Fraud Detection with Graph Neural Networks

Aleksandar Milosevic

Final project report for CM3070
for the degree of BSc Computer Science
of the University of London.

Department of Computing

Goldsmiths, University of London

March 2025

Abstract

Fraudulent financial transactions represent a major challenge, causing significant losses and eroding customer trust. Traditional detection methods often struggle with the complexity and relational nature of modern fraud schemes. This project investigates the application of Graph Neural Networks (GNNs), a deep learning approach suited for graph-structured data, to the IEEE-CIS fraud detection dataset. A heterogeneous graph was constructed representing transactions, cards, devices, locations, and networks. Both heterogeneous (using SAGE, GAT, RGCN layers within `HeteroConv`) and homogeneous GNN models were implemented and compared against a strong XGBoost baseline. A comprehensive preprocessing pipeline involving time-based splitting, robust imputation, model-specific encoding, and scaling was developed. Evaluation revealed that while GNNs showed promise on validation data (AUC up to ~0.85), they failed to generalize effectively to the temporally distinct test set (AUC ~0.65-0.82, very low AP), significantly underperforming XGBoost (AUC ~0.87, AP ~0.35). Exploratory Data Analysis confirmed substantial concept drift in both target variable and key features, identified as the primary reason for the GNNs' generalization failure. The project highlights the sensitivity of standard GNNs to non-stationarity in real-world fraud data and suggests future work should focus on time-aware features and models.

The project repository can be found at `https://github.com/amilos/fdgnn`.

Contents

1 Introduction

Fraudulent financial transactions pose a significant and growing challenge across industries. The increasing volume and sophistication of digital transactions create fertile ground for fraudsters, leading to substantial direct financial losses and indirect costs like damaged customer trust and brand reputation [Placeholder: Cite general fraud statistic source]. Consequently, developing accurate, fast, and interpretable fraud detection systems is paramount.

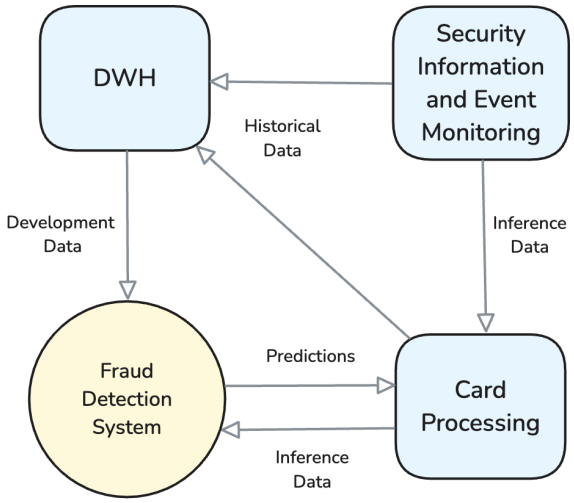

Figure 1.1: Typical context of credit card fraud detection

1.1 Limitations of Traditional Fraud Detection Methods

Traditional fraud detection methods often rely on statistical models or rule-based systems, which can struggle to keep pace with evolving fraudulent tactics. These methods focus on individual transactions or hand-crafted features, making it difficult to detect subtle or collusive patterns due to rigid decision boundaries and an inability to adapt to evolving fraudulent behaviors. As fraudsters adapt, more advanced techniques are needed—ones that can automatically learn complex representations of data and capture the relational structure of transaction networks.

Deep learning has emerged as a promising direction, having proven its effectiveness in tasks ranging from image recognition to language modeling and anomaly detection. However, many deep learning approaches in fraud detection like CNNs and RNNs, work with grid-structured data (e.g., images or sequences) and still treat each transaction as an isolated instance or, at best, a sequence of events. This approach overlooks the rich relational information contained in transactional networks, where entities such as customers, merchants, and accounts form intricate connections. These connections can reveal hidden patterns indicative of coordinated fraud that a single transaction alone cannot capture.

1.2 Graph Neural Networks (GNNs) for Fraud Detection

Graph Neural Networks (GNNs) offer a way to model these complex relational patterns. For example, major financial institutions have started leveraging GNNs to detect fraud rings, where multiple fraudulent accounts

coordinate transactions to evade detection. By analyzing the connections between these accounts and their transactions, GNNs can uncover suspicious structures that traditional models might miss. Representing transactions and entities as nodes and edges in a graph allows a GNN to propagate information across the network, learning embeddings that capture both the attributes of individual entities and the structure of their relationships. This structure-aware approach could highlight patterns of fraud that would remain invisible if the data were considered in isolation.

This project hypothesizes that explicitly modeling the relational structure of transaction data using Graph Neural Networks (GNNs) can enhance fraud detection accuracy and robustness compared to traditional methods like XGBoost. GNNs are specifically designed for graph-structured data [Kipf and Welling 2017]. By representing transactions and associated entities (cards, devices, locations, networks) as nodes and their interactions as edges in a graph, GNNs can:

- **Capture Relational Patterns:** Message passing mechanisms allow nodes to aggregate information from their neighbors, learning embeddings that reflect both node attributes and local graph structure. This enables the detection of complex patterns like collusion or unusual network activity that are invisible to models treating transactions in isolation [Liu et al. 2021; Qin et al. 2022].
- **Leverage Heterogeneity:** Real-world transaction graphs involve different types of nodes and relationships. Heterogeneous GNN architectures can learn type-specific representations and interactions, potentially leading to more nuanced models [Van Belle et al. 2022].
- **Potential for Interpretability:** While not inherently transparent, certain GNN techniques (like attention mechanisms or post-hoc explainers like GNNExplainer) offer potential pathways to understand why a transaction is flagged, by highlighting influential neighbors or subgraphs [Qin et al. 2022]. This contrasts with the often opaque nature of complex deep learning models or even large tree ensembles.

1.3 Project Aims

The core objective is to assess whether GNNs offer performance improvements over established tabular methods like XGBoost for detecting fraudulent transactions within the IEEE-CIS dataset [Howard et al. 2019]. Financial transaction data is inherently relational; transactions as events are linked with cards, devices, locations, and networks and may reveal latent patterns of fraudulent behavior. Traditional models often overlook these connections. This project hypothesizes that by explicitly modeling these relationships within a graph structure, GNNs can capture complex, potentially collusive fraud patterns more effectively than models treating transactions independently, or, more formally:

H1 GNN models, leveraging the relational structure of transaction data, will achieve higher performance (measured by ROC-AUC and Average Precision) on a time-ordered test set compared to an XGBoost baseline trained on equivalent tabular features.

The project follows the "Deep Learning on a Public Dataset" template (CM3015) and compares the performance of various GNN architectures (Heterogeneous SAGE/GAT/RGCN, Homogeneous SAGE/GAT) against a strong baseline. In order to achieve stated objectives the project will:

1. Select and preprocess a public fraud detection dataset.
2. Establish baseline models (simple deep learning architectures) for reference.
3. Implement pipeline for different GNN model architectures to capture relational structures.
4. Quantitatively evaluate performance improvements in terms of reducing false positives and negatives.
5. Consider insights gained from the model's relational understanding.

The project will begin with a baseline deep learning model—such as a simple feedforward neural network

or a commonsense heuristic—to establish a reference point. From there, it will progress to more advanced architectures, culminating in the application of GNN-based methods.

In addition to improving performance in banking fraud detection, graph-based deep learning techniques have the potential to improve security in cybersecurity threat analysis, optimize recommendation systems by better capturing user-item interactions, and even advance healthcare analytics by identifying hidden patterns in patient data. These techniques may also contribute to social network analysis for detecting misinformation campaigns, improve supply chain optimization by analyzing complex trade relationships, and support scientific research areas such as drug discovery, where molecular interactions form intricate graphs.

In summary, this project proposes to tackle the complex challenge of fraud detection by modeling transaction data as a graph and applying Graph Neural Networks for classification.

By integrating GNNs, this project aims to push beyond the limitations of traditional and even advanced deep learning models that overlook these relationships. In doing so, it meets the requirements of the CM3015 project template, selecting a public dataset for experimentation, moving from simple baselines to more complex models, and ultimately achieving improved predictive performance.

```
Words: 969
```

2 Literature Review

2.1 Credit Card Fraud Detection Using Random Forest Algorithm

The paper by [Kumar et al. 2019] presents a supervised learning approach to detecting fraudulent credit card transactions. It focuses on utilizing the Random Forest Algorithm (RFA) for improved accuracy and reliability in classification. Recognizing the escalating prevalence of both online and offline credit card fraud, the authors propose a systematic approach for cleaning, preprocessing, and partitioning transaction data into training and testing datasets before applying the RFA. The core aim is to evaluate the performance of RFA compared to existing methods like Naïve Bayes and clustering algorithms.

The proposed system involves several key steps:

- **Data Cleaning and Preprocessing:** The authors emphasize the importance of eliminating duplicate and null values from the dataset and focus primarily on two features - transaction amount and time.
- **Dataset Partitioning:** The dataset is split into training and testing subsets using a 70-30 ratio to evaluate performance.
- **Implementation of RFA:** The algorithm uses decision trees for classification, creating a robust ensemble model to enhance prediction accuracy.
- **Performance Evaluation**: A confusion matrix is employed to analyze the model's precision, recall, and overall accuracy, which reportedly reaches 90%.

The authors highlight RFA's ability to mitigate overfitting, making it advantageous for fraud detection tasks involving imbalanced datasets. Additionally, they argue that RFA outperforms other traditional methods by effectively leveraging its ensemble nature to classify fraudulent transactions with high accuracy.

The paper provides a clear and practical demonstration of the effectiveness of the Random Forest Algorithm for credit card fraud detection. The structured pipeline, from data cleaning to evaluation, offers a reproducible methodology, making it a valuable resource for foundational approaches in fraud detection systems.

Strengths

- **Focus on Practical Implementation**: The detailed description of data preprocessing and performance evaluation offers a clear roadmap for applying machine learning techniques in real-world scenarios.
- **Improved Accuracy:** Achieving 90% accuracy on a real-world dataset demonstrates the potential of RFA for detecting fraudulent transactions.
- **Ease of Implementation:** Leveraging accessible tools like Scikit-learn makes the approach feasible for deployment in practical applications.

Limitations

- **Feature Engineering Limitations:** The focus on only two features (transaction amount and time) may limit the model's capacity to generalize across different datasets with more complex patterns.
- **Comparative Scope:** While the paper claims that RFA outperforms Naïve Bayes and clustering methods, it does not compare its performance with more advanced techniques like gradient boosting or deep learning models.
- **Graph-Based Insights Missing:** For a project exploring Graph Neural Networks (GNNs), the paper lacks insights into leveraging transactional relationships and graph structures, which are critical for modeling

interconnected data in fraud detection.

Relevance: For our project, this paper offers a benchmark for comparing traditional machine learning models against your GNN-based approach. Specifically, the preprocessing steps and performance metrics discussed can guide the evaluation of the model developed in the project. Additionally, integrating features such as transaction relationships and user behavior dynamics into graph representations could significantly improve upon the results achieved with RFA.

2.2 Graph Neural Network for Credit Card Fraud Detection

The paper by [Liu et al. 2021] introduces a novel methodology for detecting credit card fraud using Graph Neural Networks (GNNs). The authors address the inadequacies of existing methods, which primarily focus on isolated aspects of fraud detection, such as transaction relationships, user behavior dynamics, or individual transaction features. To bridge these gaps, the paper proposes a Weighted Transaction Graph (TG) that integrates multiple dimensions of transaction data, such as temporal dynamics, transaction relationships, and user behavioral changes. The graph structure enables a comprehensive representation of transactions by connecting nodes (individual transactions) with weighted edges based on logical rules reflecting various transactional features.

The TG is further optimized with:

- **Sampling Policies:** A novel approach to sample neighbors based on similarity and edge weights to improve the quality of node embeddings.
- **Attention Mechanism:** Introduced into the GNN model to prioritize neighbors based on their temporal and transactional relevance. This mechanism improves the aggregation of important features and mitigates the influence of outliers.

The method employs the GraphSAGE framework for node representation, extended with the proposed sampling and attention mechanisms. A real-world dataset comprising 3.5 million transactions labeled by experts is used for training and evaluation. The results demonstrate that the proposed model significantly outperforms traditional machine learning methods (e.g., Random Forest, Logistic Regression) and advanced models like GCN and DeepWalk. Specifically, the model achieves a 6.4% improvement in F1-score compared to DeepWalk and a 10.6% improvement over Gradient Boosting Decision Trees.

This paper presents a robust and innovative approach to credit card fraud detection by effectively leveraging the capabilities of GNNs. The proposed Weighted Transaction Graph (TG) is particularly noteworthy as it encapsulates complex, multi-dimensional relationships between transactions, making it highly relevant to your project. The integration of sampling policies and an attention mechanism adds a layer of sophistication, addressing common issues like noise and sparsity in transaction graphs.

Strengths

- **Comprehensive Graph Representation:** The use of weighted edges to represent relationships ensures a richer and more interpretable graph structure.
- **Innovative Sampling and Attention Mechanisms:** These features enhance the quality of node embeddings, enabling more accurate fraud detection.
- **Real-World Validation:** The evaluation on a large, real-world dataset underlines the practical viability of the model.

Limitations

- **Dataset Specificity:** The model's reliance on domain-specific logical propositions for graph construction may limit its generalizability to other datasets or fraud scenarios.
- **Scalability Concerns:** While the paper demonstrates impressive results on a large dataset, the computational complexity of graph construction and model training could pose challenges for larger, real-time systems.
- **Limited Comparative Analysis:** Although the model outperforms existing methods, further analysis against other state-of-the-art GNN-based models could provide a clearer benchmark.

Relevance: For our project, this paper provides essential insights into constructing transaction graphs and integrating advanced mechanisms for improving detection accuracy. The TG and attention mechanism align well with your goal of leveraging GNNs on fraud datasets. Additionally, the highlighted challenges, such as feature selection for graph construction and parameter tuning, offer actionable directions for the implementation.

2.3 Credit Card Fraud Intelligent Detection Based on Machine Learning

The paper by [Mu 2022] explores advanced strategies for detecting financial fraud, particularly credit card fraud, using hybrid machine learning approaches. The research acknowledges the evolution of payment technologies and the subsequent rise in financial fraud, emphasizing the limitations of traditional rule-based and machine learning-only methods. It discusses four novel hybrid approaches that combine advanced techniques such as graph neural networks (GNNs), deep neural networks (DNNs), semi-supervised learning, and supervised-unsupervised learning hybrids to improve predictive accuracy and address challenges like unbalanced datasets and feature engineering.

The paper identifies two primary types of credit card fraud as offline and online, and introduces real-world datasets used in experiments. These datasets range from labeled transaction records from financial institutions to relational graphs representing user behavior. The study highlights various evaluation metrics, such as confusion matrices, AUC, and precision-recall curves, as benchmarks for model effectiveness.

Among the methods reviewed, significant emphasis is placed on:

- **A spatial-temporal attention GNN** that exploits the temporal and spatial characteristics of fraudulent activities.
- **A spectrum-based DNN** that leverages spectral graph coordinates for feature extraction in high-dimensional data.
- **A semi-supervised GNN** that integrates labeled and unlabeled data using hierarchical attention mechanisms.
- **A hybrid supervised-unsupervised approach** that combines balanced random forests with outlier detection for rare fraud cases.

The paper concludes with a discussion of the challenges, including real-time processing, complex social relationships in transaction data, and the scarcity of comprehensive datasets, offering future research directions.

Strengths

The paper provides a solid foundation for understanding advanced hybrid machine learning techniques in fraud detection. Its focus on GNNs aligns closely with your project goal of developing a fraud detection model leveraging GNNs. The spatial-temporal attention GNN is highly relevant as it demonstrates how

GNNs can model relationships and temporal dependencies within transaction data, critical for fraud detection. Moreover, the semi-supervised GNN method's ability to utilize both labeled and unlabeled data could inform your project's handling of imbalanced datasets.

Limitations

- **Dataset Availability**: While the approaches are innovative, their applicability is constrained by the availability of high-quality, real-world datasets. This challenge is especially pertinent to your project since fraud datasets are often proprietary and imbalanced.
- **Real-time Detection**: The paper acknowledges the lack of real-time processing capabilities in the reviewed methods, an aspect that might be crucial depending on the operational goals of your GNN model.
- **Scalability Concerns**: The integration of complex modules like hierarchical attention mechanisms and hybrid approaches could lead to computational overhead, potentially impacting scalability.

Relevance: For our project, this paper can serve as a guiding resource to adopt GNNs for modeling transaction data as a graph, utilizing the spatial-temporal and relational characteristics of fraudulent behavior. It will also serve as a guidance on practical challenges, such as dataset preparation and computational efficiency, by tailoring the discussed methods to the specific fraud dataset we've chosen.

2.4 Inductive Graph Representation Learning for Fraud Detection

The paper by [Van Belle et al. 2022] evaluates two state-of-the-art inductive graph representation learning algorithms - GraphSAGE and Fast Inductive Graph Representation Learning (FI-GRL) - on their ability to classify fraudulent transactions in a highly imbalanced credit card transaction network. The study addresses key challenges in fraud detection: dynamic networks that evolve with incoming transactions and significant class imbalance between legitimate and fraudulent transactions.

The authors constructed a heterogeneous tripartite graph representing clients, merchants, and transactions using a real-world dataset of over 3.7 million transactions, with only 0.65% labeled as fraudulent. The experimental setup incorporated advanced pre-processing, such as time-based feature engineering, and a rolling window approach for training and testing. GraphSAGE utilized supervised learning with neighborhood sampling and aggregation, while FI-GRL employed a two-step process of matrix sketching and feature extraction to generate embeddings.

Results show that both algorithms improved fraud detection when their embeddings were combined with transaction features, yielding up to a 40% increase in precision-recall area under the curve (PR-AUC) compared to baseline models. However, GraphSAGE significantly outperformed FI-GRL, leveraging its ability to integrate transaction features and heterogeneous graph structures. Additionally, graph-level undersampling demonstrated a marginal benefit for addressing class imbalance. The authors concluded that GraphSAGE's scalability and rapid inference made it more suitable for real-time fraud detection applications.

The paper provides valuable insights into leveraging inductive graph representation learning for fraud detection, directly supporting our project's focus on applying GNNs to fraud datasets. The study's detailed analysis of GraphSAGE highlights the importance of incorporating both topological and transactional data in fraud detection models. GraphSAGE's superior performance suggests that its feature aggregation and supervised learning capabilities are critical for handling real-world fraud scenarios.

Strengths

- **Practical Relevance:** The use of a real-world dataset and realistic constraints (e.g., time limits for fraud detection) ensures the findings are applicable to operational systems.
- **Comprehensive Evaluation:** The comparison between supervised (GraphSAGE) and unsupervised (FI-GRL) approaches provides a nuanced understanding of how different methods handle imbalanced and dynamic networks.
- **Scalability and Efficiency:** GraphSAGE's ability to generate embeddings for unseen nodes in under 10 milliseconds aligns well with industry requirements for real-time processing.

Limitations

- **Dataset-Specific Observations:** While the dataset is realistic, conclusions may not generalize to other fraud detection contexts without similar network structures or features.
- **Limited Comparison:** The study does not benchmark GraphSAGE or FI-GRL against other modern GNN-based fraud detection models, such as spatial-temporal attention GNNs.
- **Feature Dependence:** GraphSAGE's reliance on transaction features for superior performance underscores the need for robust feature engineering, which may limit its plug-and-play applicability.

Relevance: For our project, this paper underscores the importance of selecting a GNN architecture capable of leveraging both graph topology and node features. GraphSAGE's ability to process dynamic graphs efficiently and address class imbalance through sampling can guide the design of our model. Additionally, the experimental results highlight the need to evaluate the complementarity of graph embeddings with traditional transaction features.

2.5 Explainable Graph-based Fraud Detection via NGS

The paper by [Qin et al. 2022] introduces the Neural meta-Graph Search (NGS) framework to address the dual goals of high performance and explainability in fraud detection using Graph Neural Networks (GNNs). Traditional graph-based methods for fraud detection have shown effectiveness but often lack interpretability, which is crucial in financial domains where insights into the reasoning behind fraud detection are essential.

NGS achieves explainability by formalizing the message-passing process of GNNs into a meta-graph, which explicitly captures the semantic relationships between nodes and edges. The framework comprises three steps:

- **Meta-Graph Definition:** The message-passing mechanism of GNNs is structured as a directed acyclic graph (DAG), where edges represent relations and nodes represent intermediate states of message aggregation.
- **Differentiable Neural Architecture Search (DARTS):** This step automates the search for optimal meta-graph structures, removing the need for manual definition and allowing the framework to adapt to specific datasets.
- **Multi-meta-Graph Aggregation:** Multiple meta-graphs are aggregated to improve fault tolerance and capture diverse semantic relationships in the data.

The framework was tested on two real-world fraud detection datasets, Amazon and YelpChi, against state-of-the-art baselines. NGS outperformed all competitors, achieving improvements of up to 42% in GMean and 17% in AUC. The study also demonstrated the framework's ability to generate explainable meta-graphs, offering insights into why certain predictions were made. For instance, in the YelpChi dataset, meta-graphs highlighted patterns such as clusters of fraudulent reviews posted by the same user, a typical indicator of

click farming.

This paper provides a groundbreaking approach to addressing both the performance and explainability challenges in graph-based fraud detection, making it highly relevant to our project on Fraud Detection with Graph Neural Networks.

Strengths

- **Explainability:** By generating interpretable meta-graphs, NGS provides actionable insights into the detection process, a critical feature for adoption in sensitive domains like finance.
- **Automated Search:** The use of DARTS eliminates the need for domain-specific expertise in defining meta-graphs, enhancing adaptability across datasets.
- **Performance:** The multi-meta-graph aggregation mechanism significantly improves robustness and predictive accuracy compared to single-graph approaches.

Limitations

- **Computational Complexity:** The framework's reliance on DARTS and multi-meta-graph aggregation increases computational overhead, which might hinder scalability for large-scale datasets.
- **Limited Dataset Scope:** While the results are promising, the evaluation is restricted to two datasets. Broader testing on diverse fraud scenarios would strengthen the findings.
- **Potential Overfitting:** The reliance on dataset-specific patterns in meta-graph generation raises concerns about generalizability to other fraud detection tasks.

Relevance: For our project, the idea of formalizing message passing into a meta-graph and employing automated architecture search could be pivotal in handling dynamic fraud datasets. Additionally, incorporating explainable components would not only enhance model transparency but also aid in debugging and optimizing the GNN.

2.6 Conclusion

While prior work shows GNN potential, applying heterogeneous GNNs to the IEEE-CIS dataset with explicit handling of its temporal nature and comparison against a strong tabular baseline under concept drift remains an area for investigation.

```
Words: 2430
```

3 Design

This chapter outlines the design employed in this project to investigate the performance of Graph Neural Networks (GNNs) for fraud detection compared to a traditional machine learning baseline. It presents the end-to-end system architecture, the data processing pipeline, the graph modeling strategy, the design of GNN models, and the evaluation framework.

3.1 Architecture

The project employs a comparative experimental architecture, processing the data through a shared pipeline before feeding it into distinct modeling paths for the baseline and the GNNs.

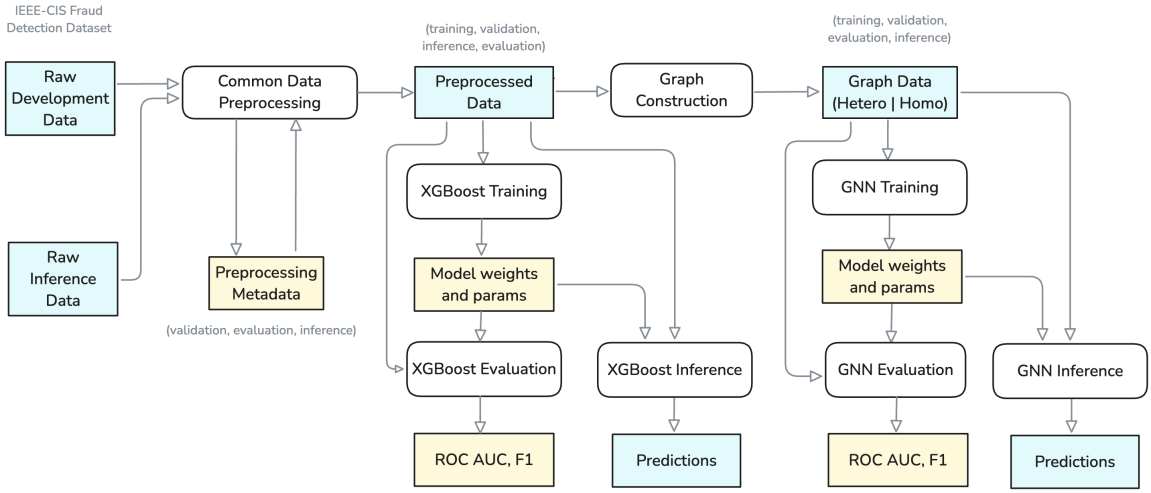

Figure 3.1: End-to-end Architecture of the Fraud Detection Model Project

Figure 3.1 Description: The diagram illustrates end-to-end architecture of the system with major components and high-level information flows. Raw data undergoes common preprocessing steps. The resulting tabular features are used to train and evaluate the XGBoost baseline. Simultaneously, the GNN-specific processed features and a snapshot containing entity IDs are used to construct a heterogeneous graph, which is then used to train and evaluate various GNN models. Performance metrics from both model types are then compared. Inferencing is performed on unseen raw data that passes through the same steps.

3.2 Design Decisions

The design choices for this project were driven by the goal of comparing GNNs against a strong baseline for fraud detection on a public, time-ordered dataset, considering the relational nature of fraud and practical project constraints.

Dataset Choice

The IEEE-CIS Fraud Detection dataset available on Kaggle [Howard et al. 2019] meets the public dataset requirement and is highly relevant to the fraud detection domain. Like many publicly available fraud detection datasets, it has inherently tabular structure features obfuscates for privacy reasons. For prototype, another

another Kaggle fraud dataset was explored but this dataset was chosen as investigation shown that, unlike for other dataset, it is possible to engineer relationships linking transactions with cards, devices, locations) makes it suitable for GNN exploration. The dataset's mix of feature types and it's obfuscated identifiers demanded a comprehensive preprocessing pipeline.

Baseline Model Choice

XGBoost was chosen for the baseline due to its proven strength on tabular data. While simpler alternatives such as Linear Regression could be used, XGBoost serves as a challenging and realistic baseline, often representing state-of-the-art performance for tabular fraud detection [Chen and Guestrin 2016]. Comparing against tabular SOTA model provides a meaningful benchmark to assess any potential advantages offered by the GNN's relational modeling capabilities.

Graph Representation and Graph Schema Modeling

A heterogeneous graph (HeteroData) was the primary design choice, naturally modeling distinct entities (transactions, cards, devices, locations, networks) and their specific relationships. This allows for potentially richer, type-aware representations compared to homogeneous graphs. Composite entity IDs (e.g., card, device profile, location) were engineered from source columns to create meaningful nodes linking related transactions, with logic included to handle missing source data. Both forward and reverse edges were included to facilitate bidirectional information flow.

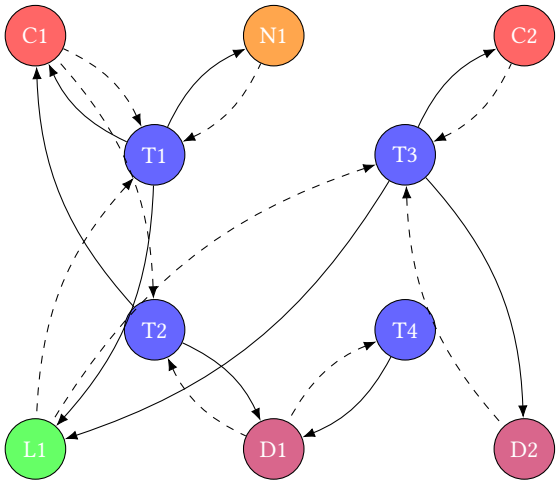

Figure 3.2: Small subgraph example illustrating heterogeneous graph

Figure 3.2 Description: The visual depicts a small subgraph, illustrating how transaction nodes connect to different entity nodes via distinct edge types, forming the heterogeneous graph structure.

GNN Model Design

A flexible GNN model design aims to accommodate both common and distinct elements of heterogeneous and homogeneous graph processing allowing for switching of convolution layer types and configurable depth and width of the GNN as illustrated in **Figure 3.3**.

- **Feature Encoding:** A dedicated Feature Encoder module was designed to create initial node representations. For transaction nodes, it combines scaled numerical features with embeddings learned for label-encoded categorical features. For entity nodes, it learns embeddings based on their graph indices. All initial repres-

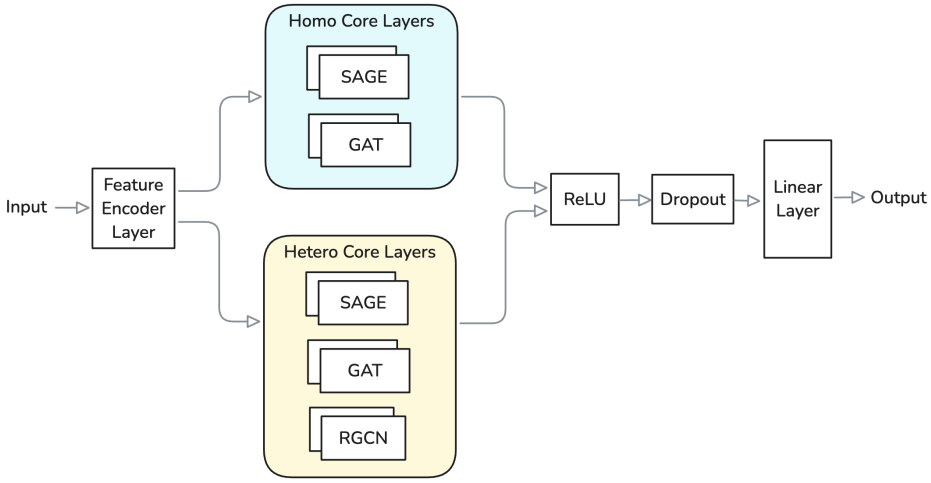

Figure 3.3: Design of GNN Model

entations are projected to a common hidden dimension using linear layers, providing standardized input for the GNN core.

- **Heterogeneous Core:** Utilizes standard convolution layers (SAGE [Hamilton et al. 2018], GAT [Veličković et al. 2018], RGCN [Schlichtkrull et al. 2017]) to handle message passing across different node and edge types.
- **Homogeneous Core:** Included for comparison, this uses a standard stack of homogeneous layers (SAGE, GAT). It requires converting the graph structure beforehand and processes a single concatenated feature matrix constructed from the Feature Encoder's output.
- **Depth/Width:** Models typically used 2-3 layers with moderate hidden dimensions (e.g., 128-256), representing common practice. ReLU activation and Dropout were used for regularization.
- **Output Layer:** A final linear layer maps the transaction node embeddings produced by the GNN core to the binary classification output (fraud/not fraud).

Evaluation Framework

The design of evaluation approach balances the exploration of advanced GNN techniques with the need for a robust baseline comparison and realistic evaluation on time-ordered data, acknowledging potential challenges like concept drift and computational constraints.

- **Time-Based Split:** A strict chronological split (e.g., 70% train, 15% validation, 15% test) over transaction timestamp is essential for realistic performance assessment given the evidence of concept drift found during EDA.
- **Metrics:** ROC-AUC and Average Precision (PR-AUC) were chosen as primary metrics due to the class imbalance inherent in fraud detection. F1-score, Precision, and Recall for the fraud class, along with the Confusion Matrix (at a default or tuned threshold), provide further practical insights into model behavior.
- **Comparison:** Evaluating all models on the identical, temporally distinct test set allows for a direct assessment of their generalization capabilities. Saving model configurations alongside weights ensures reproducibility during evaluation and inference.

3.3 Data Processing Pipeline Design

A robust and modular data processing pipeline was designed using to orchestrate data preparation for model development and inference. The pipeline is organized into functions illustrated in Figure 3.4 and is designed to avoid data leaks from test set and ensure consistent processing in all scenarios. This pipeline handles the transformation from raw CSVs to model-ready inputs.

The figure shows a flow starting from raw data (IEEE-CIS dataset CSVs) passing through pipeline functions (steps). Preprocessed data and processing metadata (encoders mappings, scaler parameters etc.) is saved at the end. In inference and evaluation runs, previously saved preprocessing metadata is used for consistency.

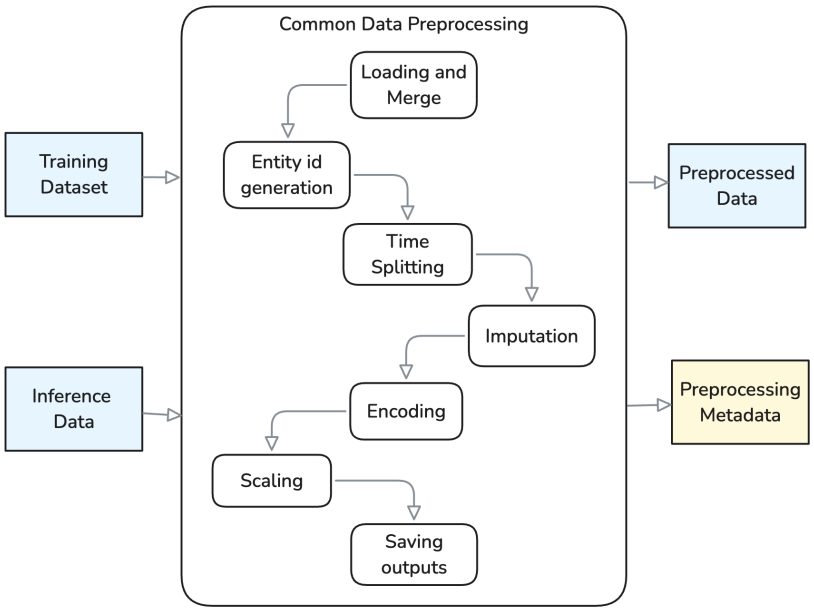

Figure 3.4: Data processing pipeline

3.4 Project Plan

During the design phase of the project, a comprehensive project plan was developed, incorporating detailed task breakdowns, effort estimates, and a planned delivery schedule. Risks were assessed to anticipate challenges that could impact the timeline or deliverables. Based on this risk assessment, a contingency reserve was built as a buffer on task level.

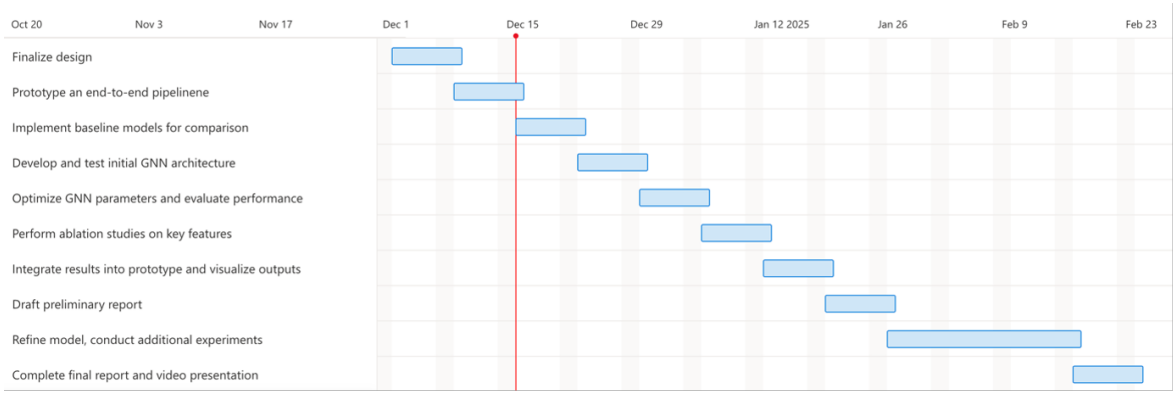

Figure 3.5: Gantt Chart with High-level Schedule

Table 3.1: Project Timeline and Effort

Week	Task	Deliverable	Start Date	Effort (hrs)
9	Finalize design	Design paper	02-Dec-2024	8
10	Develop prototype and preliminary report	Preliminary report	09-Dec-2024	10
11	Implement baseline models for comparison	Baseline model results	16-Dec-2024	12
12	Develop and test initial GNN architecture	Preliminary GNN model	23-Dec-2024	14
13	Optimize GNN parameters and evaluate performance	Tuned GNN model and metrics	30-Dec-2024	12
14	Perform ablation studies on key features	Insights on feature importance	06-Jan-2025	10
15	Integrate results into prototype and visualize outputs	Prototype with graphical outputs	13-Jan-2025	8
16	Finalize draft report	Draft project report	20-Jan-2025	8
17-19	Refine model and conduct additional experiments	Finalized GNN model	27-Jan-2025	12 per week
20	Complete final report and video presentation	Final report and video	17-Feb-2025	16

Please refer to evaluation section for the shortcomings of the plan.

```
Words: 1020
```

4 Implementation

This chapter details the implementation of the fraud detection pipeline, GNN models, and associated scripts, highlighting key technical choices and code structures. The complete code is available at public repository `https://github.com/amilos/fdgnn`.

4.1 Technology Stack

Python was selected for its rich data science ecosystem. Pandas facilitated data handling, while Scikit-learn provided robust tools for standard preprocessing (imputation, scaling, encoding) and metrics calculation. For the GNNs, PyTorch was selected for its flexibility, and PyTorch Geometric (PyG) was utilized for its specialized GNN layers and graph data structures, significantly simplifying implementation [Fey and Lenssen 2019]. Joblib and Pickle ensured consistent saving/loading of processors and data. Streamlit was used for the final demonstration app due to its rapid development capabilities.

4.2 Building blocks

Implementation includes Jupyter Notebooks, Python scripts, Python modules, unit tests and a demo app as illustrated in Figure 4.1

4.3 Data Preprocessing Pipeline Implementation

The preprocessing pipeline is orchestrated by `preprocess_main.py`, utilizing helper functions defined in `preprocess_utils.py`.

Data Ingestion

Loads the `train_transaction.csv` and `train_identity.csv` files. Merges them using the `TransactionID` (`config.ID_COL`). To manage computational resources and simulate a scenario where only recent data might be available or relevant for training, the merged DataFrame is sorted by `TransactionDT` (`config.TIMESTAMP_COL`), and a subset of the most recent $N_{ROWS\_TO\_LOAD}$ (e.g., 150,000) transactions is selected for subsequent steps. (See Appendix A for full dataset description [Howard et al. 2019]).

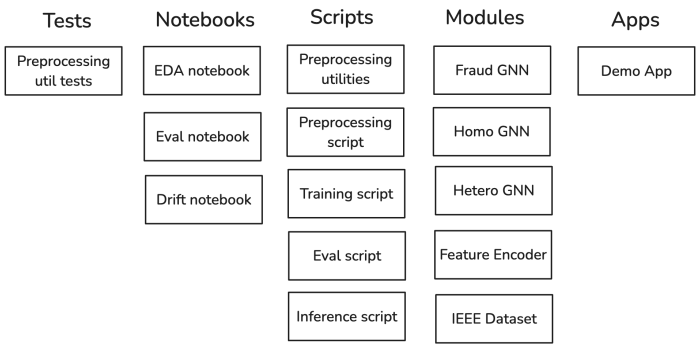

Figure 4.1: Building blocks

Listing 4.1: Example snippet from load function.

```python
# Example snippet from load function
df_trans = pd.read_csv(transaction_path, nrows=nrows, low_memory=False)
df_id = pd.read_csv(identity_path, low_memory=False) # Load full ID initially
df = pd.merge(df_trans, df_id, on=config.ID_COL, how='left')
df = df.sort_values(config.TIMESTAMP_COL)
if nrows is not None:
    df = df.tail(nrows)
```

Entity ID Engineering

This step is crucial for graph construction. Custom functions (`create_card_id` etc.) combine multiple source columns (defined in `config.py` as `CARD_ID_COLS`, etc.) into single composite string identifiers for each non-target node type (`config.DEFAULT_NON_TARGET_NODE_TYPES`). These functions incorporate logic to handle missing values in the source columns, typically by substituting `'nan'` or assigning a specific `'missing_...'` identifier if multiple source components are absent. This ensures each transaction can be linked to corresponding entity nodes in the graph. (See Appendix B for detailed function logic).

Listing 4.2: Example snippet from create_device_profile_id.

```python
# Example snippet from create_device_profile_id
def create_device_profile_id(row):
    cols = config.DEVICE_ID_COLS
    values = [str(row[c]) if pd.notnull(row[c]) else 'nan' for c in cols if c in
         row]
    if values.count('nan') >= max(1, len(values) - 1):
        return 'missing_device'
    else:
        return '_'.join(values)
df['device_profile_id'] = df.progress_apply(create_device_profile_id, axis=1)
```

Time-Based Splitting

To provide a realistic evaluation and account for potential concept drift, the data is split chronologically. The `time_based_split` function takes the feature matrix X, target vector y, and the full DataFrame `df` (containing `TransactionDT`). It sorts based on `TransactionDT`, calculates integer indices corresponding to the specified `train_size`, `val_size`, and `test_size` proportions, and returns the split DataFrames (X_{train}, X_{val}, X_{test}) and Series (y_{train}, y_{val}, y_{test}) using `.loc`. This guarantees that the validation and test sets contain data points occurring strictly after those in the training set.

Handling Fully Missing Columns

An artifact of using a time-ordered subset can be columns that contain only `NaN` values in the resulting X_{train}. These columns offer no information for fitting processors. The pipeline identifies such columns (`X_train[col].isnull().all()`) and drops them from X_{train}, X_{val}, and X_{test} to prevent errors in subsequent steps. The master lists of numerical (`num_cols`) and categorical (`cat_cols`) features are updated accordingly.

Imputation

Missing values in the *remaining* feature columns are imputed using `sklearn.impute.SimpleImputer` [Pedregosa et al. 2011]. Separate imputers are fitted *only* on X_{train}: one for numerical columns using median as strategy and one for categorical columns using `'missing'` as constant. The `fit=False` mode applies the `.transform()` method using the fitted imputers to X_{val} and X_{test}.

Encoding

This function applies different strategies based on the target `model_type`:

- **GNN Path (`model_type='gnn'`)**: All specified categorical columns (`cat_cols`) are label encoded into integer indices (0, 1, 2...). A mapping dictionary for each column is created and stored. Unseen categories during transform are mapped to -1. Additionally, target embedding dimensions for each categorical feature are calculated (e.g., based on cardinality using $\min(\text{max\_dim}, \max(2, \text{int}(\sqrt{\text{cardinality}} \times \text{factor})))$) and stored in the `encoder` dictionary. This prepares the data for `nn.Embedding` layers [Paszke et al. 2019].
- **XGBoost Path (`model_type='xgboost'`)**: Categorical columns are divided based on cardinality relative to `config.CARDINALITY_THRESHOLD_FOR_OHE`. Low-cardinality features undergo One-Hot Encoding (`OneHotEncoder` with `handle_unknown='ignore'`) [Pedregosa et al. 2011]. High-cardinality features are label encoded (similar to the GNN path). The fitted `OneHotEncoder` object, the generated OHE feature names (`ohe.get_feature_names_out()`), and the label encoding maps are stored. During transform (`fit=False`), the stored OHE feature names are crucially used to reconstruct the DataFrame, ensuring consistent columns even if a category is absent in the batch.

Scaling

Remaining numerical features are scaled using `StandardScaler` [Pedregosa et al. 2011], fitted only on X_{train} numerical data. The same fitted scaler is applied via `.transform()` to X_{val} and X_{test} for both GNN and XGBoost feature sets.

Output Saving

The final processed DataFrames (X_{train_xgb}, X_{val_xgb}, etc., and X_{train_gnn}, etc.) and corresponding labels/indices are saved into `processed_data.pkl`. A snapshot DataFrame (`full_df_original_ids`), containing only the original indices, `TransactionID`, `TransactionDT`, and the engineered entity ID columns required for graph building, is also included in this pickle file. The dictionary containing all fitted processors (`imputers`, `encoder_xgb`, `encoder_gnn`, `scaler`) and metadata (`final_num_cols`, `final_cat_cols`, `num_numerical_features`, `num_nodes_dict`) is saved separately using `joblib` to `processors.joblib`. Finally, the `sample_inference_data` function is called to generate the small, balanced test subset CSVs for the demo app.

4.4 Graph Construction

A heterogeneous graph representation is implemented using PyTorch `HeteroData` structure [Fey and Lenssen 2019] within the dataset class.

For all edge types, the *edge_index* tensor is constructed by mapping the original transaction indices and entity IDs (from *full_df_original_ids*) to their respective contiguous graph indices ($0..N-1$ for transactions and $0..M-1$ for entities of a given type).

Table 4.1: Node Types in the Heterogeneous Graph

Node Type	Description	Features / Data Stored
transaction	Represents individual transactions being analyzed.	Stores numerical and categorical features (x_{num}, x_{cat}), fraud labels (y), and data split masks (*train_mask*, *val_mask*, *test_mask*). `num_nodes` corresponds to the total unique transactions in the processed data splits.
card_id, *device_profile_id*, *network_profile_id*, *locality_id*	Represent engineered entities associated with transactions.	Initially assigned placeholder features (x). Meaningful embeddings are learned by the `FeatureEncoder` based on node indices during training. `num_nodes` for each type is determined by the count of unique IDs in the `full_df_original_ids` snapshot.

Table 4.2: Edge Types Connecting Nodes in the Graph

Canonical Edge Type	Description and Construction
(*transaction*, *linked_to_<entity>*, *<entity>*)	Connects a *transaction* node to its corresponding entity node (where *<entity>* is one of *card_id*, *device_profile_id*, etc.).
(*<entity>*, *linked_from_<entity>*, *transaction*)	Represents the reverse relationship, connecting an entity node back to the *transaction* node it is linked to.

The `IeeeFraudDetectionDataset` class implemented inherits from PyG's `InMemoryDataset`.

- **`__init__`**: Loads `processed_data.pkl` and `processors.joblib`. Calls `super().__init__()` and then loads the final graph using `torch.load(self.processed_paths[0], weights_only=False)`, ensuring it unpacks `data` and `slices`.
- **`_extract_features_and_labels`**: Helper function to get `x_num`, `x_cat`, `y`, and original indices for a given split ('train', 'val', 'test') from the loaded `processed_data`.
- **`process`**: Orchestrates the following steps: loading processed data, mapping indices, creating tensors for node features/labels/masks, iterating through entity types to create entity nodes and edge indices (using Pandas merge for efficiency [McKinney 2010]), validating the graph, and saving the collated `HeteroData` object.
  - Initializes `HeteroData()`.
  - Calls `_extract_features_and_labels` for all splits.
  - Creates mapping from original indices to contiguous graph node indices (0 to N-1) for transaction nodes.
  - Combines features (`x_num`, `x_cat`) and labels (`y`) into tensors aligned with the new contiguous indices

and assigns them to `transaction`.
  - Creates boolean `train_mask`, `val_mask`, `test_mask` tensors and assigns them.
  - Loads the `full_df_original_ids` snapshot.
  - Calls `data.validate()`.
  - Saves the graph using `torch.save(self.collate([data]), self.processed_paths[0])`.

4.5 Model Implementation

Feature Encoder

- **`__init__`**: Takes `node_types_list`, `num_nodes_dict`, `encoder_info`, `embedding_dim_other_nodes`, `hidden_dim`. Creates `nn.ModuleDicts` for embeddings (`embeddings_trans_cat`, `embeddings_other`) and linear maps. Initializes `nn.Embedding` layers with correct sizes (`cardinality+1` for transaction categoricals, `num_nodes` for others). Initializes `nn.Linear` layers to project to `hidden_dim`.
- **`forward`**: Takes `x_input_dict`. Processes `transaction` features (concats numerical + looked-up/clamped categorical embeddings) and applies linear map. Processes other node types (looks up ID embedding using `nn.Embedding.weight` for full batch) and applies linear map. Ensures an output tensor (potentially zeros if no input/nodes) of shape `[num_nodes, hidden_dim]` exists for every `node_type` in the input `node_types_list`. Returns the `x_dict` of encoded features.

HeteroGNN and HomoGNN

Implemented using PyTorch and PyTorch Geometric (PyG) with standard convolution layers.

- **GraphSAGE (`SAGEConv`)**: An inductive GNN layer that learns node embeddings by sampling and aggregating features from a node's local neighborhood [Hamilton et al., 2017]. Used with mean aggregation.
- **Graph Attention Network (`GATConv`)**: Incorporates attention mechanisms to assign different weights to different neighbors during aggregation, allowing the model to focus on more relevant connections [Veličković et al., 2018 - *cite original GAT paper*]. Used with multiple heads (typically 4).
- **Relational Graph Convolutional Network (`RGCNConv`)**: Extends GCNs to handle heterogeneous graphs with multiple edge types by learning relation-specific transformations [Schlichtkrull et al., 2018 - *cite original RGCN paper*]. Used within `HeteroConv`.
- **Heterogeneous Convolution (`HeteroConv`)**: A PyG wrapper that applies specified homogeneous convolution layers across different edge types in a `HeteroData` object, handling message passing and aggregation between different node types.

FraudGNN

- **`__init__`**: Takes metadata, counts, encoder info, and architecture hyperparameters. Determines node types and instantiates `FeatureEncoder`. It then instantiates either `HeteroGNN` or `HomoGNN` based on `model_type`, passing correct input dimensions (`hidden_channels` for both, as `FeatureEncoder` runs first). Finally, it instantiates `final_layer`.
- **`forward`**: Takes `data` (always `HeteroData` in the final version) and optional precomputed features and `homo_structure`.
  - If `x_dict_encoded` is not provided, it runs `self.feature_encoder` on `data`.
  - If `model_type == 'hetero'`, passes `x_dict_encoded` and `data.edge_index_dict` to `self.gnn`.
  - If `model_type == 'homo'`, checks for `x_homo` and `homo_structure`. Passes `x_homo` and `edge_index` to `self.gnn`. Extracts transaction outputs using `homo_structure.node_type`.

– Applies `self.final_layer` to the extracted `x_transaction`. Returns logits.

4.6 Training, Evaluation, Inference Scripts

- **`train_*.py`**: Use `argparse` for hyperparameters. Load data/processors. Instantiate model. Implement training loop with `optimizer.zero_grad()`, `model()`, `criterion()`, `loss.backward()`, `optimizer.step()`. Include validation step with `model.eval()`, `torch.no_grad()`, metric calculation (AUC). Implement `ReduceLROnPlateau` scheduler and early stopping. Save best `model.state_dict()` and `config.json` to a unique run directory. Includes pre-calculation of features before the loop for efficiency.
- **`evaluate_*.py`**: Use `argparse` to get `run_dir`. Load `config.json` from `run_dir` to get hyperparameters. Load test data and processors. Instantiate model with loaded hyperparameters. Load saved `model.state_dict()`. Run inference on the test set (pre-calculating features if needed by the model's forward). Calculate and print metrics (AUC, AP, F1, CM).
- **`infer_*.py`**: Similar to evaluation but loads raw inference data, calls `preprocess_for_inference`, loads model/config from `run_dir`, performs prediction (using appropriate strategy, e.g., GNN features-only path), and saves results.

4.7 Demo Application

Application for demonstration of model inference was built using Streamlit library. Here are its main responsibilities:

- Loads the pre-calculated prediction CSVs (`xgb_predictions.csv`, `gnn_predictions.csv`) and the raw sample data CSVs.
- Merges data based on `TransactionID`.
- Conditionally calculates calibrated GNN scores, score difference, and agreement flags.
- Conditionally filters to show top 20 GNN confidence rows. Formats data (datetime, currency, emoji).
- Displays the final table using `st.dataframe` with Pandas styling for highlighting actual fraud and cell coloring (scores).
- Includes an interactive `st.slider` to adjust the classification threshold and dynamically updates displayed confusion matrices for the sample data.

```
Words: 1858
```

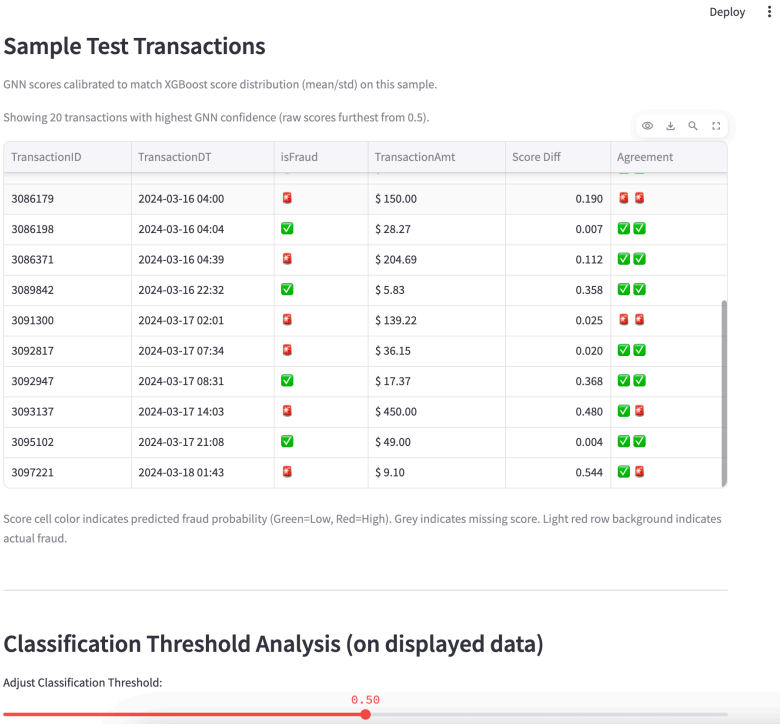

Figure 4.2: Demo app UI screenshot

5 Evaluation

5.1 Model Performance Evaluation

The primary goal of the evaluation was to quantitatively compare the effectiveness of the GNN approach against the strong XGBoost baseline in a realistic fraud detection scenario, adhering to the principles outlined in the project template. The strategy focused on assessing generalization performance on unseen, future data.

- **Test Set Definition:** A dedicated test set, comprising the final 15% of the chronologically sorted data subset (approx. 15,000 transactions out of 100,000 used), was strictly held out. This temporal separation ensures evaluation reflects the model's ability to predict future fraud based on past patterns, directly addressing the challenge of potential concept drift inherent in financial data [Bayram et al. 2020].
- **Metrics Selection:** Given the severe class imbalance observed during EDA (Appendix A), metrics sensitive to minority class performance were prioritized alongside overall discrimination measures:
  - **ROC-AUC:** Provides a threshold-independent measure of the model's ability to distinguish between fraud and non-fraud classes.
  - **Average Precision (AP / PR-AUC):** Summarizes the precision-recall curve, offering a more informative view than AUC for highly imbalanced tasks by focusing on the positive (fraud) class performance.
  - **Fraud Class F1-Score, Precision, Recall:** Calculated using a standard 0.5 classification threshold to provide insight into the practical trade-offs. F1-score balances Precision (minimizing false alarms) and Recall (minimizing missed frauds).
  - **Confusion Matrix:** Visualizes the counts of True/False Positives/Negatives at the 0.5 threshold.
- **Evaluation Procedure:** The evaluation was conducted using dedicated scripts (`evaluate_xgb.py`, `evaluate_gnn.py`). These scripts load the preprocessed test data split, load the specific trained model (either the XGBoost model file or a GNN run directory containing the saved state dictionary and configuration JSON), perform inference on the test data, and compute the aforementioned metrics using Scikit-learn functions. For GNNs, the evaluation script reconstructs the model architecture based on the saved configuration before loading weights, ensuring consistency.

Baseline Model Performance

The XGBoost model, trained on the processed tabular feature set derived from the common preprocessing pipeline, served as the performance benchmark. Its results on the test set are summarized below.

Table 5.1: XGBoost Baseline Performance on Test Set

Metric	Score
AUC	0.8735
AP (PR-AUC)	0.3487
F1 (Fraud)	0.4220
Precision (Fraud)	0.6621
Recall (Fraud)	0.3097

As indicated in Table 5.1 and Figure 5.1, the XGBoost baseline achieved strong overall discrimination (AUC 0.8735). Its Average Precision (0.3487) reflects reasonable performance on the minority class. At the default threshold, it prioritized precision (0.6621) over recall (0.3097), correctly identifying 96 fraud instances while

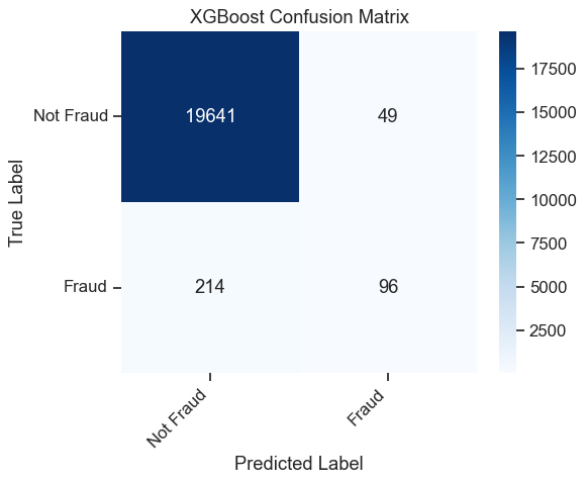

Figure 5.1: Confusion matrix for the baseline model on the test set at a 0.5 probability threshold

generating only 49 false positives, but missing 214 actual frauds. This performance sets a high standard for the GNN models.

GNN Model Performance

Multiple GNN configurations were trained and evaluated, exploring heterogeneous and homogeneous structures with different convolution layers (SAGE, GAT, RGCN) and hyperparameters (hidden dimensions, layers). A consistent observation during training was that validation AUC often reached promising levels (e.g., 0.81-0.85), suggesting the models were capable of learning from the training and validation data distributions. However, this performance did not translate to the test set. Table 5.2 summarizes the test set performance for key GNN runs.

Table 5.2: Comparison of Key GNN Model Runs on Test Set

Model Configuration (Run ID Suffix)	AUC	AP	F1 (Fraud)	Prec (Fraud)	Recall (Fraud)
homo_SAGE_h128_l3	0.8151	0.1443	0.0906	0.0485	0.6774
hetero_RGCN_h128_l2	0.7879	0.1166	0.0878	0.0471	0.6387
hetero_SAGE_h32_l1	0.6580	0.0463	0.0583	0.0309	0.5161

The results clearly show that GNN performance on the test set was substantially lower than XGBoost's. The best-performing GNN configuration tested (homo_SAGE_h128_l3) achieved a Test AUC of 0.8151, which, while better than random, is significantly below the baseline. More critically, its Average Precision was only 0.1443. The confusion matrix (Figure 5.2) reveals that this model achieved high recall (0.6774) but at the expense of extremely poor precision (0.0485), generating over 4000 false positives. This pattern of high recall coupled with very low precision was consistent across other GNN configurations, rendering them impractical for real-world use at this threshold. Simpler models (hetero_SAGE_h32_l1) performed even worse overall.

Comparative Analysis

- **Baseline Superiority:** XGBoost consistently outperformed all tested GNN variants on the primary evaluation metrics (AUC, AP) on the test set. The GNNs failed to leverage the graph structure effectively enough to surpass the strong tabular baseline in this experimental setup.
- **Generalization Failure (Validation vs. Test Gap):** The most significant finding is the dramatic drop in performance for GNNs between the validation set (where AUCs often exceeded 0.81) and the test set (where

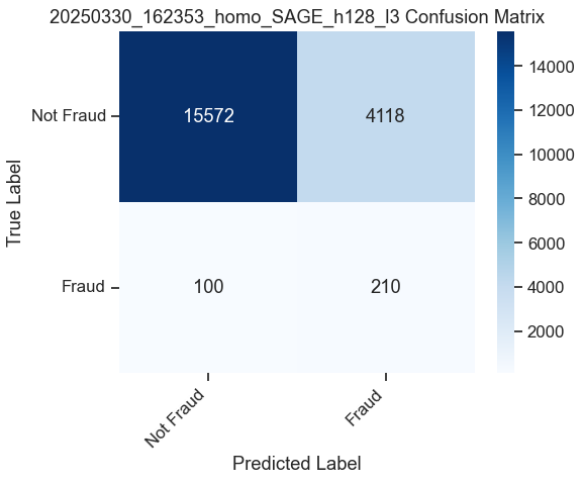

Figure 5.2: Confusion matrix for the homo_SAGE_h128_l3 GNN model on the test set at a 0.5 probability threshold

AUCs were lower and AP scores plummeted). This indicates a severe failure to generalize to data from a later time period.

- **Precision-Recall Imbalance:** GNNs consistently favored recall over precision at the default threshold, leading to high false positive rates. XGBoost offered a more balanced, higher-precision profile. While threshold tuning could adjust the GNN's operating point, the underlying lower AP suggests its precision would remain inferior to XGBoost's across various recall levels.
- **Concept Drift as Plausible Explanation:** The EDA (Appendix A) demonstrated significant target and feature drift over time. This non-stationarity is the most plausible explanation for the GNNs' poor generalization. GNNs, potentially learning intricate relational patterns specific to the training/validation data distribution, appear highly sensitive to these temporal shifts. The learned embeddings and message-passing weights likely became less effective on the drifted test data.

5.2 Critical Evaluation of the Project

This project aimed to implement and evaluate GNNs against XGBoost for fraud detection on a public dataset, aligning with the CM3015 template. We can evaluate its success against original aims, scope and standard project success criteria.

Known Limitations

1. **Dataset:** Findings are based on a 100k-150k row subset. Results might differ on the full dataset, although the observed drift is present throughout. Dataset is heavily obfuscated as 95% of columns does does not have any semantics. All identifiers are removed for privacy leaving the option to engineer pseudo identifiers for non-target nodes.
2. **Feature Engineering:** Primarily relied on existing features and basic embeddings. Lack of domain-specific, time-aware, or graph-topology features likely limited GNN potential.
3. **Graph Structure:** The engineered entity pseud-identifiers, while functional, may not represent the optimal way to structure the graph for fraud detection due to missing data and the chosen combination logic.
4. **Computational Limitations:** CPU training times for GNNs (especially homogeneous) were significant, and issues with the PyTorch MPS backend for Applce Sillicon prevented GPU acceleration. This restricted the extent of hyperparameter tuning and experimentation possible on the available hardware.

5. **GNN Architectures:** Due to scope and time constraints, exploration was limited to standard GNN layers leaving more specialized architectures for future work.
6. **Testing and QA:** Due to scope and time constraints, unit testing was developed to critical preprocessing utilies. As a result of this, productization of this code for inference would require more comprehensive functional and performace testing.

Critique and Challenges

1. **GNN Performance Goal Not Met:** The central hypothesis that GNNs would outperform XGBoost was not validated on the test set. The GNNs failed to generalize effectively, representing the main scientific/technical failure of the project in terms of achieving state-of-the-art GNN performance.
2. **Project Contingency Management:** An unforeseen and significant issue at my workplace in February and March severely disrupted my project schedule, causing me to lose approximately 20 days of planned development time. This disruption posed risks for meeting deadlines and completing my project according to the original plan. Due to significant pause on project, my progress stalled, particularly impacting exploration of additional GNN architectures and interpretability of GNN models. To mitigate this, I used all the reserve buffer for contingencies and re-prioritized tasks, reducing the time allocated to additional experiments while focusing strictly on essential functionality and critical evaluations to ensure the final product met minimum requirements. With the benefit of hindsight, it appears that my project contingency planning did not correctly account for risks.
3. **Concept Drift Mitigation:** While drift was identified, the project scope and timeline did not allow for the implementation of specific techniques (e.g., temporal features, adaptive learning, temporal GNNs) to counteract its effects, which ultimately limited GNN performance.

```
Words: 1288
```

6 Conclusion

This project aimed to evaluate the effectiveness of Graph Neural Networks (GNNs) compared to a traditional XGBoost baseline for credit card fraud detection on the IEEE-CIS dataset. A comprehensive pipeline was developed, including data preprocessing, heterogeneous graph construction based on engineered entity IDs (card, device, network, locality), time-based data splitting, and the implementation of both heterogeneous with SAGE/GAT/RGCN and homogeneous with SAGE/GAT GNN models using PyTorch Geometric.

6.1 Findings and Key Results

The primary finding was that while GNN models demonstrated promising learning capabilities on the validation set (AUCs 0.81), they failed to generalize effectively to the temporally distinct test set, exhibiting significant performance degradation (Test AUCs 0.65-0.82, Test AP 0.15). In contrast, the XGBoost baseline maintained stronger performance on the test set (AUC ~0.87, AP ~0.35), achieving a better balance between precision and recall for the minority fraud class.

Exploratory data analysis confirmed the presence of substantial **concept drift** within the dataset subset used. Both the target variable (fraud rate) and the distributions of numerous key features changed significantly over the time period covered by the train, validation, and test splits. This non-stationarity is concluded to be the main factor hindering the GNNs' generalization. The complex patterns and embeddings learned by the GNNs on earlier data likely became less relevant or inaccurate for the later test data, a challenge potentially exacerbated by the model architectures explored. While XGBoost is also affected by drift, its ensemble nature and reliance on finding optimal splits within the tabular feature space appeared more robust in this instance.

Key Results

1. **Implementation Scope:** Successfully implemented a complex end-to-end pipeline, including data processing, graph schema modelin with generation of pseudo identifier, heterogeneous graph construction, baseline (XGBoost) and multiple GNN model implementations (hetero/homo SAGE/GAT/RGCN), and evaluation scripts. This demonstrates proficiency in handling diverse data formats and applying both traditional ML and advanced GNN techniques.
2. **Methodology:** Employed sound methodological choices, including time-based splitting (critical for this domain), appropriate metrics for imbalanced data (AUC, AP), and establishing a strong baseline for comparison. The use of configuration files and run-specific directories promotes reproducibility.
3. **Problem Identification:** Successfully identified and visualized significant concept drift through EDA, providing a well-supported explanation for the primary challenge encountered – the GNNs' failure to generalize.
4. **Demonstration:** Developed a simple but functional Streamlit application showcasing the project's outputs and facilitating comparison between model predictions.

6.2 Future Work

This project highlights several important considerations when applying GNNs to real-world, time-sensitive problems like fraud detection:

1. **Interpretability:** While not explored due to scope prioritization and time constraints, understanding *why* a GNN makes a certain prediction is crucial for deployment in practice. Future work could integrate methods

like GNNExplainer [Ying et al., 2019] or leverage attention scores (from GAT) to provide insights into the model's reasoning, potentially identifying key subgraphs or features driving fraud predictions. This could also aid in debugging and model refinement.

2. **Scalability and Real-Time Inference:** The observed CPU performance issues, particularly with the homogeneous model, emphasize the need for efficient implementations and potentially specialized hardware (GPUs) for large-scale graphs. For real-time detection, inductive GNNs (like GraphSAGE) combined with efficient graph updating strategies and optimized inference pipelines are necessary [Van Belle et al., 2022]. The simplified inference approach used in the demo would need significant enhancement for a production system.
3. **Concept Drift Adaptation:** Standard GNNs assume data stationarity. The observed performance gap underscores the need for techniques specifically designed to handle drift. Future work could explore:
   - **Temporal GNNs:** Architectures explicitly incorporating time (e.g., TGN, DySAT) to model evolving patterns and relationships.
   - **Adaptive Training:** Strategies like periodic retraining on recent data windows or online learning approaches.
   - **Drift-Robust Features:** Engineering features that capture relative changes or are inherently less sensitive to temporal shifts (e.g., time deltas, feature ratios).
4. **Graph Construction and Feature Engineering:** GNN performance is highly sensitive to the input graph structure and node/edge features. Future work should involve:
   - More sophisticated entity definition and linking strategies on a dataset whose semantics and identifiers are not obfuscated for privacy. Financial organizations can use richer private datasets than what can be made public.
   - Adding graph-specific topological features (e.g., node degrees, centrality measures) to node representations.

6.3 Final Words

Ultimately, this project's outcome, where the well-established XGBoost baseline surpassed the GNN models on the temporally distinct test set, underscores a persistent observation in the field: tabular data remains a challenging frontier modality where deep learning architectures have yet to consistently dominate. Unlike domains such as computer vision or natural language processing, traditional gradient boosting methods frequently remain highly competitive, often setting the performance benchmark on structured datasets, particularly when faced with real-world complexities like concept drift.

Furthermore, the experience highlights that the success of GNNs is uniquely and critically dependent on the quality and expressiveness of the underlying graph schema modeling and feature representation. Unlike the more standardized input structures and architectural patterns found in CNNs for images or Transformers for sequences, effectively applying GNNs requires careful, often domain-specific, design choices in translating relational information into a graph structure the network can leverage. This dependence implies that GNN performance is not merely a function of the network layers chosen, but is fundamentally intertwined with the graph construction process itself – a factor likely contributing to their sensitivity observed in this project.

```
Words: 873
```

References

Barış Bayram, Bilge Köroğlu and Mehmet Gönen. 2020. 'Improving Fraud Detection and Concept Drift Adaptation in Credit Card Transactions Using Incremental Gradient Boosting Trees'. In: *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 545–550. DOI: 10.1109/ICMLA51294.2020.00091.

Tianqi Chen and Carlos Guestrin. 2016. 'XGBoost: A Scalable Tree Boosting System'. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD '16). ACM, San Francisco, California, USA, 785–794. DOI: 10.1145/2939672.2939785.

Matthias Fey and Jan Eric Lenssen. May 2019. *Fast Graph Representation Learning with PyTorch Geometric*. Software repository. MIT License. (May 2019). https://github.com/pyg-team/pytorch_geometric.

William L. Hamilton, Rex Ying and Jure Leskovec. 2018. *Inductive Representation Learning on Large Graphs*. (2018). https://arxiv.org/abs/1706.02216.

Addison Howard, Bernadette Bouchon-Meunier, IEEE CIS, inversion, John Lei, Lynn@Vesta, Marcus2010 and Prof. Hussein Abbass. 2019. *IEEE-CIS Fraud Detection*. https://kaggle.com/competitions/ieee-fraud-detection. Kaggle. (2019).

Thomas N Kipf and Max Welling. 2017. 'Semi-Supervised Classification with Graph Convolutional Networks'. *arXiv preprint arXiv:1609.02907*.

M. S. Kumar, V. Soundarya, S. Kavitha, E. S. Keerthika and E. Aswini. 2019. 'Credit Card Fraud Detection Using Random Forest Algorithm'. In: *3rd International Conference on Computing and Communication Technologies (ICCCT)*. IEEE, 149–153. DOI: 10.1109/ICCCT2019.8884656.

G. Liu, J. Tang, Y. Tian and J. Wang. 2021. 'Graph Neural Network for Credit Card Fraud Detection'. In: *2021 IEEE International Conference on Cyber-Physical Social Intelligence (ICCPSI)*. IEEE, 1–10. DOI: 10.1109/ICCPSI53130.2021.9736204.

Wes McKinney. 2010. 'Data Structures for Statistical Computing in Python'. In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman, 56–61. DOI: 10.25080/Majora-92bf1922-00a.

Duojiao Mu. 2022. 'Credit Card Fraud Intelligent Detection Based on Machine Learning'. In: *2022 IEEE International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA)*. IEEE, 1112–1117. DOI: 10.1109/EEBDA53927.2022.9744875.

Adam Paszke et al.. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. (2019). https://arxiv.org/abs/1912.01703.

F. Pedregosa et al.. 2011. 'Scikit-learn: Machine Learning in Python'. *Journal of Machine Learning Research*, 12, 2825–2830.

Zidi Qin, Yang Liu, Qing He and Xiang Ao. 2022. 'Explainable Graph-based Fraud Detection via Neural Meta-graph Search'. In: *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22)*. ACM, Atlanta, GA, USA, 4414–4418. DOI: 10.1145/3511808.3557598.

Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov and Max Welling. 2017. *Modeling Relational Data with Graph Convolutional Networks*. (2017). https://arxiv.org/abs/1703.06103.

R. Van Belle, C. Van Damme, H. Tytgat and J. De Weerdt. 2022. 'Inductive Graph Representation Learning for Fraud Detection'. *Expert Systems with Applications*, 193, 116463. DOI: 10.1016/j.eswa.2021.116463.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò and Yoshua Bengio. 2018. *Graph Attention Networks*. (2018). https://arxiv.org/abs/1710.10903.

A Dataset Description

The dataset used is the IEEE-CIS Fraud Detection dataset, publicly available on Kaggle [Howard et al. 2019]. It consists of two main tables:

1. **`train_transaction.csv`**: Contains transaction information, including `TransactionID`, `TransactionDT` (timestamp offset), `TransactionAmt`, `ProductCD`, various anonymized categorical (`M1`-`M9`) and numerical (`C1`-`C14`, `D1`-`D15`) features, card information (`card1`-`card6`), and address information (`addr1`, `addr2`, `P_emaildomain`, `R_emaildomain`). It also includes the target variable `isFraud`.
2. **`train_identity.csv`**: Contains identity information linked by `TransactionID`, including anonymized identity features (`id_01`-`id_38`) and device information (`DeviceType`, `DeviceInfo`).

The full training dataset contains over 590,000 transactions.

A.1 Data Subset Used

Due to computational limitations for GNN training and graph construction within the project timeframe, a subset of the data was used. The merged transaction and identity data was sorted chronologically by `TransactionDT`. A subset consisting of the most recent `N_ROWS_TO_LOAD` (set to 100,000-150,000 during experiments) transactions was selected for training, validation, and testing using a time-based split. This approach preserves the temporal ordering crucial for simulating real-world fraud detection but means the models were trained and evaluated on a fraction of the available data.

A.2 Exploratory Data Analysis (EDA) Highlights

1. **Target Imbalance:** The dataset exhibits significant class imbalance, typical of fraud detection problems. In the subset used, the fraud rate (`isFraud` = 1) was approximately 2-4%, varying over time (see Concept Drift section). This imbalance necessitates the use of appropriate evaluation metrics like AUC and Average Precision and techniques like class weighting during training.

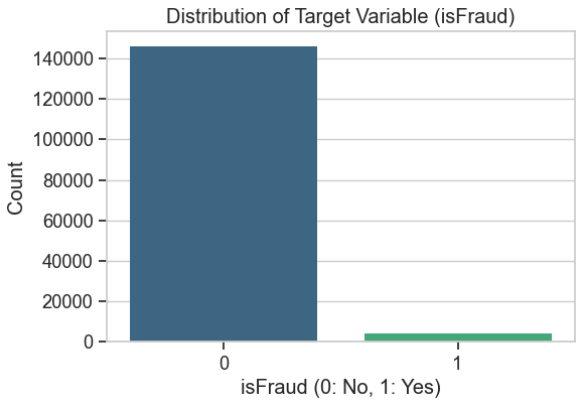

Figure A.1: Target imbalance

2. **Timestamp:** This feature represents a timedelta in seconds from an unknown reference point. It was converted to 'TransactionDays' for easier interpretation and used to create `N_TIME_BINS` (typically 10)

equal-frequency time bins for drift analysis. The distribution showed continuous activity over the period covered by the subset.

3. **Entity ID Source Columns:** An analysis of the columns used to engineer the GNN entity IDs revealed varying characteristics:
   - **Missingness:** Columns like `addr2`, `dist2`, and many `id_xx` features (especially those from the identity table) had substantial missing values, impacting the completeness of derived IDs.
   - **Cardinality:** `card1`, `addr1`, `DeviceInfo` showed high cardinality, suitable for creating distinct nodes. Others like `card4`, `DeviceType`, `M` columns had lower cardinality.
4. **Engineered Entity IDs:** The engineered IDs (`card_id`, `device_profile_id`, etc.) generally exhibited high cardinality, confirming their ability to create distinct nodes. The `card_id`, in particular, showed good selectivity, with many IDs being associated with multiple transactions, including both fraudulent and non-fraudulent ones, suggesting its potential utility in graph message passing.

A.3 Concept Drift Analysis

A dedicated analysis was performed (see `notebooks/concept_drift_analysis.ipynb`) by dividing the data subset into `N_TIME_BINS` based on `TransactionDT`.

- **Target Drift:** A clear and significant drift in the fraud rate was observed. The rate decreased initially before rising sharply in the later time bins corresponding to the validation and test periods. This non-stationarity in the target variable presents a major challenge for model generalization.

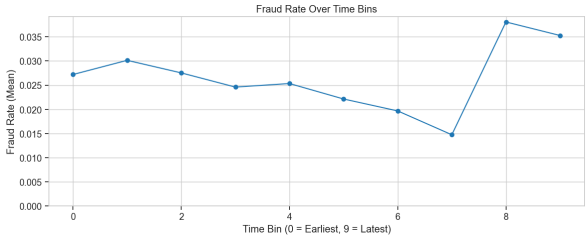

Figure A.2: Target drift

- **Feature Drift (Numerical):** Several key numerical features showed significant changes in their distributions or central tendencies over time. Notably, `id_01` trended downwards, `id_02` trended upwards, and the shapes of `D2` and `D3` distributions shifted. `TransactionAmt`'s median also increased over time.
- **Feature Drift (Categorical):** Significant shifts in category proportions were observed for important features like `ProductCD` and `M4`. The relative frequencies of different device/match statuses (`id_15`, `id_34`) also changed, particularly in the middle time bins.

Drift Conclusion

The EDA strongly indicates the presence of concept drift affecting the target variable and numerous input features within the analyzed data subset. This non-stationarity is a critical factor influencing model performance, particularly the observed drop between validation and test sets for the GNN models.

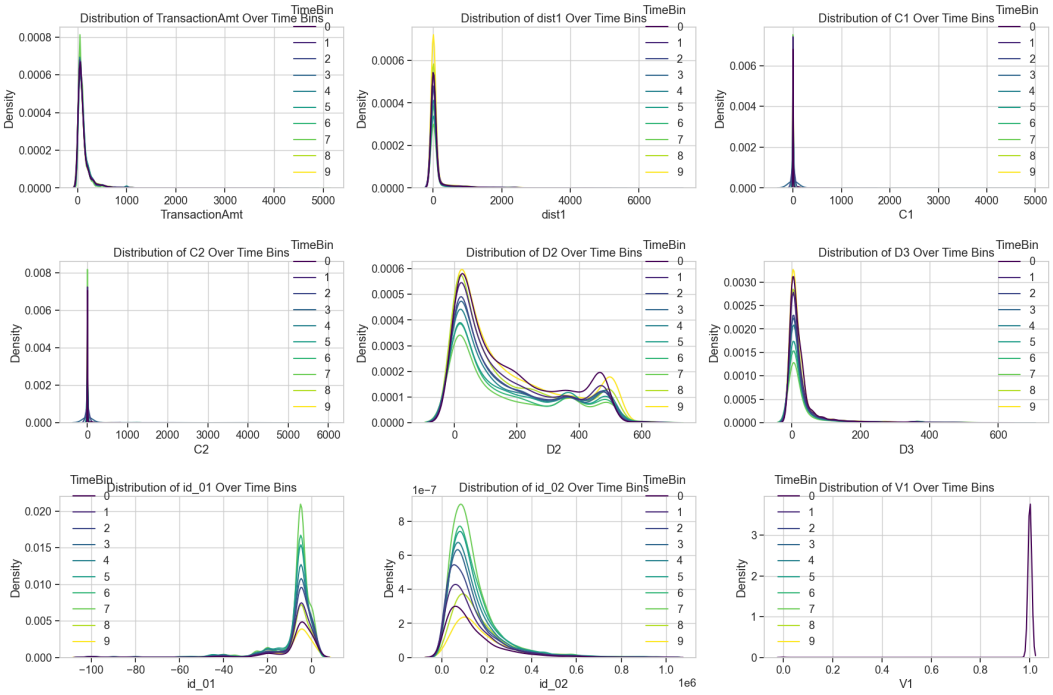

Figure A.3: Numerical feature drift

Figure A.4: Categorical feature drift

B Implementation Details

This chapter provides implementation details, including the project structure, key functions, and unit tests.

B.1 Project structure

Listing B.1: Project structure

```
fdgnn/
 data/                              # Data directory
    raw/                           # Place raw train/test csv files here
    processed/                     # Output of preprocessing
 models/                            # Saved trained models
 notebooks/                         # Jupyter notebooks
    eda.ipynb                      # EDA notebook
    model_evaluation.ipynb         # Model evaluation notebook
    concept_drift_analysis.ipynb   # Concept drift analysis notebook
 scripts/                           # Scripts for various tasks
    evaluate_gnn.py                # GNN evaluation script
    evaluate_xgb.py                # XGBoost evaluation script
    infer_gnn.py                   # GNN inference script
    infer_xgb.py                   # XGBoost inference script
    preprocess_main.py             # Runs common preprocessing
    preprocess_utils.py            # Preprocessing functions
 src/                               # Source code for GNN and configurations

    config.py                      # Constants & configurations
    gnn_model.py                   # GNN architecture definition
    gnn_dataset.py                 # PyG Dataset class definition
 tests/                             # Unit tests for preprocessing utilities
 app.py                             # Demo app for fraud scoring
 requirements.txt                   # Python dependencies
 README.md                          # This file
```

B.2 Salient Code Snippets

Listing B.2: Loading demo data

```
  def load_demo_data_with_predictions():
      try:
          trans_path = config.INFERENCE_SAMPLE_TRANS_PATH
          id_path = config.INFERENCE_SAMPLE_ID_PATH
          df_trans = pd.read_csv(trans_path)
          df_id = pd.read_csv(id_path)
          df_sample = pd.merge(df_trans, df_id, on=config.ID_COL, how='left')
          return df_sample
      except FileNotFoundError as e:
          st.error(f"Error loading data: {e}")
          return pd.DataFrame()
```

This function loads and merges transaction and identity data for demonstration purposes. It uses Streamlit's caching mechanism to optimize repeated calls.

Listing B.3: Highlighting Fraud Rows

```
def highlight_fraud(row):
    color = '#FFDDDD'
    if 'isFraud_numeric' in row and row['isFraud_numeric'] == 1:
        return [f'background-color: {color}'] * len(row)
    else:
        return [''] * len(row)
```

This function applies a red background to rows in a DataFrame that represent fraudulent transactions.

Listing B.4: Agreement flag

```
def agreement_flag(row, threshold=0.5):
    gnn_col = 'gnn_score_calibrated' if 'gnn_score_calibrated' in row else '
        gnn_score'
    xgb_pred = row['xgb_score'] > threshold if pd.notnull(row['xgb_score'])
        else None
    gnn_pred = row[gnn_col] > threshold if pd.notnull(row[gnn_col]) else None
    if xgb_pred is None or gnn_pred is None: return " "
    elif xgb_pred == gnn_pred: return "  " if not xgb_pred else "□□□□"
    elif not xgb_pred and gnn_pred: return " □□"
    else: return "□□ "
```

This function compares predictions from XGBoost and GNN models and returns an agreement flag indicating whether the predictions match.

Listing B.5: Test for Preprocessing Utility

```
def test_create_device_profile_id_less_than_3_nan(self):
    row = pd.Series({
        "id_30": "Mozilla",
        "id_31": "chrome",
        "id_32": "32",
        "id_33": "1920x1080",
        "DeviceType": "desktop",
        "DeviceInfo": "windows"
    })
    expected = "Mozilla_chrome_32_1920x1080_desktop_windows"
    actual = preprocess_utils.create_device_profile_id(row)
    self.assertEqual(actual, expected)
```

This test ensures that the `create_device_profile_id` function correctly generates a device profile ID when fewer than three fields are missing.