_scores = []
    best_threshold = 0.5
    best_f1 = 0

    for threshold in thresholds:
        y_pred = [1 if prob > threshold else 0 for prob in y_pred_prob]
        f1 = f1_score(y_test, y_pred)
        f1_scores.append(f1)
        if f1 > best_f1:
            best_f1 = f1
            best_threshold = threshold

    print(f"Best threshold: {best_threshold:.3f}")
    print(f"Best F1 Score: {best_f1:.3f}")

    # Use the best threshold for final predictions
    y_pred = [1 if prob > best_threshold else 0 for prob in y_pred_prob]

    # Calculate metrics with the best threshold
    metrics = {
        'Accuracy': accuracy_score(y_test, y_pred),
        'AUC': roc_auc_score(y_test, y_pred_prob),
        'F1': f1_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred)
```

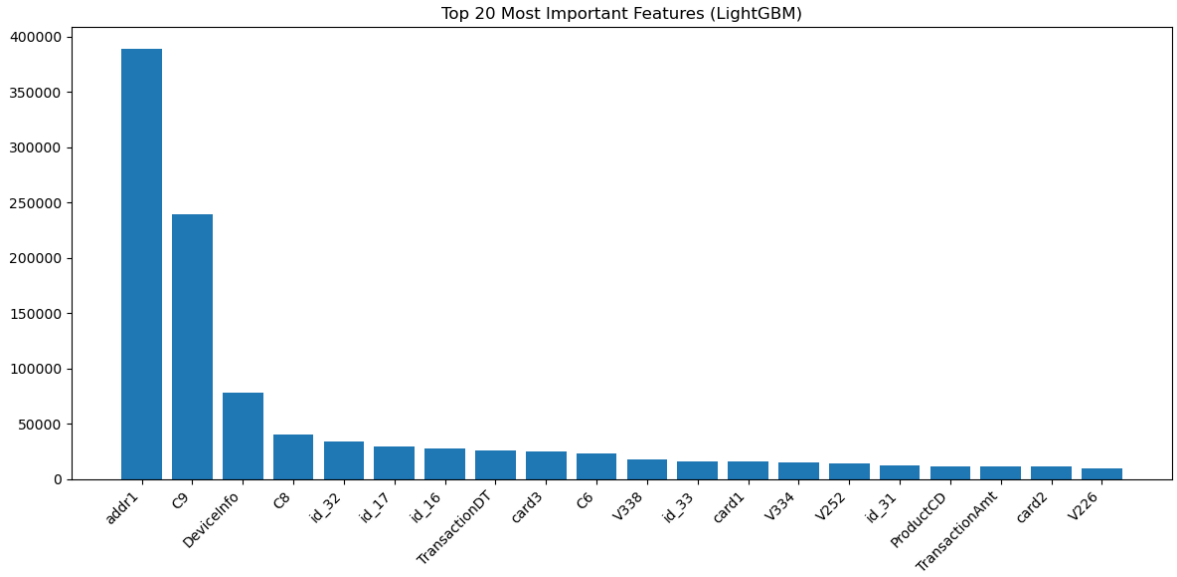
```
    }

    print("\nMetrics with optimized threshold:")
    print(metrics)
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # visualize threshold vs F1 score
    plt.figure(figsize=(15, 5))

    # Plot threshold vs F1 score
    plt.subplot(1, 3, 1)
    plt.plot(thresholds, f1_scores, 'b-')
    plt.axvline(x=best_threshold, color='r', linestyle='--', label=f'Best
    plt.xlabel('Threshold')
    plt.ylabel('F1 Score')
    plt.title('Threshold vs F1 Score')
    plt.legend()

    # Plot ROC Curve
    plt.subplot(1, 3, 2)
    fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, color='blue', lw=2, label=f"ROC curve (AUC = {roc_
    plt.plot([0, 1], [0, 1], color="gray", linestyle="--", lw=2)
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curve")
    plt.legend(loc="lower right")

    # Plot Precision-Recall Curve
    plt.subplot(1, 3, 3)
    precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
    pr_auc = auc(recall, precision)  # te thlate the area under PR curver

    # Add random classifier baseline
    no_skill = len(y_test[y_test == 1]) / len(y_test)  # Calculate ratio
    plt.plot([0, 1], [no_skill, no_skill], '--', color='red', label=f'Ran

    # Plot actual model performance
    plt.plot(recall, precision, color='green', lw=2, label=f"Precision-Re
    plt.axvline(x=metrics['Recall'], color='blue', linestyle='--',
                label=f'Best threshold metrics\nRecall: {metrics["Recall"]

    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.title("Precision-Recall Curve")
    plt.legend(loc="lower left")

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

In [23]:
```
visulize_result_lgb(bst, X_test, y_test)
```

```
Best threshold: 0.350
Best F1 Score: 0.846

Metrics with optimized threshold:
{'Accuracy': 0.9768433459285194, 'AUC': 0.9759250279376988, 'F1': 0.845656
1922365989, 'Precision': 0.8866279069767442, 'Recall': 0.808303886925795}

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.99      0.99     26583
           1       0.89      0.81      0.85      2264

    accuracy                           0.98     28847
   macro avg       0.94      0.90      0.92     28847
weighted avg       0.98      0.98      0.98     28847
```



In [52]:
```python
# get feature importance
feature_importance = bst.feature_importance(importance_type='gain')
feature_names = X_train.columns.tolist()  # get feature names

# create feature importance DataFrame and sort
importance_df = pd.DataFrame({
    'feature': feature_names,
    'importance': feature_importance
}).sort_values('importance', ascending=False)

# visualize the top 20 most important features
plt.figure(figsize=(12, 6))
plt.bar(importance_df['feature'][:20], importance_df['importance'][:20])
plt.xticks(rotation=45, ha='right')  # rotate feature names for better di
plt.title('Top 20 Most Important Features (LightGBM)')
plt.tight_layout()  # automatically adjust layout to prevent label cuttin
plt.show()
```



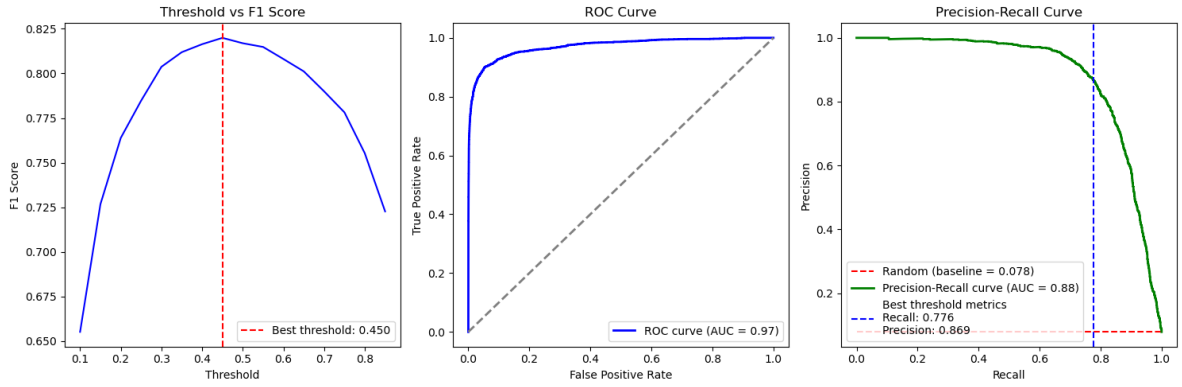
In [36]:
```
categorical_features = list(decoded_oversample_train_df.select_dtypes(inc


X = decoded_oversample_train_df.drop(columns = ['isFraud'])
y = decoded_oversample_train_df['isFraud']
# X[categorical_features] = X[categorical_features].fillna('missing').ast

X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=0.1,  # 20% as validation set
    random_state=42,
    stratify=y  # keep class ratio
)

X_test = decoded_oversample_test_df.drop(columns = ['isFraud'])
y_test = decoded_oversample_test_df['isFraud']
# X_test[categorical_features] = X_test[categorical_features].fillna('mis

for feature in categorical_features:
    X_train[feature] = X_train[feature].astype('category')
    X_val[feature] = X_val[feature].astype('category')
    X_test[feature] = X_test[feature].astype('category')


dtrain = lgb.Dataset(X_train, label=y_train)
dval = lgb.Dataset(X_val, label=y_val, reference=dtrain)
dtest = lgb.Dataset(X_test, label=y_test)

params = {
    'objective': 'binary',
    'metric': ['auc', 'binary_logloss'],
    'n_estimators': 200,
    'num_leaves': 31,
    'learning_rate': 0.1,
    'feature_fraction': 0.8
}

# 训练模型
```

```
bst_smote = lgb.train(params, dtrain, num_boost_round=200, valid_sets=[dv
```

```
/Users/mac/opt/anaconda3/lib/python3.9/site-packages/lightgbm/engine.py:20
4: UserWarning: Found `n_estimators` in params. Will use it instead of arg
ument
  _log_warning(f"Found `{alias}` in params. Will use it instead of argumen
t")
[LightGBM] [Warning] Categorical features with more bins than the configur
ed maximum bin number found.
[LightGBM] [Warning] For categorical features, max_bin and max_bin_by_feat
ure may be ignored with a large number of categories.
[LightGBM] [Info] Number of positive: 95699, number of negative: 95698
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.521294 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 86963
[LightGBM] [Info] Number of data points in the train set: 191397, number o
f used features: 382
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.500003 -> initscore=0.00
0010
[LightGBM] [Info] Start training from score 0.000010
Training until validation scores don't improve for 10 rounds
Did not meet early stopping. Best iteration is:
[200]   valid_0's auc: 0.998225 valid_0's binary_logloss: 0.048438
```

In [37]:
```
visulize_result_lgb(bst_smote, X_test, y_test)
```

```
Best threshold: 0.450
Best F1 Score: 0.820

Metrics with optimized threshold:
{'Accuracy': 0.9732381183485285, 'AUC': 0.9710772706167721, 'F1': 0.819878
6747550163, 'Precision': 0.8689416419386746, 'Recall': 0.776060070671378}

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.99      0.99     26583
           1       0.87      0.78      0.82      2264

    accuracy                           0.97     28847
   macro avg       0.93      0.88      0.90     28847
weighted avg       0.97      0.97      0.97     28847
```



3.3 CatBoost

In [26]:
```
categorical_features = list(train_df.select_dtypes(include='object').colu

X = train_df.drop(columns = ['TransactionID', 'isFraud'])
y = train_df['isFraud']
X[categorical_features] = X[categorical_features].fillna('missing').astyp

X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=0.1,  # 20% as validation set
    random_state=42,
    stratify=y  # keep class ratio
)

X_test = test_df.drop(columns = ['TransactionID', 'isFraud'])
y_test = test_df['isFraud']
X_test[categorical_features] = X_test[categorical_features].fillna('missi

for feature in categorical_features:
    X_train[feature] = X_train[feature].astype('category')
    X_val[feature] = X_val[feature].astype('category')
    X_test[feature] = X_test[feature].astype('category')

catboost = CatBoostClassifier(
    iterations=500,          # number of iterations
    learning_rate=0.1,       # learning rate
    depth=6,                 # tree depth
    cat_features=categorical_features,  # specify categorical features
    random_seed=42,
    eval_metric='AUC',  # use AUC and Logloss as evaluation metrics
    loss_function='Logloss',
    verbose=100,             # output every 100 iterations
    subsample=0.8,           # use 80% of samples for each tree
    colsample_bylevel=0.8,
)

# 训练模型
catboost.fit(X_train, y_train, eval_set=(X_val, y_val), use_best_model=Tr
```

```
0:      test: 0.7700247 best: 0.7700247 (0)     total: 183ms    remaining:
1m 31s
100:    test: 0.9565240 best: 0.9565240 (100)   total: 10.9s    remaining:
43.2s
200:    test: 0.9634073 best: 0.9634073 (200)   total: 21.9s    remaining:
32.6s
300:    test: 0.9669979 best: 0.9669979 (300)   total: 32.2s    remaining:
21.3s
400:    test: 0.9690185 best: 0.9690238 (396)   total: 42.4s    remaining:
10.5s
499:    test: 0.9702485 best: 0.9702485 (499)   total: 52.7s    remaining:
0us

bestTest = 0.9702484577
bestIteration = 499
```

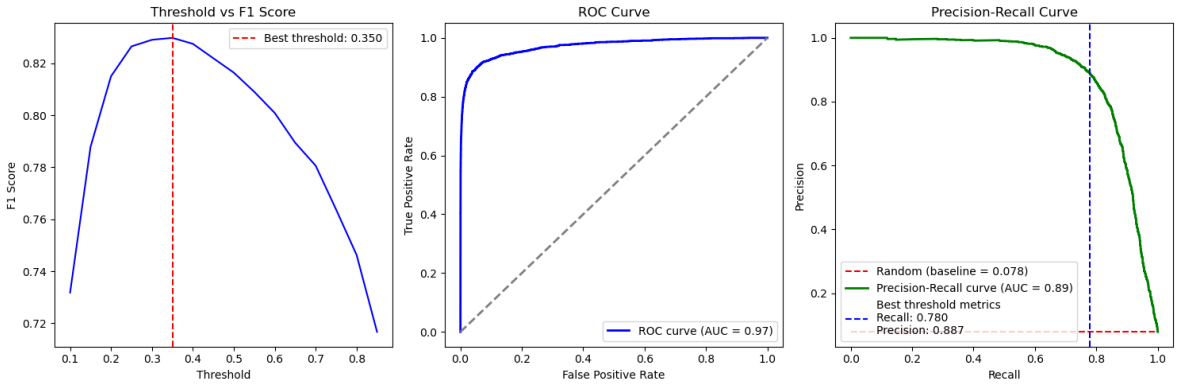
Out[26]: <catboost.core.CatBoostClassifier at 0x7fb8b1718310>

In [27]:
```python
def visulize_result_catboost(model, X_test, y_test):
    # Get predicted probabilities
    y_pred_prob = model.predict_proba(X_test)[:, 1]

    # Try different thresholds to find the best F1 score
    thresholds = np.arange(0.1, 0.9, 0.05)
    f1_scores = []
    best_threshold = 0.5
    best_f1 = 0

    for threshold in thresholds:
        y_pred = [1 if prob > threshold else 0 for prob in y_pred_prob]
        f1 = f1_score(y_test, y_pred)
        f1_scores.append(f1)
        if f1 > best_f1:
            best_f1 = f1
            best_threshold = threshold

    print(f"Best threshold: {best_threshold:.3f}")
    print(f"Best F1 Score: {best_f1:.3f}")

    # Use the best threshold for final predictions
    y_pred = [1 if prob > best_threshold else 0 for prob in y_pred_prob]

    # Calculate metrics with the best threshold
    metrics = {
        'CatBoost - Accuracy': accuracy_score(y_test, y_pred),
        'CatBoost - AUC': roc_auc_score(y_test, y_pred_prob),
        'CatBoost - F1': f1_score(y_test, y_pred),
        'CatBoost - Precision': precision_score(y_test, y_pred),
        'CatBoost - Recall': recall_score(y_test, y_pred)
    }

    print("\nMetrics with optimized threshold:")
    print(metrics)
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # 可视化
    plt.figure(figsize=(15, 5))

    # Plot threshold vs F1 score
    plt.subplot(1, 3, 1)
    plt.plot(thresholds, f1_scores, 'b-')
    plt.axvline(x=best_threshold, color='r', linestyle='--', label=f'Best
    plt.xlabel('Threshold')
    plt.ylabel('F1 Score')
    plt.title('Threshold vs F1 Score')
    plt.legend()

    # Plot ROC Curve
    plt.subplot(1, 3, 2)
    fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
    roc_auc = auc(fpr, tpr)
```

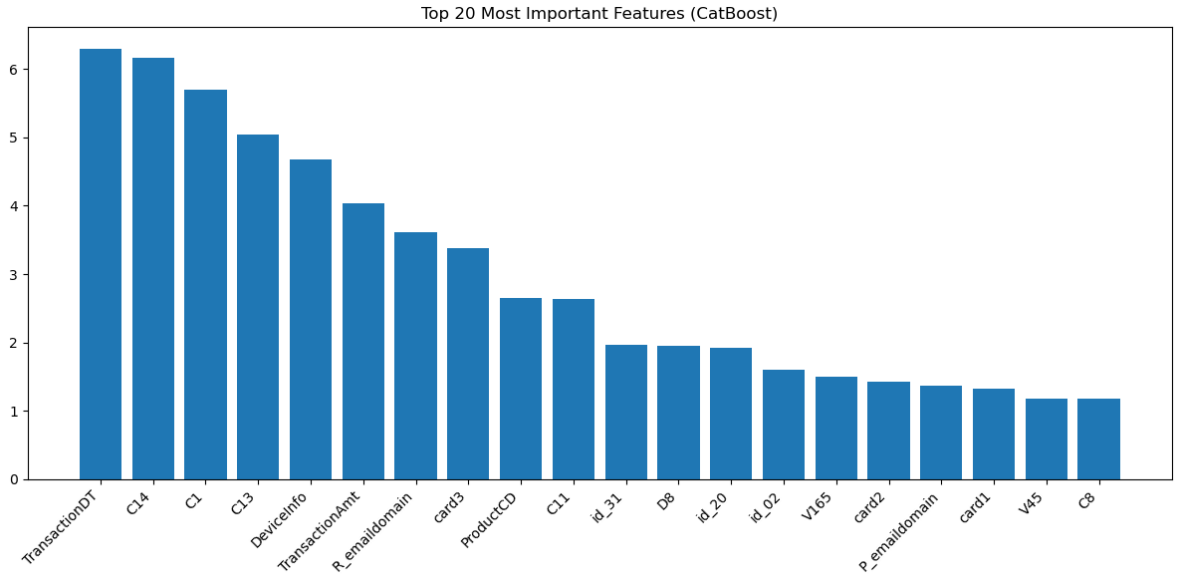
```
    plt.plot(fpr, tpr, color='blue', lw=2, label=f"ROC curve (AUC = {roc_
    plt.plot([0, 1], [0, 1], color="gray", linestyle="--", lw=2)
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curve")
    plt.legend(loc="lower right")

    # Plot Precision-Recall Curve
    plt.subplot(1, 3, 3)
    precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
    pr_auc = auc(recall, precision)  # calculate the area under PR curve

    # Add random classifier baseline
    no_skill = len(y_test[y_test == 1]) / len(y_test)  # Calculate ratio
    plt.plot([0, 1], [no_skill, no_skill], '--', color='red', label=f'Ran

    # Plot actual model performance
    plt.plot(recall, precision, color='green', lw=2, label=f"Precision-Re
    plt.axvline(x=metrics['CatBoost - Recall'], color='blue', linestyle='
                label=f'Best threshold metrics\nRecall: {metrics["CatBoost

    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.title("Precision-Recall Curve")
    plt.legend(loc="lower left")

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

In [28]:
```
visulize_result_catboost(catboost, X_test, y_test)
```

```
Best threshold: 0.350
Best F1 Score: 0.830

Metrics with optimized threshold:
{'CatBoost - Accuracy': 0.9749020695392935, 'CatBoost - AUC': 0.9715105591
673735, 'CatBoost - F1': 0.8298072402444758, 'CatBoost - Precision': 0.886
9346733668342, 'CatBoost - Recall': 0.7795936395759717}

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.99      0.99     26583
           1       0.89      0.78      0.83      2264

    accuracy                           0.97     28847
   macro avg       0.93      0.89      0.91     28847
weighted avg       0.97      0.97      0.97     28847
```



In [51]:
```
feature_importance = catboost.get_feature_importance(type='PredictionValu
feature_names = X_train.columns.tolist()

# create feature importance DataFrame and sort
importance_df = pd.DataFrame({
    'feature': feature_names,
    'importance': feature_importance
}).sort_values('importance', ascending=False)

# visualize the top 20 most important features
plt.figure(figsize=(12, 6))
plt.bar(importance_df['feature'][:20], importance_df['importance'][:20])
plt.xticks(rotation=45, ha='right')  # rotate feature names for better di
plt.title('Top 20 Most Important Features (CatBoost)')
plt.tight_layout()  # automatically adjust layout to prevent label cuttin
plt.show()
```



```
Top 20 Most Important Features and their scores:
            feature  importance
0     TransactionDT    6.297510
27              C14    6.159316
14               C1    5.695991
26              C13    5.037004
401      DeviceInfo    4.674623
1    TransactionAmt    4.039180
13    R_emaildomain    3.617050
5              card3    3.373795
2          ProductCD    2.651306
24              C11    2.641876
392           id_31    1.966792
35               D8    1.953978
388           id_20    1.922097
372           id_02    1.601275
196            V165    1.497097
4              card2    1.422898
12    P_emaildomain    1.362583
3              card1    1.317680
76               V45    1.184413
21                C8    1.171863
```

3.3.1 Dummies

In [29]:

```python
categorical_features = list(train_data_dummies.select_dtypes(include='obj

X = train_data_dummies.drop(columns = ['TransactionID', 'isFraud'])
y = train_data_dummies['isFraud']
X[categorical_features] = X[categorical_features].fillna('missing').astyp

X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=0.1,  # 20% as validation set
    random_state=42,
    stratify=y  # keep class ratio
)

X_test = test_data_dummies.drop(columns = ['TransactionID', 'isFraud'])
y_test = test_data_dummies['isFraud']
X_test[categorical_features] = X_test[categorical_features].fillna('missi

for feature in categorical_features:
    X_train[feature] = X_train[feature].astype('category')
    X_val[feature] = X_val[feature].astype('category')
    X_test[feature] = X_test[feature].astype('category')

catboost_dummies = CatBoostClassifier(
    iterations=500,         # number of iterations
    learning_rate=0.1,      # learning rate
    depth=6,                # tree depth
    cat_features=categorical_features,  # specify categorical features
    random_seed=42,
    verbose=100             # output every 100 iterations
)
```

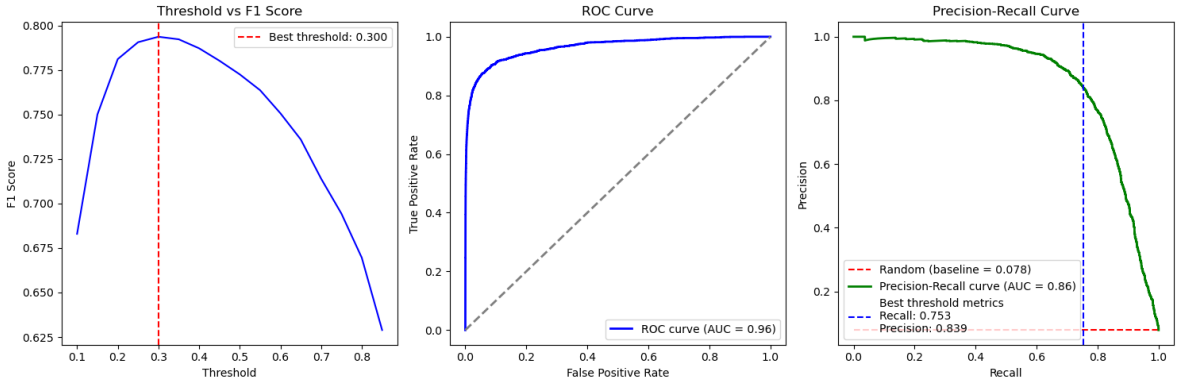
```
# train model
catboost_dummies.fit(X_train, y_train, eval_set=(X_val, y_val), use_best_
```

```
0:      learn: 0.5692068        test: 0.5690860 best: 0.5690860 (0)     to
tal: 86.5ms     remaining: 43.2s
100:    learn: 0.1145612        test: 0.1199626 best: 0.1199626 (100)   to
tal: 4.38s      remaining: 17.3s
200:    learn: 0.1020480        test: 0.1099370 best: 0.1099370 (200)   to
tal: 8.25s      remaining: 12.3s
300:    learn: 0.0943818        test: 0.1046530 best: 0.1046530 (300)   to
tal: 11.7s      remaining: 7.72s
400:    learn: 0.0879873        test: 0.1005050 best: 0.1005050 (400)   to
tal: 15.1s      remaining: 3.73s
499:    learn: 0.0834726        test: 0.0977145 best: 0.0977145 (499)   to
tal: 18.6s      remaining: 0us

bestTest = 0.09771445538
bestIteration = 499
```

Out[29]: <catboost.core.CatBoostClassifier at 0x7fb7cb498eb0>

In [30]:
```
visulize_result_catboost(catboost_dummies, X_test, y_test)
```

```
Best threshold: 0.300
Best F1 Score: 0.794

Metrics with optimized threshold:
{'CatBoost - Accuracy': 0.9692862342704613, 'CatBoost - AUC': 0.9646110741
355598, 'CatBoost - F1': 0.7936655798789007, 'CatBoost - Precision': 0.839
4088669950739, 'CatBoost - Recall': 0.7526501766784452}

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98     26583
           1       0.84      0.75      0.79      2264

    accuracy                           0.97     28847
   macro avg       0.91      0.87      0.89     28847
weighted avg       0.97      0.97      0.97     28847
```



In [ ]:

Oversample

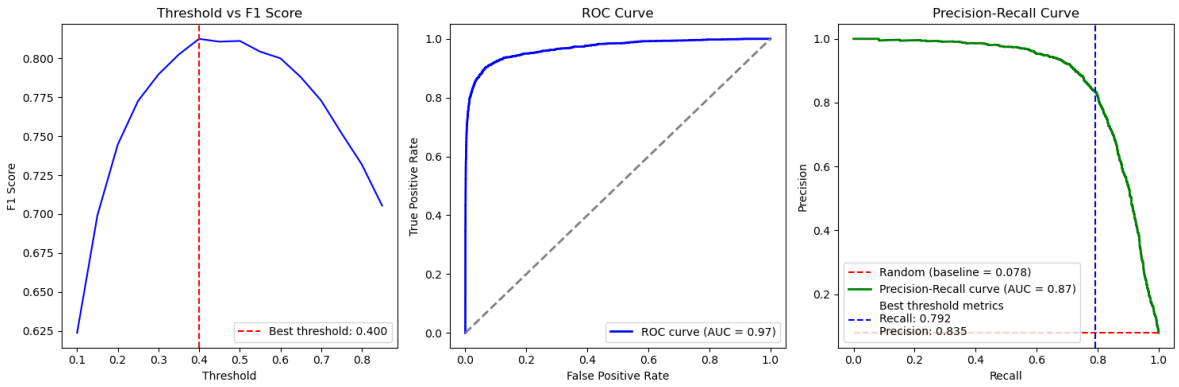
In [31]:
```
categorical_features = list(decoded_oversample_train_df.select_dtypes(inc


X = decoded_oversample_train_df.drop(columns = ['isFraud'])
y = decoded_oversample_train_df['isFraud']
# X[categorical_features] = X[categorical_features].fillna('missing').ast

X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=0.1,  # 20% as validation set
    random_state=42,
    stratify=y  # keep class ratio
)

X_test = decoded_oversample_test_df.drop(columns = ['isFraud'])
y_test = decoded_oversample_test_df['isFraud']
# X_test[categorical_features] = X_test[categorical_features].fillna('mis

for feature in categorical_features:
    X_train[feature] = X_train[feature].astype('category')
    X_val[feature] = X_val[feature].astype('category')
    X_test[feature] = X_test[feature].astype('category')


catboost_smote = CatBoostClassifier(
    iterations=500,          # number of iterations
    learning_rate=0.1,       # learning rate
    depth=6,                 # tree depth
    cat_features=categorical_features,  # specify categorical features
    eval_metric='AUC',         # use single evaluation metric
    loss_function='Logloss',
    random_seed=42,
    verbose=100              # output every 100 iterations
)

# train model
catboost_smote.fit(X_train, y_train, eval_set=(X_val, y_val), use_best_mo
```

```
0:      test: 0.9250368 best: 0.9250368 (0)     total: 215ms    remaining:
1m 47s
100:    test: 0.9947251 best: 0.9947251 (100)   total: 18.7s    remaining:
1m 13s
200:    test: 0.9969077 best: 0.9969077 (200)   total: 36.7s    remaining:
54.6s
300:    test: 0.9975357 best: 0.9975357 (300)   total: 54.4s    remaining:
35.9s
400:    test: 0.9978914 best: 0.9978914 (400)   total: 1m 11s   remaining:
17.8s
499:    test: 0.9981335 best: 0.9981335 (499)   total: 1m 30s   remaining:
0us

bestTest = 0.998133532
bestIteration = 499
```

Out[31]: <catboost.core.CatBoostClassifier at 0x7fb5cdaae0a0>

In [32]:
```
visulize_result_catboost(catboost_smote, X_test, y_test)
```

```
Best threshold: 0.400
Best F1 Score: 0.813

Metrics with optimized threshold:
{'CatBoost - Accuracy': 0.9713315076091101, 'CatBoost - AUC': 0.9678575065
043962, 'CatBoost - F1': 0.812514169122648, 'CatBoost - Precision': 0.8346
530041918957, 'CatBoost - Recall': 0.7915194346289752}

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98     26583
           1       0.83      0.79      0.81      2264

    accuracy                           0.97     28847
   macro avg       0.91      0.89      0.90     28847
weighted avg       0.97      0.97      0.97     28847
```



A.2.1 SMOTE-NC

```python
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTENC
import pandas as pd


train_df = pd.read_csv('train_split.csv')
test_df = pd.read_csv('test_split.csv')
# 1. Compute the null percentage of each column and drop the columns with null percentage greater than 80%
null_percentages = train_df.isnull().sum() / len(train_df)
columns_to_drop = null_percentages[null_percentages > 0.8].index.tolist()

print("The following columns will be deleted (null percentage > 80%):")
for col in columns_to_drop:
    print(f"{col}: {null_percentages[col]*100:.2f}%")

# 2. Drop the columns with null percentage greater than 80%
train_df_processed = train_df.drop(columns=columns_to_drop)
test_df_processed = test_df.drop(columns=columns_to_drop)

# 3. Convert boolean values to integers
bool_columns = train_df_processed.select_dtypes(include=['bool']).columns
for col in bool_columns:
    train_df_processed[col] = train_df_processed[col].astype(int)
    test_df_processed[col] = test_df_processed[col].astype(int)

# 4. Get the feature types
numerical_features = [col for col in train_df_processed.select_dtypes(include=['int64', 'float64']).columns
                      if col not in ['isFraud', 'TransactionID']]
categorical_features = train_df_processed.select_dtypes(include=['object', 'category']).columns

print("nNumber of numerical features:", len(numerical_features))
print("Number of categorical features:", len(categorical_features))

# 4. Handle missing values
for col in numerical_features:
    median_val = train_df_processed[col].median()
    train_df_processed[col] = train_df_processed[col].fillna(median_val)
    test_df_processed[col] = test_df_processed[col].fillna(median_val)

for col in categorical_features:
    mode_val = train_df_processed[col].mode()[0]
    train_df_processed[col] = train_df_processed[col].fillna(mode_val)
    test_df_processed[col] = test_df_processed[col].fillna(mode_val)

# 5. Separate features and target variables
X = train_df_processed.drop(['isFraud', 'TransactionID'], axis=1)
y = train_df_processed['isFraud']

# 6. Save the labels and IDs of the test set
test_labels = None
if 'isFraud' in test_df_processed.columns:
    test_labels = test_df_processed['isFraud']
test_ids = test_df_processed['TransactionID']
X_test = test_df_processed.drop(['TransactionID'], axis=1)
if 'isFraud' in X_test.columns:
    X_test = X_test.drop(['isFraud'], axis=1)

# 7. Standardize the numerical features
scaler = StandardScaler()
X_numerical = X[numerical_features]
X_categorical = X[categorical_features]
X_numerical_scaled = scaler.fit_transform(X_numerical)
X_numerical_scaled = pd.DataFrame(X_numerical_scaled, columns=numerical_features, index=X.index)

# 8. Standardize the test set
X_test_numerical = X_test[numerical_features]
X_test_categorical = X_test[categorical_features]
X_test_numerical_scaled = scaler.transform(X_test_numerical)
X_test_numerical_scaled = pd.DataFrame(X_test_numerical_scaled, columns=numerical_features, index=X_test.index)

# 9. Concatenate the standardized numerical features and categorical features
X_preprocessed = pd.concat([X_numerical_scaled, X_categorical], axis=1)
X_test_preprocessed = pd.concat([X_test_numerical_scaled, X_test_categorical], axis=1)

# 10. Get the indices of the categorical features
```

```python
categorical_features_idx = [X_preprocessed.columns.get_loc(col) for col in categorical_features]

# 11. Apply SMOTENC
smotenc = SMOTENC(categorical_features=categorical_features_idx, random_state=42)
X_resampled, y_resampled = smotenc.fit_resample(X_preprocessed, y)

# 12. Convert the data back to DataFrame and perform one-hot encoding
X_resampled = pd.DataFrame(X_resampled, columns=X_preprocessed.columns)
X_resampled_encoded = pd.get_dummies(X_resampled, columns=categorical_features, dtype=int)

# 13. Perform one-hot encoding on the test set
X_test_encoded = pd.get_dummies(X_test_preprocessed, columns=categorical_features, dtype=int)

# 14. Ensure the training and test sets have the same columns
missing_cols = set(X_resampled_encoded.columns) - set(X_test_encoded.columns)
for col in missing_cols:
    X_test_encoded[col] = 0

# 15. Ensure the column order is consistent
X_test_encoded = X_test_encoded[X_resampled_encoded.columns]

# 16. Create the final datasets
final_train_df = X_resampled_encoded.copy()
final_train_df['isFraud'] = y_resampled

final_test_df = X_test_encoded.copy()
if test_labels is not None:
    final_test_df['isFraud'] = test_labels

print("nDataset shapes:")
print("Original training set shape:", train_df.shape)
print("Processed training set shape:", final_train_df.shape)
print("Processed test set shape:", final_test_df.shape)
print("nFraud case ratio:")
print("- Original training data:", train_df['isFraud'].mean())
print("- After SMOTE:", final_train_df['isFraud'].mean())
if test_labels is not None:
    print("- Test set:", test_labels.mean())

'''
Output:
The following columns will be deleted (null percentage > 80%):
dist1: 100.00%
D11: 100.00%
M1: 100.00%
M2: 100.00%
M3: 100.00%
M5: 100.00%
M6: 100.00%
M7: 100.00%
M8: 100.00%
M9: 100.00%
V1: 100.00%
V2: 100.00%
V3: 100.00%
V4: 100.00%
V5: 100.00%
V6: 100.00%
V7: 100.00%
V8: 100.00%
V9: 100.00%
V10: 100.00%
V11: 100.00%
id_07: 96.48%
id_08: 96.48%
id_21: 96.47%
id_22: 96.47%
id_23: 96.47%
id_24: 96.75%
id_25: 96.49%
id_26: 96.47%
id_27: 96.47%

Number of numerical features: 381
Number of categorical features: 21
```

```
Dataset shapes:
Original training set shape: (115386, 434)
Processed training set shape: (212664, 2681)
Processed test set shape: (28847, 2681)

Fraud case ratio:
- Original training data: 0.07846705839529926
- After SMOTE: 0.5
- Test set: 0.07848303116441918'''
```