Anti-money Laundering using Graph Techniques

Ahmad Naser eddin
Doctoral Program in Computer Science
Computer Science Department
2024

Supervisor
Pedro Ribeiro, Assistant Professor, Faculty of Sciences of the University of Porto

Co-supervisors
Jacopo Bono, Research Data Scientist, Feedzai
David Oliveira Aparício, Machine Learning Scientist, Zendesk


U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

UNIVERSIDADE DO PORTO

DOCTORAL THESIS

Anti-money Laundering using Graph Techniques

Author:
Ahmad NASER EDDIN

Supervisor:
Pedro RIBEIRO

Co-supervisor:
Jacopo BONO
David APARÍCIO

A thesis submitted in fulfilment of the requirements
for the degree of Ph.D. in Computer Science

at the

Faculdade de Ciências da Universidade do Porto

2024

" Networks are everywhere. The brain is a network of nerve cells connected by axons, and cells themselves are networks of molecules connected by biochemical reactions. Societies, too, are networks of people linked by friendships, familial relationships, and professional ties. "

Albert-László Barabási (2003)

Acknowledgements

I extend my deepest gratitude to my supervisors, Professor Pedro Ribeiro, Dr. Jacopo Bono, and Dr. David Apáricio, for their invaluable help throughout my PhD. Professor Ribeiro's constant motivation has been a guiding light in tough times. Dr. Bono's sharp insights and thorough problem-solving have been crucial. Also, Dr. Apáricio's ongoing support, even after leaving Feedzai, with regular discussions and feedback, has been essential.

I am very thankful to Feedzai for providing a great environment for my PhD research. The chance to solve real-world problems here, along with access to lots of data, computational resources, and domain experts, has been key to my research success. I am particularly grateful to Pedro Bizarro for his support during my PhD. My gratitude also goes to my managers, Joao Ascensão and Hugo Ferreira, for their invaluable guidance and the impact they had on my professional growth. To my colleagues at Feedzai, especially the research team, product-AI team, and the domain experts for the valuable discussions that have significantly enriched this research and immensely aided in both my professional and personal development.

To my dear friends in Syria and Portugal your companionship and support are incredible. Also, to my colleagues from DCC for creating a supportive academic and social environment.

A special acknowledgment goes to my beloved wife, Hadeel, for her enduring love, unwavering support, and extreme patience enriching every step of my journey.

Finally, I am extremely grateful to my parents and siblings for their unconditional love and steady support, which has been the foundation of my strength and determination.

Abstract

Money laundering, the process of disguising illegally obtained assets to appear legitimate, poses significant social and economic challenges. It involves crimes like human trafficking and drug dealing. Banks implicated in undetected money laundering cases face substantial fines, underlining the necessity for effective detection mechanisms.

Detecting money laundering presents several challenges: the rarity and delayed confirmation of such events, the tendency of criminals to mimic normal financial activities, the requirement for interpretable evidence for suspicious transactions, processing large volumes of data, and the need for timely event evaluation.

Current ***anti-money laundering (AML)*** solutions, predominantly rule-based, offer clear interpretability essential for auditing but are limited by high ***false positive rates (FPRs)*** and a narrow focus on single-entity behaviors.

This thesis pivots on the premise that graphs, with their inherent capacity to represent and analyze interconnected systems. In essence, graphs are versatile enough to represent datasets where relationships between entities are important, offering a powerful tool for AML investigations. Yet, their complexity presents computational and memory challenges, particularly in real-time applications.

This research combines graphs with AML approaches, aiming to deliver a comprehensive solution that analyzes transactional relationships through a graph. Engineered for real-time decision-making and optimized memory usage, this approach represents a significant advancement in combating money laundering. The thesis introduces innovative methodologies that extract knowledge from graphs and integrate it with ***machine learning (ML)*** techniques to enhance the robustness of AML systems, establishing a framework for money laundering detection and graph information extraction. Moreover, this study extends beyond the realm of AML, offering broader applicability in various other sectors.

Key developments include the adoption of graph feature engineering for sophisticated financial data representation and an ML-integrated *"triage model"* to reduce ***false positives (FPs)*** in AML systems. Notably, the *"Walking-Profiles"* framework employs random-walks for graph feature engineering to enrich the *"triage model"*. To address latency in graph information

extraction, this thesis introduces *"Graph-Sprints"* and *"Deep-Graph-Sprints,"* harnessing real-time feature extraction and advanced deep learning techniques, respectively.

Results indicate significant performance improvements over existing systems, demonstrating enhanced predictive accuracy and speed by an order of magnitude compared to state-of-the-art methods. Future work will extend these methodologies to heterogeneous networks and diverse real-world applications, aiming to scale them for larger datasets.

Keywords: Continuous time dynamic graphs, Random-walks, Machine learning, Anti-money Laundering solutions, low-latency graph processing

Resumo

A lavagem de dinheiro, que consiste no processo de ocultação da obtenção de ativos financeiros de forma ilícita, apresenta desafios socio-económicos significativos, pois abrange crimes associados como o tráfico de seres humanos ou o tráfico de drogas. Bancos que falhem, por negligência, o reporte de casos de lavagem de dinheiro, enfrentam multas substanciais e processos judiciais, reforçando a necessidade do desenvolvimento de mecanismos de deteção eficazes.

A deteção de lavagem de dinheiro apresenta desafios diversos, nomeadamante: a raridade e a confirmação tardia desses eventos, a tendência dos criminosos em imitarem atividades financeiras normais, a necessidade de métodos interpretáveis que justiquem a sua avalição de transações suspeitas, o processamento de grandes volumes de dados e a necessidade de avaliação atempada dos casos.

As soluções atuais para ***sistemas de Anti-Money Laundering (AML)***, predominantemente baseadas em regras, oferecem interpretatividade, que é essencial para o processo de auditoria, mas sofrem de taxas ***altas de falsos positivos (FPRs)*** e estão limitadas a detetar apenas comportamentos de entidades individuais.

Esta tese parte do pressuposto de que redes oferecem uma base poderosa para investigações de AML, pois estas possuem a capacidade inerente de representar e de serem usadas para analisar sistemas interconectados. No entanto, a sua complexidade apresenta desafios computacionais e de armazenamento, que são exacerbados em aplicações em tempo real.

Este doutoramento fornece uma solução abrangente que representa as relações das transações financeiras usando redes. A nossa abordagem é direcionada para a tomada de decisões em tempo real, otimizando o uso de memória, e representa um avanço significativo no combate à lavagem de dinheiro. A tese apresenta metodologias inovadoras que extraem conhecimento das redes e integram-no nos métodos de ***machine learning (ML)***, melhorando, assim, a robustez dos sistemas AML. Esta investigação melhora não apenas os mecanismos de AML, mas também oferece uma solução que é aplicável em vários domínios.

Os desenvolvimentos-chave propostos nesta tese incluem a adoção de feature engineering aplicado em redes para obter uma representação sofisticada dos dados financeiros, e um modelo

de triagem que faz uso dessas representações em modelos de ML para reduzir falsos positivos em sistemas AML. Em particular, a framework *"Walking-Profiles"* utiliza random-walks para aprimorar o modelo de triagem. De forma a lidar com a latência na extração de features da rede, esta tese propõe *"Graph-Sprints"*, que apresenta feature engineering em tempo real, e *"Deep-Graph-Sprints"*, que expande o trabalho anterior usando técnicas avançadas de deep learning.

Os resultados obtidos durante este doutoramento apresentam melhorias significativas em relação aos sistemas existentes, demonstrando maior precisão preditiva e melhorias na latência. O trabalho futuro estenderá essas metodologias para redes heterogéneas e aplicações diversas do mundo real, visando escalabilidade para conjuntos de dados maiores.

Palavras-chave: Redes dinâmicos em tempo contínuo, Random-walks, Machine learning, Sistemas de Anti-Money Laundering, Processamento de redes de baixa latência

Contents

List of Tables

List of Figures

List of Algorithms

Glossary

AD	Automatic Differentiation
AML	Anti-Money Laundering
AP	Average Precision
AUC	Area Under Receiver Operating Characteristic Curve
CTDG	Continuous Time Dynamic Graph
DTDG	Discrete Time Dynamic Graphs
FI	Financial Institution
FP	False Positive
FPR	False Positive Rate
GNN	Graph Neural Network
ML	Machine Learning
RNN	Recurrent Neural Networks
RTRL	Real-Time Recurrent Learning
SAR	Suspicious Activity Report
TP	True Positive

Chapter 1

Introduction

This doctoral research addresses the challenge of detecting money laundering using graph techniques. This chapter introduces the challenges this work addresses and outlines the main contributions. The organization of the chapter is as follows:

- **Background and Motivation**: Section 1.1 describes the typical money laundering process, the limitations of current detection systems, and the motivation behind this research.
- **Research Contributions**: Section 1.2 lists the primary contributions of this study.
- **Research Context**: Section 1.3 offers an overview of the collaborative environment between industry and academia where this research was conducted. This section elucidates the business constraints, setting the stage for understanding the research's practical and theoretical implications.
- **Thesis Organisation**: Section 1.4 explains the structure of the thesis, detailing the contents of each chapter.
- **Bibliographic Note**: Section 1.5 lists the accepted papers and patents related to this thesis.

1.1 Background and Motivation

Money laundering concerns the legitimization of criminal proceeds by concealing their origin, resulting in around 2-5% of global GDP (€1.7-4 trillion) being laundered annually [Lannoo and

Parlour, 2021]. Underlying crimes include drug dealing, human trafficking, fraud, tax evasion, and corruption. Money laundering is, therefore, a severe and global problem affecting people, economies, governments, and the social well-being [McDowell and Novis, 2001].

The process of money laundering can be divided into three fundamental stages, namely, *placement* of illicit money into the financial system, then the *layering* phase in which criminals try to mask the origin of the money, and finally *integration* of the laundered funds into the legitimate economy. Figure 1.1 graphically represents these stages. Throughout this process, money traverses across various accounts, ***financial institutions (FIs)***, and countries, each serving as a step in the laundering journey.

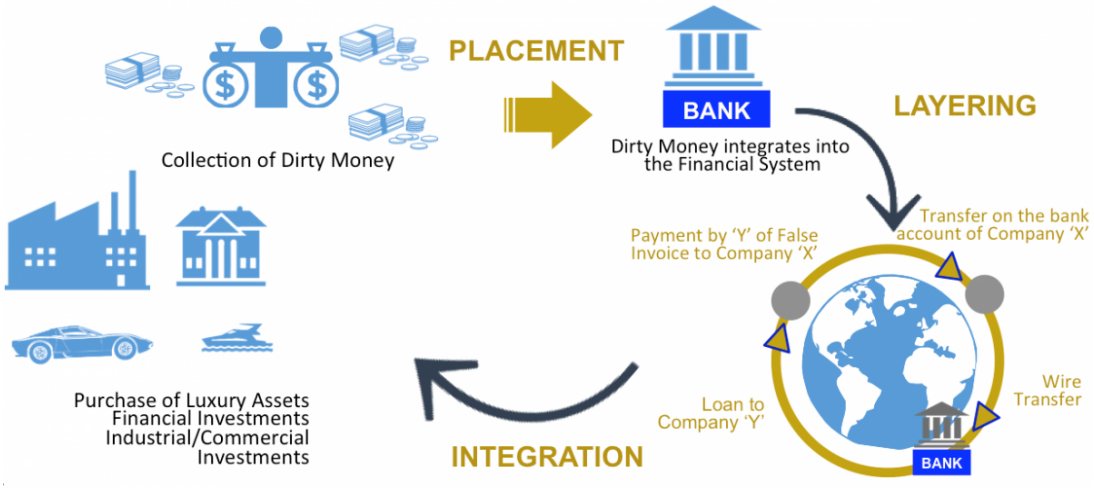


FIGURE 1.1: Illustration of the stages involved in money laundering. Adapted from [The united nations office of drugs and crime, 2020]

For FIs, undetected money laundering can lead to significant fines and reputational consequences. Example penalties in recent years include Deutsche Bank in 2017 with a fine of $630 million [Deutsche Bank fined $630m over Russia money laundering claims, 2017], ING Groep NV in 2018 with $900 million [ING to Pay $900 Million to End Dutch Money Laundering Probe, 2018], Standard Chartered in 2019 with $1.1 billion [Standard Chartered fined $1.1bn for money-laundering and sanctions breaches, 2019], Goldman Sachs in 2020 with $3.9 billion [Goldman Sachs settles 1MDB scandal with Malaysia for $3.9bn, 2020], and Danske Bank in December 2022 with $2.2 billion [Danske Bank Pleads Guilty to Fraud on U.S. Banks in Multi-Billion Dollar Scheme to Access the U.S. Financial System, 2022].

To mitigate risks of money laundering, FIs engage compliance specialists to review and analyze potential irregularities. Given the impracticality of manually assessing every transaction, banks utilize automated AML solutions to support their investigation units.

AML solutions frequently rely on rule-based systems [Li et al., 2017] to identify suspicious cases, these systems encompass a set of rules that are set in accordance with guidelines from international regulatory bodies like the Financial Action Task Force (FATF). To elucidate, one of these rule paradigms is termed *Rapid Movement of Funds*. This specific rule raises an alert when an account receives an amount of money surpassing a predetermined threshold and subsequently distributes nearly all of this amount within a short period; both the temporal window and monetary thresholds are adjustable parameters.

The complete operation of a rule-based AML system is depicted in Figure 1.2. Within this framework, transactions are continuously monitored against these rules, leading to the generation of alerts upon matching predefined criteria. Subsequent to the creation of an alert, analysts conduct thorough reviews to confirm its legitimacy. Alerts are then classified as either ***true positives (TPs)***, indicative of actual suspicious activity, or FPs, which correspond to false alarms or non-suspicious transactions. In instances where an alert is substantiated as a TP, a ***suspicious activity report (SAR)*** must be filed by the analysts.

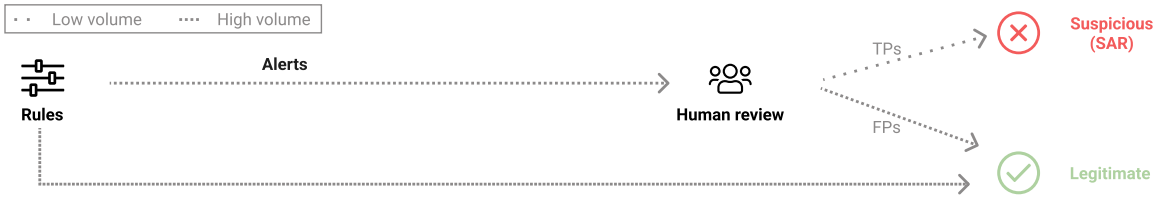


FIGURE 1.2: Overview of the traditional AML rules-based system pipeline and alert processing.

Rule-based systems are valued for their interpretability, which is essential for auditing. However, a significant limitation of these systems is their tendency to generate a high volume of FPs, as noted by Weber et al. [2018]. In fact, FPRs in such systems are reported to be as high as 95–98% [Lannoo and Parlour, 2021]. To comprehend this high rate of false positives, an understanding of the nature of money laundering is essential. Money laundering is a complex process, not a singular action. It involves a sequence of transactions strategically executed to conceal the origins of illegal funds. Furthermore, these transactions are often structured to simulate legitimate financial activities, thereby evading detection [Lorenz et al., 2020].

For a more effective identification of suspicious activities, it is imperative to perceive money laundering as a continuous flow of funds. This approach captures the essence of its ongoing, interconnected, and sometimes cyclical nature. Additionally, a comprehensive and holistic view of financial flows is required, extending beyond isolated transactions or individual accounts, to effectively understand the patterns indicative of money laundering.

To illustrate this concept, consider an example illustrated in Figure 1.3 involving a criminal attempting to disguise the source of illicit funds. The individual might execute a series of transactions, transferring money between accounts A, B, C, and D, and at times returning the funds to A from D. When each transaction is viewed in isolation within a table, the underlying suspicious pattern might be less detectable. However, representing these transactions as a flow reveals a cyclic pattern, indicative of an effort to conceal the fund's origins. This example underscores the ability to holistically understand the flow of funds within a network can provide valuable insights for identifying unusual patterns. Thus, to gain such a perspective, a shift is necessary from traditional tabular data representations to more sophisticated, network-oriented data structures like graphs.

Source	Destination
A	B
B	C
C	D
D	A

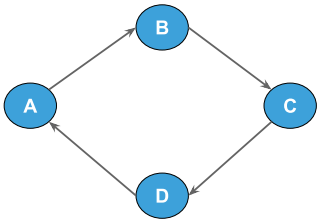


FIGURE 1.3: Comparison of cycle detection in transaction tables versus networks.

Graphs (sometimes also referred to as networks), inherently, are powerful tools for representing and analyzing interconnected systems and flows. In essence, graphs are versatile enough to depict any dataset where the significance lies in the interactions (edges) between participating entities (nodes or vertices). Once the data finds its representation in the format of a graph, numerous operations become feasible. For instance, one could detect suspicious cycles of money transfers between related accounts [Qiu et al., 2018], find the structural role of an individual within a network [Khan et al., 2010], or track money streams traversing through the network [Li et al., 2020]. Contrastingly, achieving similar insights from tabular data presents challenges. To illustrate, a graph can intuitively identify the k-hop neighbors of a node (entities distanced by k steps) through iterative edge traversal. In a tabular setup, the equivalent would necessitate join operations between tables, which tend to be computationally demanding and memory intensive.

Within the domain of ML, one central challenge when working with graphs is to represent or encode their structure, such that it can be harnessed effectively by ML algorithms. Certain strategies, particularly those under **graph feature engineering**, rely on handcrafted heuristics to

generate features that encapsulate the core characteristics of a graph, such as degree metrics (elaborated upon in Section 2.1.3). Conversely, **graph representation learning** techniques learn and encode graph structures autonomously, manifesting them in compact low-dimensional embeddings. These methods harness the advancements in deep learning and nonlinear dimensionality reduction techniques [Hamilton et al., 2017b] (elaborated upon in Section 2.3.2 and Section 3.2).

1.2 Research Contributions

In the evolving landscape of financial networks, the complex money laundering schemes have necessitated advanced techniques to monitor, trace, and flag suspicious activities. Graphs, representing interconnected transactions and accounts, provide a compelling medium to capture these complexities. Our work, hence, revolves around harnessing the potential of graph mining techniques tailored specifically for ***continuous time dynamic graphs (CTDGs)*** in the AML domain. Although our primary focus is on addressing AML challenges, it is important to highlight that our research contributions are general enough and applicable to other domains. The primary contributions of our research are outlined as follows:

1. **Creation of a framework for graph-based feature extraction:** Recognizing the significance of the insights that could be encoded in graph data, we developed the graph feature engineering framework, named *Walking-Profiles* (Section 4.2). By leveraging random-walks, this framework extracts graph-based features, that can be later used in any downstream system (e.g., ML model).

2. **Formulating a comprehensive ML pipeline for AML systems:** Addressing the issue of false alarms in AML systems, we introduce a ML-centric methodology termed the *triage model* (Section 4.3). This model processes alerts generated by pre-defined rules in AML systems. It assigns scores to these alerts, which then either facilitate the suppression of low-priority alerts or order the alert queue based on severity. An intrinsic advantage of our approach is its ability to maintain compliance and offer explainability. Since every alert originates from established rules, the process remains transparent and interpretable.

3. **Design of a real-time graph feature extraction method for CTDGs:** With the increasing volume and velocity of financial transactions, it is imperative to have techniques that are both robust and efficient. Thus, we developed *Graph-Sprints* (Section 5.2) a method

optimized for CTDGs. This method minimizes computational overheads and memory usage and makes it suitable for real-time deployment (e.g., in an AML scenario), thus addressing a limitation of *Walking-Profiles* .

4. **Design of a real-time graph representation learning method for CTDGs:** While traditional feature extraction methods often necessitate domain-specific knowledge for effective implementation, deep learning techniques can autonomously identify and learn relevant information. However, a common limitation of current deep learning techniques is their substantial latency. To address this and combine the advantages of *Graph-Sprints* and deep learning paradigms, we introduced *Deep-Graph-Sprints* (Section 6.2), a novel low-latency graph representation learning method for CTDGs.

5. **Rigorous evaluation of the proposed frameworks:** To assess the efficacy and versatility of our approaches, we undertook a comprehensive evaluation phase. We tested the performance of *Walking-Profiles* applied integrated with the *triage model* (Section 4.4), *Graph-Sprints* (Section 5.4.3), and *Deep-Graph-Sprints* (Section 6.3.3) across varied AML datasets. Further, to underscore the adaptability of our methods, we also evaluated *Graph-Sprints*, and *Deep-Graph-Sprints* on datasets from diverse domains (Sections 5.4.2, 6.3.2), demonstrating their utility beyond the AML domain.

1.3 Research Context

This research focuses on AML systems in the banking sector, specifically analyzing transactional data and entity relationships. To validate our graph mining method, we utilize internal AML datasets and publicly available datasets from different domains. Although the primary application is in the AML domain, we also aim to contribute to graph mining in general by introducing a low-latency graph feature extraction and graph representation learning algorithms.

Company Overview and Challenges

The research journey undertaken in this PhD thesis is a collaboration between Faculty of Sciences at the University of Porto and Feedzai, a leading risk prevention company. Feedzai's core product is a risk management platform that harnesses the power of ML to detect financial crime. Feedzai's clients primarily encompasses banks, payment processors, and merchants, providing them with defenses against various financial threats, ranging from illicit account

openings and transaction fraud to the primary focus of our study: money laundering. Each financial transaction processed is governed by rigorous service level agreements. For instance, in fraud detection when an account initiates a transfer or procures an online service, a critical decision - whether to approve or reject the said activity - must be rapidly executed, often within a tight timeframe of a few milliseconds. Depending on specific use-cases and client requisites, some transactions might have a processing window extending up to 200 milliseconds, especially at the 99.999^{th} percentile [Branco et al., 2020]. Throughout this document, such financial activities are collectively referred to as *transactions*.

1.4 Thesis Organization

This thesis is organized into seven principal chapters. A concise overview of each chapter is provided below:

- **Introduction:** This chapter sets the stage by providing the foundational context of the study, introducing the concept of money laundering and the typical stages employed by criminals to cleanse their illicit funds. We delve into how graph structures can adeptly capture intricate money flows and patterns. Subsequently, primary contributions of this work are enumerated. Furthermore, we articulate the scope, and provide an overview of the thesis structure. Concluding this chapter, a catalogue of published works affiliated with this research is presented.

- **Background:** This chapter establishes foundational knowledge in graph methodologies central to the thesis. It covers essential terminology, graph construction constraints, important measures, and the practical applications of graph analysis. Moreover it details the fundamentals of ML necessary for this research. The chapter then transitions to an overview of money laundering stages and traditional AML solutions,

- **Related Work:** This chapter delves deeply into the existing methodologies for extracting information from graphs, categorizing them into two primary approaches: feature engineering methods and representation learning techniques. It discusses the evolution of these methods, providing a historical and practical perspective. Also, it details the similarities and differences between these methods and our contributions, providing

a comprehensive analysis. Additionally, the chapter explores the realm of modern ML-driven AML strategies, elucidating their development and integration into current financial systems, highlighting their weaknesses.

- **Graph Feature Engineering:** This chapter introduces the concept of *Walking-Profiles*, a framework for graph feature engineering based on random-walk methodologies. It also discusses a specialized adaptation of this framework, specifically designed for the AML domain. Additionally, the chapter describes the *triage model* and evaluates its effectiveness using the AML-adapted *Walking-Profiles* framework. This evaluation is conducted using a real-world banking dataset.

- **Graph-Sprints:** This chapter presents our *Graph-Sprints* methodology, designed for real-time graph feature extraction. We detail its roots in the Random-Walk based graph feature extraction paradigm. Further, the efficacy of *Graph-Sprints* embeddings, when combined with a neural network classifier, is assessed using AML datasets and datasets from other domains, emphasizing their balance between computational efficiency and robust predictive capabilities.

- **Deep-Graph-Sprints:** This chapter introduces *Deep-Graph-Sprints*, a representation learning method that extends the *Graph-Sprints* framework using deep learning techniques. The primary objective of *Deep-Graph-Sprints* is to address the constraints inherent in the original *Graph-Sprints* method, thereby augmenting both its utility and efficiency. The efficacy of *Deep-Graph-Sprints* is empirically evaluated through its application in two tasks: node classification and link prediction. Experimental results demonstrate that *Deep-Graph-Sprints* achieves competitive performance compared to its predecessor, *Graph-Sprints*, while at the same time mitigating its previously identified limitations.

- **Conclusions and Future Work:** This concluding chapter reflects on the research journey, encapsulates the contributions, and potential avenues for subsequent investigations.

1.5 Bibliographic Note

Parts of the work of this thesis have already been published in international conferences, and workshops. A list of those is given next:

- **Triage Model:** (Section 4.3) This is in reference to our study titled "Anti-Money Laundering Alert Optimization Using Machine Learning with Graphs" and related endeavors. This research was presented at the AAAI's workshop on AI in Financial Services: Adaptiveness, Resilience & Governance [Eddin et al., 2021]. Additionally, a patent associated with this study was duly filed [Eddin et al., 2023c].

- **GuiltyWalker:** (Section 4.3.2) Our involvement here pertained to the patent of *GuiltyWalker*. Our primary contribution revolved around enhancing GuiltyWalker for circumstances where labels aren't immediately accessible [Silva et al., 2022].

- **Graph-Sprints:** (Section 5.2) The methodology we developed for real-time graph feature extraction was published at the KDD's 17th International Workshop on Mining and Learning with Graphs [Eddin et al., 2023a]. Also, an extended version of this paper that applies *Graph-Sprints* to the link prediction task was accepted at the 4th ACM International Conference on AI in Finance [Eddin et al., 2023b]. Moreover, a provisional patent about this work was submitted.

- **Deep-Graph-Sprints:** (Section 6.2) About our real-time graph representation learning methodology we are in the process of submitting a provisional patent, and after that we plan to publish a paper.

Chapter 2

Background

This chapter delves into the concepts and methodologies that serve as the foundation for this thesis. The content encompasses graph theory, graph analysis measures, and tasks. Then it delves into the ML basics necessary for this research, especially the various modes of learning in deep learning. Subsequently, it lists possible ways to integrate graph information in ML models. Then it concludes with an overview of money laundering phases and traditional AML solutions. The organization of the chapter is as follows:

- **Fundamentals of Graphs:** In Section 2.1 we discuss graph related background information, focusing on the following points:
  - **Graph concepts and terminology:** Section 2.1.1 starts with an introduction to basic graph terminologies. The narrative progresses to differentiate between various graph types, such as static versus temporal and homogeneous versus heterogeneous graphs.
  - **Graph construction considerations:** In Section 2.1.2, we outline the primary factors essential for selecting an apt graph representation tailored to distinct challenges. Then we provide a customization of these considerations in the AML use-case.
  - **Graph measures:** Section 2.1.3 explains various measures for extracting patterns and valuable information embedded within graph structures.
  - **Graph analysis tasks:** Section 2.1.4 details the graph analysis tasks important for this thesis, namely, node classification, and link prediction.

  - **Incorporating graph data into ML models:** Section 2.3 discusses two ways to extract insights from the graphs and feed them to ML models, namely, *graph feature engineering* (Section 2.3.1) and *graphs representation learning* (Section 2.3.2) techniques.

- **Fundamentals of ML:** Section 2.2 provides the fundamentals of ML necessary for this research focusing on:

  - **Classification of ML methods:** Section 2.2.1 provides a brief descriotion of the ML models that will be used throughout this thesis namely tree-based models (Section 2.2.1.1), and deep learning models 2.2.1.2.
  - **Learning mechanisms in deep learning:** Section 2.2.2 delves into the diverse learning paradigms employed in deep learning methods. Starts with an overview the various types of ***automatic differentiation (AD)*** modes (Section 2.2.2.1), then it presents an in-depth comparison of the computational and memory complexities associated with forward-mode and reverse-mode automatic differentiation (Section 2.2.2.2).

- **AML overview:** Section 2.4 provides an overview of money laundering, discussing its mechanisms and classic solutions.

  - **Phases of Money Laundering: An Overview**: Section 2.4.1 takes a closer look at the various stages involved in the illicit flow of money, offering an understanding of the sequential processes.
  - **Traditional AML Solutions:** Section 2.4.2 explores traditional AML solutions adopted by FIs. This exploration reveals the methods and practices used to address challenges in combating money laundering.

2.1 Fundamentals of Graphs

Networks serve as versatile representations for numerous systems across diverse fields, such as computer science, mathematics, biology, and chemistry [Costa et al., 2007, Febrinanto et al., 2023, Majeed and Rauf, 2020, Zhang et al., 2020b, Zhou et al., 2020]. These networks are mathematically referred to as graphs; thus, throughout this document, we will use "network" and "graph" interchangeably.

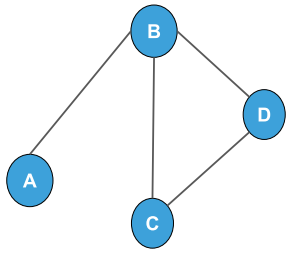
2.1.1 Graph Concepts and Terminology

A graph G consists of a set $V(G)$ of **nodes (or vertices)** and a set $E(G)$ of **edges (or links)**. While nodes denote entities, edges signify the relationships between these entities. Edges, often represented as vertex pairs (a, b) where $a, b \in V(G)$, can be categorized as directed or undirected. In **directed graphs**, an edge (i, j) illustrates a one-way relationship from i to j, but in **undirected graphs**, edges reflect a mutual relationship between the paired nodes. Both nodes and edges may possess associated types and attributes. Furthermore, the constraints governing the connections between nodes and edges, in conjunction with the presence or absence of types and attributes, as well as the temporal nature of the graph (either evolving or static), collectively determine the type of the graph. A graph **adjacency matrix** is a square matrix used to represent the connectivity of finite graph in terms of its nodes and edges. The matrix allows for the quick and concise representation and manipulation of dense graphs. The adjacency matrix is of dimensions $n \times n$, where n represents the number of nodes in the graph. In the case of an undirected graph, the matrix elements a_{ij} equal 1 if there is an edge between nodes i and j, and 0 otherwise. Due to the undirected nature of the graph, the adjacency matrix is symmetric. For directed graphs, the elements a_{ij} equal 1 if there is an arrow or edge pointing from node i to node j, and 0 otherwise. Unlike undirected graphs, the adjacency matrix for directed graphs may not be symmetric in general.

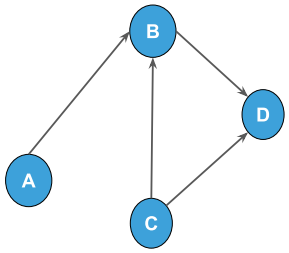
An example is provided in Figure 2.1, the left panel displays an adjacency matrix for an undirected graph, characterized by its symmetric pattern, indicative of bidirectional edges. The right panel presents an adjacency matrix for a directed graph, where the asymmetry in the matrix highlights the directionality of edges.

A **Subgraph** of G is represented as S_G, where $V(S_G) \subseteq V(G)$ and $E(S_G) \subseteq E(G)$. In simpler terms, a subgraph is a smaller graph that is formed by selecting a subset of vertices and edges from the original graph G.

G is considered a **simple graph** if it has no self-loops (i.e., it has no nodes connecting back to itself) and has no more than a single connection between any pair of nodes. We can add weights to the edges to enrich the graph – the resulting graph is called a **weighted graph**. To represent the weights in the adjacency matrix, the elements a_{ij} represent the weight of the edge between nodes i and j, instead of simply being 1 or 0. In the context of a social network, for instance, friendships can be aptly represented using an undirected graph. Within this model, individual



	A	B	C	D
A	0	1	0	0
B	1	0	1	1
C	0	1	0	1
D	0	1	1	0



FROM \ TO	A	B	C	D
A	0	1	0	0
B	0	0	0	1
C	0	1	0	1
D	0	0	0	0

FIGURE 2.1: Example of adjacency matrices for undirected and directed graphs.

users are conceptualized as nodes, while the friendships between them constitute the edges. The weight of these edges might be indicative of the duration of the relationship, thereby giving greater importance to longstanding friendships compared to more recent ones.

When analyzing systems with a stable topology and a focus on global graph statistics, **static graphs** prove to be appropriate as they depict a singular, unchanging state of the system. However, for systems that exhibit evolution, as exemplified by the continuous formation of new friendships and potential inclusion of new accounts in a social network, it becomes imperative to study time-dependent patterns. These evolving structures are referred to as **temporal** or **dynamic graphs**. Dynamic graphs can be categorized into two primary models: ***discrete time dynamic draphs (DTDGs)*** and the continuous counterparts, **CTDGs** [Rossi et al., 2020]. In the DTDG paradigm, the graph is conceptualized as a series of snapshots taken at specified time intervals. For example, one might capture a snapshot daily, which encompasses only the data of that day or all historical data up to that specific day. Within each individual snapshot, the graph remains static. Conversely, under the CTDG framework, the graph is viewed as an ongoing stream of events. An event could, for instance, constitute the addition of a new edge to the graph, leading to the graph's continuous evolution each time an event occurs. Typically,

DTDG is seen as an approximation of CTDG, primarily because handling static graphs might be computationally more efficient than dealing with their dynamic counterparts.

Graphs can be differentiated based on the distinctions in node and edge types. Within the realm of **homogeneous graphs**, there is a simplicity wherein all nodes pertain to a single category, and similarly, all edges represent just one type of relationship. For instance, in a basic social network, if we only consider users and their mutual friendships, a homogeneous graph suffices.

On the other hand, real-world networks often present complex interactions and relationships. Taking the example of a social network further, one can recognize a rich diversity of interactions. Here, entities include not just users, but also posts, comments, groups, and even events. Relationships also diversify: users author posts, like and comment on them, join groups, or attend events. **Heterogeneous graphs** are used to represent such rich systems. In heterogeneous graphs, both nodes and edges come in various types, each capturing a different facet of the overall network, thereby providing a more comprehensive view of the interactions and relationships within. Das and Soylu [2023] present a complete review of higher dimensionality graphs, including multilayer networks, multiplex graphs, colored graphs, and multipartite graphs.

This thesis concentrates specifically on CTDGs. In our context, CTDGs are defined by homogenous nodes, directed and timestamped edges, and the capability for both nodes and edges to carry attributes. The construction process of these graphs is elaborated in Section 2.1.2.2, where we detail the methodology employed to create our graph structures. These choices are motivated by the requirements of the AML use-case, where we need to process transactions on a streaming fashion, thus aligning with the CTDG framework. Additionally, for simplicity, we opt for homogeneous graphs, where nodes can represent accounts or clients—a suitable starting point for AML.

2.1.2 Graph Construction: Key Decisions and Their Impact

Graph construction is a pivotal process, significantly impacting the derivation of insights from data. There are many possible ways to represent a real-life system using graphs, representing different viewing angles and none is fundamentally correct or incorrect. The structural decisions regarding nodes, edges, their attributes, and overall graph architecture are instrumental in

effectively representing the inherent relationships and dynamics within datasets. This section discusses these considerations before focusing on their specific application in the AML context.

2.1.2.1 General Considerations in Graph Construction

Graph construction involves important decisions that fundamentally shape its analytical efficacy:

- **Node Selection:** Identifying entities in the dataset for node representation is fundamental to the graph's structure, influencing its ability to accurately model data relationships.
- **Edge Definition:** The choice between directed and undirected edges determines the nature of relational dynamics represented in the graph, affecting its interpretability and analytical value.
- **Attribute Allocation:** Assigning attributes to nodes and edges enriches the graph with multidimensional data aspects, enhancing its descriptive power.
- **Graph Dynamics:** Opting for a static or dynamic graph affects its capacity to capture and represent temporal changes, a decision that carries significant implications for data analysis over time.
- **Complexity Management:** Balancing detail with computational feasibility is crucial for maintaining both the graph's representational accuracy and practical usability.

These core decisions immensely affect the graph's potential to accurately and effectively model data relationships, thus influencing the depth and quality of insights gained.

2.1.2.2 Graph Construction in AML: A Specific Case Study

In AML domain, graph construction is a key tool for revealing complex financial networks and detecting suspicious patterns. Table 2.1 provides a detailed exposition of how general considerations are applied specifically within the AML domain.

Complementing this, Figure 2.2 visually demonstrates the transformation of tabular transaction data into a graph format, using a straightforward example. This transformation is pivotal in facilitating the effective analysis of complex transactional relationships and patterns indicative of money laundering. This section exemplifies an application of graph construction principles

Consideration	Application in AML Graph
Nodes	Clients or accounts are represented as nodes, each symbolizing a distinct entity in the financial network.
Edges	Directed edges are utilized to represent the directional flow of money, crucial for analyzing financial interactions.
Edge Timestamps	Timestamps on edges provide a chronological dimension, essential for temporal analysis in AML investigations.
Edge Weights	Weights on edges quantify amounts of money being transferred, which is crucial information for the investigation.
Graph Dynamics	A dynamic graph model is adopted, evolving with new transactions to accurately depict the current state of financial activities.

TABLE 2.1: Application of graph construction principles for AML domain.

in a specialized research context, highlighting the importance of a domain knowledge informed design towards extracting meaningful information for the AML domain.

Transaction ID	Sender	Receiver	Time stamp	Amount
1	A	D	1	20
2	B	E	2	30
3	C	D	3	25
4	E	A	4	40
5	B	D	5	50
6	C	A	10	45

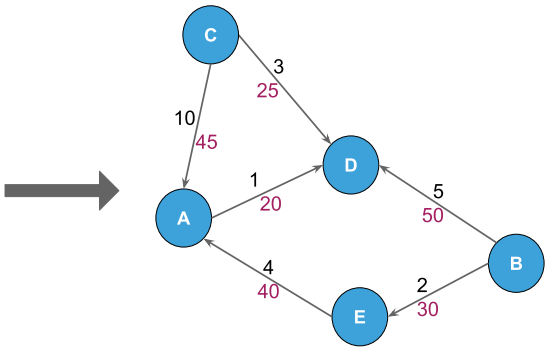


FIGURE 2.2: Transformation of transaction data into graph representation.

2.1.3 Graph Measures

A useful approach to categorizing graph mining techniques involves considering measures at the *node-level*, *subgraph-level*, and *graph-level* [Ribeiro, 2011].

2.1.3.1 Node-Level Measures

Node-Level Measures are designed to extract information specific to individual entities within a network, namely the nodes. These measure include node centrality measures or other measures like clustering coefficient. These measures are crucial for understanding the roles and significance of each node in the network. Although a detailed exploration of node measures is beyond the scope of this section, readers interested in an in-depth exploration are referred

to [Das et al., 2018]. For the interest of this thesis we will explain one node centrality measure, namely, the node degree.

Node degree is a fundamental metric in graph theory, quantifying the number of connections a node has. In an undirected graph, it is mathematically defined as:

$$\text{degree}(v) = \sum_{(u,v)\in E} e_{u,v} \tag{2.1}$$

where E denotes the set of edges in the graph, and $e_{u,v}$ represents an edge between nodes u and v. An illustrative example of node degree is presented in Figure 2.1. In the left panel of the figure, for instance, node **B** exhibits a degree of 3. The degree can be computed by summing the elements in the corresponding row or column of the adjacency matrix. Due to the matrix's symmetric nature, both row and column summations yield the same result.

In the context of directed graphs, where edges have a directional attribute, the concepts of 'in-degree' and 'out-degree' are introduced. 'In-degree' pertains to the count of incoming connections to a node, whereas 'out-degree' pertains to the count of outgoing connections. These are defined as follows:

$$\text{in-degree}(v) = \sum_{(u,v)\in E} e_{u,v} \tag{2.2}$$

$$\text{out-degree}(v) = \sum_{(v,u)\in E} e_{v,u} \tag{2.3}$$

Here, E represents the set of edges in the graph, with $e_{u,v}$ signifying an edge from node u to v.

An exemplification of in-degree and out-degree is also visible in Figure 2.1. In the right panel, node **B** has an in-degree of 2, which can be determined by summing the elements in the corresponding column in the adjacency matrix. Moreover, the out-degree of node **B**, which is 1, can be calculated by summing the elements in its respective row.

While individual metrics such as the node degree offer insight into the specific characteristics of a node, they can also be employed more broadly to portray either the entire graph or a designated neighborhood. This can be achieved by computing aggregations across the

neighborhood, yielding metrics like the average or minimum degree. Extending this idea, one can derive the distribution of a given feature (for instance, the degree distribution) across a particular neighborhood.

2.1.3.2 Subgraph-Level Measures

Subgraph-Level metrics decompose the network into its constituent sub-components, commonly termed as subgraphs. Instead of covering the entire graph or highlighting singular nodes, these measures adopt an intermediary position on the analytical spectrum. Examples encompass subgraph counting.

Subgraph Counting refers to computing the number of occurrences of subgraphs S_G within a graph G. This process necessitates a vertex-to-vertex mapping and edge-to-edge mapping as well, to preserve the structural integrity of the graphs in question. Figure 2.3 illustrates the concept: the top left panel shows a specific subgraph, the top right panel presents the main graph for identification, and the bottom panels depict four distinct occurrences of the subgraph within G, highlighted in green. It's important to observe that ABC and CDE are not considered occurrences because they have an extra edge, making them structurally different from the target subgraph.

The significance of subgraph counting extends beyond just counting; it plays a critical role in analyzing and deciphering the local topological characteristics of complex networks. This method is essential in a variety of research domains and practical applications. For instance, in graph alignment, as described by Milenković et al. [2010], and in the context of graph comparison, as explored in the work of Milo et al. [2004]. These diverse applications emphasize the versatility and analytical effectiveness of subgraph counting in the study of network structures.

Nonetheless, subgraph-level measures introduce high computational complexities. Counting subgraphs is a challenging task known to be NP-Complete. So, in practical situations, this counting is mainly done for simpler graphs that are unweighted, static, and homogeneous. More advanced techniques for complex graphs are detailed in [Kovanen et al., 2011, Ribeiro and Silva, 2014]. Typically, counting subgraphs is manageable when they are relatively small, usually containing fewer than ten nodes. For a deeper dive into subgraph counting methods, we refer the reader to the survey by Ribeiro et al. [2021].

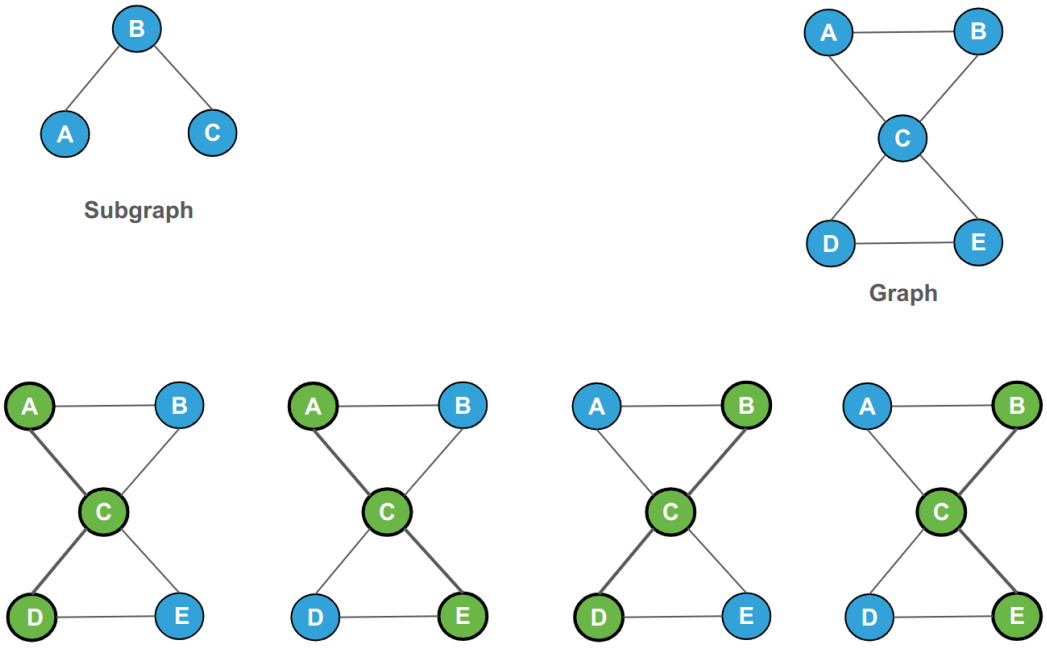


FIGURE 2.3: Demonstration of subgraph counting within a main graph.

2.1.3.3 Graph-Level Measures

Graph-Level Measures give a big picture view of the entire graph. In this thesis, we're more interested in features that describe the neighborhood or community surrounding a node of interest rather than the whole graph. In this context, we will cover basic graph-level features that offer valuable insights into our community-level analysis.

Number of Nodes and Edges function as fundamental indicators of the graph's size.

Density quantifies the proximity of a graph to being a complete (fully connected) structure. In directed graphs, density is determined using the formula:

$$D = \frac{m}{n \times (n-1)} \tag{2.4}$$

Where D stands for the density, m is the number of edges, and n denotes the number of nodes.

For undirected graphs, a slight modification is necessary due to the symmetry of such graphs. In this case, the maximum number of possible edges is halved, resulting in the density formula:

$$D = \frac{m}{\frac{n \times (n-1)}{2}} \tag{2.5}$$

For a more in-depth exploration of additional measures we recommend the review by [Costa et al., 2007].

2.1.4 Graph Analysis Tasks

This section will discuss the graph analysis tasks that were used in this thesis, namely: *node classification*, and *link prediction*.

Node Classification seeks to allocate *classes* to nodes within a given network [Bhagat et al., 2011]. In applied contexts, this term *class* can be interpreted as the *role* a node assumes within a network. As exemplified in *Struc2vec* [Ribeiro et al., 2017], the authors aim to distinguish the structural role of each node in the graph via unsupervised methodologies. Within the realm of AML, the processes can be seen as a node classification task. Specifically, given the knowledge that a particular individual is engaged in money laundering, one might investigate whether there are other individuals in the graph with similar topological characteristics. Then check whether these similar individuals are involves in money laundering as well.

Link Prediction, in its essence, attempts to forecast either absent edges or potential future edge formations within a graph [Backstrom and Leskovec, 2011]. This prediction paradigm stands central to recommender systems, wherein the objective is to recommend new product suggestions to users [Lu et al., 2015]. Further applications of link prediction encompass tasks such as the completion of knowledge graphs [Wang et al., 2020], aiming to supplement sparse knowledge bases embodied as graphs, or identifying subject-matter experts and fostering collaborations within academic or social networks [Tang et al., 2015, Wang et al., 2015].

For additional graph analysis tasks, such as community detection and subgraph embedding, we recommend consulting the survey by Makarov et al. [2021]. While tasks beyond node classification and link prediction are not the primary focus of this thesis, they represent potential future directions for applying the methods developed in this research.

2.2 Fundamentals of ML

ML explores the development of systems that enhance their performance autonomously with experience. It is a fast growing area that combines computer science and statistics, and forms the backbone of artificial intelligence and data science [Jordan and Mitchell, 2015].

2.2.1 Classification of ML Algorithms

ML algorithms essentially search for the optimal program, guided by training data, to maximize performance. These algorithms can be categorized based on how they represent candidate solutions (e.g., decision trees, neural networks), and the type of training data used.

Based on the type of training data and the use-case, key types include *supervised learning*, where the model learns from labeled examples to predict labels for new data points; *unsupervised learning*, which involves grouping or clustering similar data points without labels; and *reinforcement learning*, where a model learns to make decisions to maximize rewards based on example action sequences.

This thesis will primarily focus on supervised learning, particularly tree-based models and neural networks. For a broader perspective on ML algorithms, we refer readers to the book by Bishop [2006], and the review by Mahesh [2020].

2.2.1.1 Tree-Based Models in ML

Tree-based models, like decision trees [Bishop, 2006], random forests [Ho, 1995], and gradient boosting machines like LightGBM [Ke et al., 2017], are fundamental in machine learning. They operate using a tree-like structure, breaking complex decisions into tree-shaped decision-making processes. Decision trees, for instance, split data on specific criteria, creating branches that show different decision paths. Random forests enhance decision trees by combining multiple trees for a more robust decision. Gradient boosting machines build trees in a sequence, with each new tree fixing errors from the previous ones.

Tree-based models often outperform neural networks in medium-sized datasets and require substantially less computational resources [Grinsztajn et al., 2022]. Additionally, the simplicity of tree-based models makes them more interpretable compared to complex neural networks.

In this thesis, we will use tree-based methods within our *triage models*, as detailed in Chapter 4.

2.2.1.2 Neural Networks Models in ML

Neural networks, inspired by the human nervous system, form the basis of deep learning. They use neurons or perceptrons as fundamental units to process information. Each neuron simulates the behavior of a biological nerve cell in the human brain.

A classic example of deep learning is the multilayer perceptron, which is a mathematical function mapping inputs to outputs. This function is formed by composing many simpler functions (layers). Each neuron represents a mathematical function, contributing to the overall input-to-output transformation. Each layer in the network is defined by a weight matrix and a bias vector, whose values are learned during training, and an activation function. This learning process is guided by the network's hyperparameters, such as the learning rate.

These models, known as feedforward networks, facilitate a unidirectional flow of information from the input layer, through intermediate layers, to the output, without any feedback loops where outputs recursively influence the model itself. The inclusion of feedback connections transforms these into ***recurrent neural networks (RNNs)***. Diverse deep learning architectures exist, each with specific advantages depending on the task or data type. For example, convolutional neural networks tend to excel in processing images and videos [Xu et al., 2014]. In contrast, for sequential data like text and speech, RNNs often surpass other models in performance [LeCun et al., 2015]. This research predominantly examines RNN-like structures, as transactional data involves sequences (e.g., transactions by card) are instrumental in detecting suspicious activities both conceptually and empirically [Branco et al., 2020].

In RNNs, each element of the input sequence possesses a state vector, encapsulating historical data from previous elements, when expanded temporally, RNNs resemble deep feedforward networks with shared weights across layers.

Graph neural networks (GNNs) represent another class of neural networks, specifically tailored to operate on graph structures. Figure 2.4 contrasts the input sequences of RNNs and GNNs: single-entity sequences (over different timestamps) versus multiple-entity in a graph where the k-hop neighbors of a target node are sampled and their information is aggregated and sent to the target node.

While a comprehensive exploration of deep learning is beyond this section, essential concepts, especially learning mechanisms, are discussed in Section 2.2.2. These foundations are crucial for the development of our *Deep-Graph-Sprints* in Chapter 6.

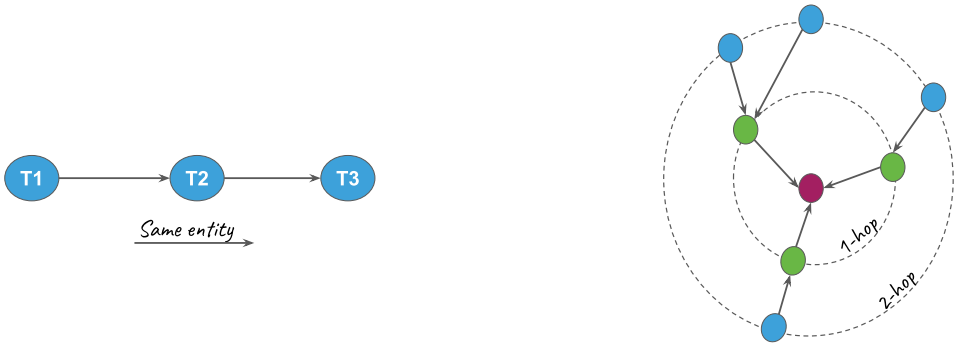


FIGURE 2.4: Simplified comparison of RNNs and GNNs.

For the curious reader, we recommend the book by Goodfellow et al. [2016] for more in-depth understanding of deep learning.

2.2.1.3 Performance Measures

Accurate evaluation of ML model performance is critical for model comparison and selection. The choice of the appropriate measure is dependent on the use case and data characteristics. For instance, in cases of unbalanced datasets, traditional accuracy - the ratio of correctly labeled instances to all instances - may be misleading. A model that only predicts the majority class could appear highly accurate.

In the context of financial crime detection, such as AML or fraud detection, it is essential to accurately identify criminal activities while avoiding the disruption of legitimate transactions, which could result in client dissatisfaction or substantial financial losses for banks or online merchants. The primary goal is to maximize TPs while minimizing FPs. This is often achieved by optimizing recall (the proportion of actual positives correctly identified) at a specific FPR, such as recall@1%FPR. However, different clients and scenarios may require the use of alternative metrics.

A prevalent metric in the context of unbalanced data is the F1 score, which combines recall and precision. The F1 score is calculated as follows: $F1\ score = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$.

Another critical measure is the ***area under the curve (AUC)*** of the receiver operating characteristics (ROC), represents the probability that the model correctly distinguishes between a positive and a negative example. A higher AUC indicates better model performance. It provides

a consolidated measure of performance across different classification thresholds, making it valuable for comparing different models.

For link prediction tasks, where we don't have class imbalance issue, we employ the ***average precision (AP)*** metric, which evaluate how well a classification model ranks positive examples higher than negative ones, AP measures the model's ability to correctly prioritize true positive cases over the entire range of its predictions. Moreover, we measure the mean reciprocal rank (**MRR**), which indicates the average rank of the positive edge. An MRR of 50% implies that the correct edge was ranked second, while an MRR of 25% implies it was ranked third. Additionally, we measure Recall@10, which represents the percentage of actual positive edges ranked in the top 10 scores for every edge.

These metrics provide a comprehensive framework for assessing the performance of ML models used in this thesis.

2.2.2 Learning Mechanisms in Deep Learning

ML algorithms depend significantly on credit assignment, a process identifying the impact of past actions on learning signals [Minsky, 1961, Sutton, 1984]. This process is essential for reinforcing successful behaviors and reducing unsuccessful ones. A deep understanding of a model's internal structure eases this task by directly linking its decisions to underlying parameters.

2.2.2.1 Overview of Automatic Differentiation (AD) Modes

The capability of assigning credit in deep learning models hinges on the differentiability of learning signals enabling the use of derivatives for this purpose [Cooijmans and Martens, 2019]. A key technique in this context is AD, used for computing derivatives in functions represented as computer programs [Baydin et al., 2018].

In AD, depending on the direction of applying the chain rule, three strategies stand out: forward mode, reverse mode (often termed backpropagation), and mixed mode. Forward mode entails multiplying the derivatives matrices from input to output. Reverse mode, a two-phase process, first executes the function to populate intermediate variables and map dependencies, then calculates derivatives in reverse order from outputs to inputs [Baydin et al., 2018]. Mixed mode combines these approaches.

In the context of this thesis we are interested in temporal models, such as RNNs, and GNN algorithms designed for temporal graphs.

AD, applicable in both RNNs and GNNs, automates the derivative calculation of model parameters. Here, we focus on RNNs to illustrate AD's principles, although these concepts are also applicable and more complex to illustrate in GNNs.

Backpropagation in GNNs or RNNs requires a forward pass for network evaluation and a backward pass for gradient computation. Complex structures like large graphs or extended sequences pose challenges, leading to techniques like truncated back propagation through time (TBPTT) [Williams and Peng, 1990]. TBPTT eliminates the need for a complete retrace through the whole data sequence at each stage thus offers computational benefits over full backpropagation. However, TBPTT struggles with long-term dependencies since the parameter updates are computed using a limited horizon in time.

An alternative, ***real-time recurrent learning (RTRL)*** (i.e., forward propagation of the gradient), facilitates online parameter updates and allows networks having recurrent connections to learn complex tasks requiring the retention of information over time periods having either fixed or indefinite length [Williams and Zipser, 1989]. However, its practicality is limited in large networks due to high computational demands. More specifically, it must retain a large matrix relating the model's internal state to its parameters. Even when this matrix can be stored at all, updating it is very expensive [Cooijmans and Martens, 2019], further details are discussed in Section 2.2.2.2.

To overcome these limitations, approximations such as UORO [Tallec and Ollivier, 2017] and KF-RTRL [Mujika et al., 2018] have been proposed. These methods aim to balance between RTRL's theoretical strengths and the practical constraints of network size and resource demands.

2.2.2.2 Computational and Memory Complexity of AD modes

This section compares the computational and memory complexities in reverse-mode and forward-mode AD within the context of neural networks. The explanation is based on an illustrative example, as explained below.

Consider a function L, represented in Equation 2.6, and illustrated in Figure 2.5, encapsulating a neural network comprising four distinct functions f, g, h, p. These functions may

be interpreted as individual layers in a feedforward network or as a sequence of events in a recurrent network configuration.

$$L(x) = p(h(g(f(x)))) \tag{2.6}$$

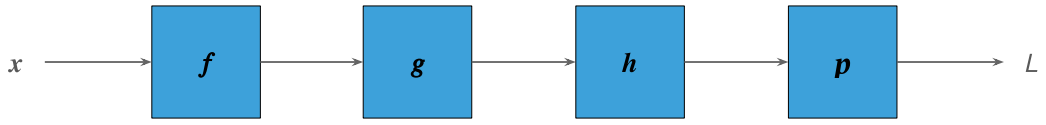


FIGURE 2.5: Representation of a composite function comprising four distinct functions.

For this example, we assume the following dimensional specifications:

- 3 layers f, g, and h, each characterized by a weight matrix of dimensions $(d \times d)$.
- A final layer p, doing a dimensional reduction from d to a scalar value, characterized by a weight matrix of dimensions $(d \times 1)$.

Considering the derivative of the error or loss $L(x)$:

$$\frac{dL}{dx} = \frac{\partial f}{\partial x} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial h}{\partial g} \cdot \frac{\partial p}{\partial h} \tag{2.7}$$

In **reverse-mode AD**, the sequential application of the chain rule, commencing from the output layer towards the input, is depicted in Equation 2.8.

$$\frac{dL}{dx} = \left[\frac{\partial f}{\partial x} \cdot \left[\frac{\partial g}{\partial f} \cdot \left[\frac{\partial h}{\partial g} \cdot \frac{\partial p}{\partial h}\right]\right]\right] \tag{2.8}$$

Conversely, **forward-mode AD** initiates the derivative computation from the input layer, as illustrated in Equation 2.9.

$$\frac{dL}{dx} = \left[\left[\left[\frac{\partial f}{\partial x} \cdot \frac{\partial g}{\partial f}\right] \cdot \frac{\partial h}{\partial g}\right] \cdot \frac{\partial p}{\partial h}\right] \tag{2.9}$$

Computational Complexity:

The associative property of matrix multiplication does not extend to computational complexity, which is significantly influenced by the order of operations.

In **reverse-mode AD**, matrix multiplication operations are as follows:

- Initial operation: $(d \times d) \cdot (d \times 1)$, resulting in $(d \times 1)$ vector with d^2 multiplications.
- Each subsequent operation also necessitates d^2 multiplications.
- Cumulatively: $3d^2$ multiplications, for our example of a sequence with length 4. Thus, given a sequence of length l then the total number of multiplications is $(l-1)d^2$. Thus the total computational complexity is $\mathcal{O}(l \times d^2)$.

For **forward-mode AD**, the operations entail:

- First operation: $(d \times d) \cdot (d \times d)$, culminating in $(d \times d)$ with d^3 multiplications.
- Second operation: Similarly, d^3 multiplications.
- Final operation: $(d \times d) \cdot (d \times 1)$, totaling d^2 multiplications.
- Overall: $2d^3 + d^2$ multiplications, for our example of a sequence with length 4. Thus, given a sequence of length l then the total number is $(l-2)d^3 + d^2$ multiplications. Thus the total computational complexity is $\mathcal{O}(l \times d^3)$.

This analysis demonstrates the considerable disparity in computational demands between reverse-mode and forward-mode AD in the context of a typical ML architecture with scalar loss.

By employing a practical parameter value of $d = 100$, it is evident that reverse-mode AD requires significantly fewer operations $(30,000)$ compared to forward-mode AD $(2,010,000)$. This discrepancy is even more pronounced when considering longer sequences or deeper models. For instance, with a sequence length $l = 100$ and $d = 100$, the computational load for reverse-mode AD amounts to $990,000$ operations, whereas forward-mode AD necessitates a staggering $98,010,000$ operations. This reveals that, in this specific example, the number of operations needed for reverse-mode AD is approximately two orders of magnitude less than that required for forward-mode AD, and this gap will increase with higher values of d.

Consequently, in prevalent neural network architectures within ML, which typically evolve from larger initial layers to more compact (often scalar) output layers, reverse-mode AD emerges

as the more efficient approach. This efficiency gain is critical in practical applications, and explains the popularity of reverse-mode AD in ML platforms.

Memory Complexity:

Reverse-mode AD involves a two-pass approach through the computational graph: an initial forward pass that evaluates the function L, and a subsequent backward pass that computes the gradient $\frac{dL}{dx}$, as outlined in Equation 2.7. During the forward pass, it is crucial to store intermediate outputs at each step, commonly known as activations. This is because the gradient of the loss with respect to the input of a layer is dependent on the output of that layer, making these intermediate values essential for the backward pass. The storage requirement scales with the number of parameters per layer and the depth of the network. In our specific example, this equates to storing three matrices of dimensions $(d \times d)$ and one vector of dimensions $(d \times 1)$. Thus the total memory complexity is $\mathcal{O}(l \times d^2)$.

Conversely, **forward-mode AD** does not need to retain all intermediate computational values. It primarily preserves the derivatives from the most recent computations, potentially reducing memory demands, particularly in networks with greater depth. For our example, the memory requisite at each step would be a matrix of derivatives having dimensions $(d \times d)$, and a vector of dimensions $(d \times 1)$ in the final step. Thus the total memory complexity is $\mathcal{O}(d^2)$.

Thus in our ongoing example since $l = 3$, then forward-mode AD requires around three times less memory than reverse-mode AD. This disparity in memory efficiency becomes more noticeable in models with increased l, meaning in models with more layers or depth. For sequence-based models, such as RNNs, l represents the sequence length, like how many transactions a specific account has made. Therefore, forward-mode AD, which goes through the computational graph in just one pass, offers enhanced memory efficiency.

2.3 Incorporating Graph Data into ML Models

While ML has seen many successes, a key challenge arises in integrating non-tabular, graph data into these models. Traditional ML methods are mainly designed for tabular data, consisting of lists of feature vectors. The question then becomes: How can graph data be effectively integrated into ML models?

A straightforward approach is using the graph's adjacency matrix, where each row represents a node's connections. However, this approach, focusing solely on direct neighbors, can miss broader structural patterns in the graph. Additionally, for large graphs, this method might be too demanding computationally for the ML model.

As shown in Figure 2.6, the objective is to map nodes from high-dimensional, complex graph structures to a lower-dimensional space. This dimensionality reduction, crucial for analyzing complex network dynamics, can be achieved through graph feature engineering techniques or via node representation learning methodologies, as discussed below:

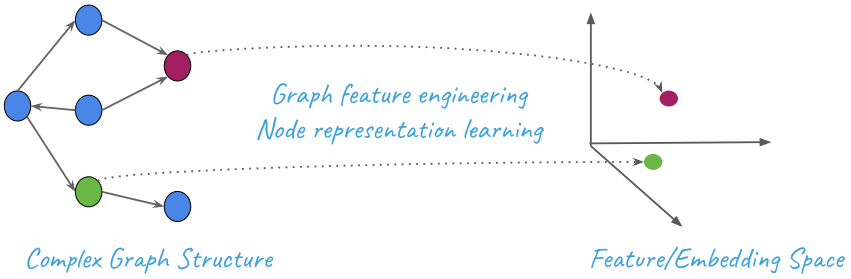


FIGURE 2.6: Mapping nodes from high-dimensional graphs to lower-dimensional space.

2.3.1 Graph Feature Engineering

Feature engineering uses graph metrics like node degree to enrich ML models with informative graph-based features, as discussed in Section 2.1.3. By incorporating these metrics into feature vectors, additional information is provided to the ML model (or rule-based systems). However, selecting the most effective metrics remains a challenge, especially considering the computational cost for large graphs. Feature engineering approaches, discussed in Section 3.1, aim to extract features from tabular or graph data manually or automatically, enabling ML models to leverage graph relationships for enhanced performance and insights. Although these approaches usually involve manual work and/or domain knowledge, they tend to be interpretable as each bin in the resulting vector has a specific meaning.

2.3.2 Graph Representation Learning

Graph representation learning focuses on transforming graphs or their components (nodes and edges) into vector spaces. This transformation simplifies capturing essential graph features, to be used in tasks such as node classification and link prediction. The graph information represented in vector forms can be seamlessly integrated with traditional ML methods.

Graph representation learning techniques can be categorized into matrix factorizations, random-walk-based methods, and deep learning methods, each offering unique features. Further details on these methods can be found in Section 3.2.

Unlike feature engineering, graph representation learning automatically derives complex feature representations from the data itself, to capture patterns and relationships within the graph structure. Thus, it requires less manual work and domain knowledge. However, the resulting representation might be less interpretable.

2.4 Fundamentals of Money Laundering

In this section we explain the phases of laundering money and the traditional solutions employed to detect it.

2.4.1 Phases of Money Laundering: An Overview

Money laundering is a carefully designed process aimed at disguising the origins of assets obtained from illegal activities, thereby allowing them to appear as though they stem from legitimate sources. As was illustrated in Figure 1.1, the procedure can be divided into three fundamental stages:

1. **Placement:** This is the initial stage where people or groups bring in illegally obtained money into the financial system. This money often comes from activities like drug trafficking, human trafficking, or financial scams and is usually put into the system through a variety of methods like cash deposits or wire transfers. By doing this, they start the process of making the money seem legitimate.

2. **Layering:** Following the placement of assets, the subsequent challenge involves concealing their illicit origins. This is achieved by engaging them in a complex web of financial activities, including transfers between different bank accounts, transactions across various banks or FIs, and even cross-border movements. The objective is to create a convoluted network of transactions, making it progressively difficult to trace these assets back to their initial illegal source.

3. **Integration:** This is the concluding stage, where the 'laundered' assets are carefully reintroduced into the legitimate economy. At this point, they are often used to invest

in legal business ventures, purchase assets, or fund lifestyles, all without immediately alerting authorities due to their now seemingly legitimate appearance.

2.4.2 Traditional AML Solutions

The most common systems to detect money laundering based on transaction data employed by banks are rule-based [Weber et al., 2018]. Such AML systems consist of a series of rules that trigger alerts based on specific transactional behaviors, as illustrated in Figure 1.2. Following the generation of these alerts, domain experts evaluate them to determine if they indicate suspicious activities or are false alarms. If deemed suspicious, a SAR is submitted to the relevant regulatory authority. However, it is noteworthy that these systems often exhibit high FPRs, with some studies indicating rates as high as 95–98% [Lannoo and Parlour, 2021]. This not only burdens resources but also demands significant time investment from domain experts. While these rules are essential as they are required by regulatory bodies to ensure compliance, their inherent rigidity can result in detection gaps. To address these constraints and enhance detection precision, modern AML solutions are increasingly integrating ML to identify complex patterns, enhancing their overall performance.

Later in Section 3.3 we detail advancements in AML systems that leverage ML systems.

Chapter 3

Related Work

This chapter evaluates existing methodologies in graph information extraction and machine-learning-driven strategies in AML, with an emphasis on their evolution, relevance to our work, and the gaps that our research aims to fill. The discussion navigates through feature engineering, representation learning techniques, and their applications in AML. This chapter is organized as follows:

- **Evolution of Feature Engineering:** Section 3.1 delves into the realm of automatic feature engineering. It is divided into two subsections: approaches for tabular data in Section 3.1.1, and methodologies specific to graph data in Section 3.1.2.
- **Graph Representation Learning:** In Section 3.2 we detail the various methodologies of graph representation learning.
- **Advancements in AML Strategies:** Section 3.3 describes advanced AML solutions. It encompasses ML-based solutions in Section 3.3.1, as well as those utilizing graph technologies in Section 3.3.2.

3.1 Evolution of Feature Engineering

Feature engineering, a crucial step in building well performing machine learning pipelines, has seen efforts to transition from manual feature crafting to more automated methods, this shift has greatly influenced our approach to graph data in AML.

3.1.1 Feature Engineering for Tabular Data

Feature engineering in tabular data has evolved significantly, incorporating a variety of advanced techniques across several key technological fronts.

Explorekit [Katz et al., 2016] generates meta-features for both individual candidate features and the target dataset as a whole. Examples of these meta-features include general statistics like the number of classes and the AUC of models using each feature. A subsequent feature selection step is applied to retain only those features that enhance the overall task performance. Despite challenges in scalability, such approaches have set the stage for more sophisticated feature transformation processes. This is exemplified in the learning feature engineering (*LFE*) framework [Nargesian et al., 2017], which employs meta-features derived from previous tasks to recommend new meta-feature without relying on model evaluation. The general nature of these meta-features, which do not require domain-specific knowledge, provides valuable insights for our *Walking-Profiles* framework in deciding which features to generate.

The integration of reinforcement learning and optimization techniques in automated feature engineering is a notable trend. Learning automatic feature engineering machine (*LAFEM*) [Zhang et al., 2019] is one example. Reinforcement learning is similarly employed in the *SAFE* method [Shi et al., 2020] and *mCAFE* [Huang et al., 2022] for feature optimization. The *CAFEM* (cross-data automatic feature engineering machine) [Zhang et al., 2020a] further underscores the potential of these techniques in enhancing feature engineering processes. Aligning with *LFE*'s approach, *CAFEM* accelerates feature engineering by applying generalized feature engineering strategies learned from diverse datasets.

Driven by the insight that meta-features overlook feature interrelationships and that generating all possible features from existing ones leads to a feature explosion, *BigFeat* [Eldeeb et al., 2022] takes a more focused approach. It evaluates the significance of existing features using tree-based models, assigning higher weights to more important features for the creation of new features. Additionally, *BigFeat* assigns importance scores to the operators themselves. The feature generation process is controlled by a predefined time or resource budget, ensuring efficiency and manageability.

While the aforementioned techniques tackle tabular data and thus are not directly applicable to our graph datasets, they offer valuable insights into the necessity of computational and space efficiency in feature engineering. Thus, we propose specific techniques to address these

challenges in *Walking-Profiles* (Chapter 4) and *Graph-Sprints* (Chapter 5).

Furthermore, domain-specific solutions have demonstrated feature engineering's adaptability. For example, the framework by Marques et al. [2020] in the financial crime domain creates features using semantic data labels. This underscores the importance of incorporating domain expertise in algorithm development, inspiring the AML-specific customization in our *Walking-Profiles* (Section 4.3.2).

For practitioners seeking an open-source Python library for automated feature engineering and selection, *AutoFeat* [Horn et al., 2020] serves as an example tool.

Overall, these advancements, while not directly integrated into our research, underscore the importance of automated feature engineering. They also provide inspiration regarding existing issues to consider in the context of automatic feature generation.

3.1.2 Feature Engineering for Graph Data

The field of feature engineering for graph data presents a fertile ground for research endeavors, as underscored by Escalante [2021].

There have been developments enhancing automation for graph data. The automatic feature engineering machine (*AFEM*) by Zhang et al. [2018] focuses on automating feature engineering for relational and graph datasets. This approach utilizes a range of feature families, such as social graph-based features. Similar to our method, *AFEM* aims to capture both the global and local aspects of networks. However, their technique computes features independently, using different methods, in contrast to our *Walking-Profiles* that relies on the same random-walks to derive all features.

Zheng [2018] focus on network functional blocks, such as paths and subgraph-augmented paths, for graph feature engineering. Their methods enhance semantic proximity search in heterogeneous graphs, revealing the applicability of these techniques in diverse graph-based applications. In our *Walking-Profiles*, a similar concept is employed as subgraph-based measures, as described in Section 4.2.1.2. This approach is inspired by the concept of *anonymous walks*.

Anonymous walks, were used by Ivanov and Burnaev [2018], to generate feature-based network embeddings. An *anonymous walk* essentially transforms a random walk by replacing each node's id with the position of its first appearance in the walk, as shown in Figure 3.1. This

technique is important to *Walking-Profiles*, as it is utilized to generate features at the subgraph level, further elaborated in Section 4.2.1.2.

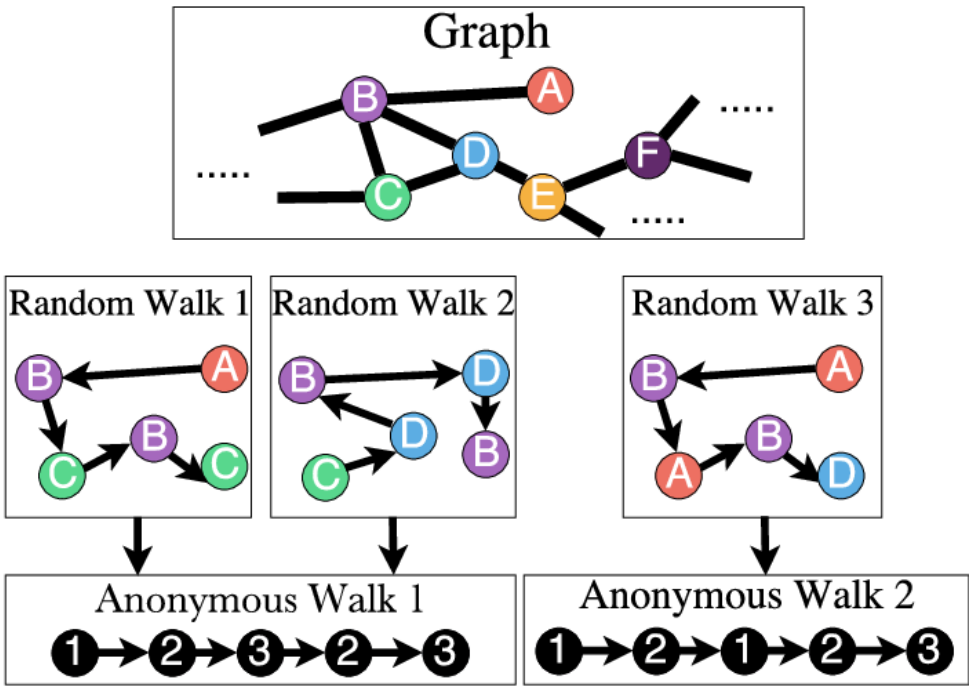


FIGURE 3.1: An example of the concept of anonymous walks [Ivanov and Burnaev, 2018].

Shirbisheh [2022] introduce a local-to-global strategy in graph ML. This method develops local features, such as betweenness centrality, and vector representations of graph nodes, using the neighbors degree frequency (*NDF*) for mapping nodes into euclidean vector spaces. This approach allows for a nuanced understanding of a node's neighborhood, facilitating neighborhood analysis in graph structures. Similarly, in our *Walking-Profiles* and *Graph-Sprints*, we utilize node degrees as a key feature to capture the structural aspects of the graph.

Concurrently, Wu et al. [2022] have developed the *AFGSL* model, which integrates deep learning and reinforcement learning techniques. This model converts tabular data into a graph-structured format and introduces a feature interaction layer based on graph structure learning. Additionally, it uses reinforcement learning to refine the graph structure. The incorporation of deep learning in *AFGSL* may impact its interpretability, aligning it more closely with representation learning methods as discussed in Section 3.2. Nonetheless, the strategy of modifying the graph structure to enhance performance resonates with our focus on the significance of selecting an effective structure for representing tabular data.

These developments in graph data feature engineering reflect an evolution towards more automated and advanced processes, aligning with our research objectives. Utilizing these

insights, we aim to develop an automated graph feature engineering framework tailored to address specific challenges in AML.

To provide an application example, the field of semantic-based image retrieval demonstrates the utilization of graph feature engineering, as detailed by Nhi et al. [2022]. This technique emphasizes extracting key features from complex data structures, paralleling the methodologies employed in our AML research.

3.1.3 Connecting Tabular and Graph Data Methods

The advancements in both tabular and graph data feature engineering exemplify the industry's shift towards more automated, intelligent systems capable of handling complex data structures. This evolution from manual to automated processes underpins our research approach, where we integrate these insights to develop our general purpose graph feature engineering frameworks and customize them into the specific context of AML domain.

3.2 Graph Representation Learning

Graph representation learning is crucial in transforming complex graph structures into usable formats for ML models. This section discusses the evolution from matrix factorization techniques, like *GraRep* and *HOPE*, to more sophisticated deep learning-driven approaches. While not all of these methods are directly used in our work, understanding their limitations and strengths has informed the development of our representation learning approach, namely, our *Deep-Graph-Sprints* method, detailed in Chapter 6.

The domain of graph representation learning is at the forefront of innovative research. For an in-depth exploration of existing methods and their applications, readers are directed to the book by Hamilton [2020], and the survey by Makarov et al. [2021].

3.2.1 Matrix Factorization-Based Techniques for Graph Representation

Matrix factorization-based approaches, sometimes referred to as graph factorization [Ahmed et al., 2013], provide foundational methods in the realm of node representation learning, drawing their inspiration from traditional dimensionality reduction techniques [Hamilton et al., 2017b].

At its core, matrix factorization decomposes a matrix into a product of several constituent matrices, facilitating the representation of data in a reduced dimensional space [Makarov et al., 2021]. In graph representation learning, the target matrix frequently corresponds to the graph's adjacency matrix or its derivatives. The main objective is to represent the underlying graph structure in this compact form.

Key algorithms in this domain include *GraRep* and *HOPE*. The *GraRep* algorithm [Cao et al., 2015] focuses on high-order graph proximities, utilizing successive powers of the adjacency matrix to reveal node relationship patterns. In contrast, *HOPE* [Ou et al., 2016] employs broader similarity metrics like Jaccard neighborhood overlaps and uses singular value decomposition to preserve asymmetric transitivity, especially in directed graphs. This method optimizes computational efficiency by retaining only the dominant eigenvalues.

However, the limitations of matrix factorization methods, such as their transductive nature requiring retraining for new nodes and computational overheads for large graphs, can restrict their applicability in dynamic graph scenarios.

For a comprehensive understanding of matrix factorization in graph representation, the survey by Chen et al. [2020] provides an in-depth analysis.

3.2.2 Random-walk Based Techniques

A *random-walk*, in the context of graph theory, is a stochastic process that derives random paths originating from a designated node. By conducting *numerous* such walks, one can approximate the inherent structure of the network. Notably, if two nodes exhibit similar random-walks, it might suggest they occupy analogous structural roles in the network.

In light of this observation, several methods proposed random-walk based approaches to generate node embeddings like, *DeepWalk* [Perozzi et al., 2014], *LINE* [Tang et al., 2015], *Node2vec* [Grover and Leskovec, 2016], *Struc2vec* [Ribeiro et al., 2017], *Metapath2vec* [Dong et al., 2017], *Role2vec* [Ahmed et al., 2019]. In essence, nodes that participated in comparable random-walks yield similar embeddings. Traditional random-walks are genuinely stochastic, choosing a random neighbor to progress the walk at each hop. *Node2vec*, however, introduces a degree of bias, using two parameters, to direct the walks to be either shallower or deeper, therefore distinguishing community structures or node roles more effectively. It is crucial to note that both *DeepWalk* and *Node2vec* primarily utilize topological data for their node embedding

generation. *Role2vec* leverages attributed random-walks to learn embeddings for each role within the graph based on functions that map feature vectors to roles. Thus, instead of learning individual embeddings for each node, embeddings are learned for each role. Limitations of these methods include their ignorance of node and edge features (except *Role2vec* that leverages node attributes), and since they are designed for static graphs they disregard temporal information. Their inherent high latency is attributable to the necessity of executing numerous walks to derive an embedding. These limitations make them not applicable in our AML scenario. In a subsequent advancement, Sajjad et al. [2019] adapted these random-walk techniques to DTDG. While this adaptation introduced a degree of efficiency, it remains constrained in its applicability, especially for CTDGs and in low-latency situations. In contrast, Node2bits [Jin et al., 2019] integrates time-related details by separating the random walks it samples into different time windows. It combines node attributes into histograms during these times. *Node2bits* closely aligns with our *Walking-Profiles*, yet there are key differences. It does not account for edge features and is designed for entity stitching rather than AML, as in our work. Unlike *Walking-Profiles*, which produce interpretable aggregated features, *Node2bits* creates binary representations. Additionally, *Node2bits* tackles space efficiency with binary hashing, whereas *Walking-Profiles* employs selective feature extraction and sliding window techniques, as detailed in Section 4.2.2. In contrast to our *Graph-Sprints*, all these methods, including *Node2bits*, demand considerable computational resources for executing walks, making them less suitable for scenarios requiring low latency.

Further developments, such as continuous-time dynamic node embeddings (*CTDNE*) [Lee et al., 2020, Nguyen et al., 2018], were introduced to provide time-aware embeddings, augmenting the *Node2vec* paradigm for CTDGs. These methodologies treat the graph as a stream of edges and perform temporal walks from seed nodes chosen through a temporally-biased distribution. Hyperbolic spaces have seen the application of temporal random-walks for embedding extraction [Wang et al., 2021a].

Moreover, the anonymous walks approach [Wang et al., 2021c] employs causal anonymized walks to encode motif-centric data (motifs refer to subgraphs that occur within a real network at a frequency higher than what statistical probability would typically predict [Ribeiro et al., 2021]). In parallel, *NeurTWs* [Jin et al., 2022b] integrate time into the anonymous walks via Neural Ordinary Differential Equations (NeuralODEs). It is essential to note that these methods, similarly to our proposed *walking-profile* and in contrast to the our proposed *graph-sprints* framework, necessitate exhaustive random-walk executions.

Recently Vital Jr et al. [2023] investigated the performance of different random-walks in the context of link prediction in static graphs, focusing on embedding information derived from various node sequence generations. Four traditional random-walks and five *Node2vec* configurations were examined across 37 networks. The findings showed minimal performance differences among the random-walks for link prediction, with a mere 3-4% difference in median AUC metrics. Exploratory walks outperformed local ones in terms of performance. Furthermore, a strong positive correlation was observed between node similarities from different walks, even for opposing walk types. The study emphasizes that different random-walks have similar performances in link prediction and capture consistent node similarity information.

A major limitation of random-walk-based approaches is their time complexity. To address efficiency, methods like *B_LIN* [Tong et al., 2006] have been developed. This method improves efficiency by leveraging two prevalent characteristics found in many real-world graphs, namely, linear correlations and community-like structures.

Apart from time complexity, random-walks also impose significant demands on main memory volume. This issue is particularly relevant as many high-speed random-walk algorithms assume the entire graph fits within the main memory, a challenge highlighted by Xia et al. [2019]. To address this, various strategies have been developed for graph partitioning and clustering. Notable among these are *METIS* [Karypis and Kumar, 1997], and *RWDISK* [Sarkar and Moore, 2010], which offer practical solutions for managing large-scale graphs efficiently. In *Walking-Profiles*, we address this memory challenge by introducing two distributed implementations of our method, and a sliding window approach as detailed in Section 4.2.2.

Random-walks have applications in domains such as collaborative filtering, recommender systems, computer vision, and more. For a comprehensive review of random-walk-based applications and the challenges they pose, the reader is directed to [Xia et al., 2019].

3.2.3 *K*-hop Neighborhood Based Methods

More recently, we see a surge in deep learning algorithms for graph representation learning. Several approaches leverage GNNs to learn functions that generate node embeddings [Hamilton et al., 2017a, Yang et al., 2020, Ying et al., 2018, Zhu et al., 2020].

Most GNN-based methods require a k-hop neighborhood on which message-passing operations lead to node embeddings. To deal with CTDGs, a simple approach is to consider

a series of discrete snapshots of the graph over time, on which static methods are applied. Such approaches however do not take time properly into account and several works propose techniques to alleviate this issue [Goyal et al., 2018, Jin et al., 2022a, Sankar et al., 2020]. To better deal with CTDGs, other works focus on including time-aware features or inductive biases into the architecture. *DeepCoevolve* [Dai et al., 2016] and *Jodie* [Kumar et al., 2019] train two RNNs for bipartite graphs, one for each node type. Importantly, the previous hidden state of one RNN is also added as an input to the other RNN. In this way, the two RNNs interact, in essence performing single-hop graph aggregations. *TGAT* [Xu et al., 2020] proposes to include temporal information in the form of time encodings, while *TGN* [Rossi et al., 2020] extends this framework and also includes a memory module taking the form of an RNN. In [Jin et al., 2020], the authors replace the discrete-time recurrent network of *TGN* with a *NeuralODE* modeling the continuous dynamics of node embeddings.

APAN [Wang et al., 2021b] proposes to reduce the latency at inference time by decoupling the more costly graph operations from the inference module. The authors propose a more light-weight inference module that computes the predictions based on a node's embedding as well as the messages recently received from interacting nodes, stored in the node's "mailbox". The mailbox is updated *asynchronously*, i.e. separated from the inference module, and involves the more expensive k-hop message passing. While *APAN* improves the latency at inference time, it sacrifices some memory since each node's state is now expanded with a mailbox, and more importantly it potentially uses outdated information at inference time due to asynchronous update of this mailbox. This algorithm addresses the need to generate low-latency embeddings, similar to our *Graph-Sprints* and *Deep-Graph-Sprints* methods. However, unlike our methods, *APAN* uses outdated information which could negatively affect its overall performance.

Moreover, towards reducing computational costs of GNNs, *HashGNN* [Wu et al., 2021] leverages MinHash (an algorithm used to efficiently estimate the similarity between sets, by hashing their elements into a smaller representative set of hash values) to generate node embeddings suitable for the link prediction task, where nodes that results in the same hashed embedding are considered similar. *SGSketch* [Yang et al., 2022] is a streaming node embedding framework uses a mechanism to gradually forget outdated edges, achieving significant speedups. Differently than our approach *SGSketch* uses the gradual forgetting strategy to update the adjacency matrix and therefore only considers the graph structure.

Liu et al. [2019] propose an algorithm for graph streams that performs node representation

updates in real-time by: 1) Identifying nodes influenced by newly added nodes (e.g., one-hop neighbors); 2) Generating embeddings for new nodes through linear summation of influenced nodes' embeddings; 3) Adjusting the embeddings of these influenced nodes. Therefore generating approximated embeddings in low latency. However, the embeddings depend only on the neighbors embeddings and ignoring the target vertex attributes.

GNNs are instrumental in analyzing vast time series data, but adapting them to large datasets is challenging due to memory constraints. While various sampling strategies exist, merging them with temporal data remains complex. Enhancing GNN's scalability for real-time applications is a critical research area Jin et al. [2023]. For a comprehensive review on GNN-based approaches for time series analysis we refer the reader to the survey by Jin et al. [2023].

To provide an application example in the realm of graph-based methodologies, these techniques are used in areas like entity alignment in knowledge graphs and cybersecurity. CG-MuAlign, utilizing node embeddings for entity alignment, exemplifies the efficiency of graph convolutional networks (a type of GNNs) in this domain [Zhu et al., 2020]. In cybersecurity, the importance of graph representations is evident in detecting complex network intrusions and anomalies, as recent developments show [Ahmetoglu and Das, 2022]. These instances demonstrate the capability of graph data in encapsulating complex relationships and their broad applicability. In the area of recommendation systems, graph embeddings enhance user experience, with Pinterest's PinSage being a notable application [Ying et al., 2018]. These systems harness graph data for understanding and forecasting user preferences, indicating the extensive application of graph-based methods.

3.3 Advancements in AML Strategies

In Section 2.4, we elaborated on the traditional rule-based systems employed for AML. These systems, while ensuring compliance and interpretability, are plagued by exceedingly high FPRs. The primary objective of our research is to address and mitigate these FPs, all the while preserving the rule-based structure to maintain compliance and provide clear explanations for the system's decisions.

Contrastingly, a considerable portion of the recent research efforts, which incorporate ML, opt to replace these rule-based systems entirely. Such approaches aim to serve a dual purpose:

reducing both FPs and false negatives. For an exhaustive review of ML-centric methodologies tailored for AML, readers can consult [Chen et al., 2018, Tiwari et al., 2020].

3.3.1 AML Solutions Leveraging ML

We can divide the ML-based AML systems into unsupervised and supervised methods. The majority use unsupervised techniques due to the lack of real-world labeled datasets available in the money laundering domain. The typical approach is to firstly cluster events, followed by anomaly detection. To address the lack of data, various strategies have been proposed. Either a fully synthetic dataset is generated [Dreżewski et al., 2012, Luna et al., 2018], or only unusual accounts are simulated within a real-world dataset [Gao, 2009, Liu et al., 2008, Tang and Yin, 2005, Wang and Dong, 2009], or one assumes that rare events within a peer group are suspicious [Larik and Haider, 2011]. One drawback of anomaly detection approaches is the assumption that suspicious activities are outliers, which may not always be the case since money launderers try to simulate legitimate behavior [Lorenz et al., 2020]. Arguably, better validations of the systems were reported in [Camino et al., 2017, Shokry et al., 2020, Yang et al., 2014] using analyst feedback, or in [Liu and Zhang, 2010] using real labeled data and where authors report a 52% recall@5%FPR.

Several approaches leverage *supervised learning*, for instance, Luo [2014] generates synthetic data and proposes a classification algorithm based on association rules to detect suspicious events. Other researchers use real-world datasets and aim to detect suspicious behavior by training classification algorithms like SVM [Keyan and Tingting, 2011] where authors report 64% recall@6%FPR, XGBoost [Jullum et al., 2020] obtaining an AUC of 82%, or after comparing various algorithms [Zhang and Trubey, 2019] in which the best model was a neural network and obtained 74% AUC. The performances of various models are hard to compare across the studies due to their different metrics and datasets.

3.3.2 ML-Enhanced AML Solutions Using Graphs

Recent work has tried to incorporate graph information in the AML system in order to capture network patterns. Weber et al. [2019] benchmarked graph convolutional networks against various supervised methods and concluded that random forest algorithms provide a better performance, despite the lack of graph-based information. Oliveira et al. [Oliveira et al., 2021]

propose *GuiltyWalker*, leveraging random-walks on a cryptocurrency graph to characterize distances to previous suspicious activity. The authors reported a 5 percentage points improvement in F1 score when including these novel features. We leveraged the *GuiltyWalker* method in our research, generalizing it for cases with label delay and incorporating it into our *Walking-Profiles* framework, as discussed in Section 4.3.2

Random-walks were also used in [Hu et al., 2019] on top of a transaction graph representing the bitcoin network. Savage et al. [2016] propose a community detection approach, from which neighborhood-centric features are extracted and ingested by a supervised ML model. On a real-world dataset, the best model was a random forest classifier achieving over 80%recall@20%FPR.

Other works propose graph-based suspiciousness scores based on money flows [Li et al., 2020, Sun et al., 2021]. These algorithms do not use a learning algorithm and instead build a detection system incorporating business knowledge about money flows. The scope is to detect novel types of money laundering activity (i.e., reducing false negatives), while our goal is to reduce incorrectly alerted events (i.e., reducing FPs).

Dreżewski et al. [2015] employed graphs to enhance the analysis of financial flows in preventing money laundering activities. Through the creation and examination of social networks derived from bank statements and the national court register, their system categorized roles within these networks, identified interconnections, and employed visual representations of the graph for analytical ease. The system's node role assignment within the graph was determined by the node's graph feature values, such as *PageRank* [Brin and Page, 1998].

In a different approach, Savage et al. [2016] developed an automated mechanism that identified money laundering activities by scrutinizing group behavior within transaction networks. This method integrates network analysis with supervised learning, utilizing both SVM and random forest algorithms. By focusing on smaller interacting groups exhibiting suspicious collective behavior, their model harnessed a plethora of demographic, network-centric, transaction-related, and temporal features crafted from domain expert insights. When tested on real-world data from the Australian transaction reports and analysis centre, this model achieved an average AUC of 92%. However, certain limitations, such as potential information leakage due to the non-temporal formation of communities, raise concerns regarding system compliance.

Considering the AML process as described in Chapter 1, an intuitive technique is to search

for cyclical patterns within a graph, recognizing that money launderers typically aim to retrieve money they've channeled through the network. While Alibaba group's approach of detecting graph cycles [Qiu et al., 2018] primarily targets fraud, it has potential applications for AML. Nonetheless, sophisticated criminals employing varied identities could easily elude simple cycle detection. The efficacy of this technique might be enhanced by integrating entity resolution to pinpoint genuine entities, followed by cycle detection in the resultant enriched graph.

Further, Wagner [2019] leveraged the *DeepWalk* algorithm [Perozzi et al., 2014] to translate bank transaction network graphs into latent vector representations for suspicious activity detection linked to money laundering. Despite achieving an average AUC of 78.9% on data from a German bank, the *DeepWalk* approach overlooks temporal aspects and specific attributes, necessitating periodic retraining, thus challenging its production viability.

Lastly, *Diga* [Li et al., 2023] presented a probabilistic diffusion model tailored for graph anomaly detection, targeting the AML domain within banking. The *Diga* model's AML inference system utilizes a subgraph sampler centered around a node, facilitated by a biased k-hop *PageRank*. Post introducing Gaussian noise to the subgraph, it undergoes denoising through a guiding classifier paired with a denoising network. Anomalies emerge from contrasting the reconstruction discrepancies between the original and denoised subgraphs. Key takeaways from this work encompass the efficacy of subgraph-level recovery on vast, sparse transaction graphs, the merits of semi-supervised methods like *Diga*'s guided diffusion in amplifying performance, and the significance of weight-sharing in conditioned graph generation to sidestep extended training durations.

Chapter 4

Walking-Profiles: A Framework for Graph Feature Engineering

Integrating graphs with ML enhances our understanding of entity interactions, enhancing the accuracy and robustness of classification models by considering both individual and collective behaviors. This chapter delves into the *Walking-Profiles* framework, a graph feature extraction approach using random-walks, provides a customization of these features to the AML domain, and discusses how graph features are engineered and transmitted to ML models. The chapter is structured as follows:

- **Motivation:** Section 4.1 explains the reasons for developing the methodologies presented in this chapter, highlighting the gaps they address in existing literature.
- **Walking-Profiles method:** Section 4.2 offers an overview of the random-walk based graph feature extraction framework. It also details our graph feature calculator, *Walking-Profiles* (Section 4.2.1).
- **Scalability techniques of Walking-Profiles:** Section 4.2.2 introduces three strategies specifically designed to enhance the scalability of computations and minimize memory requirements for the *Walking-Profiles* framework, particularly when dealing with large-scale datasets.
- **Triage model: Customizing Walking-Profiles to the AML domain:** In Section 4.3, an adaptation of the *Walking-Profiles* framework to the AML domain is presented. This section

introduces the *triage model*, an ML-based pipeline designed to decrease FPs in traditional AML systems. The model incorporates customized *Walking-Profiles* features to enhance its effectiveness in the specific context of AML.

- **Results: Evaluating the Triage Model:** Section 4.4 presents the research findings, emphasizing the effectiveness of the *triage model*. The model, which incorporates customized *Walking-Profiles* features with a LightGBM [Ke et al., 2017] classifier, successfully reduces FPs within the AML domain. Additionally, the section outlines the methodology employed to explain *triage model* predictions at two distinct levels, as detailed in Section 4.4.7.

4.1 Motivation

As discussed in Chapters 1 and 2, graphs are a key tool for understanding interconnectedness, especially useful in the AML domain. In Chapter 3, we reviewed how information is currently extracted from graphs. However, for AML, where interpretability is important, existing methods for feature engineering have gaps. They either do not fully support the temporal dynamics of the graph, or they do not take into consideration the presence of features in both nodes and edges, or they are hard to interpret. To tackle these issues, we propose a new framework, *Walking-Profiles*. This solution is designed to automatically generate graph-based features from temporal networks, filling the gaps in current methods. Additionally, we propose a tailored customization of this framework for the AML domain, developed in collaboration with domain experts. We also introduce the *triage model* to integrate machine learning and graphs into the rule-based AML solutions. The following section will provide an in-depth exploration of the *Walking-Profiles* framework, offering detailed insights into its structure and functionality.

4.2 Method

Walking-Profiles, our graph feature engineering framework, can be conceptualized as a modular component, adaptable to any data workflow seeking to integrate neighborhood insights into its decision-making process.

Upon receiving fresh data, the graph feature engineering component produces features that encapsulate the neighborhood attributes of a certain node or edge in its graph representation. This enriched neighborhood data subsequently augments the efficacy of the decision system,

which may either be an ML model or a rule-based configuration. Refer to Figure 4.1 for an example of incorporating the graph feature engineering component into a financial transaction data processing workflow.

The graph feature engineering framework requires data to be in a graph format. Therefore, transforming data from a tabular to a graph format is essential when it is not already in the desired structure. This section details the two primary steps involved:

1. **Graph Construction**: This step is necessary only when the input data is in a tabular, or other non-graph format. Its role is to convert this data into a graph format, which is required for the graph feature extraction algorithm, *Walking-Profiles*. For construction guidelines and considerations we refer the reader to Section 2.1.2. If the data is initially in a graph format, this component is not needed and can be skipped.
2. **Walking-Profiles**: The *Walking-Profiles* algorithm is employed to extract relevant features from the graph-represented data. It extracts important neighborhood information from the target node, providing insights for the decision-making system.

This approach highlights the framework's ability to handle different data representations, ensuring effective processing regardless of the data's initial format.

4.2.1 Walking-Profiles: A Random-walk-based Feature Extraction Engine

The feature extraction engine, as outlined in Algorithm 1 and Figure 4.2, processes data structured as a graph where nodes represent entities and edges their relationships. The engine follows a systematic approach to feature extraction, detailed through the following steps:

1. **Target Node(s) Selection:** Depending on the use-case, specific target node(s) are chosen as the starting point for the random-walks. For instance, entities with recent activity or entities of a particular interest might be selected.
2. **Random-walks:** This stage involves conducting random-walks starting from the selected target node(s). These random-walks explore the graph by traversing edges. During these walks, both node and edge features that are encountered are **collected** (the specific features to be gathered, denoted as $\mathcal{F}$, are hyperparameters of the method). The random-walks allow us to capture the context and relationships of the target node(s) within the graph

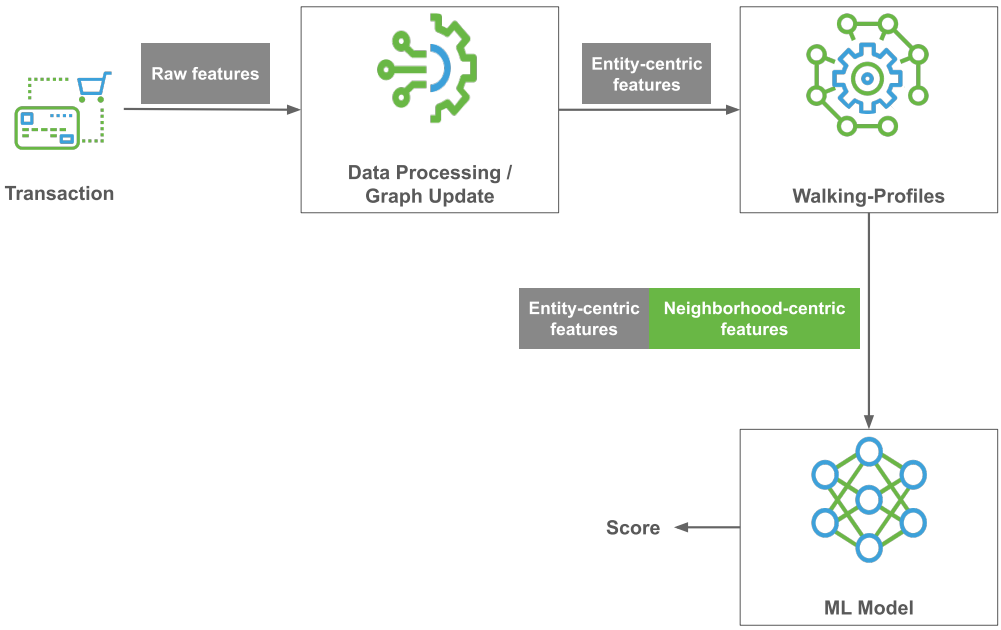


FIGURE 4.1: Simplified illustration of the *Walking-Profiles* component within transactional data processing pipeline.

structure. In our feature calculator, various types of temporal random-walks are supported. The first type (purely temporal) restricts the next explored edge in a walk to have an older timestamp than the previous edge, similar to the approach presented in Node2bits [Jin et al., 2019]. The second type of supported random-walks restricts the next explored edge to have an older timestamp than the initial node's timestamp, but not necessarily older than the previous edge's timestamp. To prevent leakage, edges that lie in the future of the random-walk starting node time cannot be traversed. The graph itself can be either undirected or directed, containing different types of nodes and edges. Depending on the use-case, random-walks can follow edge-direction or be biased by node types, edge types, node and edge features, or constrained by other factors, such as time.

3. **Summarization:** The data collected during the random-walks is then summarized into a set of relevant graph features. This summarization process processes the information gathered during the random-walks into interpretable and informative graph features that describe the characteristics of the target node(s) and their neighborhoods. The summarization involves computing aggregations of the encountered features within the walks, effectively capturing the key characteristics and patterns present in the neighborhoods of the target node(s).

Algorithm 1 *Walking-Profiles*: Random-walk based graph feature extraction engine

Require: G ▷ Updated Graph data
Require: t_nodes ▷ Target nodes
Require: K ▷ Depth of random-walks
Require: N ▷ Number of random-walks per target node
Require: $\mathcal{F}$ ▷ Node features set (e.g., degree, client_age)

for each vertex v in t_nodes **do**
  Initialize feature storage $\text{StoredFeats}_v = \{\}$
  for $w = 1$ to N **do**
    $current\_vertex = v$
    for $l = 1$ to K **do**
      Choose a random neighbor u of $current\_vertex$
      Collect features $\mathcal{F}_u$ into StoredFeats_v
      Update $current\_vertex = u$
    end for
  end for

  $\vec{S}_v \leftarrow$ Summarize features in StoredFeats_v ▷ Summarization Phase
end for

The summarization phase, plays an important role when characterizing a node's neighborhood, our engine focuses on understanding both the topology of the neighborhood (i.e., how the relations are structured) and the distributions of node or edge features across the neighborhood. The extracted graph features are categorized based on the following granularities:

- **Node/Edge level:** Data of node or edge properties traversed during random-walks starting from a certain node are collected. These features provide insights into the local properties of the target node(s) and the immediate relationships it shares with its neighbors. Further details are provided in Section 4.2.1.1.

- **Subgraph level:** Data characterizing each individual random-walk starting from a certain node is collected. This level of features gives a broader perspective of the graph by capturing the paths and patterns observed during each random-walk, offering the ability to capture specific node/edge-sequence patterns. Section 4.2.1.2 offers more detailed insights.

- **Community level:** Data characterizing the collection of random-walks starting from a certain node is gathered. These features aim to provide a higher-level abstraction of the graph's structure in proximity to the target node(s). More details can be found in Section 4.2.1.3

Figure 4.2 presents a clear example, demonstrating the *Walking-Profiles* framework in action. Given a graph and a target node, labeled A, two random walks are executed. In this simplified illustration, we focus on collecting the out-degree node feature, and the node ID. Subsequently, during the summarization phase, we calculate three levels of graph features to encapsulate the information gathered from these walks.

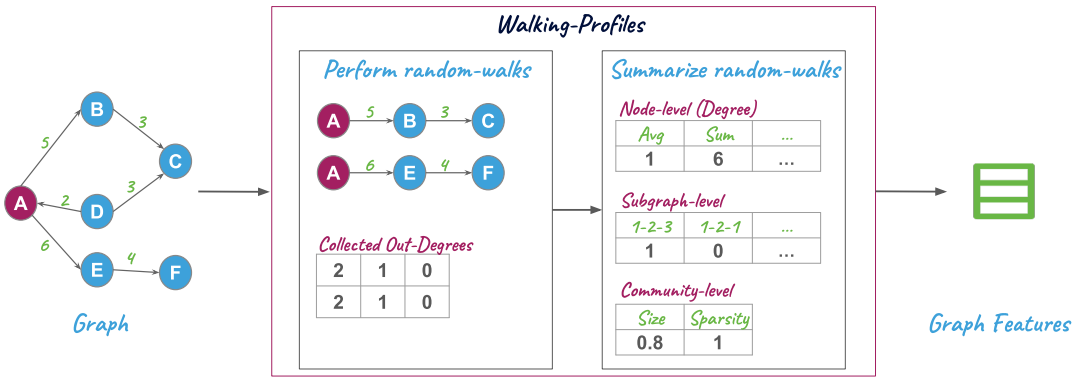


FIGURE 4.2: Illustrative example of the *Walking-Profiles* framework in action.

In the following sections, we describe in detail the extraction process of such features, and how they can be customized based on the application domain and specific use-cases. Additionally, a practical example specifically tailored to the AML context will be detailed in Section 4.3.2.

4.2.1.1 Node or Edge Level Features

Our engine is designed to differentiate between numerical and categorical attributes of nodes and edges within a graph. Based on these characteristics, it computes a variety of features. As we navigate the graph, we accumulate the values of both numerical and categorical target features. For numerical features, in particular, we take an additional step of calculating the value differences between each consecutive pair of nodes or edges.

During each random-walk, which extends up to K hops, we collect a series of values for each feature of interest, resulting in a list that can contain as many as K values. Upon completing all N random-walks, we compile a matrix with dimensions $N \times K$, representing the values for each feature of interest. It is important to note that a random-walk might terminate before reaching K hops if it encounters a node without outgoing edges. In these cases, we leave the feature values for the untraversed hops empty. This strategy is employed to prevent these partial sequences from biasing the results of later aggregation functions.

Subsequently, we characterize the distribution of these collected values using various aggregations methods. These aggregations include a range of statistical measures, such as maximum, minimum, mean, standard deviation, sum, percentiles, among others. The selection of aggregation functions acts as hyperparameters to the algorithm, providing flexibility and adaptability to the model based on the specific dataset and objectives.

One can either summarize the collected values by performing aggregations over the union of values encountered over all walks or perform aggregations on the individual walks first followed by a second aggregation over the summarized walks.

Topological information on this level can be obtained considering node degrees (or in-degree and out-degree) as numerical node properties. Entity or relationship information on this level can be obtained by considering features that are relevant to the use-case, for example, in financial data one can consider account age and transaction amount.

4.2.1.2 Subgraph Level Features

Our engine focuses on using random-walks associated with categorical features to describe distinct subgraphs and analyze the local regions around the nodes of interest. To utilize this functionality, one or more target categorical node or edge attributes must be specified. As we perform the random-walk, the system records the encountered values of the target attribute. Leveraging the idea of the anonymous walk [Ivanov and Burnaev, 2018] (illustrated in Figure 3.1), the engine represents the sequence of encountered categories (node or edge features) in an anonymous pattern. Subsequently, we count the occurrences of each pattern in random-walks starting from a specific node, and compute the ratio of occurrences to the total number of walks to generate new features.

For instance, if the node identifier is the target categorical feature and we have a set of random-walks starting at node A, such as $A - B - C$, $A - B - A$, and $A - C - D$, the corresponding anonymous walks are $1 - 2 - 3$, $1 - 2 - 1$, and $1 - 2 - 3$. We notice that the first and third random-walks correspond to the same anonymous walk, resulting in ratios of $1 - 2 - 3 : 0.66$, and $1 - 2 - 1 : 0.33$.

Due to the potentially large number of anonymous walks (subgraph patterns), we can define a reduced set of common patterns during a warm-up period or based on business knowledge. This approach enables us to efficiently compute the ratio features described above.

Subgraph-level features offer valuable insights into the topological and semantic characteristics of the neighborhood, depending on the chosen categorical feature. For instance, considering the node ID as the categorical node feature allows us to extract topological information, while analyzing the transaction currency as the categorical edge property in a financial dataset provides relevant relationship information.

Overall, our engine enables a comprehensive analysis of categorical features in random-walks, providing valuable information about subgraphs and their associations with nodes of interest.

4.2.1.3 Community Level Features

Community-level features offer valuable insights into the local regions around the target node, based on a chosen categorical node property. The process involves two main steps:

Step 1: Community Formation: Given a certain categorical node property, we define the community of the target node by the different values of that property encountered in all the walks starting from this node. To ensure community relevance, we consider only values that occur within the walks at least x times. For example, if the node property is node ID and $x = 1$, all nodes occurring in the walks will be considered part of the community of the target node.

Step 2: Data Collection and Feature Calculation: Community-level features require collecting the chosen categorical node property of the traversed nodes during the walks. After all the random-walks, we union the values (e.g., node IDs) that occurred, and then filter out values that appear less than x times.

Our *Walking-Profiles* engine calculates two community-level features:

- **Community Size:** This feature represents the number of distinct values normalized by the total number of distinct possible values in the graph. For instance, if the categorical property is the node ID, the community size is the number of distinct nodes encountered in the walks normalized by the total number of nodes in the graph.
- **Community Sparsity:** This feature is the ratio between the number of distinct values and the total number of encountered values in the walks. In the node ID example, the numerator and denominator represent the number of distinct nodes and the total number

of nodes encountered in the walks, respectively. A higher sparsity indicates nodes visited only once, while a lower sparsity suggests repeated visits to the same nodes.

Community-level features allow us to capture both topological and semantic neighborhood information. For topological insights, we can use properties like node ID as the categorical property. In social network scenarios, considering profession as the categorical node property helps understand the homogeneity of the neighborhood around a specific node.

4.2.2 Scalable Walking-Profiles for Large-Scale Data Processing

Real-world graphs can be massive, exceeding the memory capacity of a single machine. Consequently, scalable distributed implementations are required to efficiently perform random-walks on such large graphs. The independence of random-walks allows for parallelization, enabling us to propose two distinct approaches for calculating graph features in a distributed manner, and a sliding window technique.

4.2.2.1 Message-passing Approach

To compute the Random-walk-based features (Section 4.2.1), we propose using a message-passing paradigm [Attiya and Welch, 2004]. In this approach, each target node sends a message to its neighbors, conveying its information. Subsequently, the neighbors augment the received message with their own information and pass it on to their respective neighbors. This process continues iteratively for a specified depth K, representing the walk depth. After K iterations, we collect the final messages and summarize them as features. Implementing this approach can be achieved using Spark's GraphX [Xin et al., 2013] module, which provides a graph-parallel computation abstraction facilitating message passing and aggregation.

Later in Chapter 5, we propose adopting the message-passing approach with tailored adjustments towards computing graph features in a streaming setting. Considering the dynamic and continuous nature of streaming data, we can optimize the message-passing step by setting it to occur only once during the walk process. By doing so, we limit the walk depth to a single iteration ($K = 1$), which aligns perfectly with the streaming data paradigm. This strategic adaptation not only ensures timely processing of streaming data but also takes inspiration from the message-passing paradigm for effective feature computation in a streaming environment.

As a result, our approach optimally addresses the challenges posed by streaming data, enabling efficient and accurate graph feature extraction.

4.2.2.2 Distributed Joining Approach

Incorporating the principles of message-passing while operating independently from the GraphX module, the distributed joining approach is detailed in Algorithm 2, and illustrated in Figure 4.3. The methodology initiates with the construction of a distributed table, **G**, which is organized into two primary columns: **node**, listing each graph node, and **neighbors_lst**, cataloging the direct neighbors of each node.

We also define another table, **target_G**, which is essentially a subset of table *G*. This table contains the target nodes, those from which we intend to commence our random-walks, and their immediate neighbors. For clarity, we rename its columns to **t_node** and **t_neighbors_lst**. Furthermore, an additional column, **walk**, is introduced to the **target_G** table to store the resulting random-walk initiated from each target node.

The process is then iteratively conducted as follows:

1. Add a new column (or overwrite if it exists) to the **target_G** table, termed **chosen**, which is populated by randomly selecting a neighbor from the **t_neighbors_lst** column.
2. Execute a *join* operation between tables **target_G** and **G**. This is accomplished by aligning the **chosen** column from *target_G* with the **node** column in *G*. The outcome is the augmentation of a new column, **neighbors_lst**, that displays the neighbors of the nodes featured in the **chosen** column.
3. Overwrite *t_neighbors_lst* with the data from *neighbors_lst*.
4. Incorporate the values from the *chosen* column into the target node's walk list.
5. Reinitiate from the first step.

Upon completing this cycle *K* times, we achieve random-walks of depth *K*. To conduct *N* such random-walks, we can either replicate every target node across *N* distinct rows or sequentially (or in parallel, given their independence) repeat the aforementioned process *N* times.

Following this merging process, we obtain the details of the random walks. Subsequently, a step is undertaken to intersect this information with the nodes and edges features. Specifically, for the summarization phase, the random walks resulting table is joined with the tables containing the desired features to produce the feature matrices.

A tangible instantiation of such distributed tables would be the dataframes in Spark.

Algorithm 2 *Walking-Profiles*: Distributed Implementation

Require: **G**[node, neighbors_lst] ▷ Graph with nodes and their neighbors
Require: **target_G**[t_node, t_neighbors_lst] ▷ Target nodes subset of G
Require: K ▷ Depth of random-walks
Require: N ▷ Number of random-walks per target node

Add two empty column **chosen**, and **walk** to **target_G**
Duplicate each row in **target_G** N times for multiple walks
for $j = 1$ to K **do**
  chosen ← Random neighbor from **t_neighbors_lst**
  Join **target_G.chosen** with **G.node**
  Update **target_G.t_neighbors_lst** with **G.neighbors_lst**
  Append **chosen** to **target_G.walk**
end for

To provide a clearer visual representation, Figure 4.3 illustrates a 2-hop random-walk originating from two distinct target nodes, 6 and 4, resulting in walks (6, 1, 2) and (4, 5, 6) respectively.

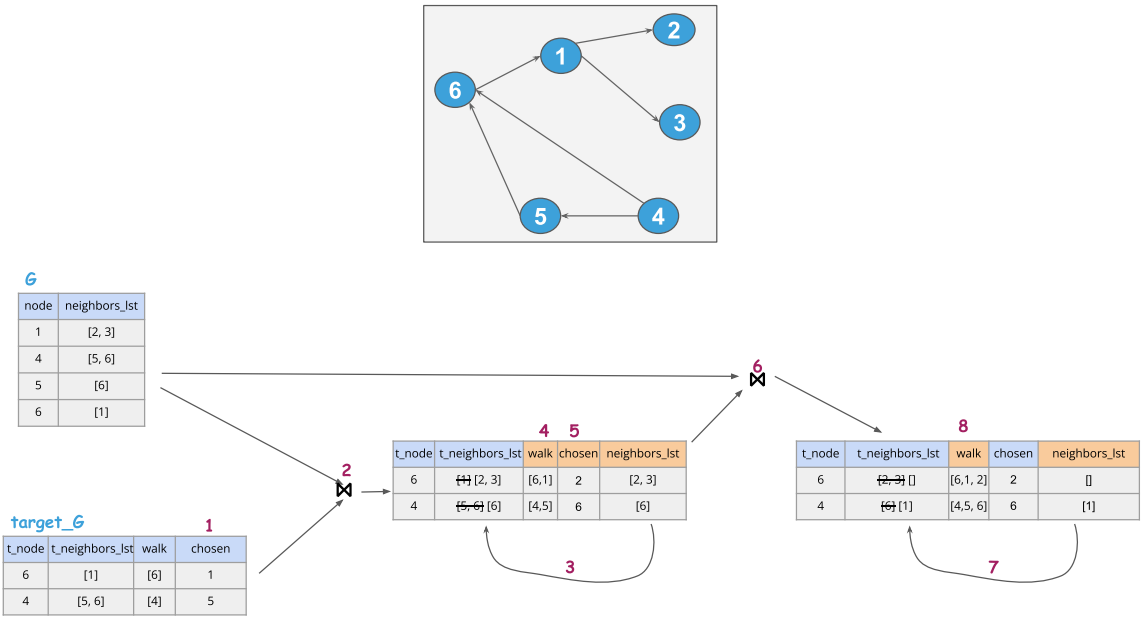


FIGURE 4.3: Applying distributed *Walking-Profiles* in a 2-hop walk example.

4.2.2.3 Sliding Window Technique for Memory Reduction

Organizations dealing with extensive real-time data often confront the challenge of representing a complete data history. Such representation necessitates significant memory and computational resources. In cases where the premise that recent data holds more relevance for decision systems, we propose the usage of a graph construction strategy anchored in the sliding window paradigm.

This method diverges from retaining the complete data chronicle in the graph. Instead, we adopt a sliding window approach, keeping only the most recent data, typically spanning over the last x units of time. By keeping only this recent subset, this intuitively leads to a reduction in memory usage. Moreover, an added advantage of this strategy is the prevention of exhaustive scans across the entire graph, which enhances computational efficiency. Consequently, this approach ensures that organizations can manage and analyze large data streams effectively without overburdening resources.

4.3 Triage Model: Integrating Walking-Profiles with AML

Predominantly, AML systems are composed of rule-based systems [Li et al., 2017] (depicted as *Rules* in Figure 4.4.a). These systems, while transparent and interpretable, often raise numerous false alarms that overwhelm human analysts with unnecessary workload (reported FPRs are around 95–98% [Lannoo and Parlour, 2021]). To address this issue, we propose an innovative ML component designed to triage alerts generated by the rules (*triage model* in Figure 4.4.a). Functioning at the level of alerts, this model retains its interpretability, and its design accommodates the inclusion of diverse features tailored to the specific demands of the AML application. The *triage model*'s outcome holds a dual purpose: it can suppress alerts with lower scores, removing them from the queue for analysis, or prioritize alerts based on their scores, leading to an organized queue of alerts for further analysis. Importantly, given that all alerts originate from the rule-based systems, the integration of the *triage model* preserves the inherent advantage of explainability.

For an enhanced efficacy, our proposed system harnesses a spectrum of feature types, ranging from entity-centric to neighborhood-centric (graph-features), subsequently processed by our *triage model* to classify suspicious activity as shown in Figure 4.4.b.

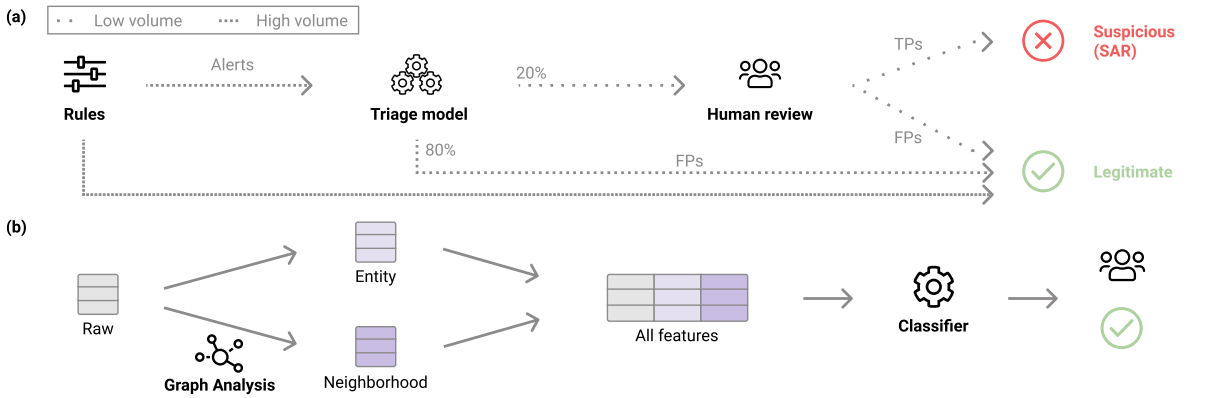


FIGURE 4.4: Overview of the full *triage model* system and details.

Subsequent subsections provide a granular exploration of the *triage model*, illustrating its adaptation from our graph feature engineering framework with an emphasis on reducing FPs. We particularly shed light on alterations made during the graph construction phase and in the realm of feature computation.

4.3.1 Graph Construction

Financial datasets usually manifest in tabular format. However, to extract neighborhood-centric insights we need to represent the data in a graph.

Financial transactions, mainly occurring between bank accounts, naturally suggest representing accounts as nodes, and transactions as edges between accounts. The direction of the edge follows the direction of the money (i.e., from sender to receiver), and edge attributes include the transaction timestamp and amount. Figure 4.5.a shows a toy example of our tabular data and how we represent it in a graph: each entity (account in this case) is represented by a node, which can have two types (Internal or External). Edges represent transactions between entities (i.e., accounts). Their direction follows the money flow, edges also have the timestamp and amount of a transaction as attributes.

Scalability challenges emerge when dealing with large transactional data, especially for financial titans processing millions of transactions daily. Acknowledging the decreasing relevance of historical data, our solution pivots to a dynamic graph architecture, using sliding windows to identify relevant data subsets. The graph undergoes an iterative update, forgetting old edges and embracing the current day's transactions, as detailed in Figure 4.5.b which shows an example of a sliding window of 60 days. Therefore, every node linked to the day's transactions

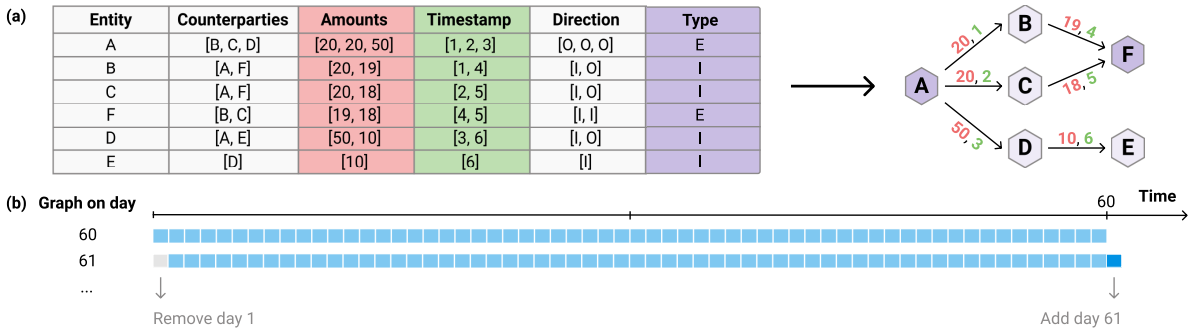


FIGURE 4.5: Graph construction: from tabular data to graph representation.

has its graph attributes computed. While our current framework operates on a daily cycle, adaptability remains at its core.

4.3.2 Customising Walking-Profiles for AML

As a concrete example of the features described in Section 4.2.1, we will discuss the AML domain. Money laundering concerns disguising the origin of illegally obtained money, typically by moving funds between various accounts and FIs creating a complex web of transactions. Using information characterizing the transaction graph is therefore relevant in this context. We held multiple sessions with AML domain experts to gain a deeper understanding of the problem and the data commonly used in their investigations. Based on these collaborative interactions, we identified the following features for computation in the AML context:

1- Node or Edge Level Features

- **Degree**: Based on the hypothesis that suspicious accounts tend to interact with a larger number of counterparties, we derive in- and out-degrees for target nodes. This helps understanding the structure of the neighborhood.
- **Delta Time**: We use the *transaction timestamp* edge feature and calculate the differences in transaction timestamps across the walks. thereby estimating the transaction velocities during the random-walks.
- **Delta Money**: By employing the *transaction amount* edge feature, we obtain the variance in the transferred sums.
- **Flow Position**: This metric, denoted as *flow position*, is derived using:

$$flow\ position = \frac{\sum(\$sent) - \sum(\$received)}{\sum(\$sent) + \sum(\$received)}$$

It ranges from -1 to 1. A -1 score indicates an account primarily receiving money, whereas a 1 indicates an account that only sends money. A 0 indicates balanced transactions, often characterizing what are usually known as money mule accounts in laundering schemes. For granularity, we evaluate this metric over varying durations, such as one week, and collect these resulting *flow position* features within the walks.

- **GuiltyWalker (GW)**: Recognizing the tendency of suspicious nodes to exist within criminal networks (See example in Figure 4.7), the *GuiltyWalker* (GW) features [Oliveira et al., 2021] aim to leverage this pattern. Unlike original *GuiltyWalker*'s random-walks that terminate upon identifying a known illicit node, our modified approach is more generalized. We allow random-walks to continue even after encountering a suspicious node, ensuring the collection of other features and enriching the *GuiltyWalker* information. In this process, the node label is adopted as the categorical node feature. Subsequently, we derive metrics such as the distance to the illicit node and the total count of illicit nodes encountered during each walk. These node labels could represent either confirmed money laundering cases or files of SARs based on historical data.

- **GuiltyWalker-delay (GWd)** *GuiltyWalker* assumes immediate feedback, i.e., that labels are immediately available for all past transactions. In AML, however, investigations are lengthy, resulting in label delays. We propose an adaptation of *GuiltyWalker* by introducing a waiting period. We start by training an ML model using entity profiles and degree features on a first training set. We use the resulting model to score a second training set and define a suitable threshold to obtain *pseudo-labels*. We then compute the *GuiltyWalker* features using the *pseudo-labels* for the unlabeled transactions in the waiting period and the actual labels otherwise. Finally, we train the *triage model* on the second training set, integrating both actual and *pseudo-labels*.

2- Subgraph Level Features

The following categorical features are used to compute subgraph level features as described in Section 4.2.1.2.

- **Currency patterns**: By using the *transaction currency* edge feature as a categorical metric, we analyze currency exchange patterns around nodes to detect potential correlations with money laundering practices.
- **Transaction patterns**: We use the node ID as the categorical feature. This provides a structural insight into the graph surrounding the target node.

3- Community Level Features

- We calculate the **community size** and **community sparsity** features leveraging node ID as detailed in Section 4.2.1.3.

4.3.3 Triage Model

4.3.3.1 Model

Our proposed *triage model* is adaptable and not restricted to any particular classifier. However, for the purpose of enhancing interpretability, we recommend utilizing tree-based models such as random forests [Ho, 1995] or LightGBM [Ke et al., 2017]. The model is trained using raw tabular data enriched with engineered features, encompassing entity-centric and neighborhood-centric attributes. The model generates a score that quantifies the degree of suspicion associated with a given transaction.

4.3.3.2 Explaining Classifier Score

The objective of this section is to explain the scores produced by our *triage model* for two key audiences: human reviewers and regulatory bodies. Reviewers seek clarification on the rationale behind raising a specific alert as suspicious, while regulators focus on justifying the exclusion of certain alerts.

Assuming that the *triage model* is grounded in a tree-based classifier (e.g., LightGBM), we employ TreeSHAP [Lundberg et al., 2020] for model interpretation due to its computational efficiency and empirical effectiveness as validated by its developers. TreeSHAP is a model-specific explanation method tailored for tree-based models. It computes exact Shapley values -quantifying each feature's impact on the final prediction- efficiently by utilizing the structure

of these models. The algorithm involves traversing the tree from root to leaves, attributing contributions to to each feature encountered. The process is repeated for all trees in the model.

To enhance comprehensibility, we aggregate explanations by categorizing features into distinct **semantic groups**. By analyzing the features employed by the model, we group similar attributes together. This categorization is expressed as key-value pairs, wherein keys represent semantic groups, and corresponding values enumerate the features associated with each group based on human interpretation.

For every event necessitating an explanation, the following procedure is undertaken:

1. Retrieve the SHAP values for the event, leveraging the TreeSHAP library.
2. Aggregate SHAP values associated with the same semantic category to form an array of aggregated SHAP values. The aggregation function can be sum, average, or any other aggregation function.
3. Create an illustrative visualization using the consolidated features and SHAP values.

With this procedure the reviewer will have a high level picture on what group of features is contributing to the event being considered of high risk or low risk. Then if the reviewer wants to see the details, they always can visualize the original shapely values of every feature.

4.4 Experiments & Results

In this section, we begin by detailing the real-world dataset employed for our experiments in Section 4.4.1. Subsequently, we delve into experiments conducted using our proposed triage ML model. These experiments utilize raw tabular data augmented with specific engineered features:

- Entity-centric features, as discussed in Section 4.4.3.
- Neighborhood-centric features, elaborated in Section 4.4.4.
- Features derived from the customized *Walking-Profiles* graph (described in Section 4.3.2), results are covered in Section 4.4.5.

4.4.1 Data

We utilize a real-world banking dataset for the following experiments. Due to privacy constraints, we cannot reveal the bank's identity nor provide exact details, but we provide approximate metrics to characterize the data where possible.

The raw dataset encompasses around half a million transfers that were *alerted* by a rule-based system (See Figure 1.2) involving 400,000 accounts and spans over approximately one year. It distinguishes accounts based on their association with the bank can be internal or external. Transfers occur in both directions between two internal accounts or between an external and an internal account. The dataset is labeled on a transaction level, with a binary label indicating whether a transaction was part of a SAR. However, we devise the proposed *triage model* to generate alerts at the account level, as is typical in AML. Moreover, in our experiments, we intend to assess accounts on a daily basis. Hence, we preprocess the raw dataset to contain aggregated daily account features, including total sent and received amounts, the counterparties, the associated timestamps, and the direction. We then extrapolate from the transactional labels to infer the account labels: if there is a suspicious transaction involving an account on a specific day, we mark that account as suspicious on that day. Importantly, this means that suspicious accounts form connected pairs in our preprocessed dataset. Suspicious accounts comprise less than 3% of the alerted batch, leading to an overwhelming 97% of FPs.

The dataset's unique categorical feature indicates the account type—either external or internal—and is binary-encoded as 0 or 1, respectively. We retain numerical features in their original form and use this dataset as a foundation to compute all subsequent features.

To elucidate the structural differences within the dataset, Figure 4.6 demonstrates the degree differences between legitimate and suspicious nodes in the dataset. The left subfigure details the in-degree exploration, showing the number of incoming connections to each node. This indicates the frequency with which different nodes receive interactions or transactions. We notice that suspicious nodes often exhibit high in-degrees, particularly when the in-degree exceeds a hundred, suggesting an increased likelihood of the node being suspicious. The right subfigure, on the other hand, displays the out-degree exploration, which counts the number of outgoing connections from each node. Similarly to the in-degree we notice that high out-degrees are associated with suspicious nodes more commonly. This comparison between in-degree and out-degree metrics is crucial for identifying distinct patterns in the network, thereby differentiating between normal and potentially suspicious nodes.

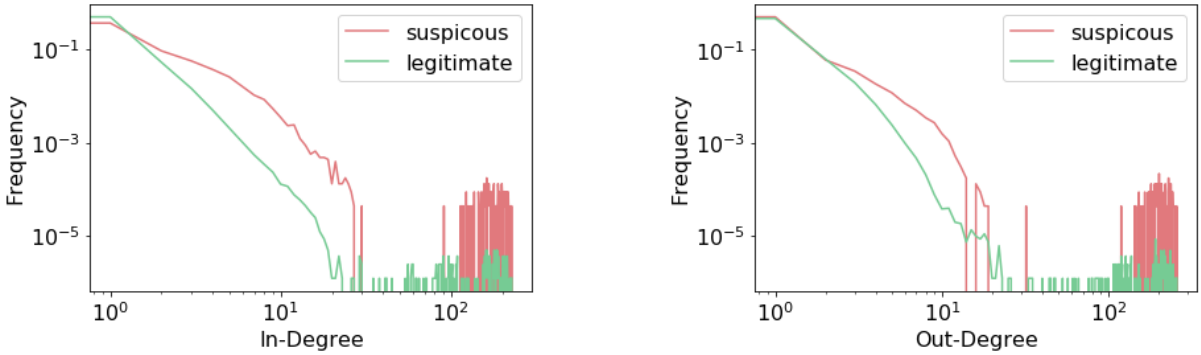


FIGURE 4.6: Data exploration: differences in degree between legitimate and suspicious nodes

Building upon the insights gained from the degree analysis in Figure 4.6, we now turn our attention to Figure 4.7 for an understanding of the network's interconnectedness in the context of AML investigations. This figure compares two separate neighborhoods, identified as weakly connected components in an AML investigation network. These components represent subgraphs where each node is accessible from every other node within the same component, ignoring edge direction, and isolated from nodes outside the component.

In Figure 4.7 blue nodes represent legitimate accounts and red nodes represent suspicious ones (were involved in a SAR). Moreover, nodes labeled with 'E' represent external accounts to the financial institution, while those marked with 'I' indicate internal accounts. The upper subfigure displays a network component composed of only legitimate accounts. This section effectively illustrates the normal transaction and interaction patterns among these accounts, providing a clear view of the typical functioning within the FI. In contrast, The lower subfigure, shows a network component consisting of suspicious accounts, marked as red nodes. This component is noteworthy for the presence of suspicious accounts showing how they are interconnected mainly through two suspicious hubs. The differences between these two components underscore the utility of graph-based analytical approaches in detecting criminal networks in AML case investigations, thereby helping in the identification of potentially illicit activities. Building on this analysis, the following discussion delves deeper into how *Walking-Profiles* can leverage these patterns of connectivity to enhance AML investigations.

4.4.2 Experimental Setup

We partition the dataset temporally into three distinct intervals: the earliest 60% is earmarked for training, the subsequent 10% for validation, and the remaining 30% for model testing. For the *GuiltyWalker* with delay features (refer to Section 4.3.2), the training subset is further divided.

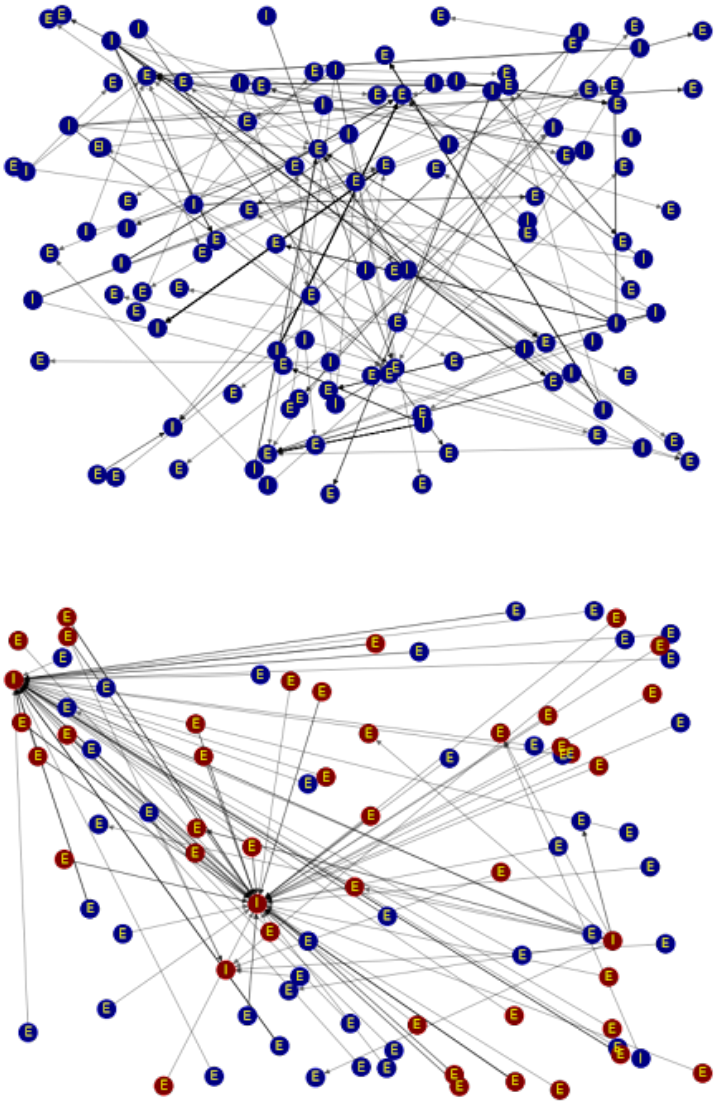


FIGURE 4.7: Compararing legitimate and suspicious account neighborhoods in an AML network.

We aim to maximize the suspicious activity captured by our *triage model* (i.e., TPs) while minimizing incorrect alerts (i.e., FPs). We choose our optimization objective to maximize recall at a specific FPR. The FPR can be chosen in accordance with the client. In our experiments, we consider *Recall@20%FPR* as our target metric, which translates to a *reduction* of the FPs by 80% compared to the rule system itself. Moreover, because most events are legitimate, the chosen FPR (i.e., 20%) roughly corresponds to the number of alerts to be reviewed to obtain a particular recall.

To illustrate the relationship between our *triage model* and the rule-based system (depicted in

Figure 1.2), refer to Figure 4.4. In the subsequent sections, we delve into the various features utilized for training the *triage model* classifier.

4.4.3 Triage Model using Entity-centric Features

4.4.3.1 Data Preprocessing and Entity-centric Features Creation

In this experiment, we augment the raw data with entity-centric features—those that capture the historical essence of an entity without delving into neighborhood details. Initially, the features we engineer for our *triage model* are tailored to capture the transactional history unique to each account. Following Branco et al. [2020], we term these features as *profiles*. Defined more specifically, profiles are arithmetic aggregations done across a specified field and within a given time window, for example, the total amount spent by an account over the preceding week. Such features empower an ML model to compare an account's long-standing history (long windows) against its recent actions (short windows), thereby understanding patterns indicative of suspicious activities.

For the purpose of our tests, we create roughly 400 profile features. Taking the account as our grouping entity, we aggregate data points on the amounts sent and received. Five distinct time frames are taken into account: a day, a week, two weeks, a month, and two months. During the aggregation phase, we employ an array of functions—sum, mean, minimum, maximum, and count. Additionally, we explore the comparative dynamics between two-time windows via ratios and differences.

Subsequently, to choosing the most important features, we employ a permutation-based feature importance method. This entails training a gradient-boosted trees model on a subset of the training data. Our aim is to obtain set of features, which account for 90% of the performance concerning our metric of interest (i.e., *Recall@20%FPR*). Consequently, we select approximately top 100 important entity features to enrich our dataset. The entire process of profile creation is orchestrated via our in-house platform. This platform automatically generates features based on the semantic labels of the data fields (e.g., entity, location, date, or amount) [Marques et al., 2020].

All experiments are executed on the real-world banking dataset, detailed in Section 4.4.1. In these experiments, the primary entities targeted for labeling are the bank accounts. Uniformly, every model deploys the raw features delineated in Section 4.4.1, accompanied by a suite of

approximately 100 profiles, each grounded on the sent and received transfer amounts tied to individual accounts. The first *triage model* we introduce is trained solely using the raw features and the aforementioned entity-centric profile features.

4.4.3.2 Optimization

We use Optuna [Akiba et al., 2019] to optimize the hyperparameters of all models. Our focus was on a set of machine learning models, which included Random Forest [Ho, 1995], Logistic Regression [Dobson and Barnett, 2018], and LightGBM [Ke et al., 2017]. The process involved training a total of 150 models, with 50 distinct models for each machine learning algorithm, selected through random sampling. The performance of these models was then evaluated based on their performance on a validation dataset. The optimal model was chosen based on this performance assessment, using the *Recall@20%FPR* metric. Detailed information about the algorithms used and the specific range of hyperparameters considered for each is provided in Table 4.1.

Algorithm	**Hyperparameter**	**Range/Values**
Logistic Regression	Alpha	[0.01 - 0.09]
	Standardize numericals	[True, False]
Random Forest	Max depth of trees	[10 - 40]
	Number of trees	[100 - 200]
	Min instances for split	[10 - 50]
LightGBM	Num of leaves	[200 - 500]
	Min data in leaf	[100 - 200]
	Learning rate	[0.01 - 0.09]

TABLE 4.1: ML algorithms and Hyperparameters ranges for *triage model*

4.4.3.3 Results

The top-performing model was a LightGBM, with a test performance close to 80%*Recall@20%FPR*. This LightGBM model is considered as our **baseline** *triage model* in subsequent experiments.

4.4.4 Enriching Triage Model with Neighborhood-centric Features

This section investigates the potential enhancement to the *triage model* using graph-based features in addition to the existing entity-centric features. We focus on three primary features: node degree, *GuiltyWalker*, and *GuiltyWalkerDelay*.

Our approach begins with the construction of a directed graph with accounts as nodes and transactions between these accounts as edges. A more detailed methodology is provided in Section 4.3.1. Given the diminishing significance of older events in current predictions, we use a sliding window approach to ensure only the most recent transactions are considered (Section 4.2.2.3). Pursuant to this, graph snapshots spanning 60-day intervals were formed based on the guidance from [Jullum et al., 2020]. We also explored different intervals for suspicious and non-suspicious activities.

The features under examination are detailed as follows:

Degree Features.

We hypothesize that the class of an account might influence its number of neighbors and the corresponding monetary flow. We thus determine both the in-degree and out-degree for each node and its adjacent neighbors. The neighboring degrees were aggregated using mean, minimum, and maximum operations. In this way, we create eight new features that characterize the number of counterparties of an account and its neighborhood. Analogously, we calculate a *weighted* version of these features by using the transferred amount as the edge weight. When these degree features were integrated into the base model, we observed an enhancement in performance by 11.6 percentage points in *Recall@20%FPR*, supporting our theory (See Figure 4.8, *+Degrees*). While weighted versions of these features (using transaction amounts as weights) were developed, their efficacy remained inferior to the standard degree features.

GuiltyWalker features.

Given the complex connections often seen in money laundering schemes, we postulate a higher likelihood of interconnected suspicious nodes. Therefore, we derived *GuiltyWalker* (GW) features [Oliveira et al., 2021], which assess proximity to such suspicious nodes through random-walks. Random-walks are generated which stop upon reaching a known illicit node or if there are no available connections. In our implementation, we run 50 random-walks for each target node. We then compute the features proposed in the original work, namely features characterizing the length of the random-walks (minimum, maximum, median, mean, standard deviation, 25th, and 75th percentile), the fraction of successful random-walks (i.e., the "hit rate"), and the number of distinct illicit nodes encountered. The inclusion of GW features boosted the model's performance by 13.4 percentage points in *Recall@20%FPR*. Remarkably,

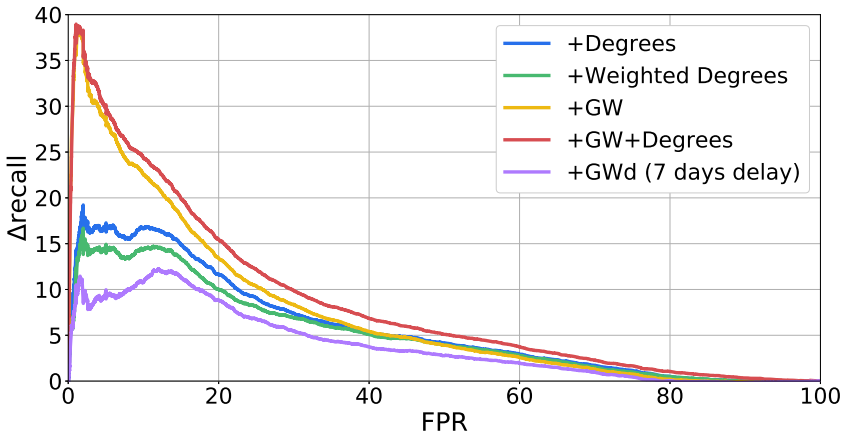


FIGURE 4.8: *Triage model:* Impact of graph features on performance.

GW performance peaks, with up to a 38% improvement, were observed in lower FPR regions (Figure 4.8, *+GW*).

In addition, we combined both degree and *GuiltyWalker* features with the base model to determine if they offer complementary insights. Our findings indicated some overlap, yet the combined model outperformed models with individual features, with an enhancement of 15.5 percentage points in *Recall@20%FPR* (Figure 4.8, *+GW+Degrees*).

GuiltyWalker-delay features (GWd).

Real-world evaluations indicate that while GW features effectively reduce FPs, the foundational GW algorithm [Oliveira et al., 2021] presumes daily updates of previous labels—a scenario not always feasible in real banking AML contexts. Recognizing delays in actual banking operations, we modified the GW algorithm to utilize model scores and thresholds for generating pseudo-labels for recent events. This is expanded upon in Section 4.3.2. Our threshold optimization involved an extensive grid search across several values. Under a 7-day label delay, an optimum gain in *Recall@20%FPR* was observed for a threshold of 0.25.

Subsequent tests with varying label delays from 1 to 30 days confirmed that reduced delays led to superior performance. Interestingly, even with a one-month labeling delay, the GWd features significantly augmented the baseline model's performance (Figure 4.9).

In a final set of tests, we assessed the combined performance of degree and GWd features. Although the degree features improved TP rates, they did not exceed the performance achieved by solely using degree features (compare blue line in Figure 4.8 with the red line in Figure 4.9).

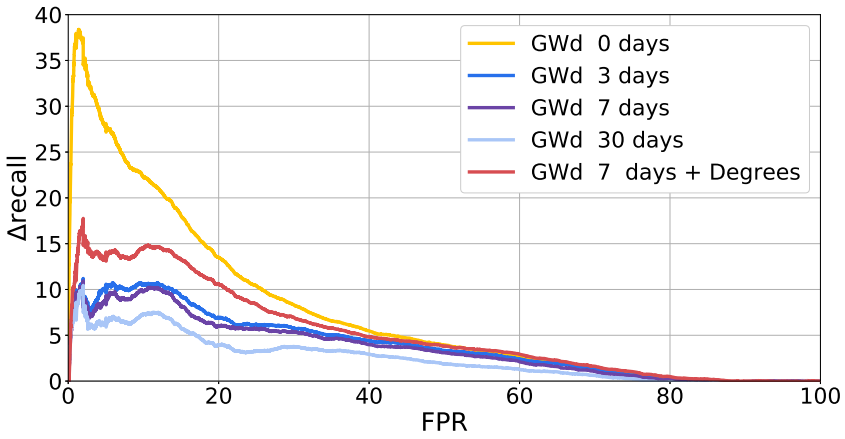


FIGURE 4.9: *triage model:* Impact of label delay on performance.

This was consistent even when the degree features were derived from the reduced dataset intended for GWd models. Considering the computational efficiency, the degree features emerge as an optimal choice in such scenarios.

4.4.5 Enriching Triage Model with Walking-Profiles Features

In this section we assess the added value of enriching a baseline that uses entity-centric features with the WalkingProfiles features detailed in Section 4.3.2 Our aim is to leverage a *triage model* and perform binary classification to predict whether an alerted activity is indeed suspicious or is an FP. We use a LightGBM [Ke et al., 2017] classifier as the *triage model* and Optuna [Akiba et al., 2019] is used for hyperparameter tuning. Similary to previous experiments (Section 4.4.3.2) we use 50 models and *Recall@20%FPR* as our target metric.

Similarly to previous section, our approach begins with the construction of a directed graph with accounts as nodes and transactions between these accounts as edges. A more detailed methodology is provided in Section 4.3.1. Given the diminishing significance of older events in current predictions, we use a sliding window approach to ensure only the most recent transactions are considered (Section 4.2.2.3). Pursuant to this, graph snapshots spanning 60-day intervals were formed based on the findings from [Jullum et al., 2020].

To calculate the graph features, we use the WalkingProfiles AML customization (Section 4.3.2). We choose the number of walks equal to $N = 50$ and the walk depth equal to 5.

In figure 4.10, we show the difference in recall to the baseline vs. FPR. Results show that by adding graph-based features, we obtain an improvement up to 12%*Recall@20%FPR* assuming a

label delay of 1 day for the *GuiltyWalker* features and an improvement of 7%*Recall@20%FPR* assuming a label delay of 7 days. Furthermore, we evaluate the model without using the *GuiltyWalker* features and obtain an improvement of 5%*Recall@20%FPR*. It is important to note that Figure 4.10 can not be directly compared with Figure 4.8, and Figure 4.9 due to the use of two different baselines.

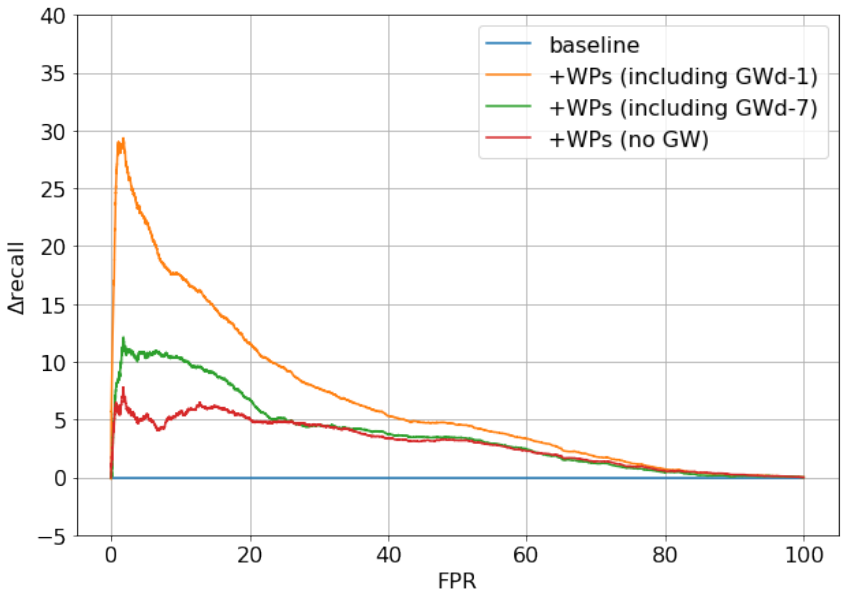


FIGURE 4.10: *Triage model:* Impact of integrating *Walking-Profiles* graph features.

Finally, regarding the **inference time** in this setup, our model processes 20 events per second in the used AML datasets.

4.4.6 Assessing Sliding Window Effects on Triage Model Performance

In the previous experiments, we built a dynamic graph using a sliding time window of 60 days (Section 4.3.1). We now wondered how changing this window affects the *triage model* performance. Moreover, since our experiments showed that connections to known suspicious accounts are important features, we investigate whether keeping a more extended memory for such suspicious accounts compared to legitimate ones is helpful. To this end, we use a different window for each event type (legitimate vs. suspicious) and retrain our best model for a realistic case of label delays, which uses only degrees features as described in the previous section. Similar results were obtained when retraining the best model without label delay (using degrees+GW features, data not shown). We perform a grid search for values of these time windows between 0 days (i.e., events are not used in the graph at all) up to 90 days. For brevity,

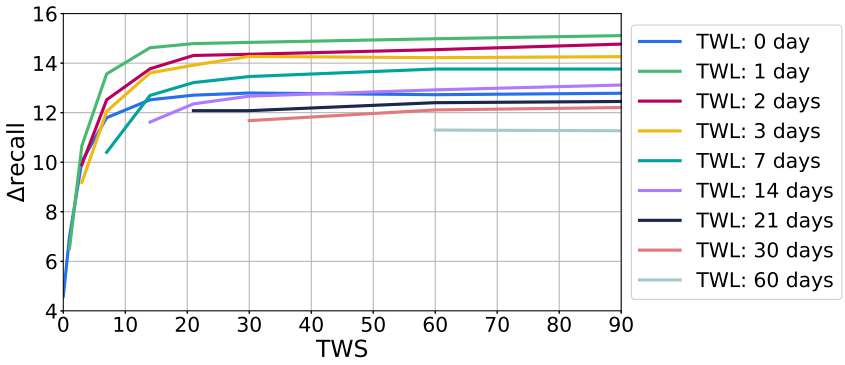


FIGURE 4.11: *Triage Model:* Impact of varying time window size on model performance, illustrating the balance between Time Windows for Suspicious Events (TWS) and Time Windows for Legitimate Events (TWL). The figure highlights how different time window sizes influence the predictive performance of the model.

we refer to the *time window for legitimate events as* ***TWL*** and to the *time window for suspicious events as* ***TWS***. Firstly, we find that, for any value of TWS, the best performance is achieved for a TWL equal to one day. Secondly, the performance increases only marginally when increasing the TWS beyond 30 days (Figure 4.11). Therefore, we can construct a good model efficiently by keeping only one day of legitimate events and 30 days of suspicious events in our graph. Importantly, having separate time windows for legitimate and suspicious events implies knowing the label at least after the duration of the smallest time window. Thus, for a label delay of 7 days, the best model we can construct efficiently would be using a TWL of 7 days and a TWS of 30 days. Nonetheless, it is interesting that we can significantly reduce the data needed to construct the graph without sacrificing performance.

4.4.7 Interpreting the Triage Model Through TreeSHAP

This experimental phase aims to understand the underlying mechanics of our *triage model*, with a specific focus on facilitating comprehension for two pivotal stakeholders: human reviewers and regulatory entities. To achieve this depth of insight, the TreeSHAP technique, as elucidated in Section 4.3.3.2, is harnessed.

In order to construct semantic groups of features, a manual analysis of the enriched features is undertaken. Some illustrative features from this analysis include:

- **Receivers magnitude:** Comprising features that aggregate node degrees of recipients receiving funds from the node being studied.

- **Money sent:** Including features that combine data about money leaving accounts in the nearby area. For example, the "Average amount sent" feature reveals information about the amount of money exchanged in the local network.

Figures 4.12 and 4.13 provide an illustrative example of the level two (aggregated) and level one (detailed) explication processes for a suspicious case that the model accurately assigned a high score. Figure 4.12 showcases a prominent red color, indicating heightened suspicion stemming from variables like neighborhood magnitude, the account's internal nature, and substantial transactional sums within its radius. For a granular examination, refer to Figure 4.13, which discloses specific money amount values and node degrees responsible for the elevated model score.

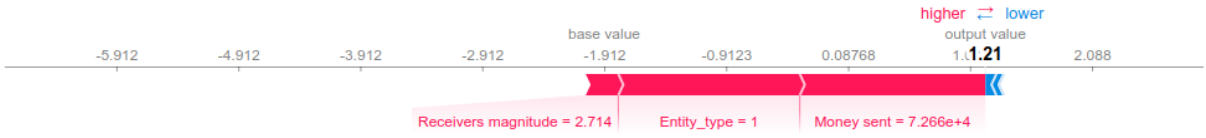


FIGURE 4.12: *Triage model:* Aggregated explanation for a suspicious case.

FIGURE 4.13: *Triage model:* Detailed explanation for a suspicious case.

In contrast, Figures 4.14 and 4.15 illustrate the model's reasoning for accurately assigning a low score to a **legitimate case**. Figure 4.14 demonstrates how aggregated features contribute to a decreased score, indicative of diminished suspicion (a potential FP of the rules system). For a comprehensive exploration, delve into Figure 4.15.

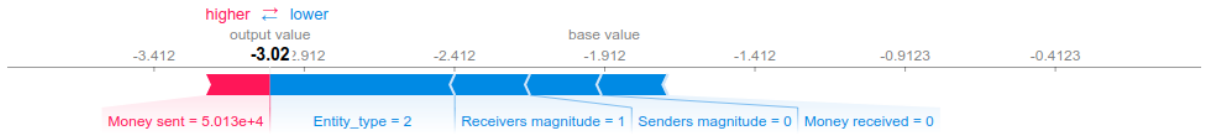


FIGURE 4.14: *Triage model:* Aggregated explanation for a legitimate case.

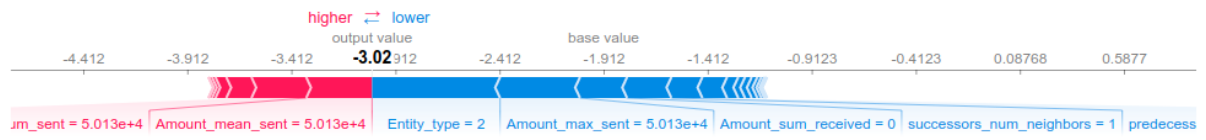


FIGURE 4.15: *Triage model:* Detailed explanation for a legitimate case.

4.5 Summary

In summary, graphs play a crucial role in detecting new patterns as they supplement traditional ML techniques by providing a more holistic view of the target entity's behavior through the analysis of its interactions within a network. This broader perspective can lead to more sophisticated and insightful classification models, ultimately advancing our ability to recognize and understand complex patterns in data.

Within this chapter, we present a dynamic graph feature engineering framework that exhibits the following characteristics:

- Works with temporal graphs (edges are timestamped)
- Flexibility in considering any node or edge feature.
- Customizability to suit various scenarios.
- Generation of explainable features for better interpretability.
- Calculation of graph features on the node, subgraph, and community levels.

Furthermore, we demonstrate the practical application of this framework in the AML domain, working alongside rule-based systems in a novel pipeline in Section 4.3. The aim is to reduce FPRs of rule-based systems while ensuring compliance with regulatory standards.

Chapter 5

Graph Sprints: A Method for Low-latency Graph Feature Engineering

Real-world datasets often have a dynamic graph structure, characterized by evolving relationships between data points, as seen in social networks, financial datasets, and biological systems. ML models, particularly GNNs, excel in handling these datasets, but face challenges with CTDGs due to high computational costs in embedding computations. This chapter presents the *Graph-Sprints* method, a random-walk based graph feature extraction framework designed for low-latency solutions in large data and high-frequency contexts like financial transaction, focusing on leveraging the latest information for enhanced capabilities. The organization of the chapter is as follows:

- **Random-walk Based Features:** Section 5.1 initiates with an overview of the Random-walk based graph feature extraction framework, as detailed in Chapter 4, Section 4.2.1.
- **Graph-Sprints method:** Section 5.2 elaborates on the derivation of our *Graph-Sprints* method from the random-walk based framework, delving into the intricate details of the method.
- **Memory reduction techniques:** Section 5.2.5 introduces two strategies tailored to minimize the memory demands of our *Graph-Sprints* approach.
- **Theoretical analysis:** Sections 5.3.1 and 5.3.2 Undertake a rigorous theoretical analysis, exploring its equivalence to random-walks and its complexity dynamics.

- **Experiments and results:** Section 5.4 Culminates with a presentation of our findings, showcasing how the *Graph-Sprints* features, when combined with a neural network classifier, manage to be both time-efficient and retain robust predictive performance, when compared against the more time-intensive GNNs.

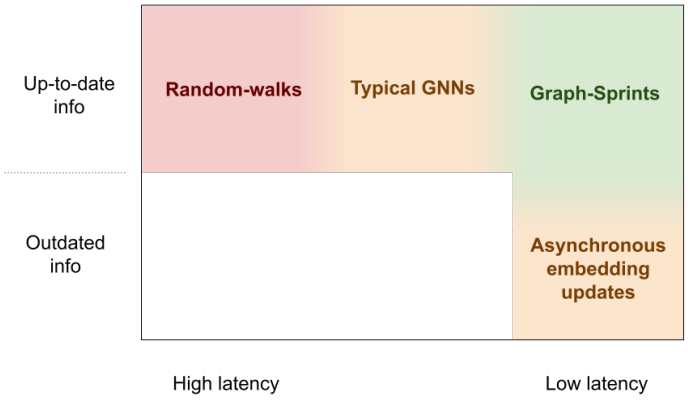


FIGURE 5.1: Overview of various approaches in CTDGs.

5.1 Random-walk Based Features

Prior to delving into the intricacies of our *Graph-Sprints* framework, it is relevant to provide a brief summary of the random-walk based feature extraction framework. This framework, extensively detailed in Section 4.2.1, lays the foundation upon which our subsequent discussions will be anchored. Following this overview, we will elucidate the efficient computational techniques underpinning *Graph-Sprints*. The random-walk based feature extraction framework essentially encompasses the following sequential steps, resulting in the creation of a node feature vector for a specific seed node (Figure 5.2).

1. **Select the seed node.** This selection depends on the use-case, and for CTDGs typically one considers entities involved in new activity, for instance if the change on the graph is adding a new edge between two nodes, then each of these two nodes could be a candidate for a seed node.

2. **Perform random-walks starting from the seed nodes.** During the random-walks, relevant data such as node or edge features of the traversed path are collected. The type of random-walks influences what neighborhood is summarized in the extracted features. Walks can be (un)directed, biased, and/or temporal.

3. **Summarize collected data.** The data collected over walks is aggregated into a fixed set of features, characterizing each seed node's neighborhood. Examples of such aggregations are the average of encountered numerical node or edge features, the maximum of encountered out-degree, etc.

The computation of these features is costly, because multiple random-walks need to be generated for each seed node. For CTDGs, one would have to compute such features each time an edge arrives. This is infeasible for high-frequency use-cases such as fraud detection in financial transactions, where a decision about a transaction needs to be made in a few milliseconds. In the next section, we derive an efficient approximation to the above random-walk based features.

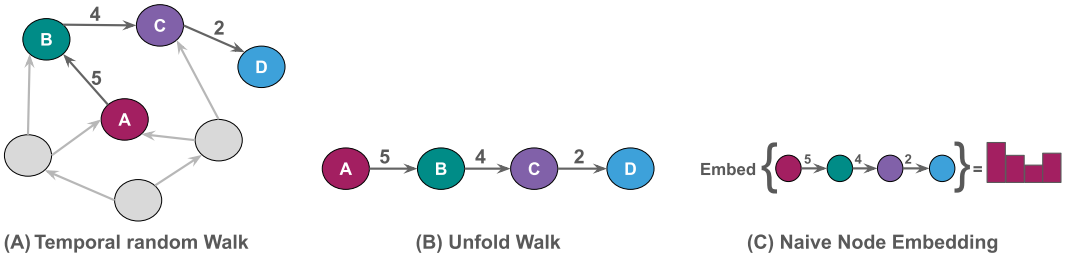


FIGURE 5.2: Conversion of random-walks to histograms.

5.2 Method

In this section, we propose approximations to random-walk based features described in Section 5.1. Our aim in this section is to optimize the computation of such features by exploiting recurrence and eliminating the need to execute full random-walks. As shown in Figure 5.3, given a temporal graph where edges have a timestamp feature (numbers) representing the time that a relationship was created, the left banner of Figure 5.3 illustrates temporal random-walk is traversed from the most recent interaction A-B towards older interactions. As shown in the right banner, one can compute similar embeddings to the ones in Figure 5.2 in a streaming setting, from only the new edge and the existing embeddings of the involved nodes..

FIGURE 5.3: Streaming histograms from temporal random-walks.

5.2.1 Assumptions

For our approximations to be reliable, we make the following assumptions: the input graph is a CTDG with directed edges (we will relax this assumption later), edges have timestamps and the temporal walks respect time, in the sense that the next explored edge is older than the current edge. With these assumptions, one can unfold any directed temporal walk as a time-series (Figure 5.2A and 5.2B).

5.2.2 Streaming Histograms as Node Embeddings

Given the above assumptions, we now formalize the approximation of random-walk based aggregations described in Section 5.1.

In this framework, we do not consider random-walks with a fixed number of hops, and instead consider infinite walks, on top of which we compute embeddings analogously to exponential moving averages. The importance of older information compared to newer is controlled by a factor α between 0 and 1. A larger α gives more weight to features further away in the walk (or in the past), and we can therefore consider α the parameter that replaces the number of hops. Formally, let $\vec{S}_i$ be a histogram with L bins, represented as an L-dimensional vector and characterizing the distribution of a feature f in the neighborhood of node i. A full infinite walk starting at node 0 computes the histogram $\vec{S}_0$ as:

$$\vec{S}_0 = \sum_{i=0}^{\infty} \alpha^i (1-\alpha) \vec{\delta}(f_i) \tag{5.1}$$

where $\sum$ is adding vectors, α is a discount factor between 0 and 1, controlling the importance of distant information in the summary $\vec{S}_0$, and i denotes the hops of the walk ($i = 0$ being the newest node, or in other words the seed node of the infinite walk). f_i is the feature value at node i or edge i, and $\vec{\delta}(f_i)$ is an L-dimensional vector with element $\vec{\delta}_j = 1$ if the feature value f_i falls within bin j and $\vec{\delta}_j = 0$ for all other elements. Equation 5.1 then implements a streaming counts per bin, where older information is gradually forgotten. If the feature f_i is a node feature, then the value is taken from the current node. If it is an edge feature, then the feature value is taken from the edge connecting the current node and the chosen neighbor.

One could compute multiple such summaries per node, one for each node or edge feature of interest, and together they would summarize a neighborhood. The key idea is that we can now approximate the infinite random-walks, i.e., the infinite sum of equation 5.1, by performing only a finite number of $k \geq 1$ hops, followed by choosing a random neighbor of the last encountered node and choosing an available summary $\vec{S}_k$ of that neighbor randomly, where $\vec{S}_k$ is defined as

$$\vec{S}_k = \sum_{i=0}^{\infty} \alpha^i (1-\alpha) \vec{\delta}(f_{i+k}) \tag{5.2}$$

With this strategy, we can approximate the summary $\vec{S}_0$ from equation 5.1 recurrently using

$$\vec{S}_0 \approx \sum_{i=0}^{k-1} \alpha^i (1-\alpha)\vec{\delta}(f_i) + \alpha^k \vec{S}_k \tag{5.3}$$

Compared with equation 5.1, one now truncates the sum after k terms. Note that whenever the last histogram $\vec{S}_k$ is normalized such that the bins sum to 1, e.g. using a uniform initialization for terminal nodes, equation 5.3 guarantees that all subsequent histograms will be normalized in the same way. Since we are interested in low-latency methods, we take the limit of $k = 1$ and Equation 5.3 becomes a streaming histogram:

$$\vec{S}_0 \leftarrow (1-\alpha)\vec{\delta}(f_0) + \alpha \vec{S}_1 \tag{5.4}$$

The hyperparameter α can be chosen to depend on the number of hops or on time. When discounting by hops, this discount factor α is a fixed number between 0 and 1. When discounting by time, the factor is made dependent on the difference in edge timestamps, for example exponentially or hyperbolically.

Using equation 5.4, one could approximate N (biased) random-walks by sampling N neighbors (non-uniformly), and subsequently combining the resulting histograms, e.g., by averaging. This would require performing N 1-hop look-ups each time.

Instead of that, we can increase efficiency even further by removing any stochasticity and updating a node's histogram at each edge arrival, combining the histograms of the two nodes involved in the arriving edge, as shown in equation 5.5:

$$\vec{S}_0 \leftarrow \beta \vec{S}_0 + (1-\beta)\left((1-\alpha)\vec{\delta}(f_0) + \alpha \vec{S}_1\right) \tag{5.5}$$

In this way we combine all neighbors' information implicitly using a moving average over time.

Hyperparameter β is another discount factor between 0 and 1, controlling how much to focus on recent information in contrast to older information and which can optionally depend on time. In this way, we can update histograms in a fully streaming setting, using only information of each arriving edge. We term this procedure *Graph-Sprints* and summarize it in algorithm 3.

Compared to equation 5.4, one can observe that the remaining sampling over single-hop

neighbors is abolished, at the cost of imposing a more strict dependence on time. The advantage of algorithm 3 is that no list of neighbors needs to be stored. Moreover, algorithm 3 can be applied in parallel to both the source node and the destination node, and therefore edges are not required to be directed. In fact, while we derived equation 5.5 from random-walks, the attentive reader can notice that it can be interpreted as a special case of message passing where all neighbor summaries are aggregated using a weighted average, with weights that are biased by recency, and where the average is computed in a streaming fashion over time.

One special type of feature are the degree features (in- and out-degree). To avoid accumulating degrees over time, we propose to implement a streaming count of degrees per node. Every time an edge involving node u arrives, we compute

$$d_u = d_u \exp\left(-\Delta t / \tau_d\right) + 1 \tag{5.6}$$

where d_u denotes either in- or out-degree of node u, Δt denotes the time differences between the current edge involving node u and the previous one, and τ_d is a timescale for the streaming counts.

Algorithm 3 *Graph-Sprints*: Real-time graph feature extraction engine (Equation 5.5)

Require: *EdgeStream* ▷ Stream of arriving edges $e_{i,j}$
Require: $\mathcal{F}$ ▷ Set of features for GS (e.g., node degree)

for $e_{v,u} \in EdgeStream$ **do**
  Get $\vec{S}_u, \vec{S}_v$ ▷ Current summaries of nodes u,v
  $\vec{S}_v^\star \leftarrow \alpha \vec{S}_v$ ▷ Multiply all bins by α
  for $f \in \mathcal{F}$ **do**
    if value(f) in bin j **then**
      $\vec{S}_{vj}^\star \leftarrow \vec{S}_{vj}^\star + (1-\alpha)$ ▷ Add (1-α) to bin j
    end if
  end for
  $\vec{S}_u \leftarrow \beta \vec{S}_u + (1-\beta)\vec{S}_v^\star$ ▷ Updated summary of node u.
end for

5.2.2.1 Choosing Histogram Bins

Essential hyperparameters of this method are the choices of the boundaries of the histograms bins. We propose to use one bin per category for categorical features. If the cardinality of a certain feature is too high, we propose to form bins using groups of categories. For numerical features, one can plot the distribution in the training data and choose sensible bin edges, for

example on every 10th percentile of the distribution. The framework is not constrained by one choice of bins, as long as they can be updated in a streaming way.

5.2.3 Streaming Community Features

In our effort to better understand the area around the target node, we introduce two distinctive features, elucidating their integration within our framework. It is worth noting that the computation of these features remains optional, tailored to the specific requirements of the use-case in question. Specifically, the features under discussion are: *community diversity* and *community size*.

Community diversity

To estimate community diversity in the streaming context, we leverage the assumption that for more diverse community there is a higher probability that individual random-walks contain different information. Therefore, we propose a feature that estimates the variety over the various random-walks. The feature is similar to the *inception score* [Charikar, 2002] (or variety score) used to evaluate images generated using generative adversarial networks. The score is calculated as:

$$\text{Vscore} = \frac{1}{N} \sum_i KL(p_i || \sum_i p_i / N) \tag{5.7}$$

In other words, the variety score Vscore of a node is the average (across all random-walks) of the KL divergence between the distribution characterizing an individual random-walk, p_i, to the average of the distributions over all random-walks, $\sum_i p_i / N$.

In our case, we compute the variety score between the various histogram summaries of a node. The above metric can be generalized to use other divergence metrics, e.g. the cosine distance between the histograms. When only a single histogram is kept per node, one can calculate the variety score between the histograms of the direct neighbors of the node of interest. However, if we use Equation 5.5, we only access the current node's histogram $\vec{S}_0$ and the latest neighbor's histogram $\vec{S}_1$ and we can use the following streaming version of Equation 5.7:

$$\text{Vscore}_{t+1} \leftarrow \gamma \text{Vscore}_t + (1 - \gamma) \mathcal{D}(\vec{S}_0, \vec{S}_1) \tag{5.8}$$

Here, $\mathcal{D}$ stands for the divergence measure used (e.g. KL divergence, cosine distance), $\vec{S}_0$ approximates the average of histograms of all neighbors and the variety score is now updated as a moving average of the divergences instead of an actual average over all neighbors.

Finally, when only hashed histograms are used (see Section 5.2.5), one can use e.g. the Hamming, or cosine distance as a divergence metric.

Community size (Path Length)

Community size is more difficult to estimate in the streaming setting. We propose to compute a streaming path length histogram $\vec{p}$, which for each new node is initialized with L bins, where the first bin has a value equal to 1 and the rest are zeros. Every time an edge $e_{u,v}$ arrives from node u to v, one can update the path length histogram of node u as follows

$$\vec{p}_u \leftarrow \beta\vec{p}_u + (1-\beta)\left(\vec{p}_v \cdot U\right) \tag{5.9}$$

This equation is very similar to equation 5.5, but the histogram $\vec{p}_v$ is updated by multiplication with an $(L \times L)$ square matrix U. The matrix U has the diagonal directly above the main diagonal equal to 1, $U_{i,j} = \delta_{i+1,j}$, as well as $U_{L,L} = 1$. In this way, the multiplication by U moves each bin to the right in the histogram, while the last bin acts as an absorbing state. One can then easily verify that each hop results in a histogram with bins shifted to the right, i.e. increasing the path length by one unit, while the last bin accounts for all paths with length larger or equal to L. As before, β ensures that older paths are gradually forgotten and newer information dominates. Clearly, this feature cannot be included in the proposed similarity hashing framework described in Section 5.2.5, since the multiplication by matrix U is not compatible with the hashing method.

5.2.4 GuiltyWalker Features in Streaming Context

In some use-cases, the node features may change values without the node having activity. For instance, in the AML use-case, one node attribute is a label stating whether a node was considered suspicious or not. This information depends on a review process that could happen many days after the node had activity and was incorporated into the graph. *GuiltyWalker* [Oliveira et al., 2021] features, leverage this label information to calculate the distance to illicit nodes and are shown to be helpful to detect suspicious entities in the AML use-case. To deal with this scenario we propose updating the histograms dedicated to such features of the whole K-hop neighborhood once a change happens (e.g., a delayed label arrives), where K will be

a hyperparameter for the *GuiltyWalker* feature. Concretely, the *GuiltyWalker* feature will be a histogram containing two categories, legitimate and suspicious. For nodes of interest, these histograms are updated in the usual way described in the previous section. However, once a label arrives to an existing node, we update the K-hop neighborhood in the following way. We apply the same update formulas and the same α that we used in the previous steps (feature extraction step), but in the opposite direction. Instead of the target node collecting neighbors' histograms to update its histogram, the labeled node sends its histogram to the neighbors to update their histograms. We always maintain the same temporal order i.e., we use older nodes' histograms to update the more recent histograms.

5.2.5 Reducing Memory Footprint

The space complexity of the *Graph-Sprints* approach (algorithm 3) is

$$M = |\mathcal{V}| \sum_{f \in \mathcal{F}} L_f \tag{5.10}$$

where $|\mathcal{V}|$ stands for the number of nodes, L_f stands for the number of bins of the histogram for feature f, and $\mathcal{F}$ stands for the set of features chosen to collect in histograms. In case this memory is too high, we propose the following methods to reduce memory further.

5.2.5.1 Reducing Embedding Size using Similarity Hashing

Following the similarity hashing approach proposed in Jin et al. [2019], we extend the method to the streaming setting. All histograms as defined in the previous sections are normalized (in the sense that bin values sum to 1), and we can concatenate them into one vector $\vec{S}_{tot}$. We can now define a hash mapping by choosing k random hyperplanes in $\mathbb{R}^M$ defined by unit vectors $\vec{h}_j, j = 1, \ldots, k$.

The inner product between the histograms vector and the k unit vectors results in a vector of k values, each value θ_j can be calculated using the dot product of the unit vector $\vec{h}_j$ and the histogram vector $\vec{S}_{tot}$, as illustrated in Equation 5.11. We use the superscript t to denote the current time step.

$$\theta_j^t = \vec{h}_j \cdot \vec{S}_{tot}^t \tag{5.11}$$

One can binarize the representation of the hashed vector by taking the *sign* of the above θ_j^t.

Therefore, the resulting space complexity per node is k, replacing the number of bins in the memory M by the number of hash vectors k.

Importantly, the hashed histograms can be updated without storing any of the original histograms. Combining equations 5.4 and equation 5.11 and denoting $\vec{\delta}(\vec{f})$ the concatenation of the $\vec{\delta}$ vectors for all collected features, we get

$$\theta_j^{t+1} = \theta_j^t \cdot \alpha + \vec{h}_j \cdot \vec{\delta}(\vec{f}) \cdot (1 - \alpha) \tag{5.12}$$

Therefore, we can compute the next hash θ_j^{t+1} or $\text{sign}(\theta_j^{t+1})$ directly from the previous θ_j^t and the new incoming features $\vec{\delta}(\vec{f})$.It is also important to note that this hashing scheme is preserved when averaging.

Below, we show that averaging various histograms followed by hashing is equivalent to hashing histograms and averaging the hashes. Assuming N histograms with L bins $\vec{S}_i$, h an L-dimensional unit vector and $\bigoplus$ representing element-wise addition, we have

$$\begin{aligned}
&\vec{S}_{avg} = \frac{1}{N} \bigoplus_{i=1}^{N} \vec{S}_i \\
&\Rightarrow \theta = \vec{h} \cdot \vec{S}_{avg} = \frac{1}{N} \vec{h} \cdot \left[\sum_{i=1}^{N} s_{i,1}, \ldots, \sum_{i=1}^{N} s_{i,L} \right]^T \\
&\Rightarrow \theta = \frac{1}{N} \cdot \left(\sum_{i=1}^{N} h_1 s_{i,1} + \cdots + \sum_{i=1}^{N} h_L s_{i,L} \right) \\
&\Rightarrow \theta = \frac{1}{N} \sum_{i=1}^{N} \vec{h} \cdot \vec{S}_i = \frac{1}{N} \sum_{i=1}^{N} \theta_i
\end{aligned} \tag{5.13}$$

5.2.5.2 Reducing Embedding Size using Feature Importance

One can reduce the needed memory by relying on feature importance techniques. One possibility is to train a classifier on the raw node and/or edge features and determine feature importances, after which only the top important features are used in the *Graph-Sprints* framework. Or similarly train on all bins and decide the bins to be used based on their importance in the classification task. Thus, either reducing the number of features, or the number of bins within the features, or both.

5.3 Graph-Sprints Theoretical Analysis

5.3.1 Equivalence between Graph-Sprints and Random-walks

In equation 5.3, we discussed a framework approximating random-walk based node embeddings. We can show that this strategy indeed leads to the same result as when using random-walks in expectation. The probability that a path passes through a node a at hop t+1 is given by

$$P_{t+1}(a) = \sum_{b|(a,b)\in E} P(s_a \in \Omega_b) \frac{1}{|\Omega_b|} P_t(b) \tag{5.14}$$

Where $(a, b) \in E$ means (a, b) is an existing edge, i.e., we sum over all neighbors of node a. $P(s_a \in \Omega_b)$ means the probability that the set of summaries Ω_b of node b contain a summary s_a for which the edge (a, b) was the last added information. $|\Omega_b|$ stands for the number of summaries stored at node b and $P_t(b)$ stands for the probability that a path passed through node b at hop t.

The probability that a summary s_a exists in the set of summaries Ω_b of its neighbor b, is related to the number of summaries $|\Omega_b|$ and degree of node b referred to as D_b and can be written as illustrated in equation 5.15.

$$P(s_a \in \Omega_b) = \frac{|\Omega_b|}{D_b} \tag{5.15}$$

Note that in the case that $|\Omega_b| > D_b$, equation 5.15 does not reflect a probability, since in expectation multiple summaries will have node a as last added information, but the equations still hold.

By substituting equation 5.15 in equation 5.14 we result in equation 5.16 which defines a random-walk with unweighted edges.

$$P_{t+1}(a) = \sum_{b|(a,b)\in E} \frac{1}{D_b} P_t(b) \tag{5.16}$$

A weighted version can be straightforwardly implemented by choosing a neighbor not randomly, but according to the edge weights, leading to a factor w_{ab} to be added in equation 5.14 and 5.16.

5.3.2 Graph-Sprints: Complexity Analysis

In this section, we analyse the computational complexity of the proposed *Graph-Sprints* framework.

We start with the complexity of algorithm 3. The multiplication of f vectors with a scalar has complexity $\sum_f L_f$, where L_f stands for the number of bins for feature f and is equal to the dimensionality of the vector. The lookup of the correct bin for a value of feature f is $\mathcal{O}(\log(L_f))$. This has to be repeated for each feature, resulting in a complexity of $\mathcal{O}(\sum_f \log(L_f))$. Finally, the moving average of two vectors has complexity $3\sum_f L_f$. The total complexity of algorithm 3 is therefore $\mathcal{O}(\sum_{\mathbf{f}} \mathbf{L_f})$.

Secondly, for community diversity (equation 5.8), we consider cosine distance as the divergence measure as in our experiments. The cosine distance is computed by performing 3 inner products between the $\sum_f L_f$ dimensional vectors, and therefore has complexity $\mathcal{O}(\sum_f L_f)$.

Lastly, path lengths (equation 5.9) are computed by a matrix-vector multiplication between an $(L_p \times L_p)$ dimensional matrix and a L_p dimensional vector, where L_p are the number of bins summarizing the path lengths. The complexity of such multiplication is $\mathcal{O}(\mathbf{L_p^2})$.

Notably, one can improve upon this by parallelizing over the features f, as well as parallelizing the scalar-vector products and moving average of two vectors, resulting in $\mathcal{O}\left(\max_{\mathbf{f}}(\log \mathbf{L_f})\right)$ for algorithm 3, $\mathcal{O}(\mathbf{1})$ for equation 5.8 and $\mathcal{O}(\mathbf{L_p})$ for equation 5.9.

Once the *Graph-Sprints* features are computed, one needs to pass the resulting feature vector through a classifier. The complexity will depend on the chosen classifier, for instance, in case, a LightGBM model is used, for which $\mathcal{O}(\text{depth})$ is the inference complexity where depth stands for the cumulative depth of the decision trees.

5.4 Experiments & Results

5.4.1 Experimental Setup

We assess the quality of the graph based features generated by the **Graph-Sprints** framework on two different tasks, namely, node classification and link prediction.

5.4.1.1 Baselines

As a first baseline, we reproduce a state-of-the-art GNN model for CTDGs, the *TGN* [Rossi et al., 2020], which leverages a combination of memory modules and graph-based operators to obtain node representations. As an important note, we mention that the pytorch geometric [Fey and Lenssen, 2019] implementation of *TGN* was used, for which the sampling of neighbors uses a different strategy than the original *TGN* implementation. Indeed, the original paper allowed to sample from interactions within the same batch as long as they are older, while the pytorch geometric implementation does not allow within-batch information to be used. As also noted in the pytorch geometric documentation, we believe the latter to be more realistic. As a consequence, our *TGN* results are not directly comparable with the originally published *TGN* performances. In any case, the *Graph-Sprints* embeddings were computed using the same batch size and therefore also do not have access to within-batch information, allowing a fair comparison between the algorithms.

Two variations of the *TGN* architecture were used. First, *TGN-attn* was implemented, which was the most powerful variation in the original paper but is expected to be slower due to the graph-attention operations. Second, *TGN-ID* was implemented, which is a variation of the *TGN* where no graph-embedding operators are used, and only the embedding resulting from the memory module is passed to the classification layers.

A third baseline we use is *Jodie* [Kumar et al., 2019]. We use the *TGN* implementatin of Jodie, where instead of using Graph attention embeddings on top of the memory embedding, a time projection embedding module is used and where the loss function is otherwise identical to the *TGN* setting. For a fair comparison with *TGN* we use the same memory updater module, namely, gated recurrent units.

The *TGN-ID* and *Jodie* baselines do not require sampling of neighbors, and were therefore chosen as lower-latency baselines compared to TGN-attn.

5.4.1.2 Optimization

We use Optuna [Akiba et al., 2019] to optimize the hyperparameters of all models, conducting 100 trials with the Tree-structured Parzen Estimator (TPE) sampler and 40 initial warmup trials. Each model trains using early stopping with a patience of 10 epochs, where the early stopping metric computed on the validation set as AUC for node classification and AP for link prediction.

All models were trained using a batch size of 200 edges. Table 5.1 shows the ranges of the tuned hyperparameters.

TABLE 5.1: Hyperparameters ranges for *Graph-Sprints* and baseline methods.

Method	Hyperparameter	min	max
GS	α	0.1	1
GS	β	0.1	1
GNN/GS	Learning rate	10^{-4}	10^{2}
GNN/GS	Dropout perc	0.1	0.3
GNN/GS	Weight decay	10^{-9}	10^{3}
GNN/GS	Num of dense layers	1	3
GNN/GS	Size of dense layer	32	256
GNN	Memory size	32	256
GNN	Neighbors per node	5	10
GNN	Num GNN layers	1	3
GNN	Size GNN layer	32	256

5.4.1.3 Graph-Sprints and Classifier

For each arriving edge, we apply the *Graph-Sprints* feature update (algorithm 3) to both the source node and the destination node in parallel. All edge features are used for the computation of the *Graph-Sprints* features, and for each feature bin edges are chosen as the 10 quantiles computed on the training data. Since the *Graph-Sprints* framework only creates features, a classifier is implemented for the classification tasks. We chose to implement a neural network consisting of dense layers, normalization layers, and skip-connections across every two dense layers. Hyperparameter optimization proceeds in two steps. First, default parameters for the classifier are used to optimize the discount factors of the *Graph-Sprints* framework, α and β. For this step, 50 models are trained. Subsequently, hyperparameter optimization of the classifier follows same approach as TGN, training 100 models.

In all experiments, we test the following three cases. Firstly, we train the classifier using only raw features (Raw). We then train the classifier using only the *Graph-Sprints* features (GS). Finally, we train the classifier using both raw and *Graph-Sprints* features (GS+Raw).

5.4.1.4 Node Classification vs Link Prediction

For the node classification task on the Wikipedia, Reddit and Mooc datasets, we concatenate the source and destination node embeddings and feed the concatenated vector to the classifier, as is usual for datasets where labels are on the edge level.

For the link prediction task, negative edges are generated following the same approach as the original *TGN* paper [Rossi et al., 2020], a negative edge is sampled for every positive one. We perform the link prediction task both in the *transductive* and *inductive* settings. In the *transductive* setting, negative edges are sampled on the same graph used for training. Conversely, in the *inductive* setting, the sampled negative edges are constrained to include at least one new node which was not used in the training graph.

5.4.2 Public Datasets Experiments

5.4.2.1 Datasets

We use three publicly available datasets [Kumar et al., 2019] from the social and education domains. We detail their main characteristics in Table 5.2. All datasets are CTDGs and are labeled. Each dataset is split into train, validation, and test sets respecting time (i.e., all events in the train are older than the events in validation, and all events in validation are older than the events in the test set). In the public datasets, we adopt the identical data partitioning strategy employed by the baseline methods we compare against, which also utilized these datasets.

TABLE 5.2: Information and data partitioning strategy for public datasets.

	Wikipedia	**Mooc**	**Reddit**
#Nodes	9,227	7,047	10,984
#Edges	157,474	411,749	672,447
Label type	editing ban	student drop-out	posting ban
Positive labels	0.14%	0.98%	0.05%
Used split (%)	70-15-15	60-20-20	70-15-15

5.4.2.2 Task Performance

In Table 5.3 we report the average test AUC ± std for the **Node classification** task. Our approach involved retraining the best model obtained after hyperparameter optimization, using 10 different random seeds. Our models, Raw, GS, and GS+Raw, use the same ML classifier but differ in the features employed for training. Raw uses raw edge features, GS uses *Graph-Sprints* histograms, and GS+Raw combines both. We identify the **best** model and highlight the second best model. We can observe that on all datasets, the best model for node classification uses a variation of our Graph-Sprint framework (either GS or GS+Raw). To provide an overview, we include a column showing the average rank in Table 5.3, which represents the mean ranking computed from all datasets.

TABLE 5.3: *Graph-Sprints:* Node classification results using public datasets.

Method	AUC ± std			Average rank
	Wikipedia	Mooc	Reddit	
Raw	58.5 ± 2.2	62.8 ± 0.9	55.3 ± 0.8	6
TGN-ID	88.9 ± 0.2	63.0 ± 17	61.3 ± 2.0	4.3
Jodie	87.2 ± 0.9	63.7 ± 16.7	61.9 ± 2.0	4
TGN-attn	86.6 ± 2.8	75.8 ± 0.4	67.9 ± 1.6	3
GS	**90.7 ± 0.3**	75.0 ± 0.2	**68.5 ± 1.0**	1.6
GS+Raw	89.2 ± 0.4	**76.5 ± 0.3**	63.7 ± 0.4	2

In table 5.4 we report the average test AUC ± std, along with the AP ± std for the **Link prediction** task. Results were again computed after retraining the best model obtained through hyperparameter optimization, utilizing 10 distinct random seeds. We report results on both *Transductive (T)* or *Inductive (I)* settings. We can observe that the *Graph-Sprints* model is the best for link prediction on the Mooc dataset. On the Reddit dataset, the *Graph-Sprints* model is best in the transductive setting, and the second best in the inductive settings. In the Wikipedia dataset, the performance is slightly worse than the best baselines. To offer a comprehensive view, we have included a column in Table 5.4 that displays the average rank. This represents the mean ranking derived from all datasets, calculated using AP.

TABLE 5.4: *Graph-Sprints:* Link prediction results using public datasets.

	Method	Wikipedia		Mooc		Reddit		Average rank
		AUC	AP	AUC	AP	AUC	AP	
T	**TGN-ID**	95.6 ± 0.2	95.8 ± 0.1	80.4 ± 5.8	75.0 ± 6.1	94.7 ± 0.7	93.2 ± 1.0	4
	Jodie	94.3 ± 0.3	94.5 ± 0.3	85.1 ± 1.8	80.0 ± 3.5	94.9 ± 1.2	93.4 ± 1.7	3
	TGN-attn	**97.0 ± 0.3**	**97.3 ± 0.3**	80.3 ± 8.1	75.6 ± 8.4	96.1 ± 0.4	95.1 ± 0.7	2.6
	GS	92.5 ± 0.6	92.9 ± 0.7	82.7 ± 0.8	81.1 ± 0.7	96.1 ± 0.2	95.3 ± 0.3	3
	GS+Raw	92.1 ± 0.4	92.6 ± 0.4	**85.4 ± 0.3**	**83.7 ± 0.3**	**96.8 ± 0.1**	**96.1 ± 0.2**	2.3
I	**TGN-ID**	92.2 ± 0.2	92.8 ± 0.2	68.5 ± 8.6	63.5 ± 6.7	93.4 ± 0.6	92.2 ± 0.8	3.6
	Jodie	87.0 ± 0.6	89.1 ± 0.7	71.1 ± 2.2	66.1 ± 2.9	92.3 ± 1.3	90.8 ± 1.9	4.6
	TGN-attn	**94.5 ± 0.2**	**95.0 ± 0.2**	71.4 ± 4.1	66.9 ± 3.9	**95.0 ± 0.4**	**94.3 ± 0.5**	2.6
	GS	92.0 ± 0.3	91.7 ± 0.4	78.2 ± 0.6	76.5 ± 0.6	92.7 ± 0.5	92.7 ± 0.6	2.6
	GS+Raw	91.4 ± 0.2	91.1 ± 0.3	**83.0 ± 0.5**	**80.3 ± 0.5**	93.5 ± 0.4	92.2 ± 0.5	2.3

5.4.2.3 Inference Runtime

We compare the latency of our framework to baseline GNN architectures. For this purpose, we run 200 batches of 200 events on the external datasets, Wikipedia, Mooc, and Reddit using the node classification task. We compute the average time over 10 runs. Both models were running on Linux PC with 24 Intel Xeon CPU cores (3.70GHz) and a NVIDIA GeForce RTX 2080 Ti GPU

(11GB). As depicted in Figure 5.4, our *Graph-Sprints* consistently outperforms other baselines (*TGN-attn*, *TGN-ID*, *Jodie*) in the node classification task while also demonstrating a significantly lower inference latency. Compared to TGN-attn, the GS achieves better classification results but is close to an order of magnitude faster (Figure 5.4).

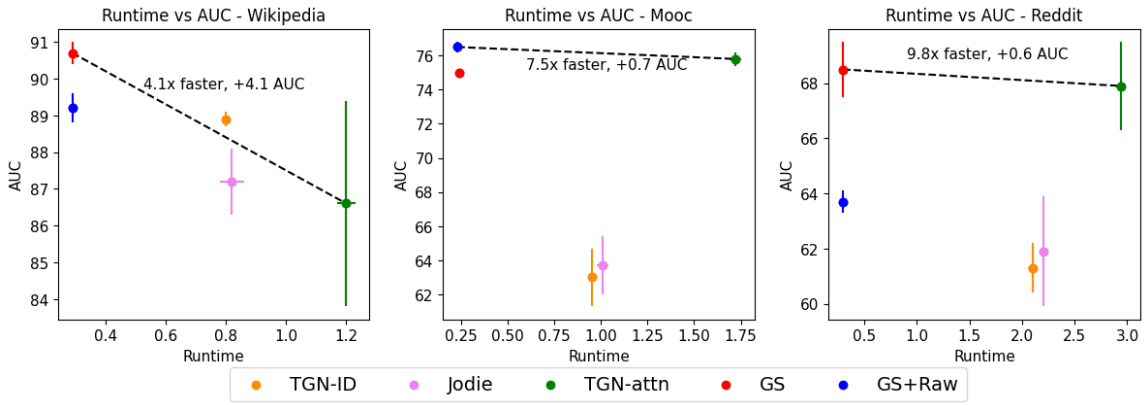


FIGURE 5.4: *Graph-Sprints:* Trade-off between AUC and runtime.

To investigate the impact of graph size on runtime, Figure 5.5 showcases our observations. Notably, in the utilized datasets, TGN's runtime increases as the number of edges in the dataset grows, requiring more time to score 200 batches. Conversely, since GS does not require neighborhood sampling, it exhibits constant inference time regardless of the graph size. Furthermore, the speedups achieved by *Graph-Sprints* are expected to be significantly higher in a big-data context, where the data is stored in a distributed manner rather than in memory as in our current experiments. In such scenarios, graph operations used in graph-neural networks like *TGN-attn* would incur even higher computational costs.

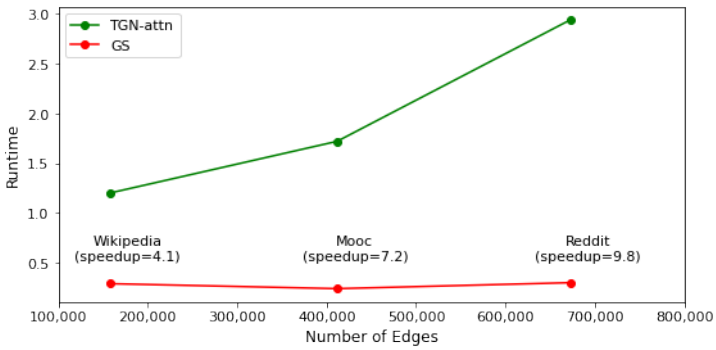


FIGURE 5.5: *Graph-Sprints:* Speedup vs. number of edges: The speedups increase almost linearly with the number of edges in the graph.

Recently, APAN [Wang et al., 2021b] has attempted to build a low-latency framework for CTDGs. Their approach consisted of performing the expensive graph operations asynchronously, out of the inference loop. In that way, they achieved inference speeds of 4.3ms per batch on the

Wikipedia dataset, but we cannot directly compare those results with ours (GS: 1.4ms, TGN-attn: 6ms) due to the different setup and hardware. Importantly, their approach achieves low-latency by sacrificing up-to-date information at inference time. Indeed, the inference step is performed without access to the most recent embeddings, because the expensive graph operations to compute the embeddings are performed asynchronously.

5.4.2.4 Memory Reduction

Both the Wikipedia and Reddit datasets consist of 172 edge features. By calculating *Graph-Sprints* with 10 quantiles per feature, along with incorporating in/out degrees histograms and time-difference histograms, we obtain a node embedding of 1742 features (one feature per histogram bin). In our experimental setup, similar to state-of-the-art approaches, we concatenate the source and destination node embeddings for source label prediction, resulting in a 3484-feature vector. To reduce the size of the node embeddings, we propose a similarity hashing-based memory reduction technique (Section 5.2.5)). Our experiments, as presented in Table 5.5, demonstrate that our technique reduces storage requirements sacrificing the AUC in the node classification task. In the Reddit dataset, storage can be reduced to 50% with a 0.6% AUC sacrifice or to 10% with a 2% AUC sacrifice. In the Mooc dataset we can reduce necessary memory to 25% with a 1% sacrifice in AUC. In the Wikipedia dataset, a reduction in storage to a only 0.12% of the original features with a 4.3% AUC sacrifice. The reduction percentage can be fine-tuned as a hyperparameter, considering the use case and dataset, to strike a balance between precision and memory trade-off.

TABLE 5.5: *Graph-Sprints:* Impact of memory reduction on node classification performance.

Space used	Wikipedia	Mooc	Reddit
100%	90.7 ± 0.3	75.0 ± 0.2	68.5 ± 1.0
50%	90.8 ± 0.1	75.0 ± 0.1	67.9 ± 1.1
25%	91.1 ± 0.1	74.9 ± 0.3	65.1 ± 1.9
10%	90.9 ± 0.2	74.0 ± 0.3	66.5 ± 0.9
0.5%	89.7 ± 0.3	-	58.0 ± 2.6
0.12%	86.4 ± 0.3	-	55.2 ± 1.1

5.4.3 AML experiments

In money laundering, the criminals' objective is to hide the illegal source of their money by moving funds between various accounts and FIs. In these experiments, our objective is to enrich

a triage classifier (detailed in Chapter 4, Section 4.3) with graph-based features generated by our *Graph-Sprints* framework.

5.4.3.1 Datasets

We evaluate the *Graph-Sprints* framework in the AML domain using two real-world banking datasets. Due to privacy concerns, we can not disclose the identity of the FIs nor provide exact details regarding the node features. We refer to the datasets as *FI-A* and *FI-B*. The graphs in this use-case are constructed by considering the accounts as nodes and the money transfers between accounts as edges. Table 5.6 shows approximate details of these datasets.

TABLE 5.6: Information and data partitioning strategy for AML datasets.

	FI-A	**FI-B**
#Nodes	≈400000	≈10000
#Edges	≈500000	≈2000000
Positive labels	2-5%	20-40%
Duration	≈300 days	≈600 days
Edges/day (mean ± std)	1500 ± 750	3000 ± 5000
Used split	60%-10%-30%	60%-10%-30%

5.4.3.2 Task Performance

As before, we train the neural network classifier that uses raw node features only, i.e., no graph information is present (Raw). We compare that baseline performance against models that include only *Graph-Sprints* features (GS), and models that use both *Graph-Sprints* features and raw features (GS+Raw). Finally, we train the same GNN architectures as in the public datasets (*TGN-ID*, *Jodie*, and *TGN-attn*).

Due to privacy considerations, we are unable to disclose the actual obtained AUC values. Instead, we present the relative improvements in AUC (ΔAUC) when compared to a baseline model that does not utilize graph features. In this context, the baseline model corresponds to a ΔAUC value of 0, and any increase in AUC compared to the baselines is represented by positive values of ΔAUC.

Table 5.7 displays the average ΔAUC test values ± std achieved by retraining the best model after hyperparameter optimization using 10 random seeds. We identify the **best** model and highlight the second best model. We compare our GS variations and other state-of-the-art baselines. Our GS variations exhibit the most favorable outcomes in both datasets, with an

approximate 3.3% improvement in AUC for the *FI-A dataset* and a 27.8% improvement in AUC for the *FI-B dataset*. To provide an overview, we include a column showing the average rank which represents the mean ranking computed from the two datasets.

TABLE 5.7: *Graph-Sprints:* Node classification results using AML datasets.

Method	**ΔAUC ± std**		**Average rank**
	FI-A	**FI-B**	
TGN-ID	+0.1 ± 0.1	+24.4 ± 0.2	4
Jodie	+0.0 ± 0.1	+24.5 ± 0.2	4
TGN-attn	+0.3 ± 0.7	<u>+25.1 ± 0.3</u>	2.5
GS	<u>+1.8 ± 0.5</u>	**+27.8 ± 0.4**	1.5
GS+Raw	**+3.3 ± 0.3**	+20.1 ± 3.9	3

5.5 Summary

This chapter introduced the *Graph-Sprints* framework, which enables the computation of time-aware embeddings for CTDGs with minimal latency. The study demonstrates that the *Graph-Sprints* features, when combined with a neural network classifier, achieve competitive predictive performance compared to state-of-the-art methods while having a significantly faster inference time, up to approximately an order of magnitude improvement.

In future work, it would be interesting to extend the *Graph-Sprints* framework to heterogeneous graphs, and explore how GNNs could inherit some of the strengths of Graph-Sprints.

Chapter 6

Deep-Graph-Sprints: Low-latency Node Representation Learning method

Our investigations in Chapters 4 and 5 have demonstrated the promising knowledge embedded in graphs representing data. These findings underscore the importance of graph feature engineering and representation learning in translating graph relationship information into an embedding space, facilitating its use in ML models.

In Chapter 5, we introduced *Graph-Sprints*, a rapid graph feature engineering method. *Graph-Sprints*, showed a high speed and competitive performance with state-of-the-art methods. However, it is essential to address the inherent limitations associated with its feature engineering nature. These include the requirement for distinct tuning phases for parameters and a dependence on domain expertise. Furthermore, *Graph-Sprints* uniformly applies the same forgetting coefficient across features, which may not be optimal. Moreover, the tendency to produce a high-dimensional embeddings, poses a memory challenge (detailed in Section 6.1).

Deep learning emerges as a solution to these challenges, offering automatic parameter learning, enhanced model accuracy through complex data relationship learning, and adaptability in dynamic environments. It also simplifies model development by reducing routine extract-transform-load tasks and infrastructure complexities, enabling direct feature extraction from raw data.

This chapter focuses on enhancing *Graph-Sprints*, our graph feature engineering method

discussed in Chapter 5, through the development of a deep learning-enhanced variant, *Deep-Graph-Sprints*. This new iteration aims to overcome the limitations of the original method, enhancing its practical application and effectiveness. The chapter is structured as follows:

- **Graph-Sprints recap and limitations:** Section 6.1 begins with a summary of *Graph-Sprints* and its limitations.
- **Deep-Graph-Sprints method:** Section 6.2 details the development of the *Deep-Graph-Sprints* method, including its architecture in Section 6.2.1, learning mechanisms in Section 6.2.3, gradients calculation details in Section 6.2.4, and parameter updates in Section 6.2.5.
- **Experiments and results:** Section 6.3 concludes with our findings, showcasing the efficiency and robust performance of *Deep-Graph-Sprints* compared to traditional GNNs, and its competitive edge over *Graph-Sprints* while addressing its limitations.

6.1 Graph-Sprints Recap and Limitations

Prior to delving into the *Deep-Graph-Sprints* methodology, a review of the foundational *Graph-Sprints* approach, as elaborated in Chapter 5, is essential. The *Graph-Sprints* algorithm updates a node's state at time t, represented as $\vec{S}_t$, by employing Formula 6.1. It integrates the previous state of the target node ($\vec{S}_{t-1}$), the interacting node's state ($\vec{S}^*_{t-1}$), and the new edge features (F_t).

The parameters α and β serve as forgetting coefficients. Specifically, β modulates the balance between a node's past state ($\vec{S}_{t-1}$) and its present state, facilitating a balance between historical and current data. Conversely, α adjusts the emphasis between self-information and neighboring node information. Concerning the $\vec{\delta}$ function, it functions as an encoding mechanism. Given the features values, $\vec{\delta}$ constructs a series of concatenated histograms. Each histogram corresponds to a distinct feature, where the bin corresponding to the actual value of the feature is marked with a '1', other bins have a value '0'.

$$\vec{S}_t = \beta\vec{S}_{t-1} + (1-\beta)\left((1-\alpha)\vec{\delta}(F_t) + \alpha\vec{S}^*_{t-1}\right) \tag{6.1}$$

Although *Graph-Sprints* exhibits rapid processing capabilities and is competitive with state-of-the-art techniques, a critical examination of its limitations reveals several challenges in practical applications:

- **Tuning as Separate Processes**: A primary challenge in *Graph-Sprints* is the complex tuning required for the feature engineering parameters, particularly α and β. The process necessitates distinct tuning phases for the feature extraction component (*Graph-Sprints*) and the subsequent decision-making model, such as a neural network classifier. This tuning involves initially adjusting the α and β parameters of *Graph-Sprints* using a model with default settings. Following this, the model is fine-tuned using the optimized α and β values. This iterative process, due to its independence of steps, can be time-consuming and labor-intensive. Section 6.2.2.1 explains how *Deep-Graph-Sprints* addresses this concern.

- **Limited Model Expressivity**: In *Graph-Sprints*, the α and β parameters (i.e., forgetting coefficients) are scalars, leading to a uniform forgetting coefficient being applied across all features. This uniform application potentially constrains the model's expressivity. For instance, in scenarios where the immediacy of information from one feature outweighs that of another, the model's inability to differentiate between these varying temporal relevancies due to the scalar nature of the forgetting coefficients becomes evident. Similarly, the differentiation between attributes of a node and those of its neighbors is limited when only a single forgetting coefficient is employed. Such uniformity in temporal and relational weighting diminishes the model's capacity to distinctly represent and process the temporal dynamics inherent in different features or relational contexts. The approach of *Deep-Graph-Sprints* to this issue is explained in Section 6.2.2.2.

- **Histogram Bin Edge Definition Challenges**: A significant limitation within the *Graph-Sprints* methodology arises from the requirement to specify bin edges for histograms in the feature encoding function $\vec{\delta}$. This process, crucial for representing each feature accurately, can be approached either by utilizing domain expertise or through an automated tuning procedure. Regardless of the chosen method, this task proves to be particularly burdensome in scenarios involving datasets with a high feature count. When domain knowledge is applied, the challenge lies in accurately determining the appropriate bin edges that meaningfully represent the feature distribution. Conversely, if an automated tuning approach is selected, it often involves a computationally intensive process, as

it requires iterative adjustments to find the optimal bin configuration. The technique *Deep-Graph-Sprints* uses to resolve this matter is elaborated in Section 6.2.2.3.

- **Large State Size**: The decision to employ a high number of bins per histogram in *Graph-Sprints* results in substantially large node states, escalating both memory and space requirements. Illustratively, a dataset featuring 100 distinct attributes, with each attribute discretized into 10 bins, results in a state representation consisting of 1000 bins. This complexity is further increased in tasks such as link prediction, wherein the states of both source and destination nodes are requisite, effectively doubling the ML model's input to a 2000-dimensional vector. Moreover, managing the state size emerges as a significant challenge, necessitating a thorough calibration of bin edges for each attribute. To mitigate these issues, we have introduced memory reduction techniques as outlined in Section 5.2.5. However, these techniques are not without their limitations, particularly due to the prerequisite of forming histograms prior to executing similarity hashing. Additionally, in the context of using feature importance to reduce dimensionality, training a model with the complete histograms is still required to determine feature significance. The approach of *Deep-Graph-Sprints* to this issue is explained in Section 6.2.2.3.

- **Challenges in Adapting to Heterogeneous Graphs**: Adapting *Graph-Sprints* to heterogeneous graphs, which include different node and edge types, presents challenges not addressed by the current model. These challenges include integrating states from various types, where achieving a consistent encoding dimension and interpreting bins across types is complex. Another challenge involves developing distinct forgetting coefficients for each node type, coupled with a feature encoding process that depends on the node type. This leads to an expansion in the parameter space that requires detailed tuning. The increased parameter diversity calls for advanced tuning strategies. These challenges limit the broader applicability of *Graph-Sprints* to heterogeneous graphs. We detail how *Deep-Graph-Sprints* could address this in Section 6.5.

The aforementioned limitations underscore the need for an adaptation of the *Graph-Sprints* methodology. Such enhancements are vital to augment its efficacy and broaden its applicability across a spectrum of real-world scenarios. Recognizing these challenges, this chapter introduces a significant evolution of the *Graph-Sprints* method: the *Deep-Graph-Sprints*. This adaptation is designed to address the previously mentioned limitations, offering a more robust and versatile

solution. In the following sections, we will delve into the details of *Deep-Graph-Sprints*, providing an in-depth analysis and assessment of its architecture, functionalities, and performance.

6.2 Method

This section introduces an advanced variation of the *Graph-Sprints* method, leveraging deep learning techniques to learn its parameters and address the limitations mentioned in Section 6.1.

We propose a three-step methodology, each step representing an incremental increase in complexity. The first step aims to automate the learning of the scalar values α and β, thereby streamlining the tuning process. In the second step, we enhance the model's expressivity by evolving α and β from scalars to vectors, thereby allowing a unique forgetting parameter for each feature. This adaptation significantly increases the algorithm's flexibility and adaptability. The final step further extends the model's sophistication: in addition to learning vectorized forgetting coefficients ($\vec{\alpha}$ and $\vec{\beta}$), it replaces the encoding function $\vec{\delta}$ with a learnable mapping from feature space to embedding space. This alteration not only reduces the dimensionality of the resultant state but also eliminates the need for histogram bin edge determination, thereby simplifying the feature engineering process and reducing the memory footprint of the model. Section 6.2.2 details each step, explaining their methodologies and principles.

6.2.1 Architecture and Workflow

The *Deep-Graph-Sprints* is developed to handle a continuous flow of edge data. As shown in Figure 6.1, the system processes each incoming edge to derive a task-specific score, applicable for any ML task such as classification.

The *Deep-Graph-Sprints* method is divided into two key components:

1. **Embedding Component (DGS):** This segment is dedicated to representation learning, where each node or edge in the graph is mapped from high-dimensional, complex graph structures to a lower-dimensional embedding space. Refer to Figure 2.6 for an illustration of node embedding.
2. **Neural Network Classifier (NN):** This part is responsible for decision-making processes, such as classification. It uses the embedding provided by the DGS component to generate a task specific score.

The DGS component is particularly noteworthy for its role in regularly updating the embeddings of nodes or edges, thereby enriching them with detailed attributes and relationships context within the network.

These embeddings are then input into the neural network classifier, which is tailored to specific applications. The classifier assigns a score based on the comprehensive data contained within the embeddings. For instance, in node classification, the network evaluates each node associated with a new edge, with the score reflecting the network's interpretation from the representations provided by the DGS.

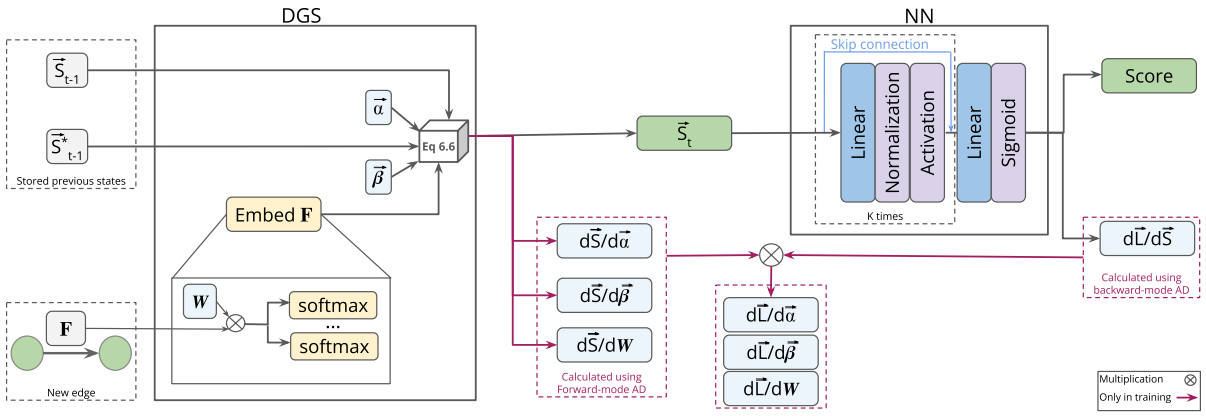


FIGURE 6.1: *Deep-Graph-Sprints:* Architecture. Eq 6.6 in the figure, represents Equation 6.6

6.2.2 Deep-Graph-Sprints Approaches

The integration of deep learning into *Graph-Sprints* primarily aims to automate the learning of parameters during training. This eliminates the need for manual tuning and domain-specific knowledge for setting hyperparameter values. Referring to the *Graph-Sprints* formula (see Equation 6.1), our focus is on learning the hyperparameters α, β, and the encoding function $\vec{\delta}$.

The process is categorized into three approaches, progressing from simple to complex methods.

6.2.2.1 DGS-1: Learning Scalar Parameters α and β

This approach, aligned with the *Graph-Sprints* model (refer to Equation 6.1), integrates the learning of hyperparameters α and β into the neural network training phase. This method eliminates the need for separate tuning processes. Initially set at 0.5, both α and β are refined during training, akin to neural network weights, to reduce loss and enhance model efficacy. As

scalar values within the [0, 1] range, α and β maintain consistency with the *Graph-Sprints* method, uniformly applying the forgetting coefficient to all features. This integration ensures that the optimization of α and β occurs concurrently with the neural network's parameter adjustments, thereby facilitating an end-to-end training process.

6.2.2.2 DGS-2: Learning Vectorized Parameters $\vec{\alpha}$ and $\vec{\beta}$

In this advanced *Deep-Graph-Sprints* variant, we enhance the learning mechanism by vectorizing α and β into $\vec{\alpha}$ and $\vec{\beta}$, thereby assigning unique forgetting coefficients to each feature. As outlined in Equations 6.2a and 6.3a, this vectorization aligns with the feature count f, marking a departure from the original scalar-based *Graph-Sprints* model. For implementation, coefficients within $\vec{\alpha}$ and $\vec{\beta}$ are repeated according to the bin count b_i for each i^{th} feature, as shown in Equations 6.2b and 6.3b. This ensures they are compatible with the dimensions of the state vector. Specifically, for each feature i, individual forgetting coefficients α_i, and β_i are learned and subsequently replicated b_i times, resulting in vectors $\vec{\alpha_i}$, and $\vec{\beta_i}$. These vectors are then concatenated to form the comprehensive forgetting coefficient vectors $\vec{\alpha}$ and $\vec{\beta}$, as detailed in Equations 6.2a and 6.3a.

$$\vec{\alpha} = \bigoplus_{i=1}^{f} \vec{\alpha_i} \tag{6.2a}$$

$$\vec{\alpha_i} = \alpha_i \cdot \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}_{b_i \times 1} \tag{6.2b}$$

$$\vec{\beta} = \bigoplus_{i=1}^{f} \vec{\beta_i} \tag{6.3a}$$

$$\vec{\beta_i} = \beta_i \cdot \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}_{b_i \times 1} \tag{6.3b}$$

This modification allows for distinct forgetting coefficients per feature, enhancing the model's adaptability. The application of this refined forgetting mechanism involves an element-wise multiplication (Hadamard product [Horn, 1990]) between $\vec{\alpha}$, $\vec{\beta}$, and the corresponding state and feature encodings, as described in Equation 6.4.

$$\vec{S}_t = \vec{\beta} \odot \vec{S}_{t-1} + (1 - \vec{\beta}) \odot \left((1 - \vec{\alpha}) \odot \vec{\delta}(F_t) + \vec{\alpha} \odot \vec{S}^*_{t-1} \right) \tag{6.4}$$

In this configuration, each feature retains a singular forgetting coefficient (consistent across all its bins) to ensure that histograms within every feature sum to one, thus preserving the integrity of the state update formula (Equation 6.1).

Analogous to the scalar learning method in DGS-1, these vectorized parameters are learned during the training process, informed by the neural network's loss function, thus enabling an end-to-end training. Moreover, the added advantage of assigning different forgetting coefficients to individual features considerably elevates the representational capacity of the embeddings. By providing a feature-specific forgetting mechanism, it more precisely reflects the distinct characteristics of each feature. This is especially critical in scenarios where the significance of recent information varies distinctly among different features. Such a tailored approach ensures that the model not only maintains its efficiency and simplicity but also gains in adaptability and accuracy, particularly in complex data environments.

Detailed insights into the learning mechanism and the computation of gradients for these vectorized parameters are elaborated in Sections 6.2.3 and 6.2.4.

6.2.2.3 DGS-3: Advanced Learning of Feature Embeddings W

In this advanced version of *Deep-Graph-Sprints*, DGS-3, the model undergoes a pivotal transformation by replacing the $\vec{\delta}$ function with an embedding matrix W. This matrix W serves to project features into an embedding space, subsequently updating the state with these derived embeddings.

A key aspect of this approach is ensuring that the generated embeddings are compatible with our foundational method's requirements. Each embedding should ideally be a set of histograms, with each histogram's values summing to one and remaining within the [0, 1] range.

Considering a single histogram, traditional normalization methods like Min-Max or absolute value normalization can maintain values within this range but do not guarantee their summation to one. An alternative method of normalizing each element by the total sum can be employed, yet this may not consistently preserve the 0 to 1 range, especially if negative values are present. A preliminary normalization or adjustment by the absolute magnitude of the minimum value could be applied, but this method might change the original meaning of the data and is affected by outliers.

A more effective solution is the utilization of the softmax function ($\vec{\sigma}$), as expressed in Equation 6.5. The softmax function converts a vector into a probability distribution, where the sum of all probabilities equals to one and each element lies between 0 and 1, thereby offering a probabilistic interpretation.

In DGS-3, the softmax function is selected for its aforementioned benefits, though exploration of other methods is reserved for future research.

$$\vec{\sigma}(\vec{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{6.5}$$

DGS-3 enhances expressiveness and memory utilization by employing multiple softmax functions, each applied to the product of a segment of the embedding matrix W_i and feature vector F_t. The use of multiple softmax functions helps in reducing Jacobian dimensionality (as elaborated in Section 6.2.4), potentially leading to improved memory efficiency. Additionally, this approach allows for a structure where different segments of the model can respond to various aspects of the input data. This is analogous to the way multi-head attention mechanisms in transformers [Vaswani et al., 2017] operate, where different 'heads' learn distinct mappings or functions based on the same input data. For a detailed understanding of how each softmax function is applied and to visualize the parallelism in learning across these segments, readers are directed to Section 6.2.4.3, where a comprehensive example is provided.

The count of softmax functions (x), a crucial hyperparameter, governs the transition from feature space to embedding space (WF_t). The outputs from the softmax functions are concatenated, forming a feature representation akin to that in DGS-1 and DGS-2, with each softmax function now paralleling a feature.

Consistent with DGS-2, DGS-3 maintains α and β as vectors, assigning one value per softmax function, contrasting the one value per feature in DGS-2.

The state update mechanism is detailed in Equation 6.6, where $\vec{\sigma}$ signifies the softmax function (utilized in a vector form notation to clarify the nature of the output), $\bigoplus_{i=1}^{x}$ represents concatenation of x softmax function outputs, and $W_i F_t$ indicates the product of a subset of the embedding matrix W_i and the features F_t. The practical application of this mechanism, including the computation and integration of these components, is comprehensively outlined in Algorithm 4.

$$\vec{S}t = \vec{\beta} \odot \vec{S}_{t-1} + (1 - \vec{\beta}) \odot \left((1 - \vec{\alpha}) \odot \left(\bigoplus_{i=1}^{x} \vec{\sigma}(W_i F_t) \right) + \vec{\alpha} \odot \vec{S}^{*}_{t-1} \right) \tag{6.6}$$

This architectural innovation addresses key limitations of the *Graph-Sprints* model. It removes the dependency on domain-specific knowledge for setting bin edges and allows for more precise control over the size of the resultant state or embedding, governed by the dimensions of the embedding matrix W. This approach also advances adaptability and flexibility in handling heterogeneous graph structures as discussed in Section 6.5. Algorithm 4 details state calculation during inference.

Algorithm 4 *Deep-Graph-Sprints*: Graph Representation Learning (Equation 6.6)

Require: $EdgeStream$ ▷ Stream of arriving edges $e_{i,j}$
Require: f ▷ Number of input features
Require: s ▷ Embedding size
Require: x ▷ Number of softmax functions
Require: $W[s \times f]$ ▷ Learnt embedding matrix
Require: $\vec{\alpha}$ ▷ Learnt forgetting coefficient
Require: $\vec{\beta}$ ▷ Learnt forgetting coefficient

$W_{reshape} \leftarrow W.\text{reshape}(x, s/x, f)$ ▷ Reshape W for softmax application
for $e_{v,u} \in EdgeStream$ **do**
  Get $\vec{S}_u, \vec{S}_v$ ▷ Summaries of nodes u, v $(t-1)$
  Get F_t ▷ Features of edge $e_{v,u}$
  $enc\_feats \leftarrow \text{MatrixMultiply}(W_{reshape}, F_t)$ ▷ Encode features
  $norm\_feats \leftarrow \vec{\sigma}(enc\_feats, \text{axis} = -1)$ ▷ Normalize features
  $norm\_feats \leftarrow norm\_feats.reshape(\text{s})$ ▷ Reshape Normalized features
  $\vec{S}_u \leftarrow \vec{\beta} \odot \vec{S}_u + (1 - \vec{\beta}) \odot ((1 - \vec{\alpha}) \odot norm\_feats + \vec{\alpha} \odot \vec{S}_v)$ ▷ Update node u summary
  $\vec{S}_v \leftarrow \vec{\beta} \odot \vec{S}_v + (1 - \vec{\beta}) \odot ((1 - \vec{\alpha}) \odot norm\_feats + \vec{\alpha} \odot \vec{S}_u)$ ▷ Update node v summary
end for

6.2.3 Learning Mechanisms in Deep-Graph-Sprints

Given the DGS-3 architecture discussed above, we will now delve into the learning mechanisms for the *Deep-Graph-Sprints* parameters. As we will discuss in the next paragraphs, the *Deep-Graph-Sprints* architecture allows for an efficient implementation of mixed-mode AD.

As discussed in Section 2.2.2, three strategies for AD are prominent, depending on the application of the chain rule: forward-mode AD, reverse-mode AD (commonly known as backpropagation), and a mixed-mode that combines elements of both.

In contexts involving temporal data, particularly with GNNs applied to temporal graphs or RNNs, backpropagation presents significant memory challenges. Specifically, the whole subgraph that is in the causal past of an event (i.e. the whole subgraph that could have influenced the current event) for GNNs, or sequence for RNNs must be maintained in memory to facilitate backpropagation of Jacobians. Memory usage scales with computation length, necessitating the storage of all intermediate values.

This requirement often becomes a bottleneck in terms of memory and computational efficiency. As a workaround, truncated backpropagation is employed, allowing the use of only part of the sequence or graph for Jacobian propagation. Several GNN algorithms, such as TGN [Rossi et al., 2020], implement truncated backpropagation to limit the backward phase to a single step, covering a one-hop neighborhood. While effective in reducing memory load, this truncation compromises the learning of long-term dependencies.

Conversely, forward-mode AD does not require tracking of intermediate values, eliminating the need to traverse back through the entire graph or sequence. However, in typical ML architectures where the output dimension (e.g., loss) is much smaller than the input dimension, forward-mode AD becomes more computationally intensive than reverse-mode AD.

The complexity of forward-mode AD limits its applicability in typical ML scenarios, as detailed in Section 2.2.2.2. Nonetheless, forward-mode AD is applicable in situations requiring a manageable number of Jacobian computations, offering efficient Jacobian propagation through computational graphs, and learning long term dependencies.

In the *Deep-Graph-Sprints* method, the feasibility of forward-mode AD is supported by three considerations. First, in *DGS-1*, the scalar nature of α and β reduces computational complexity. Second, *Deep-Graph-Sprints* is dominated by elementwise multiplications, where

different elements of a state vector are not mixed together, in *DGS-2* and *DGS-3*, each feature or softmax function corresponds to a single scalar within $\vec{\alpha}$ and $\vec{\beta}$, constraining the dependency of every bin in the state vector to specific bins in these vectors. Third, the implementation of multiple softmax functions in *DGS-3* limits each state element's dependency to segments of the embedding matrix W. The subsequent sections will address the calculation specifics and analyze the computational and memory complexities in Section 6.2.4.

These factors collectively justify the selection of forward-mode AD for the differentiation process in the DGS models.

Consequently, given our architecture illustrated in Section 6.2.1 and Figure 6.1, our method consists of two components, the DGS component which could be one of the variations (*DGS-1*, *DGS-2*, or *DGS-3*), and the NN classifier component. The design of *Deep-Graph-Sprints* methodically incorporates forward-mode AD for learning the DGS component, aligning with specific computational and memory considerations inherent in graph-based data contexts. In contrast, the subsequent NN classifier component, processing the embeddings generated by the DGS component, utilizes reverse-mode AD differentiation. This hybrid approach effectively leverages the strengths of both paradigms, namely, learning long term dependencies, and ensuring efficient learning while accommodating the memory constraints and structural complexities of graph data.

Given the specific architecture, standard Jacobian tools were Insufficiently efficient especially to calculate the Jacobians of the DGS component parameters. Consequently, we handcrafted and implemented the Jacobians from scratch to ensure precise alignment with the model's requirements, optimizing both performance and computational efficiency, as detailed in the following Section 6.2.4.

6.2.4 Gradient Calculations in Deep-Graph-Sprints

This section delves into the core computational mechanics of the *Deep-Graph-Sprints* methodology, with a specific focus on the gradient calculation process integral to the optimization of the learnable parameters. *Deep-Graph-Sprints* involves two components DGS and NN, as discussed in Section 6.2.1.

In the NN classifier component, number of learnable parameters varies based on model architecture, primarily involving the network's weights. As discussed in Section 6.2.3, the

parameters of the NN classifier component are optimized using backpropagation, and to implement that we leverage the functionalities of a standard deep learning platform, namely, PyTorch [Fey and Lenssen, 2019].

Central to the DGS model are three key parameters: α, β, and the learnable feature mapping matrix W, which facilitates the transformation from feature space to embedding space. Notably, this W matrix replaces the $\vec{\delta}$ feature encoding function utilized in the third variant of the model (DGS-3). The optimization of these parameters is dependent upon the accurate computation of their gradients in relation to the defined loss function, denoted as $\vec{L}$. Therefore, it becomes imperative to systematically compute the partial Jacobians $\frac{d\vec{L}}{d\vec{\alpha}}$, $\frac{d\vec{L}}{d\vec{\beta}}$, and $\frac{d\vec{L}}{dW}$. This process entails the adjustment of $\vec{\alpha}$, $\vec{\beta}$, and W after the processing of each data batch, based on their respective gradients with respect to the loss function. Due to the specificities of the learning paradigm for these parameters, as discussed in Section 6.2.3, we need to implement their gradient calculation and optimization process from scratch. In the following subsections, we present a detailed explanation for the computation of these gradients. Each parameter's Jacobian computation is broken down into a sequential, step-by-step process, elucidating the mathematical and algorithmic underpinnings that facilitate the *Deep-Graph-Sprints* model's learning algorithm. This comprehensive breakdown aims to provide clarity and enhance the replicability of the gradient calculation procedure within the *Deep-Graph-Sprints* method.

Importantly, in the *Deep-Graph-Sprints* method variations (DGS-2 and DGS-3), the parameters $\vec{\alpha}$ and $\vec{\beta}$ are conceptualized as vectors, diverging from the scalar form used in DGS-1. This distinction is crucial, yet the foundational equations remain applicable in both scenarios. $\vec{\alpha}$, and $\vec{\beta}$ are interpreted as an aggregation of f forgetting coefficients α_i, and β_i, as demonstrated in Equations (6.2a and 6.3a), where f represents the count of input features, and α_i is the associated forgetting coefficient for the i^{th} feature.

This architectural choice, allocating one coefficient per feature, is an important detail for the efficiency of Jacobian calculation, a critical aspect that enables efficient forward-mode AD with minimized computational complexity as discussed in Section 6.2.3.

6.2.4.1 Alpha ($\vec{\alpha}$)

This section focuses on the computation of the Jacobians of the parameter $\vec{\alpha}$ by applying the chain rule. This fundamental concept in calculus is crucial for understanding how changes

in $\vec{\alpha}$ affect the loss function $\vec{L}$, given the state $\vec{S}$, which represents the NN model input. The relationship is mathematically expressed in Equation 6.7.

$$\frac{d\vec{L}}{d\vec{\alpha}} = \frac{d\vec{L}}{d\vec{S}} \cdot \frac{d\vec{S}}{d\vec{\alpha}} \tag{6.7}$$

We see that the resulting Jacobian depends on two parts $\frac{d\vec{L}}{d\vec{S}}$ representing the NN classifier component, and $\frac{d\vec{S}}{d\vec{\alpha}}$ representing the DGS component.

For the NN classifier component, $\frac{d\vec{L}}{d\vec{S}}$ is derived using backpropagation. For practical implementation, this derivation process is efficiently facilitated by PyTorch [Fey and Lenssen, 2019] built-in functions, allowing for streamlined computation. About the second component (i.e., $\frac{d\vec{S}}{d\vec{\alpha}}$) representing the *DGS* component, here we use the forward-mode AD to calculate the Jacobians. As mention in Section 6.2.3, due to the specificities of our model, standard Jacobian tools were insufficient. Consequently, we handcrafted and implemented the Jacobians from scratch to ensure precise alignment with the model's requirements.

As defined by our state computation formula (see Equation 6.4), the state $\vec{S}_t$ is influenced both directly and recursively by the parameters $\vec{\alpha}$ and $\vec{\beta}$, as it depends on the previous states of the target node $\vec{S}_{t-1}$, and its neighbor $\vec{S}^*_{t-1}$, which are in turn functions of $\vec{\alpha}$, $\vec{\beta}$, and all preceding states. For the sake of clarity and to avoid notational complexity, we choose to omit this recursive dependency in the notation when computing $\frac{d\vec{S}}{d\vec{\alpha}}$, as illustrated in Equation 6.8.

$$\frac{d\vec{S}_t}{d\vec{\alpha}} = \frac{d\vec{S}_t}{d\vec{S}_{t-1}} \odot \frac{d\vec{S}_{t-1}}{d\vec{\alpha}} + \frac{d\vec{S}_t}{d\vec{S}^*_{t-1}} \odot \frac{d\vec{S}^*_{t-1}}{d\vec{\alpha}} + \frac{\partial\vec{S}_t}{\partial\vec{\alpha}} \tag{6.8}$$

The calculation of the partial Jacobians, $\frac{\partial\vec{S}_t}{\partial\vec{\alpha}}$, is a key step in this process. This calculation, essential for understanding the direct impact of $\vec{\alpha}$ on the state S_t, is shown in Equation 6.9.

$$\frac{\partial\vec{S}_t}{\partial\vec{\alpha}} = (1 - \vec{\beta}) \odot \left(\vec{S}^*_{t-1} - \vec{\delta}(F_t)\right) \tag{6.9}$$

By integrating in the *Deep-Graph-Sprints* algorithm (referenced in Equation 6.4) with the Jacobian chain rule (outlined in Equation 6.8), we arrive at a recursive formulation as depicted in Equation 6.10.

$$\frac{d\vec{S}_t}{d\vec{\alpha}} = \vec{\beta} \odot \frac{d\vec{S}_{t-1}}{d\vec{\alpha}} + (1 - \vec{\beta}) \odot \vec{\alpha} \odot \frac{d\vec{S}^*_{t-1}}{d\vec{\alpha}} + \frac{\partial \vec{S}_t}{\partial \vec{\alpha}} \tag{6.10}$$

Finally, by substituting the values of the partial Jacobians formula (Equation 6.9, we derive Equation 6.11.

$$\frac{d\vec{S}}{d\vec{\alpha}} = \vec{\beta} \odot \frac{d\vec{S}_{t-1}}{d\vec{\alpha}} + (1 - \vec{\beta}) \odot \left(\vec{\alpha} \odot \frac{d\vec{S}^*_{t-1}}{d\vec{\alpha}} + \vec{S}^*_{t-1} - \vec{\delta}(F_t) \right) \tag{6.11}$$

Memory and Computational Complexities for $\vec{\alpha}$ Derivatives

Equation 6.11 underscores the recursive nature of the gradient calculation, highlighting the necessity to maintain a historical record of the Jacobian states for each node within the graph. Practically, to manage this, we implement a storage mechanism for the most recent Jacobian, $\frac{d\vec{S}_{t-1}}{d\vec{\alpha}}$, for each node. Upon the introduction of a new edge, these stored Jacobians are utilized to compute the updated Jacobians, which are then used to refresh the stored values. Therefore, the storage requirement entails maintaining an $n \times s$ matrix for a graph with n nodes, each with a state vector of length s.

About the computational complexity a key characteristic of our method is the element-wise multiplication between the state vectors $\vec{S}$ and the parameter vectors $\vec{\alpha}$. This operation significantly optimizes the computational complexity involved in the calculation of Jacobians. For a state vector $\vec{S}$ of length s, the Jacobian with respect to $\vec{\alpha}$ is outlined in Equation 6.12. This process, critical for assessing the impact of changes in $\vec{\alpha}$ on the state vector $\vec{S}$, has a computational complexity of $\mathcal{O}(s)$, a reflection of its element-wise nature. The same applies for the element wise multiplication between the encoded features vector and $\vec{\alpha}$.

Conversely, if the operation were matrix multiplication, the computational complexity would rise to $\mathcal{O}(s^2)$. This increase is due to the more complex interdependency where each element of the product is influenced by all the values in $\vec{\alpha}$. Therefore, the element-wise multiplication approach not only simplifies the computational procedure but also improves efficiency, making it a strategically beneficial choice for the applicability of forward-mode AD in our model.

$$\frac{\partial \vec{S}}{\partial \vec{\alpha}} = \begin{bmatrix} \frac{\partial S_1}{\partial \alpha_1} \\ \frac{\partial S_2}{\partial \alpha_2} \\ \vdots \\ \frac{\partial S_s}{\partial \alpha_s} \end{bmatrix} \tag{6.12}$$

Therefore, the arrival of a new edge involves calculating the partial Jacobians as outlined in (6.9) and integrating these with the pre-stored Jacobians as formulated in Equation 6.11. The resultant computational complexity can be expressed as $\mathcal{O}(s + s + s + a)$, simplifying to $\mathcal{O}(s + a)$, under the assumption that the encoding function $\vec{\delta}$ incurs a computational complexity denoted by a. Where a=$\mathcal{O}(\sum_f \log(L_f))$, where L_f stands for the number of bins for feature f, as detailed in Section 5.3.2.

This formulation not only streamlines the understanding of the recursive nature of the gradient calculations in the *Graph-Sprints* method but also defines the practical aspects of implementing such a system, particularly in terms of storage and computational complexity.

6.2.4.2 Beta ($\vec{\beta}$)

In parallel to the methodology applied for $\vec{\alpha}$, this section is devoted to the computation of the Jacobians of the parameter $\vec{\beta}$ within the *Deep-Graph-Sprints* method. Employing the chain rule, we determine the impact of $\vec{\beta}$ on the loss function $\vec{L}$, in relation to the state $\vec{S}$, the input to the model. This relationship is represented in Equation 6.13.

$$\frac{d\vec{L}}{d\vec{\beta}} = \frac{d\vec{L}}{d\vec{S}} \cdot \frac{d\vec{S}}{d\vec{\beta}} \tag{6.13}$$

The resulting Jacobian depends on two parts $\frac{d\vec{L}}{d\vec{S}}$ representing the NN classifier component, and $\frac{d\vec{S}}{d\vec{\beta}}$ representing the DGS component.

Similar to the process for $\vec{\alpha}$, $\frac{d\vec{L}}{d\vec{S}}$ is obtained through backpropagation. The term $\frac{d\vec{S}}{d\vec{\beta}}$ then captures how changes in $\vec{\beta}$ affect the state $\vec{S}$. We use the forward-mode AD to calculate the Jacobians. As mention in Section 6.2.3, due to the specificities of our model, standard Jacobian tools were insufficient. Consequently, we handcrafted and implemented the Jacobians from scratch to ensure precise alignment with the model's requirements.

As per the $\vec{\beta}$ parameter, and given our state computation formula (see Equation 6.1), the state $\vec{S}_t$ depends on $\vec{\beta}$, and also depends on the previous states of the target node and the neighbor, $\vec{S}_{t-1}$ and $\vec{S}^*_{t-1}$, which also depend on $\vec{\beta}$. Thus, the Jacobian can be further expanded as shown in Equation 6.14, which decomposes the Jacobian into more granular components.

$$\frac{d\vec{S}_t}{d\vec{\beta}} = \frac{d\vec{S}_t}{d\vec{S}_{t-1}} \odot \frac{d\vec{S}_{t-1}}{d\vec{\beta}} + \frac{d\vec{S}_t}{d\vec{S^*}_{t-1}} \odot \frac{d\vec{S^*}_{t-1}}{d\vec{\beta}} + \frac{\partial \vec{S}_t}{\partial \vec{\beta}} \tag{6.14}$$

The calculation of the partial Jacobian $\frac{\partial \vec{S}_t}{\partial \vec{\beta}}$ leveraging the *Deep-Graph-Sprints* formula (Equation 6.4) is crucial in understanding the direct influence of $\vec{\beta}$ on the state S_t, as demonstrated in Equation 6.15.

$$\frac{\partial \vec{S}_t}{\partial \vec{\beta}} = \vec{S}_{t-1} - \left((1 - \vec{\alpha}) \odot \vec{\delta}(F_t) + \vec{\alpha} \odot \vec{S^*}_{t-1} \right) \tag{6.15}$$

Employing the *Deep-Graph-Sprints* formula (referenced in Equation 6.4), we proceed with a modular-focused reformulation of the Jacobian $\frac{d\vec{S}_t}{d\vec{\beta}}$. This is achieved by representing it through the partial Jacobians of $\vec{S}$, as illustrated in Equation 6.16. This reformulation not only simplifies the understanding but also facilitates easier application in practical scenarios.

$$\frac{d\vec{S}_t}{d\vec{\beta}} = \vec{\beta} \odot \frac{d\vec{S}_{t-1}}{d\vec{\beta}} + (1 - \vec{\beta}) \odot \vec{\alpha} \odot \frac{d\vec{S^*}_{t-1}}{d\vec{\beta}} + \frac{\partial \vec{S}_t}{\partial \vec{\beta}} \tag{6.16}$$

Finally, by substituting the values of the partial Jacobians formula (Equation 6.15, we derive Equation 6.17.

$$\frac{d\vec{S}_t}{d\vec{\beta}} = \left(\vec{S}_{t-1} + \vec{\beta} \odot \frac{d\vec{S}_{t-1}}{d\vec{\beta}} \right) - \left((1 - \vec{\alpha}) \odot \vec{\delta}(F_t) + \vec{\alpha} \odot \vec{S}^*_{t-1} \right) + (1 - \vec{\beta}) \odot \left(\vec{\alpha} \odot \frac{d\vec{S}^*_{t-1}}{d\vec{\beta}} \right) \tag{6.17}$$

Memory and Computational Complexities for $\vec{\beta}$ Derivatives

The approach for computing the Jacobians of $\vec{\beta}$, as detailed in Equation 6.16, mirrors the methodology applied to $\vec{\alpha}$, with a similar recursive structure necessitating the tracking of past Jacobian states for each graph node. For operational efficiency, a storage system is employed to retain the latest Jacobian, $\frac{d\vec{S}_{t-1}}{d\vec{\beta}}$, for each node. The arrival of a new edge triggers the utilization of these stored Jacobians for the computation of updated values, subsequently updating the

storage.

The data storage architecture remains consistent with the $\vec{\alpha}$ Jacobian process, requiring an $(n \times s)$ matrix corresponding to a graph of n nodes, each represented by a state vector of length s.

The computational complexity for the $\vec{\beta}$ Jacobian, including feature encoding and integration with stored Jacobians as per Equation 6.16.

Similarly to the computational complexity of $\vec{\alpha}$, element-wise multiplication between state vectors $\vec{S}$ and parameters $\vec{\beta}$ optimizes Jacobian calculations in our method. For $\vec{S}$ of length s, the Jacobian with respect to $\vec{\beta}$ (Equation 6.18) has computational complexity $\mathcal{O}(s)$. This also applies to the multiplication with the encoded features vector and $\vec{\beta}$. In contrast, matrix multiplication would lead to a complexity of $\mathcal{O}(s^2)$, making element-wise multiplication a more efficient choice.

$$\frac{\partial \vec{S}}{\partial \vec{\beta}} = \begin{bmatrix} \frac{\partial S_1}{\partial \beta_1} \\ \frac{\partial S_2}{\partial \beta_2} \\ \vdots \\ \frac{\partial S_s}{\partial \beta_s} \end{bmatrix} \tag{6.18}$$

Therefore, computing the Jacobians of $\frac{d\vec{S}_t}{d\vec{\beta}}$ maintain a computational complexity of $\mathcal{O}(s + a)$. This is assuming that the encoding function $\vec{\delta}$ possesses a computational complexity characterized by a. Where a=$\mathcal{O}(\sum_f \log(L_f))$, where L_f stands for the number of bins for feature f, as detailed in Section 5.3.2.

This structured approach to gradient computation for $\vec{\beta}$, while akin to that of $\vec{\alpha}$, reinforces the method's coherence in handling recursive calculations.

6.2.4.3 Embedding Matrix (W)

In the third approach of the *Deep-Graph-Sprints* methodology, the embedding matrix parameter W plays a pivotal role. This approach modifies the state computation formula, integrating a learnable mapping from feature space to embedding space, as opposed to the encoding function $\vec{\delta}$. The revised state computation formula is presented in Equation 6.6, where the softmax function is applied to the product of W and feature vector F_t.

The calculation of the Jacobians of W is essential for understanding how changes in the embedding matrix affect the overall model. Employing the chain rule, we establish a relationship between the loss $\vec{L}$ and W, as defined in Equation 6.19.

$$\frac{d\vec{L}}{dW} = \frac{d\vec{L}}{d\vec{S}} \cdot \frac{d\vec{S}}{dW} \tag{6.19}$$

The resulting Jacobian depends on two parts $\frac{d\vec{L}}{d\vec{S}}$ representing the NN classifier component, and $\frac{d\vec{S}}{dW}$ representing the DGS component.

As per the Jacobians of the loss in respect to the state (i.e., $\frac{d\vec{L}}{d\vec{S}}$), it is obtained using backpropagation, as discussed when detailing the Jacobians of the parameters $\vec{\alpha}$, and $\vec{\beta}$.

About the second term (i.e.,$\frac{d\vec{S}}{dW}$) representing the *DGS* component, here we use the forward-mode AD to calculate the Jacobians.

To further elucidate the second component, $\frac{d\vec{S}}{dW}$ representing the relationship between W and $\vec{S}_t$, given our state computation formula (see Equation 6.6), the state $\vec{S}_t$ depends on W, and also depends on the previous states of the target node and the neighbor, $\vec{S}_{t-1}$ and $\vec{S}^*_{t-1}$, which also depend on W. Thus, we decompose the Jacobian of the state $\vec{S}_t$ with respect to W into more granular components, as shown in Equation 6.20.

$$\frac{d\vec{S}_t}{dW} = \frac{d\vec{S}_t}{d\vec{S}_{t-1}} \odot \frac{d\vec{S}_{t-1}}{dW} + \frac{d\vec{S}_t}{d\vec{S^*}_{t-1}} \odot \frac{d\vec{S^*}_{t-1}}{dW} + \frac{\partial\vec{S}_t}{\partial W} \tag{6.20}$$

Utilizing Equation 6.6, the expression of $\frac{d\vec{S}_t}{dW}$ is simplified through the application of the partial Jacobians of $\vec{S}$. This methodology yields the Jacobian of $\vec{S}$ relative to W, as detailed in Equation 6.21.

$$\frac{d\vec{S}_t}{dW} = \vec{\beta} \odot \frac{d\vec{S}_{t-1}}{dW} + (1 - \vec{\beta}) \odot \vec{\alpha} \odot \frac{d\vec{S}^*_{t-1}}{dW} + \frac{\partial\vec{S}_t}{\partial W} \tag{6.21}$$

A critical step in this process is the calculation of the partial Jacobian $\frac{\partial\vec{S}_t}{\partial W}$, which directly quantifies the impact of W on the state S_t. This is expressed in Equation 6.22, where $\bigoplus_{i=1}^{x}$ represents the concatenation of the results of x softmax functions, and $W_i F_t$ represents the result of multiplying the i^{th} subset of W with the features.

$$\frac{\partial \vec{S}_t}{\partial W} = (1 - \vec{\beta}) \odot (1 - \vec{\alpha}) \odot \frac{\partial \left(\bigoplus_{i=1}^{x} \vec{\sigma}(W_i F_t) \right)}{\partial W} \tag{6.22}$$

The softmax function, denoted as $\vec{\sigma}$ and applied to an input vector $\mathbf{x}$, is formally defined by Equation 6.5. This definition implies that the softmax value of each element in $\mathbf{x}$ is influenced by all other elements in the vector. As a result, the Jacobian of the softmax function with respect to its input vector forms a two-dimensional Jacobian matrix, reflecting the interdependencies among the vector elements. The values of these Jacobians varies depending on their position within the Jacobian matrix, specifically whether they are diagonal or off-diagonal elements, as shown in Equation 6.23.

$$\frac{\partial \vec{\sigma}(\vec{x})_i}{\partial x_j} = \begin{cases} \sigma(x)_i \cdot (1 - \sigma(x)_j) & \text{if } i = j, \\ -\sigma(x)_i \cdot \sigma(x)_j & \text{if } i \neq j. \end{cases} \tag{6.23}$$

Memory and Computational Complexities for W Derivatives:

The Jacobian computation for the embedding matrix W, as formulated in Equation 6.21, follows a recursive calculation parallel to that used for $\vec{\alpha}$ and $\vec{\beta}$. This process entails tracking historical Jacobian states for each node in the graph. To enhance operational efficiency, we maintain a dedicated storage system to store the most recent Jacobian, $\frac{d\vec{S}_{t-1}}{dW}$, for each node. The introduction of new edges activates the use of these stored Jacobians to facilitate the calculation of updated values, which are then used to update the stored Jacobians.

Optimizing Memory Footprint and Computational Complexity Using Multiple Softmax Functions:

Considering an embedding matrix W of dimensions $s \times f$, where s is the embedding size and f is the number of input features, a conventional application of a single softmax function would typically yield Jacobians of dimensions $f \times s^2$. In our *Deep-Graph-Sprints* model, we address this memory complexity issue by employing multiple softmax functions. Each softmax function is designated to manage a segment of the resulting state, as illustrated in Equation 6.24. Utilizing x softmax functions, each softmax corresponds to a subset of h rows in the embedding matrix $h = s \div x$. Therefore, we effectively reduce the memory requirement to $f \times h \times s$. This is the memory required per node in the graph, thus, the memory complexity for a graph with n nodes is $n \times (f \times h \times s)$, each node associated with a state vector of length s, and processed using x

softmax functions.

$$\frac{\partial \vec{S}}{\partial W} = \begin{cases} \frac{\partial \vec{S}i}{\partial W_{j//f,j\%f}} & \text{if } i \ \& \ j \in \text{ same } \vec{\sigma}, \\ 0 & \text{otherwise.} \end{cases} \tag{6.24}$$

To illustrate with an example, consider an input size $f = 3$, a state size $s = 4$, and a number of softmax functions $x = 2$. Consequently, the number of rows in the embedding matrix W per softmax is $h = 2$.

$$W = \begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \\ W_{41} & W_{42} & W_{43} \end{bmatrix}, F = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix}$$

The calculation of $\bigoplus_{i=1}^{x=2} \vec{\sigma}(W_i F_t)$ (done in the forward pass, when calculating the state) can be visualized as:

$$\bigoplus_{i=1}^{x=2} \vec{\sigma}(W_i F_t) = \begin{bmatrix} \vec{\sigma} \left(\begin{bmatrix} W_{11}F_1 + W_{12}F_2 + W_{13}F_3 \\ W_{21}F_1 + W_{22}F_2 + W_{23}F_3 \end{bmatrix} \right) \\ \vec{\sigma} \left(\begin{bmatrix} W_{31}F_1 + W_{32}F_2 + W_{33}F_3 \\ W_{41}F_1 + W_{42}F_2 + W_{43}F_3 \end{bmatrix} \right) \end{bmatrix}$$

To calculate the Jacobians of S with respect to W, applying Equation 6.24 we get:

$$\frac{\partial \vec{S}}{\partial W} = \begin{bmatrix} \frac{\partial S_1}{\partial W_{11}} & \frac{\partial S_1}{\partial W_{12}} & \frac{\partial S_1}{\partial W_{13}} & \frac{\partial S_1}{\partial W_{21}} & \frac{\partial S_1}{\partial W_{22}} & \frac{\partial S_1}{\partial W_{23}} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\partial S_2}{\partial W_{11}} & \frac{\partial S_2}{\partial W_{12}} & \frac{\partial S_2}{\partial W_{13}} & \frac{\partial S_2}{\partial W_{21}} & \frac{\partial S_2}{\partial W_{22}} & \frac{\partial S_2}{\partial W_{23}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{\partial S_3}{\partial W_{31}} & \frac{\partial S_3}{\partial W_{32}} & \frac{\partial S_3}{\partial W_{33}} & \frac{\partial S_3}{\partial W_{41}} & \frac{\partial S_3}{\partial W_{42}} & \frac{\partial S_3}{\partial W_{43}} \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{\partial S_4}{\partial W_{31}} & \frac{\partial S_4}{\partial W_{32}} & \frac{\partial S_4}{\partial W_{33}} & \frac{\partial S_4}{\partial W_{41}} & \frac{\partial S_4}{\partial W_{42}} & \frac{\partial S_4}{\partial W_{43}} \end{bmatrix}$$

The analysis of the Jacobians $\frac{\partial \vec{S}}{\partial W}$ reveals that they comprise a collection of x blocks, each independent and with dimensions of $h^2 \times f$.

Thus, the computational complexity for every segment is $(h^2 \times f)$, and given that we have

x segments, then the total computational complexity for deriving W, as per Equation 6.21, is $\mathcal{O}(s + (s \times h \times f))$. It is worth noting that the calculation can be done in parallel since segments are independent.

In conclusion, the total computational complexity is linear with respect to the state size s, assuming a fixed h. This is desirable and similar to backpropagation. Furthermore, it is feasible to fix h to a predetermined computational complexity.

6.2.5 Parameter Updating Mechanisms in Deep-Graph-Sprints

The **Deep-Graph-Sprints** method, as outlined in Section 6.2.1, integrates two components: the DGS and the NN classifier components. Each component encompasses a set of learnable parameters, which are subject to iterative updates subsequent to the processing of each data batch. In the context of the NN classifier component, the Adam optimizer [Kingma and Ba, 2014]—a refined algorithm for first-order gradient-based optimization of stochastic objective functions, predicated on adaptive estimates of lower-order moments—is employed for parameter updates. This implementation is sourced from Pytorch [Fey and Lenssen, 2019].

In contrast, the DGS component utilizes vanilla gradient descent instead of Adam to update the parameters.

Specifically, after processing a specific batch b, the Jacobians are calculated for the learnable parameters based on the batch data, yielding Jacobian values $\frac{d\vec{L_b}}{d\vec{\alpha}}$, $\frac{d\vec{L_b}}{d\vec{\beta}}$, and $\frac{d\vec{L_b}}{dW}$, (as discussed in Section 6.2.4). The parameter update equations are then applied: equation 6.25 for the α parameter, equation 6.26 for the β parameter, and equation 6.27 for updating the embedding matrix W.

$$\vec{\alpha} := \vec{\alpha} - \left(\eta \cdot \frac{d\vec{L_b}}{d\vec{\alpha}} \right) \tag{6.25}$$

$$\vec{\beta} := \vec{\beta} - \left(\eta \cdot \frac{d\vec{L_b}}{d\vec{\beta}} \right) \tag{6.26}$$

$$W := W - \left(\eta \cdot \frac{d\vec{L_b}}{dW} \right) \tag{6.27}$$

The parameters of both the DGS and NN classifier components are updated simultaneously.

Looking ahead, future work might involve exploring the integration of more advanced optimization algorithms for the DGS component parameters. For instance, the adoption of the Adam optimizer.

6.3 Experiments and Results

6.3.1 Experimental Setup

The efficacy of the *Deep-Graph-Sprints* methodology was evaluated through two distinct tasks: node classification and link prediction. This evaluation utilized five datasets, comprising three open-source external datasets and two proprietary datasets from the AML domain.

6.3.1.1 Baselines

A first basic baseline is Raw, which uses the raw edge features to train an ML classifier, aiming to demonstrate the classifier's performance in the absence of graph-related information.

Another pivotal baseline is *Graph-Sprints*, the graph feature engineering method detailed in Chapter 5. This benchmark aims to ascertain whether *Deep-Graph-Sprints* maintains competitive performance while addressing the limitations of *Graph-Sprints*, discussed in Section 6.1. The comparison includes two variants: GS, which utilizes *Graph-Sprints* histograms, and GS+Raw, which integrates *Graph-Sprints* histograms with raw edge features.

Consequently, while Raw, GS, and GS+Raw baselines employ the same machine learning classifier types, they differ in their feature sets: Raw utilizes raw edge features, GS is based on *Graph-Sprints* histograms, and GS+Raw combines both sets of features.

An additional baseline, termed *Fixed-DGS*, represents a variant of *Deep-Graph-Sprints* where the DGS parameters remain static during training. The purpose of this baseline is to highlight the performance enhancement attributed to the learning of DGS parameters.

Another baseline set comprises state-of-the-art GNN methods, specifically TGN [Rossi et al., 2020] and *Jodie* [Kumar et al., 2019], as previously used in *Graph-Sprints* (Section 5.4.1.1). Our TGN implementation, based on PyTorch Geometric [Fey and Lenssen, 2019], differs from the original by restricting within-batch neighbor sampling. For consistency, *Graph-Sprints* and *Deep-Graph-Sprints* also employ the same constraints.

Two TGN variants were used: *TGN-attn*, aligning with the original paper's robust variant, and *TGN-ID*, a simplified version focusing solely on memory module embeddings. *Jodie*, utilizing a time projection embedding with gated recurrent units. *TGN-ID* and *Jodie* baselines, which do not necessitate neighbor sampling, were chosen for their lower-latency attributes compared to *TGN-attn*.

GNN baselines (*TGN-ID*, *TGN-attn*, and *Jodie*) all used a node embedding size of 100

6.3.1.2 Optimization

The hyperparameter optimization process adopted a methodology parallel to that in *Graph-Sprints*, utilizing Optuna [Akiba et al., 2019] for training 100 models. Initial warmup trials were conducted through random sampling, followed by the application of the TPE sampler. Each model incorporated an early stopping mechanism, triggered after 10 epochs without improvement. This criterion, based on the AUC for node classification and AP for link prediction, ensured efficient training. Table 6.1 enumerates the hyperparameters and their respective ranges employed in the *Deep-Graph-Sprints* tuning process, while Table 5.1 details the ranges for the baselines.

TABLE 6.1: Hyperparameters ranges for *Deep-Graph-Sprints*

Hyperparameter	**Min**	**Max**
DGS learning rate (η)	10^{-4}	10^{3}
Number of softmaxes	10	50
Learning rate	10^{-4}	10^{2}
Dropout percentage	0.1	0.3
Weight decay	10^{-9}	10^{3}
Number of dense layers	1	3
Size of dense layer	32	256

Importantly, for all *Deep-Graph-Sprints* variants (DGS-1, DGS-2, and DGS-3), the state size is consistently set at 100, ensuring a fair comparison with other GNN baseline models (*TGN-ID*, *TGN-attn*, and *Jodie*). In future experiments, we propose to maintain a constant h while varying the number of softmaxes.

6.3.2 Public Datasets Experiments

6.3.2.1 Datasets

This study performs the evaluation of the *Deep-Graph-Sprints* method using three publicly available datasets, from social and educational domains. These datasets are identical to those used in the assessment of *Graph-Sprints*, and their details are elaborated in Section 5.4.2.1.

6.3.2.2 Task Performance

In assessing our method, we focused on two tasks: node classification and link prediction. The results for **node classification** are detailed in Table 6.2, displaying the average test AUC ± std for each dataset. To obtain these figures, we retrained the best model identified through hyperparameter optimization across 10 different random seeds.

The variations of our *Deep-Graph-Sprints* approach, DGS-1, DGS-2, and DGS-3, as described in Section 6.2.2, were also evaluated. Along with these, GNN baselines (*TGN-ID*, *TGN-attn*, and *Jodie*) and DGS-3 all utilized the same embedding size of 100.

We highlighted the **best** and second-best performing models in each dataset. Our analysis shows that *Deep-Graph-Sprints* variants performed exceptionally well in the Wikipedia and Mooc datasets, achieving the highest scores, while *Graph-Sprints* performed the second-best. In the Reddit dataset, our methods ranked second, behind *Graph-Sprints*. To provide an overview, we include a column showing the average rank, in Table 6.2, which represents the mean ranking computed from all datasets.

The lower performance of DGS-3 in the Wikipedia dataset, compared to other *Deep-Graph-Sprints* variations, is noteworthy. Initial analysis indicates that this is due to the data preprocessing method for the Wikipedia dataset. While using a consistent standardization approach across all datasets, omitting this step improved DGS-3's performance to 89.2 ± 1.9. Nonetheless, for consistency in the results table, the performance with standard preprocessing is reported. Future research will focus on understanding and resolving the factors behind this difference in DGS-3's performance.

For the link prediction task, average test AUC $\pm$ std and AP $\pm$ std are reported in Table 6.3, achieved by retraining the hyperparameter-optimized model with 10 random seeds. We evaluated both transductive (T) and inductive (I) settings, the former predicting future links of

TABLE 6.2: *Deep-Graph-Sprints:* Node classification results using public datasets.

Method	AUC ± std			Average rank
	Wikipedia	Mooc	Reddit	
Raw	58.5 ± 2.2	62.8 ± 0.9	55.3 ± 0.8	10
Fixed-DGS	87.3 ± 1.7	72.7 ± 0.3	64.5 ± 0.4	6.3
TGN-ID	88.9 ± 0.2	63.0 ± 17	61.3 ± 2.0	6
Jodie	87.2 ± 0.9	63.7 ± 16.7	61.9 ± 2.0	8
TGN-attn	86.6 ± 2.8	75.8 ± 0.4	67.9 ± 1.6	5.7
GS	90.7 ± 0.3	75.0 ± 0.2	**68.5 ± 1.0**	3.7
GS+Raw	89.2 ± 0.4	76.5 ± 0.3	63.7 ± 0.4	4.3
DGS-1	88.2 ± 0.6	73.8 ± 0.5	65.8 ± 0.8	4
DGS-2	**91.0 ± 0.3**	75.2 ± 0.3	67.2 ± 0.4	3
DGS-3	83.3 ± 3.7	**78.7 ± 0.6**	68.0 ± 1.9	4

nodes observed during training, and the latter involved predictions for nodes not encountered during training.

Deep-Graph-Sprints variants (DGS-1, DGS-2, and DGS-3), are detailed in Section 6.2.2.

We identified the **best** and second-best models. *Deep-Graph-Sprints* excelled in link prediction, consistently ranking first or second across datasets and settings (T and I). Notably, in the Mooc dataset, it outperformed the second-best model by about 7% in AP. To provide an overview, we have included a column in Table 6.3 that displays the average rank. This represents the mean ranking derived from all datasets, calculated using AP.

Optimized values varied depending on the use case and dataset. We discuss a few examples focusing on three essential parameters: DGS learning rate (η), number of softmaxes, and learning rate. Figure 6.2 presents the AP for each trained model in the Mooc dataset for the link prediction inductive task, indicating lower values of DGS learning rate (η) generally yield better results, contrary to the learning rate (NN classifier's component learning rate). The optimal number of softmax functions was found to be the minimum (10). For a comprehensive view of all tuned parameters, refer to Figure 6.3.

In the Wikipedia dataset, used for link prediction in a transductive setting, as shown in Figure 6.4, patterns slightly differ. Optimal DGS learning rates were around 0.5, and 20 softmax functions seemed preferable for this dataset.

TABLE 6.3: *Deep-Graph-Sprints:* Link prediction results using public datasets.

	Method	Wikipedia		Mooc		Reddit		Average rank
		AUC	AP	AUC	AP	AUC	AP	
T	**TGN-ID**	95.6 ± 0.2	95.8 ± 0.1	80.4 ± 5.8	75.0 ± 6.1	94.7 ± 0.7	93.2 ± 1.0	7
	Jodie	94.3 ± 0.3	94.5 ± 0.3	85.1 ± 1.8	80.0 ± 3.5	94.9 ± 1.2	93.4 ± 1.7	6.3
	TGN-attn	**97.0 ± 0.3**	**97.3 ± 0.3**	80.3 ± 8.1	75.6 ± 8.4	96.1 ± 0.4	95.1 ± 0.7	4.7
	Fixed-DGS	90.9 ± 1.1	91.0 ± 1.2	82.9 ± 0.2	80.8 ± 0.3	94.6 ± 0.1	93.9 ± 0.2	7.3
	GS	92.5 ± 0.6	92.9 ± 0.7	82.7 ± 0.8	81.1 ± 0.7	96.1 ± 0.2	95.3 ± 0.3	5
	GS+Raw	92.1 ± 0.4	92.6 ± 0.4	85.4 ± 0.3	83.7 ± 0.3	**96.8 ± 0.1**	96.1 ± 0.2	3.7
	DGS-1	91.3 ± 0.4	91.3 ± 0.5	83.5 ± 0.2	81.7 ± 0.3	95.6 ± 0.2	94.9 ± 0.2	6
	DGS-2	93.5 ± 1.3	93.6 ± 1.4	83.7 ± 4.1	82.1 ± 3.5	96.6 ± 0.1	**96.2 ± 0.2**	3
	DGS-3	96.4 ± 0.8	96.8 ± 0.7	**91.7 ± 0.4**	**90.4 ± 0.5**	96.0 ± 0.2	95.3 ± 0.2	2
I	**TGN-ID**	92.2 ± 0.2	92.8 ± 0.2	68.5 ± 8.6	63.5 ± 6.7	93.4 ± 0.6	92.2 ± 0.8	5.3
	Jodie	87.0 ± 0.6	89.1 ± 0.7	71.1 ± 2.2	66.1 ± 2.9	92.3 ± 1.3	90.8 ± 1.9	7.7
	TGN-attn	94.5 ± 0.2	95.0 ± 0.2	71.4 ± 4.1	66.9 ± 3.9	**95.0 ± 0.4**	**94.3 ± 0.5**	3.3
	Fixed-DGS	89.2 ± 0.8	89.8 ± 0.9	75.4 ± 0.4	72.5 ± 0.4	92.8 ± 1.2	91.5 ± 1.5	5.7
	GS	92.0 ± 0.3	91.7 ± 0.4	78.2 ± 0.6	76.5 ± 0.6	92.7 ± 0.5	92.7 ± 0.6	3
	GS+Raw	91.4 ± 0.2	91.1 ± 0.3	83.0 ± 0.5	80.3 ± 0.5	93.5 ± 0.4	92.2 ± 0.5	3.7
	DGS-1	88.8 ± 0.7	88.5 ± 1.2	71.8 ± 0.7	70.0 ± 1.3	90.7 ± 1.3	86.2 ± 2.0	8
	DGS-2	91.8 ± 0.8	91.6 ± 1.4	73.4 ± 1.8	71.0 ± 1.7	91.8 ± 0.5	87.3 ± 1.3	6
	DGS-3	**94.8 ± 0.7**	**95.5 ± 0.7**	**90.7 ± 0.5**	**89.5 ± 0.5**	93.8 ± 0.4	92.1± 0.6	2.3

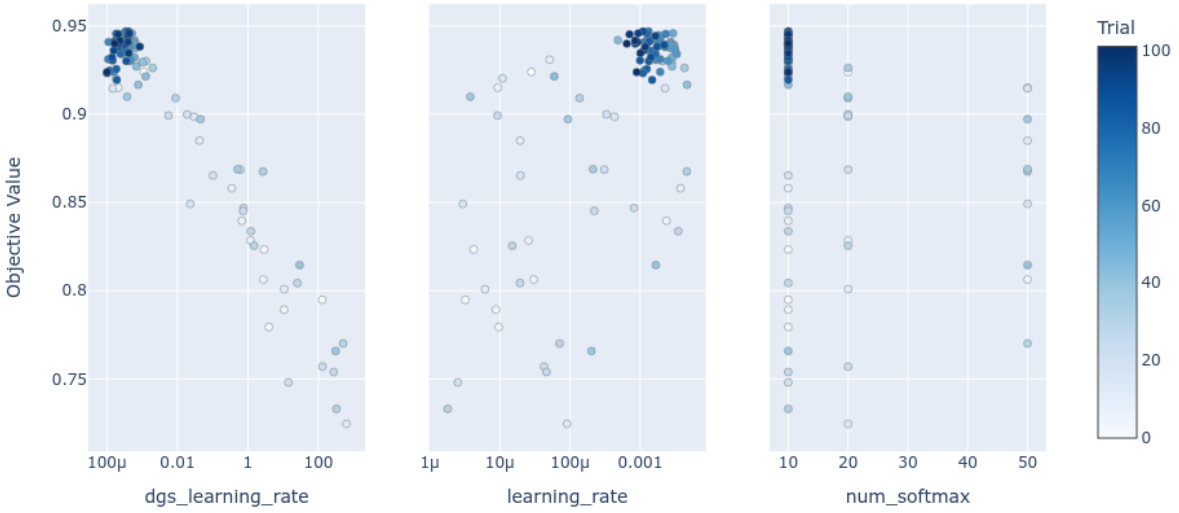


FIGURE 6.2: Example of hyperparameters' influence on inductive link prediction in the Mooc dataset, with the objective value measured as AP

6.3.2.3 Inference Runtime

In evaluating *Deep-Graph-Sprints*, our primary goal was to ensure inference times comparable to *Graph-Sprints* while addressing its limitations. We conducted latency comparisons between *Deep-Graph-Sprints*, *Graph-Sprints*, and baseline GNN models. This involved running 200 batches of 200 events on each external datasets (Wikipedia, Mooc, and Reddit) for the node classification

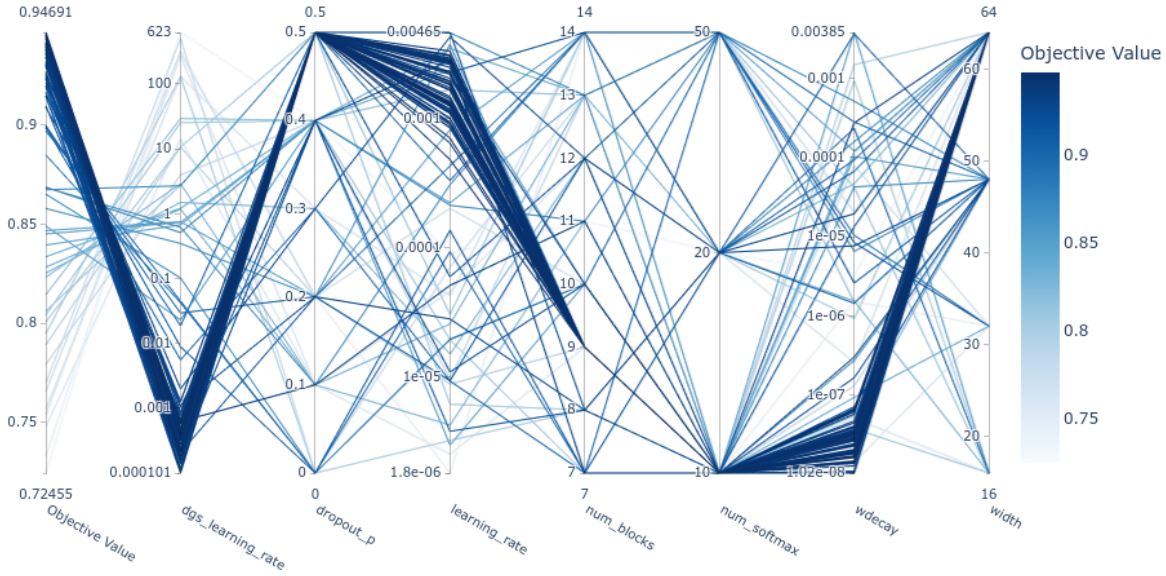


FIGURE 6.3: Example: Influence of all *Deep-Graph-Sprints* hyperparameters on link prediction in the Mooc dataset, with the objective value measured as AP.

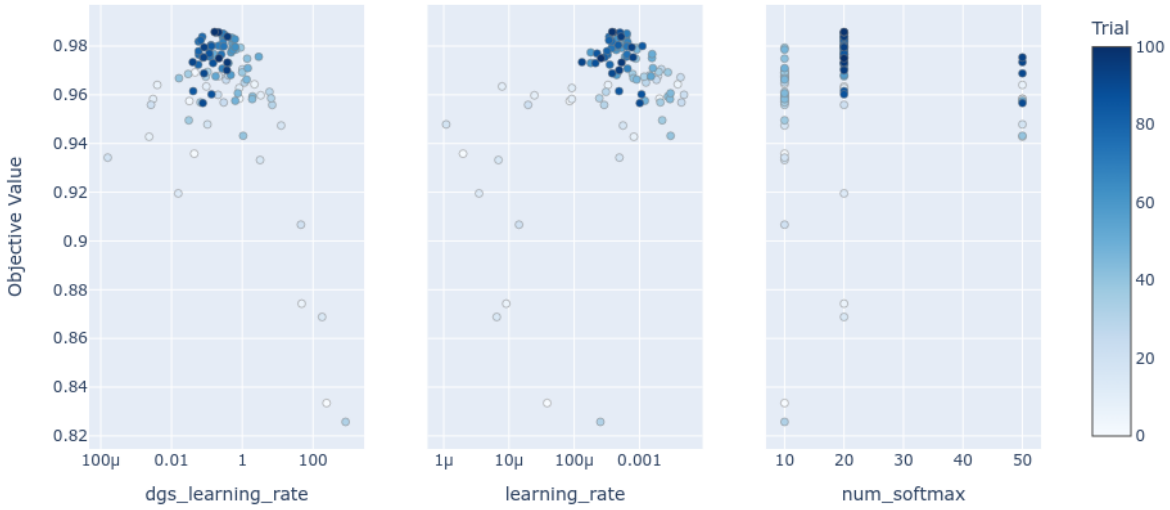


FIGURE 6.4: Example of hyperparameters' influence on transductive link prediction in the Wikipedia dataset, with the objective value measured as AP.

task, averaging times over 10 iterations. Tests were performed on a Linux PC equipped with 24 Intel Xeon CPU cores (3.70GHz) and an NVIDIA GeForce RTX 2080 Ti GPU (11GB). According to Figure 6.5, *Deep-Graph-Sprints* not only matched but also surpassed *Graph-Sprints* in inference speed, particularly in the Wikipedia and Reddit datasets (0.24 vs 0.29 seconds). This improvement is attributed to the higher feature count (172) in these datasets, increasing *Graph-Sprints* feature volume and thus, potentially, its processing time. In contrast, the Mooc dataset, with only 7 edge features, exhibited a smaller disparity in running times (0.24 vs 0.28

seconds), *Graph-Sprints* being faster.

When compared to GNN baselines (TGN-attn, *TGN-ID*, *Jodie*), *Deep-Graph-Sprints* demonstrated markedly lower inference latency, being over 12 times faster than *TGN-attn* in the Reddit dataset. Additionally, it delivered competitive classification performance in Reddit and surpassed *TGN-attn* in Mooc. However, its unexpectedly low performance in Wikipedia warrants further investigation.

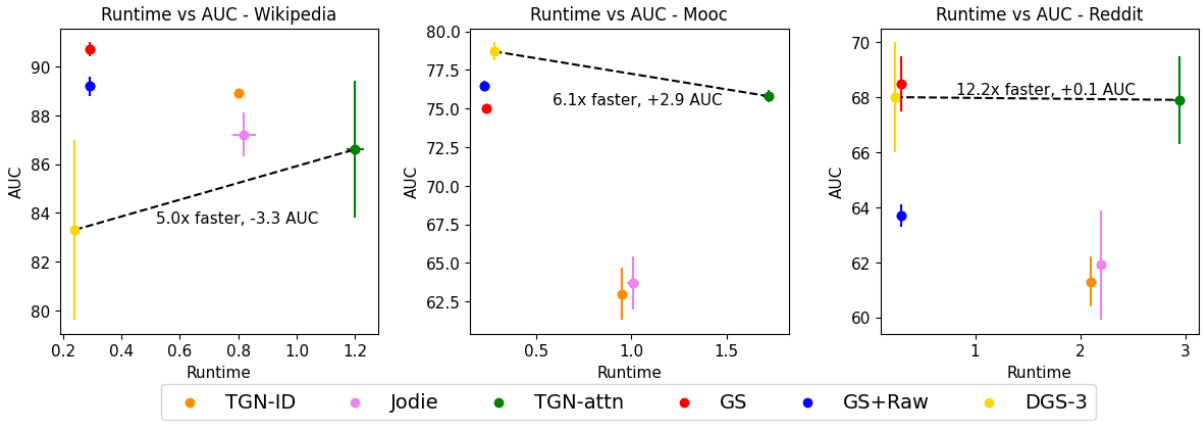


FIGURE 6.5: *Deep-Graph-Sprints:* Trade-off between AUC and runtime.

6.3.2.4 Analysing learning procedure

The evolution of the learning process can be seen through the visualization of parameter adjustments across successive epochs. The figures provided elucidate the dynamic alterations within the parameters, offering insights into the model's convergence behavior.

Figure 6.6 illustrates the variation of the α and β scalar parameters during the learning process in the DGS-1 approach, in the node classification task in the Mooc dataset. Initially set at 0.5, α generally decreases over time, indicating reduced emphasis on neighboring node information. Conversely, β tends to increase, suggesting a greater reliance on the previous state of the target node.

In the node classification task for Mooc using the DGS-3 variant of *Deep-Graph-Sprints*, α and β are vectors. Therefore, we report their average values. Figure 6.7 displays the changes in these average parameters alongside the validation results, providing a comparison between the adjustments in average parameters and validation performance.

This comparison reveals that on average both parameters are reducing in the initial epochs and there is a jump close to epoch 75. However, this aggregated view is not very precise.

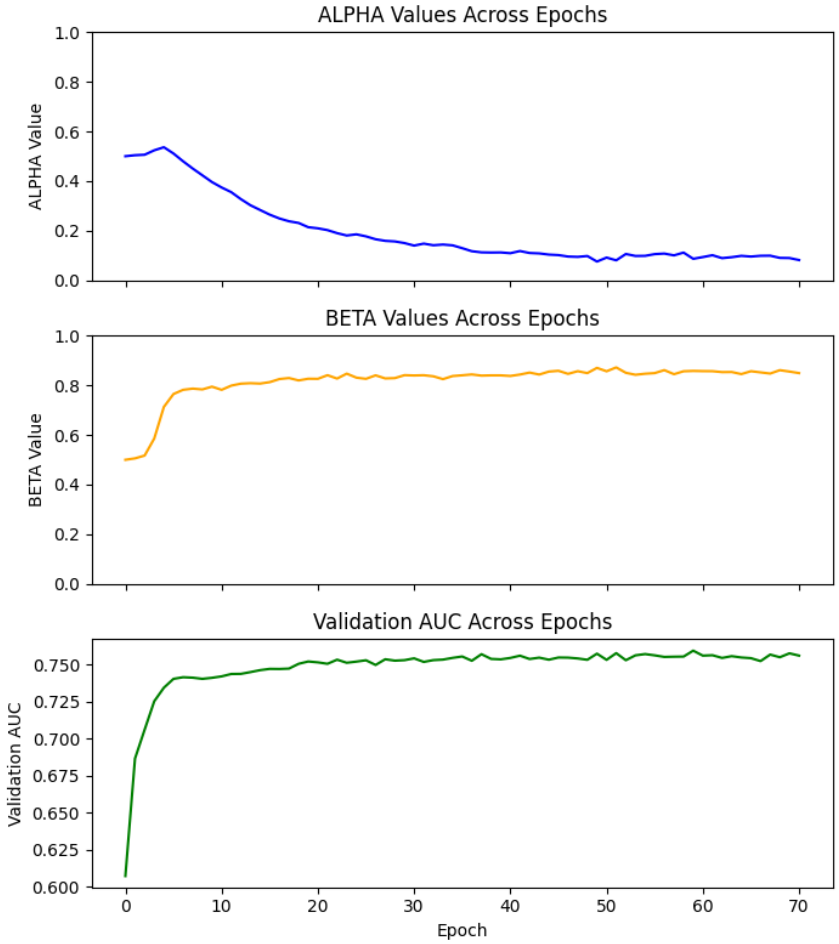


FIGURE 6.6: *DGS-1:* Example comparing α and β parameter changes with validation performance in node classification in the Mooc dataset.

Looking at the initial and last value within each bin of the ALPHA and BETA vectors (see Figure 6.8), we see that almost all bins in ALPHA tend to reduce, however, in BETA we notice that approximately half of the beta parameters increase while the other decrease. This fact underscores the importance of the implementation of β as a vector

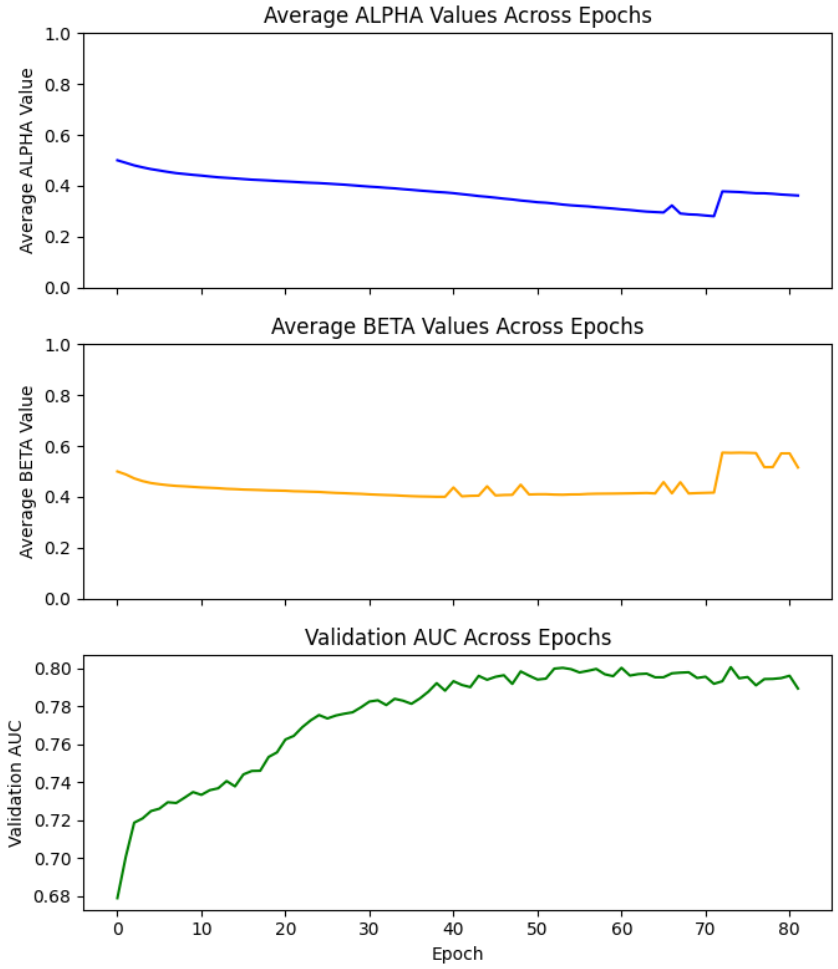


FIGURE 6.7: *DGS-3:* Example comparing average α and β parameter changes and validation performance in node classification in the Mooc dataset.

6.3.3 AML Experiments

In the context of money laundering, where criminals aim to conceal the unlawful origins of their capital by transferring it through multiple accounts and FIs, our experimental focus is specifically to elevate the efficacy of a *triage model*, which is comprehensively described in Chapter 4, Section 4.3, by integrating the *Deep-Graph-Sprints* method. This integration involves replacing the classifier and the graph-based feature generation steps with the capabilities of the *Deep-Graph-Sprints* method. Consequently, the *Deep-Graph-Sprints* method is employed as the primary *triage model*, a detailed architecture of which is presented in Figure 4.4.

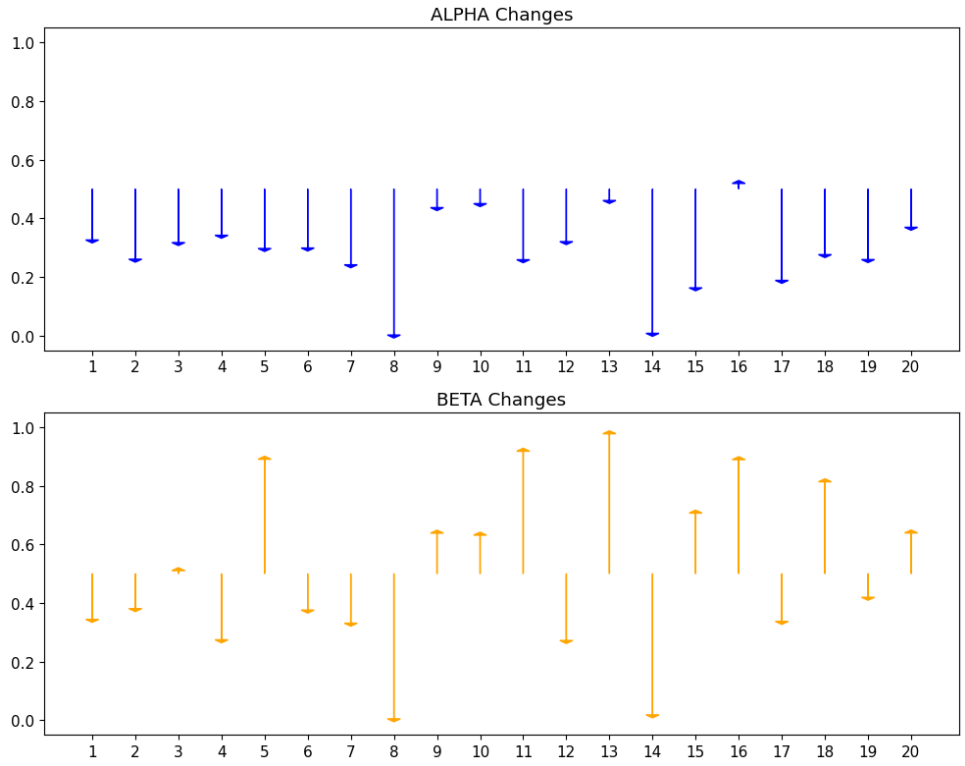


FIGURE 6.8: *DGS-3:* Evolution of per-bin α and β parameters throughout training. The x-axis represents the 20 bins per vector, while the y-axis shows the adjusted values of these bins after tuning. Initially, all bins start at 0.5, and then their values evolve to optimize model performance.

6.3.3.1 Datasets

We assess the *Deep-Graph-Sprints* method in two real-world banking datasets. These datasets are identical to those used in the assessment of *Graph-Sprints*, and their details are elaborated in Section 5.4.3.

6.3.3.2 Task Performance

In line with our *Graph-Sprints* experiments, we employed several baseline models for comparison. The first baseline is a neural network classifier that relies solely on raw node features (i.e., entity-centric features), without incorporating any graph information, termed 'Raw'. Additional baselines include 'GS', utilizing exclusively *Graph-Sprint* features, and 'GS+Raw', combining both *Graph-Sprints* and raw features. Furthermore, we replicated the training of three GNNs architectures used in public datasets: *TGN-ID*, *Jodie*, and *TGN-attn*.

For our *Deep-Graph-Sprints* models, we evaluate the three distinct approaches (explained in Section 6.2.2), labeled as DGS-1, DGS-2, and DGS-3.

All models underwent optimization with an equal optimization budget. Following this, the optimal hyperparameters (those yielding the best performance on the validation dataset) were selected. Each model was then run using 10 distinct random seeds, and the average AUC along with the standard deviation were calculated. This approach mirrors the methodology applied in the node classification task on public datasets (as detailed in Section 6.3.2). Due to confidentiality constraints, the specific AUC values are not disclosed. Instead, we present the relative AUC improvements (ΔAUC) compared to a baseline model lacking graph features ('Raw'). In this experiment, the baseline model is assigned a ΔAUC of 0, with any enhancement in recall over the baselines indicated by positive ΔAUC values.

The results are summarized in Table 6.4. We marked the **best** and the second-best models for clarity. This analysis includes *Deep-Graph-Sprints* models, *Graph-Sprints* variants, and other GNNs baselines. The various *Deep-Graph-Sprints* approaches showed promising results, with DGS-3 being the best model in the *FI-A dataset* by achieving a 3.6% increase in AUC. And a notable 26.9% improvement in AUC for the *FI-B dataset*, demonstrating the effectiveness of *Deep-Graph-Sprints* in this domain. To provide an overview, we include a column showing the average rank which represents the mean ranking computed from the two datasets.

TABLE 6.4: *Deep-Graph-Sprints:* Node classification results using AML datasets.

Method	**ΔAUC ± std**		**Average rank**
	FI-A	**FI-B**	
TGN-ID	+0.1 ± 0.1	+24.4 ± 0.2	8
Jodie	+0.0 ± 0.1	+24.5 ± 0.2	8
TGN-attn	+0.3 ± 0.7	+25.1 ± 0.3	6.5
Fixed-DGS	+2.0 ± 0.3	25.3 ± 0.2	4.5
GS	+1.8 ± 0.5	**+27.8 ± 0.4**	3.5
GS+Raw	+3.3 ± 0.3	+20.1 ± 3.9	5.5
DGS-1	+1.8 ± 0.3	25.8 ± 0.7	4.5
DGS-2	+3.2 ± 0.1	+26.7 ± 0.2	3
DGS-3	**+3.6 ± 0.2**	+26.9 ± 0.3	1.5

In this experiment, particularly with the *FI-B* dataset (Figure 6.9), higher DGS learning rates proved beneficial up to a certain threshold, unlike the learning rate. As for the number of softmax functions, 20 and 50 exhibited similar performance.

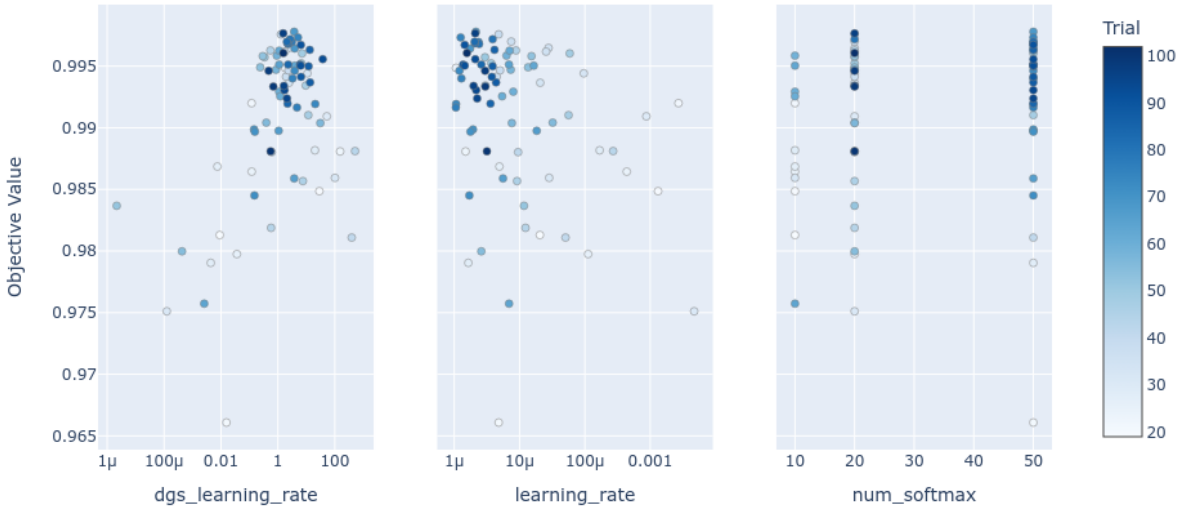


FIGURE 6.9: Example of hyperparameters' influence on node classification in FI-B dataset, with the objective value measured as AUC.

6.4 Summary

This chapter introduced *Deep-Graph-Sprints*, a novel approach to representation learning in CTDGs. This method efficiently learns time-sensitive embeddings, balancing rapid processing with resource efficiency. A pivotal aspect of *Deep-Graph-Sprints* is its capability to overcome the limitations inherent in the *Graph-Sprints* model, as detailed in Section 6.1. Furthermore, *Deep-Graph-Sprints* successfully navigates the challenges faced by existing GNNs in learning long-term dependencies. This is accomplished through the strategic implementation of forward-mode AD, a significant enhancement in advancing the capabilities of GNNs.

Extensive experiments were conducted to rigorously assess *Deep-Graph-Sprints*. As detailed in Tables 6.2, 6.3, and 6.4, our findings show that *Deep-Graph-Sprints* often surpasses both *Graph-Sprints* and state-of-the-art GNN baselines in performance. This is particularly evident in node classification and link prediction tasks, with notable results in the Mooc dataset. The method's performance in both transductive and inductive settings highlights its robustness and adaptability across various CTDG scenarios.

In terms of inference speed, *Deep-Graph-Sprints* maintains similar or faster processing times compared to *Graph-Sprints*, especially in high-feature datasets like Wikipedia and Reddit. This efficiency is a significant advantage over slower GNN models, such as *TGN-attn*, with *Deep-Graph-Sprints* being over an order of magnitude faster in certain cases.

In conclusion, the *Deep-Graph-Sprints* method allows to achieve state-of-the-art performance with end-to-end learning of parameters and therefore less tuning effort than the graph sprint method, while keeping the desirable low-latency properties.

6.5 Future Work

Future developments in the *Deep-Graph-Sprints* method are envisioned to encompass several pivotal areas of enhancement.

Advanced Optimization Techniques:

The incorporation of an advanced optimization algorithms such as Adam could potentially replace the current use of stochastic gradient descent in updating the DGS parameters during forward-mode AD. The adoption of these advanced techniques is anticipated to facilitate the learning of optimal parameters and therefore more easily achieving good performance.

Extension to Heterogeneous Graphs:

A significant area of future research for the *Deep-Graph-Sprints* method is its adaptation to heterogeneous graph structures. This evolution aims to extend the method's applicability across a diverse array of graph-based problem domains. One viable approach is to employ distinct embedding matrices for different node or edge types, where each matrix maps varied input features to a consistent embedding dimension. This method, however, presents challenges in mini-batch training, necessitating the segregation of edges by type for effective mapping.

An alternative approach involves implementing a unique update process for each node or edge type, resulting in the creation of multiple sub-embeddings. These sub-embeddings would then be concatenated to form a unified final embedding. The key consideration here is determining whether to learn a singular embedding matrix W per node or edge type, or to learn a complete set of parameters — $\vec{\alpha}$, $\vec{\beta}$, and W — for each type. The decision on this architecture is likely to be dependent on the use cases.

Central to this adaptation is that the structure of the model needs to be defined, after which the parameters are learned autonomously during training, obviating the need for manual adjustments as required in *Graph-Sprints*.

Input-Dependent Parameters:

A significant enhancement under consideration involves enabling input-dependent adaptability for the parameters alpha ($\vec{\alpha}$) and beta ($\vec{\beta}$). This advancement aims to amplify the model's responsiveness to diverse data inputs. As such, the state update equation would undergo modification, integrating trainable matrices W_1, and W_2 of dimension $(s \times f)$, where s represents the embedding size and f denotes the number of input features.

To ensure the parameter values remain within the range [0, 1], a sigmoid function per feature, denoted as ψ, can be employed. The parameters are thus defined as:

$$\vec{\alpha} = \psi(W_1 F_t),$$

and

$$\vec{\beta} = \psi(W_2 F_t).$$

The revised state update formula, encapsulated in Equation 6.28, effectively integrates these changes:

$$\vec{S}_t = \psi(W_2 F_t) \odot S_{t-1} + (1 - \psi(W_2 F_t)) \odot \Big((1 - \psi(W_1 F_t)) \odot \vec{\sigma}(W F_t) + \psi(W_1 F_t) \odot \vec{S}^*_{t-1} \Big) \quad (6.28)$$

This modification enhances the accuracy of the *Deep-Graph-Sprints* model in processing varying inputs.

These future directions represent critical steps towards augmenting the sophistication and versatility of the *Deep-Graph-Sprints* model, positioning it at the forefront of graph representation learning tools.

Chapter 7

Conclusions and future work

Money laundering primarily seeks to hide the origins of illicit funds originating from crimes such as drug trafficking, human trafficking, fraud, tax evasion, and corruption. This is achieved by strategically transferring these funds through a network of interlinked accounts to represent them as legitimate assets. An easy laundering of these funds supports underlying criminal activities, subsequently impacting individuals, economies, governmental stability, and social well-being [McDowell and Novis, 2001].

To incentivize FIs in their efforts against money laundering, governmental regulations outline criteria indicating which transfers merit review. Institutions are met with rigorous sanctions for breaches, including substantial fines for overlooked laundering activities.

In this doctoral research, we have adopted a graph-based perspective to address money laundering detection, elucidating innovative strategies and delving into the complexities of transaction networks.

Our journey through AML systems in banking underscored the significance of transactional data and the relationships between entities. This exploration led to the creation of the *Walking-Profiles* framework—a graph feature engineering approach grounded in the dynamics of random-walks to extract pivotal graph-based features.

Moreover, in response to the persistent issue of false alerts in AML systems, we introduced the *triage model*. This ML approach evaluates alerts against established AML criteria, assigning them relevance scores to streamline and clarify the decision-making process.

Considering the escalating volume of financial transactions, we presented *Graph-Sprints*, crafted for CTDGs. Prioritizing computational efficiency, this technique is adept for real-time implementations.

Advancing towards a comprehensive solution that overcomes the limitations of *Graph-Sprints*, the *Deep-Graph-Sprints* methodology was created. This approach represents a low-latency representation learning method, combining the advantages of *Graph-Sprints* with the capabilities of deep learning methods.

Comprehensive evaluations across diverse datasets validated the adaptability and potency of our methodologies. Our proposed methods, namely, *Walking-Profiles*, *Graph-Sprints*, and *Deep-Graph-Sprints* were subjected to performance assessments, emphasizing their applicability within and beyond the AML sphere.

In summation, this research contributes to the broader field of graph representation learning. Moreover, it sets the foundation for future research to enhance these methods and explore their application in a range of potential areas.

7.1 Main Contributions

1. **Creation of a framework for Graph-based Feature Extraction:** Recognizing the significance of the insights that could be encoded in graph data, we developed the graph feature engineering framework, named *Walking-Profiles* (Section 4.2). By leveraging random-walks, this framework extracts graph-based features, that can be later used in any downstream system (e.g., ML model).

2. **Formulating a Comprehensive ML Pipeline for AML Systems:** Addressing the prevalent issue of false alerts in AML systems, we introduce an ML-centric methodology termed the *triage model* (Section 4.3). This model processes alerts generated by pre-defined rules in AML systems. It assigns scores to these alerts, which then either facilitate the suppression of low-priority alerts or order the alert queue based on severity. An intrinsic advantage of our approach is its ability to maintain compliance and offer clear explainability. Since every alert originates from established rules, the process remains transparent and interpretable.

3. **Design of a Real-time Graph feature engineering Approach for CTDGs:** With the increasing volume and velocity of financial transactions, it is imperative to have techniques

that are both robust and efficient. Thus, we developed *Graph-Sprints* (Section 5.2) a method optimized for CTDGs. This method minimizes computational overheads and memory usage, making it suitable for real-time deployment (e.g., in an AML scenario).

4. **Design of a Real-time Graph Representation Learning Approach for CTDGs:** We present *Deep-Graph-Sprints*, a pioneering approach in real-time graph representation learning, as detailed in Section 6.2. This methodology successfully addresses the high latency challenges of contemporary deep learning methods while eliminating the need for manual tuning and domain-specific knowledge required by traditional feature extraction techniques. *Deep-Graph-Sprints* combines the principles of *Graph-sprints* and deep learning, offering an innovative solution for CTDGs.

5. **Rigorous Evaluation of the Proposed Frameworks:** To assess the performance and broad applicability of our approaches, we undertook a comprehensive evaluation phase. We tested the performance of both *triage model* (Section 4.4) and *Graph-Sprints* (Section 5.4.3) across varied AML datasets. Further, to underscore the adaptability of our methods, we also evaluated *Graph-Sprints* on datasets from diverse domains (Section 5.4.2), demonstrating its utility beyond the AML domain.

7.2 Future Research Directions

In this section, we start by enumerating the limitations of this research, followed by a discussion on future directions to enhance and expand the utility of our methodologies.

7.2.1 Limitations

Acknowledged limitations include:

- **Data Confidentiality:** This research utilizes a combination of publicly accessible datasets and proprietary datasets from Feedzai. Due to privacy constraints, specific details pertaining to the internal AML datasets cannot be disclosed.
- **Homogeneity Assumption:** The methodologies proposed herein operate under the assumption that all nodes and edges within the graph are homogenous, as elaborated in Section 2.1.1.

- **Algorithmic Scalability:** While the algorithm is designed for low latency, evaluating its performance and efficiency on significantly large graphs (more than a million nodes) remains a topic future work.

Further research will seek to address these limitations and enhance the framework's adaptability.

1. Expanding the scope of the *Deep-Graph-Sprints* framework to more complex graphs, such as heterogeneous graphs. Given the complexity and richness of data these graphs offer, exploring such avenue could extend the utility of our frameworks, further details in Section 6.5.

2. Assessing the scalability and efficiency of *Graph-Sprints*, the *Deep-Graph-Sprints* when applied to larger networks could provide insights into their potential real-world applicability across various sectors.

3. It is crucial to explore the application of the *Graph-Sprints*, and *Deep-Graph-Sprints* frameworks in diverse tasks, including community detection. Extending their use to other practical domains, such as fraud detection, is important to fully leverage their potential and broaden their applicability in graph-based research. This exploration will not only validate their versatility but also potentially reveal new insights and improvements in diverse areas of study.

4. Exploration of advanced optimization methods, notably the Adam algorithm, to enhance or replace Stochastic Gradient Descent (SGD) in the *Deep-Graph-Sprints* framework. This strategy is aimed at improving convergence efficiency and strengthening model robustness, further details in Section 6.5.

5. Enhancement of the *Deep-Graph-Sprints* model's responsiveness by making αandβ parameters depend on data inputs, further details in Section 6.5.

7.3 Closing Remarks

This Ph.D. is a collaboration between the University of Porto and Feedzai, a leading risk prevention company.

We hope our work towards detecting money laundering leveraging graphs lays the ground for more applications of graph-based techniques in the fight against financial crime.

Bibliography

Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd international conference on World Wide Web*, pages 37–48, 2013. [Cited on page 39.]

Nesreen K Ahmed, Ryan A Rossi, John Boaz Lee, Theodore L Willke, Rong Zhou, Xiangnan Kong, and Hoda Eldardiry. role2vec: Role-based network embeddings. *Proc. DLG KDD*, pages 1–7, 2019. [Cited on page 40.]

Huseyin Ahmetoglu and Resul Das. A comprehensive review on detection of cyber-attacks: Data sets, methods, challenges, and future research directions. *Internet of Things*, page 100615, 2022. [Cited on page 44.]

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019. [Cited on pages 70, 73, 92, and 124.]

Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004. [Cited on page 57.]

Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 635–644, 2011. [Cited on page 21.]

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018. [Cited on page 26.]

Smriti Bhagat, Graham Cormode, and S Muthukrishnan. Node classification in social networks. In *Social network data analytics*, pages 115–148. Springer, 2011. [Cited on page 21.]

C Bishop. Pattern recognition and machine learning. *Springer google schola*, 2:531–537, 2006. [Cited on page 23.]

Bernardo Branco, Pedro Abreu, Ana Sofia Gomes, Mariana SC Almeida, João Tiago Ascensão, and Pedro Bizarro. Interleaved sequence rnns for fraud detection. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3101–3109, 2020. [Cited on pages 7, 24, and 69.]

Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998. [Cited on page 46.]

Ramiro Daniel Camino, Radu State, Leandro Montero, and Petko Valtchev. Finding suspicious activities in financial transactions and distributed ledgers. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 787–796. IEEE, 2017. [Cited on page 45.]

Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pages 891–900, 2015. [Cited on page 40.]

Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002. [Cited on page 86.]

Fenxiao Chen, Yun-Cheng Wang, Bin Wang, and C-C Jay Kuo. Graph representation learning: a survey. *APSIPA Transactions on Signal and Information Processing*, 9:e15, 2020. [Cited on page 40.]

Zhiyuan Chen, Ee Na Teoh, Amril Nazir, Ettikan Kandasamy Karuppiah, Kim Sim Lam, et al. Machine learning techniques for anti-money laundering (aml) solutions in suspicious transaction detection: a review. *Knowledge and Information Systems*, 57(2):245–285, 2018. [Cited on page 45.]

Tim Cooijmans and James Martens. On the variance of unbiased online recurrent optimization. *arXiv preprint arXiv:1902.02405*, 2019. [Cited on pages 26 and 27.]

L da F Costa, Francisco A Rodrigues, Gonzalo Travieso, and Paulino Ribeiro Villas Boas. Characterization of complex networks: A survey of measurements. *Advances in physics*, 56(1):167–242, 2007. [Cited on pages 12 and 21.]

Hanjun Dai, Yichen Wang, Rakshit Trivedi, and Le Song. Deep coevolutionary network: Embedding user and item features for recommendation. *arXiv preprint arXiv:1609.03675*, 2016. [Cited on page 43.]

Danske Bank Pleads Guilty to Fraud on U.S. Banks in Multi-Billion Dollar Scheme to Access the U.S. Financial System. Us department of justice, 2022. URL `https://www.justice.gov/opa/pr/danske-bank-pleads-guilty-fraud-us-banks-multi-billion-dollar-scheme-access-us-financial`. Accessed: 24-08-2023. [Cited on page 2.]

Kousik Das, Sovan Samanta, and Madhumangal Pal. Study on centrality measures in social networks: a survey. *Social network analysis and mining*, 8(1):13, 2018. [Cited on page 18.]

Resul Das and Mucahit Soylu. A key review on graph data science: The power of graphs in scientific studies. *Chemometrics and Intelligent Laboratory Systems*, page 104896, 2023. [Cited on page 15.]

Deutsche Bank fined $630m over Russia money laundering claims. The guardian, 2017. URL `https://www.theguardian.com/business/2017/jan/31/deutsche-bank-fined-630m-over-russia-money-laundering-claims`. Accessed: 24-08-2023. [Cited on page 2.]

Annette J Dobson and Adrian G Barnett. *An introduction to generalized linear models*. CRC press, 2018. [Cited on page 70.]

Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 135–144, 2017. [Cited on page 40.]

Rafał Dreżewski, Jan Sepielak, and Wojciech Filipkowski. System supporting money laundering detection. *Digital Investigation*, 9(1):8–21, 2012. [Cited on page 45.]

Rafał Dreżewski, Jan Sepielak, and Wojciech Filipkowski. The application of social network analysis algorithms in a system supporting money laundering detection. *Information Sciences*, 295:18–32, 2015. [Cited on page 46.]

Ahmad Naser Eddin, Jacopo Bono, David Aparício, David Polido, Joao Tiago Ascensao, Pedro Bizarro, and Pedro Ribeiro. Anti-money laundering alert optimization using machine learning with graphs. *arXiv preprint arXiv:2112.07508*, 2021. [Cited on page 9.]

Ahmad Naser Eddin, Jacopo Bono, David Aparício, Hugo Ferreira, João Ascensão, Pedro Ribeiro, and Pedro Bizarro. From random-walks to graph-sprints: a low-latency node embedding framework on continuous-time dynamic graphs. *arXiv preprint arXiv:2307.08433*, 2023a. [Cited on page 9.]

Ahmad Naser Eddin, Jacopo Bono, David Aparício, Hugo Ferreira, João Tiago Ascensão, Pedro Ribeiro, and Pedro Bizarro. From random-walks to graph-sprints: a low-latency node embedding framework on continuous-time dynamic graphs. In *Proceedings of the Fourth ACM International Conference on AI in Finance*, pages 176–184, 2023b. [Cited on page 9.]

Ahmad Naser Eddin, Jacopo Bono, João Tiago Barriga Negra Ascensão, and Pedro Gustavo Santos Rodrigues Bizarro. Triaging alerts using machine learning, May 11 2023c. US Patent App. 17/831,199. [Cited on page 9.]

Hassan Eldeeb, Shota Amashukeli, and Radwa ElShawi. Bigfeat: Scalable and interpretable automated feature engineering framework. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 515–524. IEEE, 2022. [Cited on page 36.]

Hugo Jair Escalante. Automated machine learning—a brief review at the end of the early years. *Automated Design of Machine Learning and Search Algorithms*, pages 11–28, 2021. [Cited on page 37.]

Falih Gozi Febrinanto, Feng Xia, Kristen Moore, Chandra Thapa, and Charu Aggarwal. Graph lifelong learning: A survey. *IEEE Computational Intelligence Magazine*, 18(1):32–51, 2023. [Cited on page 12.]

Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. [Cited on pages 92, 113, 114, 122, and 123.]

Zengan Gao. Application of cluster-based local outlier factor algorithm in anti-money laundering. In *2009 International Conference on Management and Service Science*, pages 1–4. IEEE, 2009. [Cited on page 45.]

Goldman Sachs settles 1MDB scandal with Malaysia for $3.9bn. Bbc news. https://www.bbc.com/news/business-53529075, 2020. Accessed: 24-08-2023. [Cited on page 2.]

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org. [Cited on page 25.]

Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. Dyngem: Deep embedding method for dynamic graphs. *arXiv preprint arXiv:1805.11273*, 2018. [Cited on page 43.]

Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? *Advances in Neural Information Processing Systems*, 35:507–520, 2022. [Cited on page 23.]

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016. [Cited on page 40.]

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017a. [Cited on page 42.]

William L Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020. [Cited on page 39.]

William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017b. [Cited on pages 5 and 39.]

Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995. [Cited on pages 23, 64, and 70.]

Franziska Horn, Robert Pack, and Michael Rieger. The autofeat python library for automated feature engineering and selection. In *Machine Learning and Knowledge Discovery in Databases: International Workshops of ECML PKDD 2019, Würzburg, Germany, September 16–20, 2019, Proceedings, Part I*, pages 111–120. Springer, 2020. [Cited on page 37.]

Roger A Horn. The hadamard product. In *Proc. Symp. Appl. Math*, volume 40, pages 87–169, 1990. [Cited on page 108.]

Yining Hu, Suranga Seneviratne, Kanchana Thilakarathna, Kensuke Fukuda, and Aruna Seneviratne. Characterizing and detecting money laundering activities on the bitcoin network. *arXiv preprint arXiv:1912.12060*, 2019. [Cited on page 46.]

Yiran Huang, Yexu Zhou, Michael Hefenbrock, Till Riedel, Likun Fang, and Michael Beigl. Automatic feature engineering through monte carlo tree search. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 581–598. Springer, 2022. [Cited on page 36.]

ING to Pay $900 Million to End Dutch Money Laundering Probe. Bloomberg, 2018. URL `https://www.bloomberg.com/news/articles/2018-09-04/ing-to-pay-784-million-in-fines-to-settle-dutch-criminal-case`. Accessed: 24-08-2023. [Cited on page 2.]

Sergey Ivanov and Evgeny Burnaev. Anonymous walk embeddings. In *International conference on machine learning*, pages 2186–2195. PMLR, 2018. [Cited on pages 37, 38, and 55.]

Di Jin, Mark Heimann, Ryan A Rossi, and Danai Koutra. node2bits: Compact time-and attribute-aware node representations for user stitching. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 483–506. Springer, 2019. [Cited on pages 41, 52, and 88.]

Di Jin, Sungchul Kim, Ryan A Rossi, and Danai Koutra. From static to dynamic node embeddings. *arXiv preprint arXiv:2009.10017*, 2020. [Cited on page 43.]

Di Jin, Sungchul Kim, Ryan A Rossi, and Danai Koutra. On generalizing static node embedding to dynamic settings. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, pages 410–420, 2022a. [Cited on page 43.]

Ming Jin, Yuan-Fang Li, and Shirui Pan. Neural temporal walks: Motif-aware representation learning on continuous-time dynamic graphs. In *Advances in Neural Information Processing Systems*, 2022b. [Cited on page 41.]

Ming Jin, Huan Yee Koh, Qingsong Wen, Daniele Zambon, Cesare Alippi, Geoffrey I Webb, Irwin King, and Shirui Pan. A survey on graph neural networks for time series: Forecasting, classification, imputation, and anomaly detection. *arXiv preprint arXiv:2307.03759*, 2023. [Cited on page 44.]

Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015. [Cited on page 22.]

Martin Jullum, Anders Løland, Ragnar Bang Huseby, Geir Ånonsen, and Johannes Lorentzen. Detecting money laundering transactions with machine learning. *Journal of Money Laundering Control*, 2020. [Cited on pages 45, 71, and 73.]

George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997. [Cited on page 42.]

Gilad Katz, Eui Chul Richard Shin, and Dawn Song. Explorekit: Automatic feature generation and selection. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 979–984. IEEE, 2016. [Cited on page 36.]

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. *Advances in neural information processing systems*, 30:3146–3154, 2017. [Cited on pages 23, 50, 64, 70, and 73.]

Liu Keyan and Yu Tingting. An improved support-vector network model for anti-money laundering. In *2011 Fifth International Conference on Management of e-Commerce and e-Government*, pages 193–196. IEEE, 2011. [Cited on page 45.]

Roheena Q Khan, Malcolm W Corney, Andrew J Clark, and George M Mohay. Transaction mining for fraud detection in erp systems. *Industrial engineering and management systems*, 9(2): 141–156, 2010. [Cited on page 4.]

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. [Cited on page 122.]

Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, 2011. [Cited on page 19.]

Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2019. [Cited on pages 43, 92, 94, and 123.]

Karel Lannoo and Richard Parlour. Anti-money laundering in the eu: Time to get serious. ceps task force report 28 jan 2021. *UNSPECIFIED*, 2021. [Cited on pages 1, 3, 33, and 60.]

Asma S Larik and Sajjad Haider. Clustering based anomalous transaction reporting. *Procedia Computer Science*, 3:606–610, 2011. [Cited on page 45.]

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015. [Cited on page 24.]

John Boaz Lee, Giang Nguyen, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. Dynamic node embeddings from edge streams. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 5(6):931–946, 2020. [Cited on page 41.]

Xiangfeng Li, Shenghua Liu, Zifeng Li, Xiaotian Han, Chuan Shi, Bryan Hooi, He Huang, and Xueqi Cheng. Flowscope: Spotting money laundering based on graphs. In *AAAI*, pages 4731–4738, 2020. [Cited on pages 4 and 46.]

Xujia Li, Yuan Li, Xueying Mo, Hebing Xiao, Yanyan Shen, and Lei Chen. Diga: Guided diffusion model for graph recovery in anti-money laundering. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4404–4413, 2023. [Cited on page 47.]

Xurui Li, Xiang Cao, Xuetao Qiu, Jintao Zhao, and Jianbin Zheng. Intelligent anti-money laundering solution based upon novel community detection in massive transaction networks on spark. In *2017 fifth international conference on advanced cloud and big data (CBD)*, pages 176–181. IEEE, 2017. [Cited on pages 3 and 60.]

Xi Liu, Ping-Chun Hsieh, Nick Duffield, Rui Chen, Muhe Xie, and Xidao Wen. Real-time streaming graph embedding through local actions. In *Companion proceedings of the 2019 world wide web conference*, pages 285–293, 2019. [Cited on page 43.]

Xuan Liu and Pengzhu Zhang. A scan statistics based suspicious transactions detection model for anti-money laundering (aml) in financial institutions. In *2010 International Conference on Multimedia Communications*, pages 210–213. IEEE, 2010. [Cited on page 45.]

Xuan Liu, Pengzhu Zhang, and Dajun Zeng. Sequence matching for suspicious activity detection in anti-money laundering. In *International conference on intelligence and security informatics*, pages 50–61. Springer, 2008. [Cited on page 45.]

Joana Lorenz, Maria Inês Silva, David Aparício, João Tiago Ascensão, and Pedro Bizarro. Machine learning methods to detect money laundering in the bitcoin blockchain in the presence of label scarcity. In *Proceedings of the first ACM international conference on AI in finance*, pages 1–8, 2020. [Cited on pages 3 and 45.]

Jie Lu, Dianshuang Wu, Mingsong Mao, Wei Wang, and Guangquan Zhang. Recommender system application developments: a survey. *Decision Support Systems*, 74:12–32, 2015. [Cited on page 21.]

Devendra Kumar Luna, Girish Keshav Palshikar, Manoj Apte, and Arnab Bhattacharya. Finding shell company accounts using anomaly detection. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, pages 167–174, 2018. [Cited on page 45.]

Scott M Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. From local explanations to global understanding with explainable ai for trees. *Nature machine intelligence*, 2(1):56–67, 2020. [Cited on page 64.]

Xingrong Luo. Suspicious transaction detection for anti-money laundering. *International Journal of Security and Its Applications*, 8(2):157–166, 2014. [Cited on page 45.]

Batta Mahesh. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).[Internet]*, 9(1):381–386, 2020. [Cited on page 23.]

Abdul Majeed and Ibtisam Rauf. Graph theory: A comprehensive survey about graph theory applications in computer science and social networks. *Inventions*, 5(1):10, 2020. [Cited on page 12.]

Ilya Makarov, Dmitrii Kiselev, Nikita Nikitinsky, and Lovro Subelj. Survey on graph embeddings and their applications to machine learning problems on graphs. *PeerJ Computer Science*, 7:e357, 2021. [Cited on pages 21, 39, and 40.]

Paulo César Gonçalves Marques, Miguel Ramos de Araújo, Bruno Casal Laraña, Nuno Miguel Lourenço Diegues, Pedro Cardoso Lessa e Silva, and Pedro Gustavo Santos Rodrigues Bizarro. Semantic-aware feature engineering, March 19 2020. US Patent App. 16/567,761. [Cited on pages 37 and 69.]

John McDowell and Gary Novis. The consequences of money laundering and financial crime. *Economic Perspectives*, 6(2):6–10, 2001. [Cited on pages 2 and 137.]

Tijana Milenković, Weng Leong Ng, Wayne Hayes, and Nataša Pržulj. Optimal network alignment with graphlet degree vectors. *Cancer informatics*, 9:CIN–S4744, 2010. [Cited on page 19.]

Ron Milo, Shalev Itzkovitz, Nadav Kashtan, Reuven Levitt, Shai Shen-Orr, Inbal Ayzenshtat, Michal Sheffer, and Uri Alon. Superfamilies of evolved and designed networks. *Science*, 303 (5663):1538–1542, 2004. [Cited on page 19.]

Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961. [Cited on page 26.]

Asier Mujika, Florian Meier, and Angelika Steger. Approximating real-time recurrent learning with random kronecker factors. *Advances in Neural Information Processing Systems*, 31, 2018. [Cited on page 27.]

Fatemeh Nargesian, Horst Samulowitz, Udayan Khurana, Elias B Khalil, and Deepak S Turaga. Learning feature engineering for classification. In *Ijcai*, volume 17, pages 2529–2535, 2017. [Cited on page 36.]

Giang Hoang Nguyen, John Boaz Lee, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. Continuous-time dynamic network embeddings. In *Companion proceedings of the the web conference 2018*, pages 969–976, 2018. [Cited on page 41.]

Nguyen Thi Uyen Nhi, Thanh Manh Le, et al. A model of semantic-based image retrieval using c-tree and neighbor graph. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 18(1):1–23, 2022. [Cited on page 39.]

Catarina Oliveira, João Torres, Maria Inês Silva, David Aparício, João Tiago Ascensão, and Pedro Bizarro. Guiltywalker: Distance to illicit nodes in the bitcoin network. *arXiv preprint arXiv:2102.05373*, 2021. [Cited on pages 45, 63, 71, 72, and 87.]

Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114, 2016. [Cited on page 40.]

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014. [Cited on pages 40 and 47.]

Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12):1876–1888, 2018. [Cited on pages 4 and 47.]

Leonardo FR Ribeiro, Pedro HP Saverese, and Daniel R Figueiredo. struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 385–394, 2017. [Cited on pages 21 and 40.]

Pedro Ribeiro. *Efficient and scalable algorithms for network motifs discovery*. PhD thesis, PhD thesis, University of Porto, 2011. [Cited on page 17.]

Pedro Ribeiro and Fernando Silva. Discovering colored network motifs. In *Complex Networks V*, pages 107–118. Springer, 2014. [Cited on page 19.]

Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys (CSUR)*, 54(2):1–36, 2021. [Cited on pages 19 and 41.]

Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. In *ICML 2020 Workshop on Graph Representation Learning*, 2020. [Cited on pages 14, 43, 92, 94, 111, and 123.]

Hooman Peiro Sajjad, Andrew Docherty, and Yuriy Tyshetskiy. Efficient representation learning using random walks for dynamic graphs. *arXiv preprint arXiv:1901.01346*, 2019. [Cited on page 41.]

Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th international conference on web search and data mining*, pages 519–527, 2020. [Cited on page 43.]

Purnamrita Sarkar and Andrew W Moore. Fast nearest-neighbor search in disk-resident graphs. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 513–522, 2010. [Cited on page 42.]

David Savage, Qingmai Wang, Pauline Chou, Xiuzhen Zhang, and Xinghuo Yu. Detection of money laundering groups using supervised learning in networks. *arXiv preprint arXiv:1608.00708*, 2016. [Cited on page 46.]

Qitao Shi, Ya-Lin Zhang, Longfei Li, Xinxing Yang, Meng Li, and Jun Zhou. Safe: Scalable automatic feature engineering framework for industrial tasks. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1645–1656. IEEE, 2020. [Cited on page 36.]

Vahid Shirbisheh. Local graph embeddings based on neighbors degree frequency of nodes. *arXiv preprint arXiv:2208.00152*, 2022. [Cited on page 38.]

Amr Ehab Muhammed Shokry, Mohammed Abo Rizka, and Nevine Makram Labib. Counter terrorism finance by detecting money laundering hidden networks using unsupervised machine learning algorithm. In *International Conferences ICT, Society, and Human Beings*, 2020. [Cited on page 45.]

Maria Inês Silva, David Oliveira Aparício, Ahmad Naser Eddin, Jacopo Bono, João Tiago Barriga Negra Ascensão, and Pedro Gustavo Santos Rodrigues Bizarro. Graph traversal for measurement of fraudulent nodes, June 23 2022. US Patent App. 17/553,265. [Cited on page 9.]

Standard Chartered fined $1.1bn for money-laundering and sanctions breaches. The guardian, 2019. URL `https://www.theguardian.com/business/2019/apr/09/standard-chartered-fined-money-laundering-sanctions-breaches`. Accessed: 24-08-2023. [Cited on page 2.]

Xiaobing Sun, Jiabao Zhang, Qiming Zhao, Shenghua Liu, Jinglei Chen, Ruoyu Zhuang, Huawei Shen, and Xueqi Cheng. Cubeflow: Money laundering detection with coupled tensors. In *PAKDD (1)*, pages 78–90. Springer, 2021. [Cited on page 46.]

Richard Stuart Sutton. *Temporal credit assignment in reinforcement learning*. University of Massachusetts Amherst, 1984. [Cited on page 26.]

Corentin Tallec and Yann Ollivier. Unbiased online recurrent optimization. *arXiv preprint arXiv:1702.05043*, 2017. [Cited on page 27.]

Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077, 2015. [Cited on pages 21 and 40.]

Jun Tang and Jian Yin. Developing an intelligent data discriminating system of anti-money laundering based on svm. In *2005 International conference on machine learning and cybernetics*, volume 6, pages 3453–3457. IEEE, 2005. [Cited on page 45.]

The united nations office of drugs and crime. United nations, 2020. URL `https://www.unodc.org/unodc/en/money-laundering/laundrycycle.html`. [Cited on page 2.]

Milind Tiwari, Adrian Gepp, and Kuldeep Kumar. A review of money laundering literature: the state of research in key areas. *Pacific Accounting Review*, 2020. [Cited on page 45.]

Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast random walk with restart and its applications. In *Sixth international conference on data mining (ICDM'06)*, pages 613–622. IEEE, 2006. [Cited on page 42.]

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. [Cited on page 109.]

Adilson Vital Jr, Filipi N Silva, and Diego R Amancio. Comparing biased random walks in graph embedding and link prediction. *arXiv preprint arXiv:2308.03636*, 2023. [Cited on page 42.]

Dominik Wagner. Latent representations of transaction network graphs in continuous vector spaces as features for money laundering detection. *SKILL 2019-Studierendenkonferenz Informatik*, 2019. [Cited on page 47.]

Hongwei Wang, Hongyu Ren, and Jure Leskovec. Entity context and relational paths for knowledge graph completion. *arXiv preprint arXiv:2002.06757*, 2020. [Cited on page 21.]

Lili Wang, Chenghan Huang, Weicheng Ma, Ruibo Liu, and Soroush Vosoughi. Hyperbolic node embedding for temporal networks. *Data Mining and Knowledge Discovery*, 35(5):1906–1940, 2021a. [Cited on page 41.]

Peng Wang, BaoWen Xu, YuRong Wu, and XiaoYu Zhou. Link prediction in social networks: the state-of-the-art. *Science China Information Sciences*, 58(1):1–38, 2015. [Cited on page 21.]

Xingqi Wang and Guang Dong. Research on money laundering detection based on improved minimum spanning tree clustering and its application. In *2009 Second international symposium on knowledge acquisition and modeling*, volume 2, pages 62–64. IEEE, 2009. [Cited on page 45.]

Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 international conference on management of data*, pages 2628–2638, 2021b. [Cited on pages 43 and 96.]

Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. Inductive representation learning in temporal networks via causal anonymous walks. *arXiv preprint arXiv:2101.05974*, 2021c. [Cited on page 41.]

Mark Weber, Jie Chen, Toyotaro Suzumura, Aldo Pareja, Tengfei Ma, Hiroki Kanezashi, Tim Kaler, Charles E Leiserson, and Tao B Schardl. Scalable graph learning for anti-money laundering: A first look. *arXiv preprint arXiv:1812.00076*, 2018. [Cited on pages 3 and 33.]

Mark Weber, Giacomo Domeniconi, Jie Chen, Daniel Karl I Weidele, Claudio Bellei, Tom Robinson, and Charles E Leiserson. Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics. *arXiv preprint arXiv:1908.02591*, 2019. [Cited on page 45.]

Ronald J Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990. [Cited on page 27.]

Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989. [Cited on page 27.]

Wei Wu, Bin Li, Chuan Luo, and Wolfgang Nejdl. Hashing-accelerated graph neural networks for link prediction. In *Proceedings of the Web Conference 2021*, pages 2910–2920, 2021. [Cited on page 43.]

Yu Wu, Xin Xi, and Jieyue He. Afgsl: Automatic feature generation based on graph structure learning. *Knowledge-Based Systems*, 238:107835, 2022. [Cited on page 38.]

Feng Xia, Jiaying Liu, Hansong Nie, Yonghao Fu, Liangtian Wan, and Xiangjie Kong. Random walks: A review of algorithms and applications. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 4(2):95–107, 2019. [Cited on page 42.]

Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First international workshop on graph data management experiences and systems*, pages 1–6, 2013. [Cited on page 57.]

Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962*, 2020. [Cited on page 43.]

Li Xu, Jimmy SJ Ren, Ce Liu, and Jiaya Jia. Deep convolutional neural network for image deconvolution. In *Advances in neural information processing systems*, pages 1790–1798, 2014. [Cited on page 24.]

Carl Yang, Aditya Pal, Andrew Zhai, Nikil Pancha, Jiawei Han, Charles Rosenberg, and Jure Leskovec. Multisage: Empowering gcn with contextualized multi-embeddings on web-scale

multipartite networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2434–2443, 2020. [Cited on page 42.]

Dingqi Yang, Bingqing Qu, Jie Yang, Liang Wang, and Philippe Cudre-Mauroux. Streaming graph embeddings via incremental neighborhood sketching. *IEEE Transactions on Knowledge and Data Engineering*, 35(5):5296–5310, 2022. [Cited on page 43.]

Yan Yang, Bin Lian, Lian Li, Chen Chen, and Pu Li. Dbscan clustering algorithm applied to identify suspicious financial transactions. In *2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 60–65. IEEE, 2014. [Cited on page 45.]

Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983, 2018. [Cited on pages 42 and 44.]

Jianyu Zhang, Françoise Fogelman-Soulié, and Christine Largeron. Towards automatic complex feature engineering. In *Web Information Systems Engineering–WISE 2018: 19th International Conference, Dubai, United Arab Emirates, November 12-15, 2018, Proceedings, Part II 19*, pages 312–322. Springer, 2018. [Cited on page 37.]

Jianyu Zhang, Jianye Hao, Françoise Fogelman-Soulié, and Zan Wang. Automatic feature engineering by deep reinforcement learning. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2312–2314, 2019. [Cited on page 36.]

Jianyu Zhang, Jianye Hao, and Françoise Fogelman-Soulié. Cross-data automatic feature engineering via meta-learning and reinforcement learning. In *Advances in Knowledge Discovery and Data Mining: 24th Pacific-Asia Conference, PAKDD 2020, Singapore, May 11–14, 2020, Proceedings, Part I 24*, pages 818–829. Springer, 2020a. [Cited on page 36.]

Yan Zhang and Peter Trubey. Machine learning and sampling scheme: An empirical study of money laundering detection. *Computational Economics*, 54(3):1043–1063, 2019. [Cited on page 45.]

Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):249–270, 2020b. [Cited on page 12.]

Vincent W Zheng. Engineering graph features via network functional blocks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 5749–5753, 2018. [Cited on page 37.]

Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020. [Cited on page 12.]

Qi Zhu, Hao Wei, Bunyamin Sisman, Da Zheng, Christos Faloutsos, Xin Luna Dong, and Jiawei Han. Collective multi-type entity alignment between knowledge graphs. In *Proceedings of The Web Conference 2020*, pages 2241–2252, 2020. [Cited on pages 42 and 44.]