Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Artificial Intelligence

Péter Dániel Markó

AI-BASED GRAPH EMBEDDING TECHNIQUES FOR SOFTWARE PRODUCT DATA

Bachelor's Thesis

ADVISORS

László Toka (DSc), Máté Nagy (PhD)

BUDAPEST, 2024

Table of Content

HALLGATÓI NYILATKOZAT

Alulírott **Markó Péter Dániel**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2024. 12. 05.

...…….

Markó Péter Dániel

Kivonat

A szoftverfejlesztésben az olyan feladatok, mint a funkciók implementálása, hibák javítása, dokumentáció készítése és a rutin tevékenységek kezelése jelentős kihívásokat jelentenek, különösen a hosszú élettartamú, összetett szoftverrendszerek esetében. Ezeket a kihívásokat tovább fokozza a fejlesztők gyakori cserélődése, mivel az ilyen termékek élettartama gyakran meghaladja az eredeti fejlesztőcsapatok részvételének időtartamát. Ez hatékonyságcsökkenéshez és megnövekedett költségekhez vezet, mivel az új fejlesztők nehezen értik meg az előző évek munkáját közvetlen útmutatás nélkül. A töredezett adatforrások, beleértve a forráskódot, a dokumentációt, a specifikációkat és a hibajegyeket, tovább súlyosbítják a problémát, megnehezítve az összes rendelkezésre álló információ hatékony kihasználását.

Bár a mesterséges intelligencia (MI) és a mélytanulás számos területen jelentős sikereket ért el, alkalmazásuk a szoftverfejlesztésben korlátozott maradt, gyakran szűken fókuszálva az egymodális eszközökre, mint például a kódgenerálás. A nagy nyelvi modellek például elsősorban szöveges adatokra támaszkodnak, és nem képesek feltárni a mélyebb szemantikai kapcsolatokat a különböző források között. Ennek következtében a dokumentációból, projektmenedzsment eszközökből és naplófájlokból származó értékes kontextuális információk gyakran kihasználatlanok maradnak, csökkentve az ilyen modellek hatékonyságát.

Szakdolgozatomban egy gráf alapú megközelítést javasolok összetett szoftvertermékek modellezésére, amely több adatmodalitást integrál egy egységes, szemantikailag gazdag tudásgráfba. A megközelítés értékeléséhez négy modellel végeztem kísérleteket - TransE, GAE, VGAE és GLM - olyan gráfokon, amelyeket különböző strukturális és csomóponti jellemzőkkel építettem fel. Ezeket a kísérleteket az ágyazások minőségét mérő (Quality of Embedding, QEmb) mutató vezérelte is vezérelte, amely felméri az ágyazások minőségét és segít optimalizálni a gráfstruktúrákat és adatforrásokat. Míg a TransE küzdött a valós gráfok összetettségével, és a VGAE a legtöbb esetben rosszabbul teljesített, mint a GAE, addig a GLM kiváló eredményeket ért el, de erőforrás-korlátok akadályozták az alkalmazását. A GAE felhasználásával egy ajánlórendszert építettem, amely jobb teljesítményt mutatott a szövegalapú alapvonalaknál, mint például a BM25 és a Sentence BERT. A GAE kihasználta a

tudásgráf strukturális és relációs mélységét, az ajánlások 42%-a kapott magas relevanciaértékelést (4 vagy 5), szemben az SBERT esetében mért 30%-kal. Emellett a szakértői értékelések változó nézőpontokat tártak fel az ajánlások hasznosságáról, kiemelve mind a pontos egyezések, mind a kapcsolódó kontextus megragadásának fontosságát.

Több modalitás integrálásával és a tudásgráfok relációs struktúrájának kihasználásával ez a megközelítés mélyebb betekintést nyújt a fejlesztői feladatkiosztásokba és a problémamegoldásba. Ezek az eredmények aláhúzzák a gráf alapú megközelítésekben rejlő lehetőségeket a hatékonyság és a szoftverminőség javítására összetett fejlesztési környezetekben.

Abstract

In software development, tasks such as implementing features, resolving bugs, creating documentation, and handling routine activities pose significant challenges, particularly in long-lived, complex software systems. These challenges are intensified by the frequent turnover of developers, as the lifespan of such products often exceeds the involvement of the original development teams. This leads to inefficiencies and increased costs, with new developers struggling to understand years of prior work without direct guidance. Fragmented data sources, including source code, documentation, specifications, and bug reports, further exacerbate the problem, making it harder to retrieve all available information effectively.

While artificial intelligence (AI) and deep learning have achieved substantial success across numerous fields, their application in software development has remained limited, often focusing narrowly on unimodal tools like code generation. Large Language Models (LLMs), for instance, rely primarily on textual data and fail to reveal deeper semantic relationships across diverse sources. Consequently, valuable contextual information from documentation, project management tools, and log files often goes unused, reducing the effectiveness of such models.

This work proposes a graph-based approach to model complex software products, integrating multiple data modalities into a unified, semantically rich knowledge graph. To evaluate this approach, I conducted experiments with four models - TransE, GAE, VGAE, and GLM - on graphs constructed with different structural and node feature configurations. These experiments were guided by the Quality of Embedding (QEmb) indicator, which assesses embedding quality and informs the optimization of graph structures and data sources. While TransE struggled with the complexity of real-world graphs, and VGAE performed worse than GAE in most cases, GLM achieved excellent results but was limited by resource constraints. I constructed a recommendation system using GAE, which demonstrated superior performance over text-based baselines such as BM25 and Sentence BERT. GAE utilizes the structural and relational depth of the knowledge graph, achieving 42% of recommendations rated as highly relevant (4 or 5) compared to 30% for SBERT. Additionally, expert evaluations revealed varying

perspectives on recommendation usefulness, highlighting the importance of capturing both exact matches and related context.

By integrating multiple modalities and using the relational structure of knowledge graphs, this approach provides deeper insights into developer assignments and issue resolution. These results underscore the potential of graph-based approaches to improve efficiency and software quality in complex development environments.

1 Introduction

Software products can be incredibly large and complex, often containing millions of lines of code spread across several modules and components. By incorporating multiple layers of abstraction, integrating diverse technologies, and providing a wide range of functionalities understanding and maintaining the product is becoming an increasingly challenging task for developers. Large-scale software systems inherently become complex over time, especially when developed by multiple people across many years, even with consistent adherence to best practices. Several factors contribute to this challenge, with these being just some of the key examples:

- Continuous development leads to evolving functionality and architecture.
- Enforcing design principles, if they exist, becomes increasingly difficult as the project grows.
- Diverse development practices and coding styles as new developers bring in their preferred methods, causing inconsistencies.
- Developers frequently rotate on projects, often only staying involved for a few years resulting in many developers having only a few years of professional experience. In one particular project at Ericsson, a quick study found that the ratio of junior to senior developers is one to four, illustrated in Figure 1.
- Increase in interdependencies among components that create tightly coupled modules, making modifications complex and risky.
- Updating documentation becomes progressively challenging as code changes.
- Integration of external libraries and APIs which adds dependencies that are prone to change, making maintenance and compatibility more challenging.

These examples highlight just some of the factors that drive complexity in large projects, yet there are countless additional challenges specific to each unique codebase, team structure, and software lifecycle. The software in this study is a telecommunication software where in addition to these general complexities, other unique challenges arise that further increase the difficulty of understanding and maintaining such systems and emphasize the need for advanced tools:

- Telecommunication systems often require five-nines (99.999%) availability, especially to support critical infrastructure like emergency services. Achieving this level of reliability demands extensive performance optimizations and low-latency processing, leading to heavily abstracted and highly optimized code that can be challenging to understand and maintain.
- Frequent signaling across processes, modules, and network functions changes in both format and meaning as it moves through the system. For example, a message may start as an internal signal and later be converted to a standardized 3GPP message, altering its interpretation slightly at each step.
- Telecommunication systems often rely on specialized languages and protocols that require domain expertise, making it harder for new developers to quickly understand the system and adding to overall complexity.

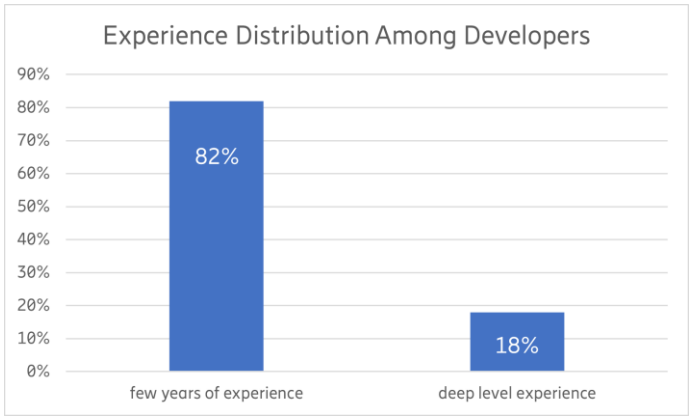

Figure 1: Due to the long lifespan of legacy codebases, there are often shifts in the people working on the project. This figure shows a snapshot of the experience distribution among developers for a specific project at Ericsson.

In such extensive and long-term projects, developers spend a substantial portion of their time navigating existing codebases instead of writing new code or implementing features. Minelli et al. highlight that developers, regardless of their experience level, must frequently search through various data sources like documentation, version control histories, and issue trackers to gather context on specific areas of the project [1]. Less experienced developers, in particular, face challenges in understanding the tangled details of these systems, but even experts are affected by the vast size and complexity of these repositories. In another study Damevski et al. found that, during feature location tasks, developers encounter difficulties with effective code search, underscoring the need for better integration between search, navigation, and debugging tools [2]. These challenges

impose a significant cost burden on tech companies, and retaining experienced developers can be expensive. Due to the seriousness of this issue and the rapid advancements in artificial intelligence (AI), recent research has increasingly focused on developing AI-driven tools that can improve developers' understanding of software and guide them through complex development processes.

This paper introduces a solution that uses the interconnected structure of software repositories, documentation, and issue tracking systems to build a unified representation of the project. Similar to recommendation systems in social networks or on platforms like Netflix, which use relationships within their data to suggest friends or content, this approach relies on the connections within software projects to create practical recommendations. By representing the project as a graph and applying Graph Neural Network (GNN) methods, we can identify links between components - such as issues, code files, and developer contributions - to recommend developers who are well-suited to handle new issues or to highlight files likely involved in specific bugs. To evaluate this approach, I compared four models: TransE, GAE, VGAE, and GLM, demonstrating the strengths and limitations of each. The possible applications of this graph-based method are broad, making it a promising tool for navigating and improving complex software projects.

To illustrate the limitations of current solutions and the potential benefits of the approach proposed in this paper, consider the example graph in Figure 2. As mentioned before, in real-world settings, developers often rotate on and off projects, leading to gaps in available expertise. This example shows how a model based solely on text comparisons would struggle to identify a suitable developer for a new issue. If a new issue (Issue-4) arrives, which is similar to Issue-2 and Issue-1 on textual level, but developers Person-1 and Person-2 are no longer available, traditional text-based methods may fail to suggest Person-3, even though Person-3 has experience working on a file associated with Issue-2. In cases where Issue-3’s description differs textually, these models would overlook Person-3 as a potential recommendation, while the solution in this paper would recommend Person-3.

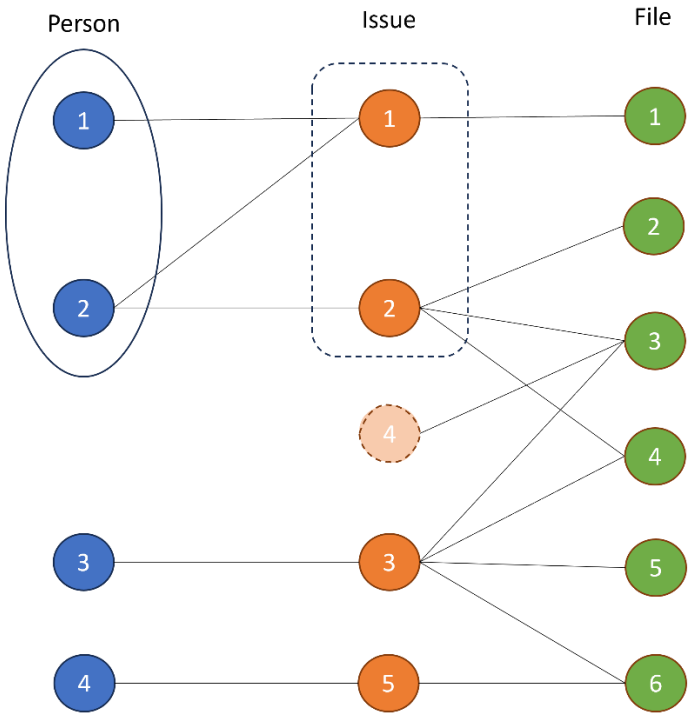

Figure 2: Illustration of a knowledge graph showing relationships between issues, files, and developers. Here, traditional text-based recommendation models struggle to suggest Person-3 for a new issue (Issue-4), as they rely solely on textual similarity. Although Person-3 has worked on a related file, they might be overlooked if descriptions differ. This highlights the advantage of a graph-based approach that captures structural relationships.

2 Problem statement

Despite recent advancements in AI and considerable research efforts, significant challenges remain. Tools like Codium assist with code generation but only operate at the function level, lacking insight into broader system architecture. Similarly, GitHub Copilot accelerates coding tasks by providing code suggestions yet is limited to code-level assistance without understanding the larger project-specific context. Likewise, ChatGPT and similar models cannot incorporate private or large datasets due to legal and context-size restrictions, making it impractical to analyze entire repositories and associated tickets, especially when data sizes are in the gigabyte range. These tools focus on a single type of data - analyzing source code and sometimes comments - but fall short in capturing the rich, interconnected relationships within a software project. This highlights the necessity of novel approaches that integrate diverse data sources, enabling developers to make more informed decisions and solve problems more effectively. Software development involves more than just code; it encompasses a complex system of elements, including issue reports, documentation, commit histories, and the developers themselves. Unimodal models focusing solely on textual information from source code miss the broader context provided by other data sources and cannot identify implicit connections between different entities. As illustrated in Figure 2 from the introduction, current solutions may fail to recommend relevant developers or files when textual similarity alone is insufficient to uncover underlying relationships in complex cases.

The scattered nature of information poses a substantial challenge. There are various components developers must navigate to understand a complex telecommunications software codebase. As developers often examine these sources independently, their understanding can become fragmented and inefficient. The absence of a unified view prevents the discovery of valuable insights hidden within the relationships between these entities.

To overcome these shortcomings, a graph-based approach offers a promising solution. By constructing a knowledge graph, the diverse data sources can be integrated into a single, unified representation. Nodes could represent entities like code files, functions, issues, documentation, and developers, while could represent the relationships between them. In Figure 3, a section of an example knowledge graph illustrates the

complexity of connections within the codebase. This representation makes implicit connections explicit, allowing developers to see how different components are interconnected. It addresses the limitations of unimodal models by modelling complex relationships that are not readily apparent when data sources are considered separately.

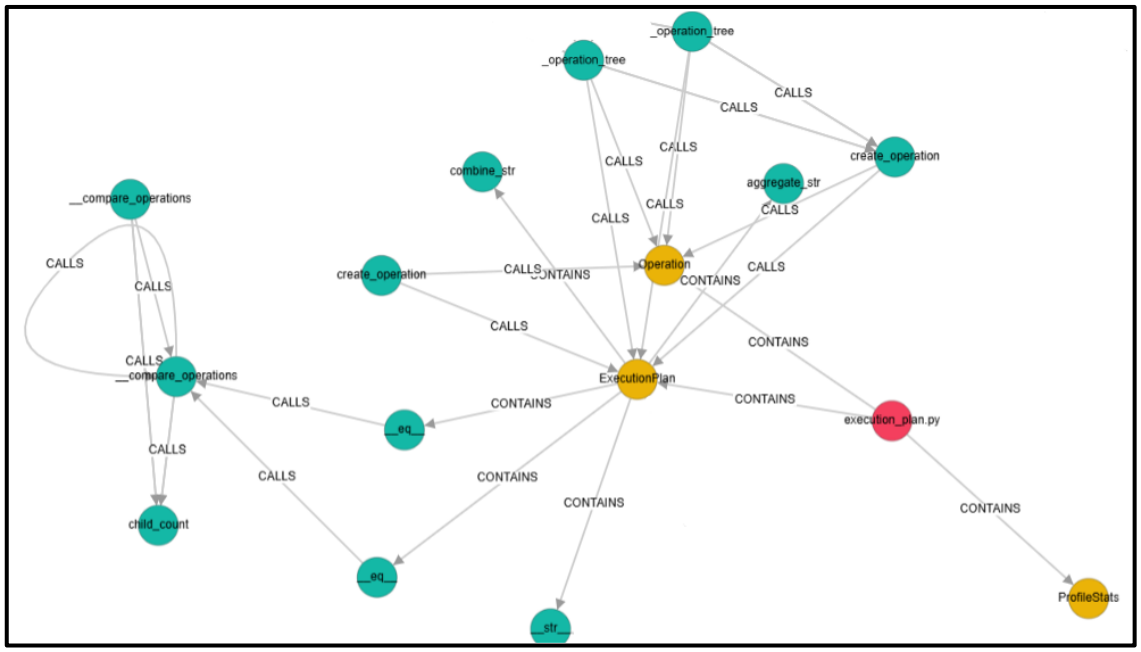

Figure 3: Example knowledge graph illustrating the complex relationships within a codebase. Nodes represent functions, classes, and files, while edges indicate relationships such as function calls and containment. This visualization highlights the intricate connections that can exist in large-scale software systems, underscoring the potential for a graph-based model to support deeper insights and effective recommendations [3].

The primary goal of this model is to recommend developers and files related to specific issues. As a result, developers can access necessary information more effectively, enhancing both decision-making and productivity. By adopting a semantically rich, graph-based representation, the diverse data modalities within a software project can be visualized and embedded into a single cohesive model. This allows for the development of an AI-powered recommendation system that does not yet exist in current tools, providing targeted suggestions for developer assignments and potential areas for bug investigation. This approach supports a deeper understanding of the project structure, reducing the time spent on manual information retrieval tasks by recommending related components and enabling developers to focus on productive activities such as coding, debugging, and implementing new features.

3 Related work

As for the recent years, the integration artificial intelligence into software development has emerged as a promising direction, reshaping the development practices. It also attracted significant research interest, leading by the rapid advancement of large language models (LLMs) and text-based encoders. Utilizing these novel approaches for understanding code, vulnerability detection, fault localization, code generation and developer recommendation resulted in some highly ranked publications.

In this section, I categorize the related work into distinct areas. First, I review foundational knowledge graphs and their applications in various domains. Then, I detail papers that focus on graph-based data representations in non-recommendation tasks within software development. In the next paragraph, I elaborate on non-GNN-based recommendation systems applied to software development. Finally, I discuss GNN-based recommendation frameworks and their adaptability across different contexts. This structured approach provides a comprehensive view of existing methodologies and positions my work within the current research landscape by showing how each category addresses various aspects of software development.

Knowledge graphs (KGs) have also shown significant promise in various domains, including software development, by providing structured, semantically rich representations of complex relationships. As noted in the study by Lu et al. KGs contribute to tasks like intelligent code recommendation, API suggestion, and vulnerability detection, yet the focus on software-specific KGs remains limited, failing to address the full complexity of development environments [4]. Similarly, the survey by Abu-Salih discusses advancements in domain-specific KGs in fields like healthcare and finance but highlights the lack of standardized construction techniques and evaluation metrics in the software engineering domain [5]. These limitations reveal a gap in applying KGs effectively to software development. In contrast, my work extends KG applications by constructing a software-specific KG that integrates diverse data sources, enabling deeper semantic connections and enhancing recommendation tasks critical to software development.

Graph-based data representations have proven valuable in enhancing various non-recommendation tasks within software development. For instance, Qiu et al. introduced

MGVD, a vulnerability detection framework using multiple graphs to capture code dependencies and control flow, improving precision in identifying vulnerable segments [6]. Similarly, Nayak et al. employed knowledge graphs to automate test case generation by extracting entities and relationships from engineering documents, aiding functional testing [7]. Furthermore, in another study an augmented code graph approach was proposed for software defect prediction, combining structural and semantic features to detect defect-prone components [8]. Additionally, DeepLink proposed by Xie et al. enhances issue-commit link recovery through code knowledge graphs, improving artifact traceability in project management [9], while another paper demonstrated that graph-based learning for fault localization significantly increases accuracy and fault coverage [10]. While these studies effectively utilize graph data, they are primarily focused on defect prediction, vulnerability detection, and fault localization rather than recommendation. My work diverges by applying graph-based representations specifically to recommendation tasks, integrating multiple data modalities and employing GNNs to enhance recommendation accuracy and utility in complex software environments.

Non-GNN-based recommendation systems have been applied to support software development tasks, such as artifact suggestion and reviewer assignment. Abu-Salih et al. developed a personalized recommender system for mobile app development, employing semantic web technologies and user profiling to recommend tools and code snippets for targeted developer needs [11]. Similarly, in 2021 a knowledge graph embedding-based approach was introduced to recommend APIs in Mashup development by capturing co-invocation patterns [12]. Additionally, Sülün et al.'s RSTrace+ suggests reviewers using artifact traceability links, incorporating recency data to improve recommendation accuracy [13]. While effective within specific applications, these methods do not use GNNs, limiting their ability to capture deeper relational insights in complex software datasets. My work addresses this by applying GNNs to enhance recommendation quality and adaptability across diverse software environments.

Graph neural networks play a crucial role in recommendation systems, effectively using complex, multi-relational data structures to improve recommendation quality. Wu et al. demonstrate GNNs' strength in capturing diverse relationships, from social to sequential interactions, to improve user-item recommendations [14]. GNNs are also applied in personalized recommendation contexts, as seen in studies like the study by Yang et al., which highlight the flexibility of GNNs across domains, achieving high

precision by effectively modeling complex data patterns [15]. While these works illustrate the adaptability of GNNs for recommendation systems, my approach uniquely adapts GNNs to software development environments. By integrating diverse data sources specific to software development, such as code, commit, and issue tracking, my method addresses the distinct needs of development tasks - an area that stays relatively underexplored within GNN-based recommendation frameworks.

In conclusion, my work addresses the gap in multi-source integration and relational depth by applying GNNs to diverse, software-specific data, creating a more adaptable recommendation system suited for complex, large-scale projects.

4 Theoretical background

In this work, the primary aim is to develop a general framework that improves the efficiency of current software development practices by recommending relevant components of the product. To achieve this, a heterogeneous knowledge graph was first constructed, that integrates several diverse modalities of a long-lived and complex software system.

An essential part of this framework involves creating a low-dimensional vector space representation of the knowledge graph. This is a useful concept, because in this space seemingly independent objects can be positioned based on real-world relationships between them. The shared vector space introduces a valuable concept of distance, which is especially useful in information retrieval. Unlike traditional databases, where queries provide only binary results - whether an object exists in the database or not - a vector-based approach enables more nuanced queries by returning objects that are similar to the target based on closeness in the vector space. This captures semantic relatedness, which aligns well with human intuition: often, we have blurred ideas of what we are searching for, and receiving suggestions based on similarity can help us find relevant results, even if we cannot think of the exact term or concept. This approach allows us to map the complexities of real-world relationships into a format that supports both precise retrieval and more abstract associations.

Initially, I examined text embeddings using SBERT (Sentence-BERT), which transforms text into vectors capturing semantic meaning, allowing retrieval based on similarity rather than exact word matches. Mentioning other common, traditional method BM25, which cannot handle synonyms or related concepts, SBERT's embeddings provide flexible representations suited for unstructured language data.

Subsequently, I introduce the Graph Auto-Encoder (GAE) proposed by Kipf et al. architecture which is a neural network driven approach to graph embedding [16]. For the recommendation experiments, I train the GAE model and compare its results to those achieved with a solution based on the SBERT text encoder, originally proposed by Reimers and Gurevych [17]. This comparison highlights the advantages of using GAE for capturing complex relationships within the knowledge graph, thereby providing a more effective tool for supporting software development activities.

4.1 Common approach

A text encoder is a model that converts text into numerical representations, or embeddings, which can then be processed by machine learning models. Text encoding is particularly useful for tasks such as semantic search, recommendation, and text classification. One prominent model used for text encoding is Sentence-BERT or SBERT. SBERT is specifically optimized to produce embeddings for entire sentences, rather than individual words. SBERT embeds sentences into a vector space where similar sentences lie close together.

Text encoders work with discrete units of text known as tokens. Tokenization is the process of breaking down text into individual tokens, which can be words or subword units. For example, the sentence "Developers often need support" could be tokenized into individual words or subwords, such as ["Develop", „ers," "often," "need," "support"]. For large documents, encoders may apply chunking as well, which divides long text into manageable segments, ensuring that each chunk fits within the model's input length limits. This is especially important for models like SBERT, because the input size is fixed. By tokenizing and chunking, text is transformed into a structured format that an encoder can process, even for lengthy documents.

SBERT and similar encoders generate text embeddings by assigning each sentence or chunk a unique position in a high-dimensional vector space. This allows the model to capture relationships between sentences based on semantic similarity, rather than relying solely on shared keywords. For instance, SBERT might place "software development" and "coding practices" near each other in vector space due to their related meanings, even if they do not share identical words.

This vector-based approach is useful for information retrieval and recommendation tasks, where the goal is to find relevant content based on similarity in meaning. Unlike traditional statistical methods like BM25 which was proposed by Robertson et al., which rely on exact word matches and cannot account for synonyms or related terms, embeddings are more flexible and can capture nuanced meanings [18]. This capability makes embeddings highly effective for applications involving unstructured, diverse language data, where semantic similarity is often more valuable than direct word overlap.

4.2 Knowledge graph

The concept of knowledge graphs was first introduced by Google in 2012 with the announcement of the "Google Knowledge Graph" [19]. A knowledge graph is a structured, directed graph where entities are represented as nodes and the relationships between them as directed edges. Since 2012, KGs have gained wide application across domains, thanks to their capability to capture semantic richness and interconnected information.

As noted in the Related Works section, KGs are extensively used in fields such as biomedicine, where they model relationships between diseases, drugs, and adverse events to uncover new medical insights. A subgraph of such KG is visualized in Figure 4, highlighting its path-based reasoning applications. Social networks like Facebook rely on KGs to map connections between users, their interests, and activities, supporting personalized content recommendations. Search engines also use KGs to link concepts and entities, enhancing search accuracy and relevance.

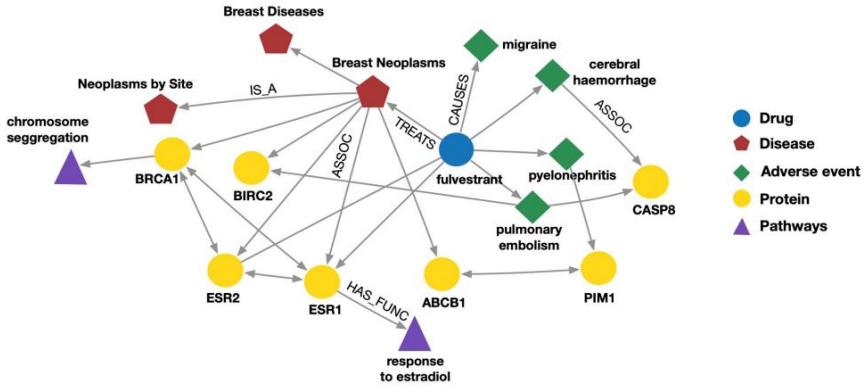

Figure 4: A subgraph of a medical knowledge graph, showing relationships between drugs, diseases, adverse events, proteins, and pathways. Fulvestrant, for example, treats breast neoplasms and links to adverse events like migraine, aiding in visualizing and analyzing complex medical interdependencies [20].

The key components of knowledge graphs are RDF (Resource Description Framework) triplets, consisting of a subject, predicate, and object. For instance, as shown in Figure 4, a medical knowledge graph includes triplets like *(Fulvestrant, treats, Breast Neoplasms)*. This triplet specifies that the drug Fulvestrant is used to treat Breast Neoplasms, highlighting a relationship within the medical domain. By aggregating many such triplets, a knowledge graph provides a comprehensive view of relationships within

a system. To uncover indirect relationships, path-based reasoning can be used. For instance, through paths like *(Fulvestrant, treats, Breast Neoplasms)* and *(Breast Neoplasms, associated with, BRCA1)*, the graph can help infer potential genetic links or targets for treatment by tracing connections across drugs, diseases, and genes. This reasoning approach enables discovering insights that aren't immediately obvious, revealing complex dependencies across medical entities. Hence, by aggregating many such triplets, a knowledge graph provides a comprehensive view of the system's structure and behavior.

Knowledge graphs possess several key properties. Firstly, they are naturally incomplete, meaning they may not contain every relationship or entity. However, this does not mean the missing information is irrelevant; it may simply not have been captured yet. This incompleteness highlights the importance of knowledge graph completion tasks, which aim to fill in missing connections. For example, in social networks like Facebook, incomplete knowledge graphs may result in missed connections between people who share mutual friends or live in the same neighborhood, thus impacting recommendation accuracy. Secondly, knowledge graphs are dynamic and continuously expand as new information is added, which is essential in fields like software development where systems are constantly evolving. Finally, one of the main strengths of knowledge graphs is their dual nature of being machine-interpretable and human-readable.

However, the size and complexity of knowledge graphs can make it challenging for humans to discover new insights and reason through the data. Machine interpretability is therefore essential for processing the information in knowledge graphs efficiently, as traditional methods would struggle with such complex queries. In the next section, we explore how knowledge graphs can be transformed into machine-readable vector spaces, enabling advanced analytical techniques.

4.3 TransE

Embedding is a process in machine learning that transforms discrete objects like words, nodes, or entire graphs into continuous, low-dimensional vector spaces, transforming complex data into a machine-interpretable form. Resulting in entities being represented as numerical vectors. Building on the earlier discussion, the use of a low-dimensional vector space allows us to perform tasks like link prediction and semantic reasoning with increased computational efficiency, using the preserved relationships

from the original graph. This vector representation enables algorithms to perform various downstream tasks like link prediction, reasoning, or path queries. Figure 5 illustrates the basic concept of embedding a graph into this vector space.

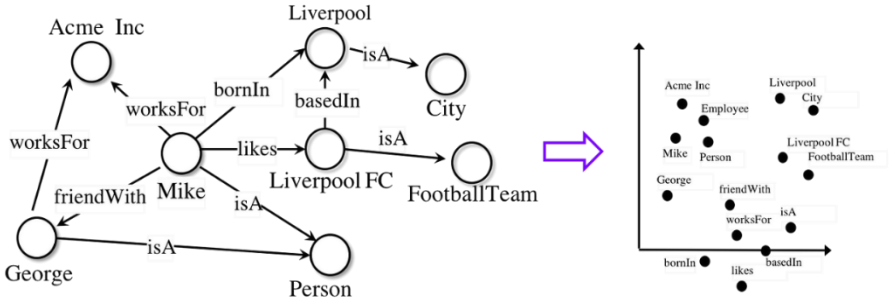

Figure 5: Illustration of embedding a graph. Entities from the original graph are mapped to a low-dimensional vector space, enabling efficient machine learning analysis while preserving structural information.

Knowledge graph embedding (KGE) is a unique type of embedding because it involves embedding both the nodes (entities) and the edges (relationships) of a graph. This approach is critical because the value of a knowledge graph lies not just in its individual entities but in the complex network of connections between them. By embedding both nodes and edges, KGE captures the full richness of the graph's structure, allowing for more precise and accurate modeling of relationships and interactions. Unlike embeddings that focus solely on individual entities' attributes, KGE preserves the relational context in which entities exist, making it particularly powerful for tasks where understanding relationships is vital. Various approaches exist for efficiently embedding knowledge graphs, each suited to different downstream tasks. A survey by Wang Q. et al. provides a detailed overview of these methods [21].

Translation-based KGE models, such as TransE proposed by Bordes et al. [22], model relationships as translations in the embedding vector space. The fundamental idea is that for a valid triplet *(h, l, t)*, the embedding of the head entity plus the embedding of the relation approximates the embedding of the tail entity: $h + l \approx t$. This translation mechanism allows TransE to effectively capture hierarchical relationships within the knowledge graph.

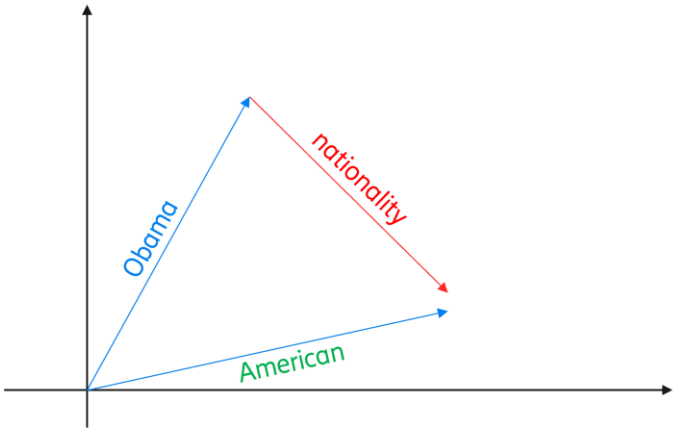

Figure 6: Illustration of embedding a knowledge graph using the TransE approach. Nodes and relationships are mapped into a vector space where relationships approximate geometric translations in the vector space.

TransE is particularly effective for simple one-to-one relationships but may struggle with more complex relationships like one-to-many or many-to-many due to its simplicity. In cases where multiple tails correspond to a single head and relation, TransE cannot distinguish between them, as it would map them to the same point in embedding space because of the way the model is trained. This limitation implies that distinct entities could become indistinguishable in the embedding space, which is not desirable. To address these limitations, extensions like TransH and TransR were proposed. TransH, introduced by Wang Z. et al., allows entities to have different representations when involved in different relationships, accommodating more complex relational patterns [23]. TransR, proposed by Lin Y. et al., introduces separate spaces for entities and relationships, enabling it to model multi-relational data more effectively by projecting entities and relations into distinct vector spaces [24].

Training the TransE model involves modifying the embedding vectors from iteration to iteration so that for valid triplets (h, l, t), the vector $h + l$ is close to t, while for invalid or corrupted triplets (h', l, t'), the distance $h' + l$ to t' is maximized. To create a corrupted triplet, either the head or the tail is replaced by a random entity, but not both at the same time as see in *(1)*:

$$S'_{(h,l,t)} = \{(h', l, t) | \, h' \in E\} \cup \{(h, l, t') | \, t' \in E\} \qquad (1)$$

While the process of adjusting the embedding vectors is typically achieved through a margin-based ranking loss function, defined in *(2)*:

$$L = \sum_{(h,l,t)\in S} \sum_{(h',l,t')\in S'_{(h,l,t)}} [\gamma + d(h+l,t) - d(h'+l,t')]_+ \qquad (2)$$

Where γ is the margin that separates positive and negative triplets, $d(h+l,t)$ is the distance between the positive triplet embeddings and $d(h'+l,t')$ is the distance for the corrupted triplet. The equation above is visualized in Figure 7.

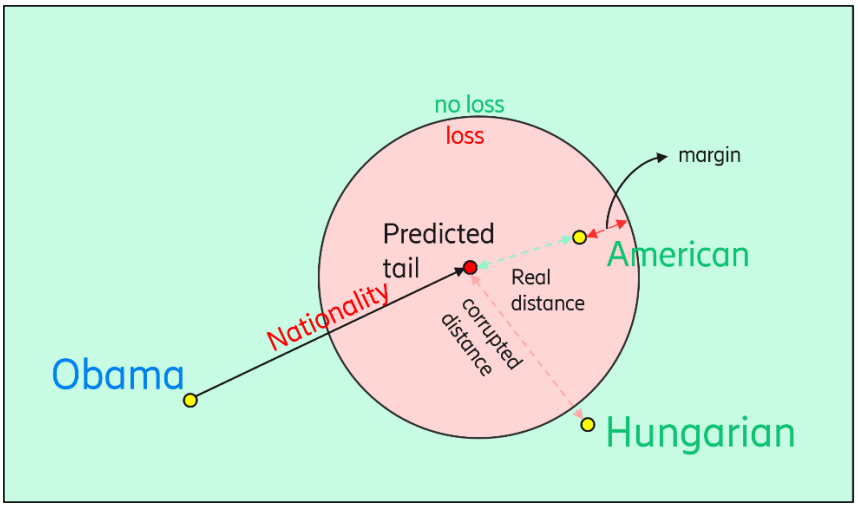

Figure 7: Visualization of margin in TransE embeddings. The margin defines a threshold around the predicted tail (American) for the relationship 'Nationality' between nodes. If a corrupted entity (Hungarian) falls within the distance of 'real distance + margin,' it results in a loss, indicating a closer-than-desired similarity to the true tail. If outside this margin, there is no loss, suggesting adequate separation between correct and incorrect entities.

The training process involves minimizing this loss function using optimization techniques like Stochastic Gradient Descent (SGD), which is the optimizer used in the original paper. During each iteration, real triplets - those that exist in the knowledge graph - are selected, and corresponding negative triplets are generated by the process defined above. The model then modifies the embeddings to ensure that positive triplets are closer together – in case on a non-zero margin closer by the margin - in the vector space than their negative equivalent.

4.4 Graph Auto-Encoder

Autoencoder is a type of neural network designed to compress input data into a lower dimension form and then reconstruct it back to its original state. The concept was published by Bank et al. in the publication [25]. The main goal is to learn a compact but meaningful representation of the data. The lower dimension vector space to which the

original data transformed is usually called latent space. Autoencoders are useful for tasks such as data compression, noise reduction, and feature extraction because they thrive in finding patterns in data, allowing for better understanding and processing. They are particularly good at reducing the size of data without losing vital information, making them useful for various applications, including anomaly detection and recommendations also in cases when the size of the input data is huge. This concept was extended to graph-structured data and called Graph Auto-Encoders. The aim of GAE is to learn low-dimensional embeddings of nodes while preserving the graph's structural information. The motivation for using GAE arises from the need to capture both complex relationships and node-specific information within the graph, which traditional methods like TransE proposed by Bordes et al. might not fully address [22]. While TransE models relationships as linear translations in the embedding space, it focuses solely on the connections between nodes without considering their individual attributes or features. GAE, on the other hand, incorporates node features into its neural network architecture, allowing it to learn from both the structure and the content of each node. This capability enables GAE to model more complex patterns and capture richer relational and attribute-based information, building on the foundational Graph Neural Network framework proposed by Scarselli et al. [26].

The architecture of a GAE has two main components shown in Figure 8: an encoder and a decoder. Both the encoders and decoders could be arbitrary models. In the original paper, the encoder consists of Graph Convolutional Networks (GCN) layers proposed by Kipf et al. [27]. While the decoder is a simple dot product calculation. The output of a GAE is an adjacency matrix holding probability values that indicate the likelihood of a link between each pair of nodes.

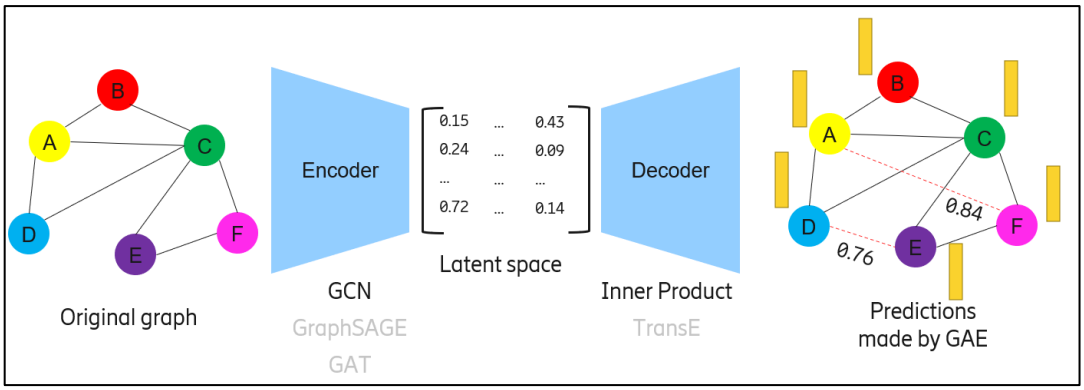

Figure 8: The architecture of GAE consists of an encoder and a decoder part. The encoder processes the input graph and converts the nodes into a low-dimensional vector representation where these embeddings inherit the structure of the graph. The decoder part reconstructs the graph based on the latent space representation.

The role of Graph Convolutional Network layers is to map each node in a graph to a low-dimensional embedding through multiple layers of non-linear transformations based on the graph's structure. A key strength of GCNs and other GNN models is their ability to accept graphs of arbitrary size and structure as input, unlike other deep neural networks that require input data in a fixed format. This flexibility makes GCNs powerful tools for processing graph data. However, this also means that GCNs operate differently from traditional neural networks. For example, processing a graph with a GCN cannot use the sliding window technique seen in Convolutional Neural Networks (CNNs) for images proposed by O'Shea and Nash, because graphs do not have a fixed direction or order like images do [28].

Instead, GCN layers aggregate information from each node's neighbors, allowing the embeddings to capture local structural features. In this sense, the neighborhood of each node defines the architecture of the GCN. In a GCN, the number of layers determines the number of hops from which information is aggregated. For each node, the neighboring nodes define a computational graph, which is a subgraph of the original graph. This computational graph includes as many layers of neighboring nodes, or 'hops,' as there are GNN layers, shaping the neural network structure illustrated in Figure 9. Further exploring this concept, a Convolutional Neural Network could be considered as a special case of a Graph Neural Network where the neighboring nodes are the pixels in an image, which have a fixed size and order. GNNs broaden this scope by allowing arbitrary graphs with varying degrees for each node. Additionally, CNNs are not permutation invariant,

meaning that changing the order of the input pixels affects the output, whereas GNNs are designed to handle such permutations without impacting the results.

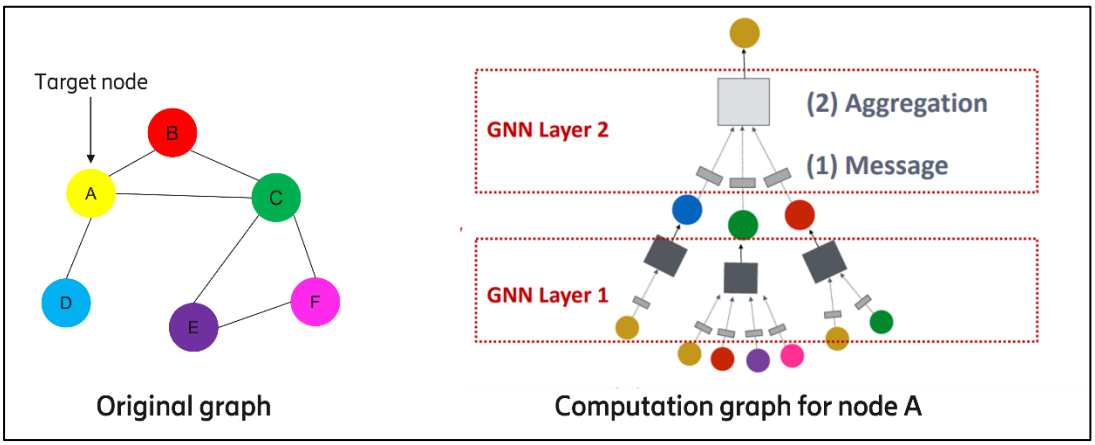

Figure 9: Illustration of node embedding computation in a Graph Neural Network. The original graph (left) shows nodes and their connections. The computation graph (right) demonstrates how node A learns its embedding by aggregating information from neighboring nodes [29].

The training of GNN models, such as GAEs, involves a concept known as message passing. This process consists of two main steps. First, each target node collects messages from its neighboring nodes. These messages are transformed through learned parameters and then aggregated to create a single message. Initially the message of a node is its' node feature. In Figure 9 in the computational graph the small grey rectangles (1) represent the message transformation, while the big gray squares (2) represent the message aggregation process. Second, the target node passes this aggregated message to its neighbors. This iterative process allows information to flow through the graph, enabling nodes to capture local structural features based on their surroundings. Therefore, in general we can write these steps with the following equations *(3)* and *(4)*:

$$m_u^{(l)} = MSG^{(l)}\left(h_u^{(l-1)}\right), u \in \{N(v) \cup v\} \qquad (3)$$

$$h_v^{(l)} = AGG^{(l)}\left(\{m_u^{(l)}, u \in \{N(v)\}, m_v^{(l)}\right) \qquad (4)$$

In these equations, GNN message passing is formulated for a node v at layer l. *(3)* represents the message function, where $m_u^{(l)}$ is the message sent from a neighboring node $u \in \{N(v) \cup v\}$ to v, and is computed based on the hidden state $h_u^{(l-1)}$ from the previous layer. The aggregation function in *(4)* combines the messages $m_u^{(l)}$ from all neighbors

$N(v)$ and the node itself to update the hidden state $h_v^{(l)}$ for node v at the current layer. During training, our goal is to find the optimal parameters for the transformations and aggregations applied during message passing. The parameters I adjust include the weight matrices used for transforming a node's own hidden vector and those used for aggregating messages from neighboring nodes in each layer. While each node has its own computational graph, these parameters are shared across all nodes but can differ between layers. *(5)*, *(6)* and *(7)* show the training with an L layer GNN network, where the message aggregation is a simple mean calculation:

$$h_v^{(0)} = x_v \qquad (5)$$

$$h_v^{(l+1)} = \sigma\left(\boldsymbol{W_l} \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + \boldsymbol{B_l} h_v^{(l)}\right), \forall l \in \{0, \ldots, L-1\} \qquad (6)$$

$$z_v = h_v^{(l)} \qquad (7)$$

This set of equations describes the layer-wise update rule for a GCN. Initially, each node v is assigned a feature vector written in *(5)*, where x_v represents the initial features of node v. For each layer l, the GCN updates the hidden representation of node v by aggregating information from its neighboring nodes and combining it with a self-transformation, resulting in the new representation show in *(6)*. In this formulation, W_l is a learnable weight matrix responsible for scaling the aggregated information from neighboring nodes, while B_l is another learnable weight matrix used to transform the hidden representation of node v itself. An activation function σ is applied elementwise to the result to introduce non-linearity. After L layers of aggregation and transformation, the final embedding for node v, denoted z_v, is given as in *(7)*. This iterative process allows each node to progressively integrate information from its neighborhood, with the parameters W_l and B_l learned to optimize the GCN's performance on the target task. This architecture is also referred to as a deep encoder because the output of one encoder layer serves as the input for the next layer. This stacking of layers allows the model to capture information from progressively larger neighborhoods around each node. To train this GCN model for link prediction, we minimize a loss function that distinguishes between positive examples (existing edges) and negative examples (non-edges). Using a binary cross-entropy loss, the model is optimized to correctly predict edges versus non-edges by

adjusting the learnable parameters W_l and B_l at each layer l. An optimizer, such as Adam or stochastic gradient descent (SGD), iteratively updates these parameters based on gradients from the loss, enhancing the model's ability to capture graph structure effectively.

As it can be seen, in the context of GCNs, the neighborhood of each node defines the architecture of the network. The number of layers in the GCN determines the number of hops from which a target node aggregates messages. Therefore, this approach remains scalable, with memory and computational requirements staying manageable even as the graph size grows. Rewriting the above equations with matrix operations helps us to understand this, see Appendix for computational details. This matrix-based approach allows GAE models to efficiently aggregate information over large graphs by utilizing the sparsity of adjacency and degree matrices. As parameters are shared across all nodes and edges, memory requirements stay manageable, even as graph size grows. This scalability makes GAEs well-suited for large, complex systems, where entities and relationships are numerous. Additionally, by using the same aggregation parameters for all nodes, the model's parameters grow sublinearly with the graph size, allowing it to handle new nodes in evolving graphs without retraining.

One crucial aspect of GNNs is the neighborhood aggregation function. It must be permutation invariant to ensure consistent results regardless of the order in which neighbors are processed, but apart from this there are no other constraints. Different GNN architectures used as encoders in GAE mainly differ in how they perform this aggregation. For example, in GCNs, the aggregation function often computes the average of the input messages. Other prominent encoders include GraphSAGE proposed by Hamilton W. et al. [30] and Graph Attention Networks (GAT) proposed by Veličković et al. [31], which primarily differ in how they perform message aggregation. While GraphSAGE improves upon GCNs by incorporating the node’s own features during aggregation, using techniques like mean, LSTM, or MLP to aggregate neighbor information before combining it with the node’s features, resulting in richer node embeddings, GAT introduce a self-attention mechanism that dynamically learns the importance of each neighbor during aggregation, allowing the model to focus on the most relevant neighbors and produce more nuanced embeddings.

The decoder can be a simple function, such as a dot product, or a more complex neural network. The goal is to ensure that similar nodes (those connected in the graph)

have embeddings that produce a high value when passed through the decoder, indicating a strong connection, while dissimilar nodes produce a low value. The training process minimizes a loss function, often a cross-entropy loss, which measures the difference between the original adjacency matrix and the reconstructed one. This encourages the model to learn embeddings that accurately capture the graph's structure.

Training GAEs in a supervised way can be challenging since it needs predefined labels for each node to represent their final embeddings. But autoencoders, including GAEs, thrive in unsupervised training, where the graph structure itself acts as the supervision needed for learning, also called self-supervised learning. In this setup, the encoder creates embeddings for the nodes, and a decoder tries to reconstruct the adjacency matrix from these embeddings, capturing relationships between nodes without needing explicit labels. This makes GAEs effective for a variety of graph-related tasks.

A non-trivial part of self-supervised training in GAEs is splitting the graph into training, validation, and test sets without breaking the structure. Unlike datasets like images, where each sample stands alone, graphs have interconnected nodes, so acting the same way is a wrong approach. There are two main ways to do this: the transductive and inductive settings. In the transductive setting, we keep the whole graph structure untouched and split the edges into training, validation, and test sets. During training, some edges are used for message passing, while others are held back to see if the model can predict them later. For validation and testing, these held-back edges are added to check how well the model has learned to predict missing links. In the inductive setting, we take a different approach by splitting the graph into separate subgraphs depicted in Figure 10. This allows the model to be tested on new, unseen nodes or subgraphs, showing whether it can generalize beyond the specific data it was trained on. During training, the GNN only learns from one of these subgraphs, and then it is validated and tested on entirely different parts of the graph that it has not encountered before. This setup helps assess whether the model can apply what it is learned to new sections and still predict links accurately.

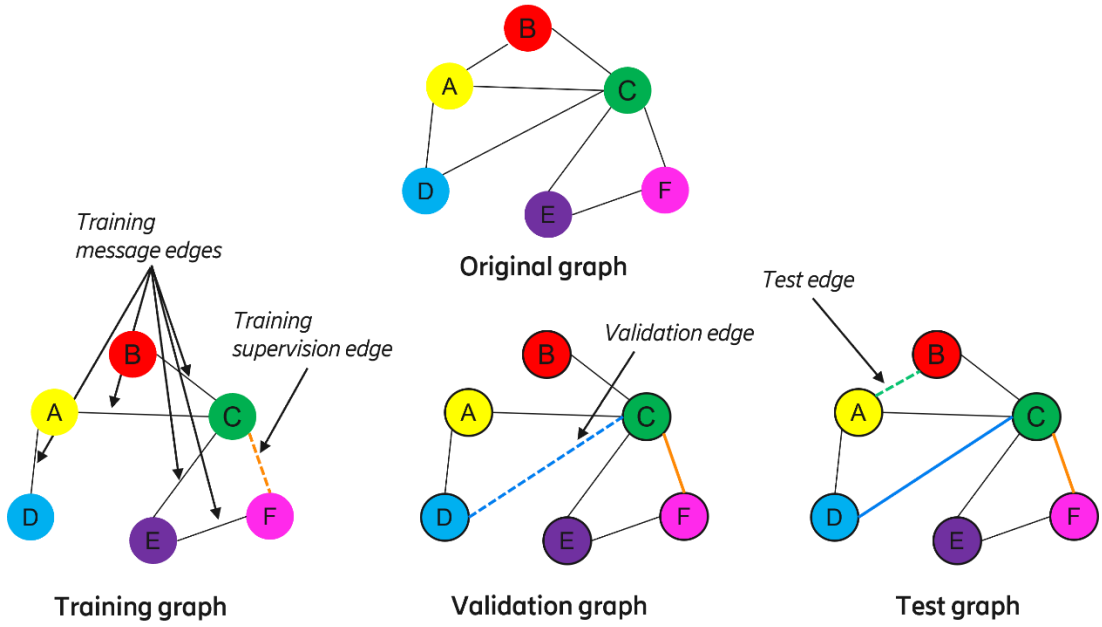

Figure 10: Inductive setting for GAE with graph splits. The original graph is divided into training, validation, and test subgraphs. In the training graph, message edges (black) propagate information, while supervision edges (orange) are held out for link prediction. The validation graph introduces additional edges (blue) for tuning, and the test graph includes unseen edges (green) to assess model generalization. This setup ensures the model learns without information leakage across splits.

Doing link prediction in this setting, two distinct graph splits are necessary: one for message passing during training, where nodes learn embeddings from their neighbors within the training subgraph, and another for validation and testing, where the model must generate embeddings for nodes in unseen subgraphs without any retraining. This separation is essential to avoid any information leakage between the sets, ensuring a fair evaluation of the model's predictive abilities. By constructing separate training, validation, and testing subgraphs, the model is prevented from memorizing patterns from the training data, so it must rely on the learned message passing functions to make predictions in unfamiliar parts of the graph. A possible splitting on edges on an example graph is depicted in Figure 10.

For evaluating link prediction performance, typical binary classification metrics like Accuracy, Precision, and Recall can be used. However, since I am often predicting probabilities rather than exact classifications, ROC AUC is particularly useful. Additionally, rank-based metrics like Mean Rank, Mean Reciprocal Rank, and Hits@K help evaluate how well the model ranks true links among possible links, framing the task as a recommendation problem. Explanations about these metrics can be found in Appendix in Table 8.

A key advantage of using Graph Auto-Encoders in the inductive setting is that, when a new node is added to the graph, there is no need to retrain the entire model. Instead, a simple forward pass through the model generates the embedding for the new node by aggregating information from its neighbors using the existing learned parameters. This makes the model efficient and adaptable, especially in dynamic environments like evolving software systems, where nodes and connections are frequently updated.

4.4.1 Variational Graph Auto-Encoder

The Variational Graph Auto-Encoder (VGAE) proposed by Kipf and Welling builds upon the Graph Auto-Encoder by introducing probabilistic modeling through latent variables, making it effective for tasks involving noisy or incomplete graph data [16]. While GAE learns deterministic node embeddings from graph structure and features, VGAE models embeddings as distributions, capturing uncertainty for more robust and generalizable representations. Inspired by Kingma and Welling's Variational Auto-Encoder (VAE), VGAE incorporates variational inference to approximate complex posterior distributions [32].

In VGAE, the encoder employs a two-layer Graph Convolutional Network to parameterize the posterior distribution $q(Z \mid X, A)$, where Z represents the latent node embeddings, X the node features, and A the adjacency matrix. Unlike GAE, the encoder outputs both the mean (μ) and variance (σ^2) for each node embedding, from which stochastic latent variables are sampled. The decoder, shared with GAE, is an inner product operation that reconstructs the adjacency matrix, where the probability of an edge between two nodes is calculated as $p\left(A_{ij} = 1 \mid z_i,\ z_j\right) = \sigma(z_i^T z_j)$, with σ being the logistic sigmoid function.

A key difference between GAE and VGAE lies in their optimization objectives. GAE minimizes reconstruction loss between input and reconstructed graphs, while VGAE maximizes a variational lower bound on data likelihood, combining reconstruction with a regularization term - the Kullback-Leibler (KL) divergence. This regularization ensures learned latent distributions remain close to a Gaussian prior, promoting smooth and interpretable embeddings. The lower bound is given as in (8):

$$\mathcal{L} = \mathbb{E}_{q(Z \mid X,A)}[\log p(A|Z)] - KL[\, q(Z \mid X, A) \,\|\, p(Z)\,] \qquad (8)$$

where the first term encourages accurate reconstruction of the graph and the second term regularizes the latent space.

The probabilistic framework of VGAE makes it particularly effective for graphs with sparse or noisy connections. By modeling embeddings as distributions rather than fixed points, the model can represent uncertainty and generalize better in unseen scenarios. This property is especially beneficial in tasks like link prediction, where missing or noisy edges are common. Moreover, VGAE performs well even on featureless graphs, where the input features X are replaced by an identity matrix, relying solely on the graph structure for learning embeddings.

The VGAE has been shown to achieve superior results compared to GAE in tasks like link prediction on citation networks, as demonstrated in the original work. By taking advantage of the strengths of variational inference, VGAE effectively balances reconstruction accuracy with robust regularization, making it a powerful tool for graph representation learning.

4.5 Graph Language Model

The Graph Language Model (GLM), proposed in August 2024 by Plenz and Frank, introduces a transformative approach to integrating language models (LMs) with graph-structured data. Traditional methods often separate semantic encoding (handled by LMs) and structural reasoning (handled by Graph Neural Networks, or GNNs) [33]. This division can limit model performance when both modalities - textual and structural - are important for a task, as it requires each component to operate independently under distinct principles. GLM addresses this challenge by unifying textual and structural processing within a single framework. Taking advantage of the strengths of pre-trained LMs like T5 introduced by Raffel et al., GLM incorporates graph-specific reasoning through early fusion mechanisms, which allow it to process both textual and graph inputs without requiring additional GNN layers [34]. This integration is particularly beneficial for knowledge graphs, where facts are represented as triplets (e.g. (Entity1, Relationship, Entity2)), as it enables GLM to reason jointly over graph and textual data. Figure 11 illustrates the GLM architecture.

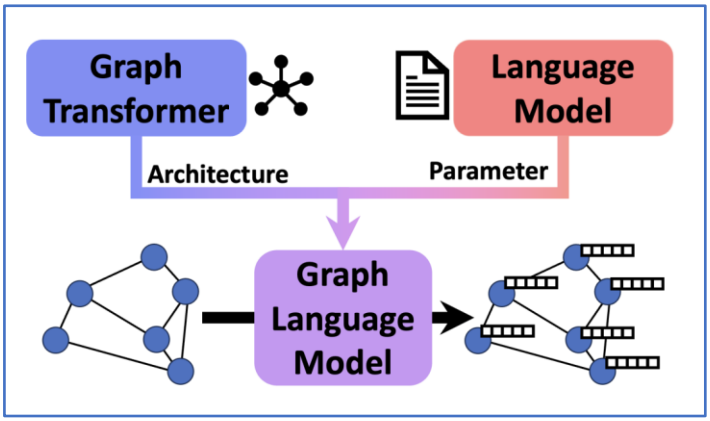

Figure 11: The Graph Language Model (GLM) integrates Graph Transformer architecture with Language Model parameters, enabling unified reasoning over graph structures and text for enhanced knowledge graph tasks [33].

To prepare input graphs, GLM employs a Levi graph transformation, converting edges into nodes to explicitly represent relationships, followed by an extension where each graph element is tokenized into sequences that align with LM tokenization principles. This preprocessing step ensures the graph structure can be processed as textual data, enhancing the model's ability to integrate both modalities. GLM also modifies the Positional Encoding (PE) typically used in transformers to suit graph structures, encoding relative distances between nodes rather than absolute positions. Two variants of GLM exist:

- **Local GLM (ℓGLM)**: restricts attention to tokens within the same triplet
- **Global GLM (gGLM)**: permits attention across the entire graph while introducing biases that prioritize closer nodes.

This adaptability allows GLM to handle tasks with varying degrees of graph complexity. The model's self-attention mechanism extends the standard transformer formula as seen in (9):

$$Attention(Q, K, V) = Softmax\left(\frac{QK^T}{\sqrt{d}} + B_p + M\right) * V \qquad (9)$$

where Q, K, V represent the query, key, and value matrices, $B_p$introduces graph-specific biases, and M applies attention masks to encode graph connections.

A key strength of GLM lies in its ability to jointly encode graph and textual inputs through modality-specific attention mechanisms. The model processes interactions

among graph-to-graph (G2G), text-to-text (T2T), graph-to-text (G2T), and text-to-graph (T2G) tokens, ensuring both modalities contribute meaningfully to reasoning tasks like relation classification or knowledge graph population. The preprocessing pipeline visualized in Figure 12 highlights how textual and structural components are seamlessly fused within the model. By initializing from pre-trained language model parameters, GLM achieves significant efficiency and avoids the costly training procedures required by models built from scratch.

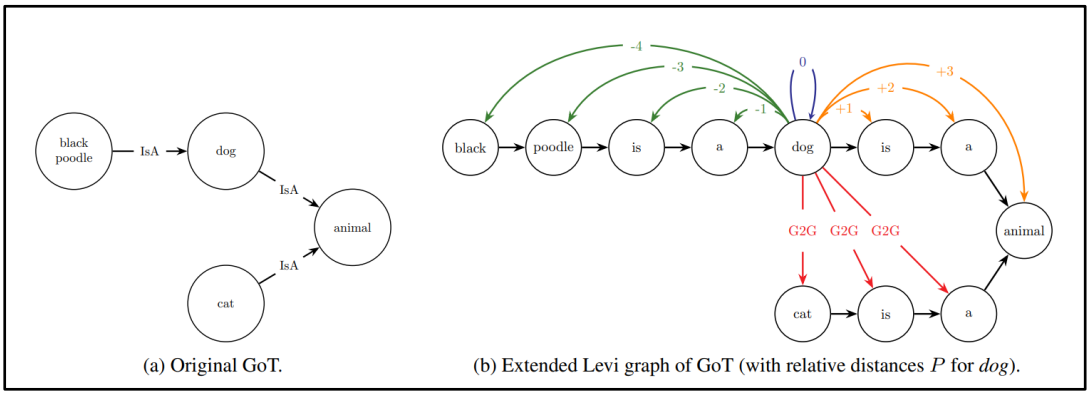

Figure 12: The figure illustrates the graph processing in the Graph Language Model (GLM). Panel (a) shows the original Graph of Triplets (GoT), where nodes represent entities, and edges indicate relationships. Panel (b) depicts the Extended Levi Graph, a transformation of the GoT into a token-based representation with relative distances (PPP) for the token "dog." This extended representation enables the GLM to use both graph structure and textual semantics by encoding relationships as nodes and assigning attention scores based on relative positions in the graph [33].

The empirical performance of GLM demonstrates its superiority over LM-only and GNN-based approaches, particularly in tasks requiring joint reasoning across modalities. In supervised and zero-shot settings, GLM consistently achieves higher accuracy, emphasizing its ability to taking into account both graph structure and semantic information effectively.

- **Local GLM** thrives in tasks with localized reasoning, where small-scale graph structures dominate.
- **Global GLM** is optimal for tasks requiring long-range dependency modeling, as it captures interactions across the graph.

In conclusion, GLM unifies language models and graph reasoning into a single framework, representing a paradigm shift in graph representation learning. This integration offers a powerful, efficient, and scalable solution for knowledge graph tasks.

5 Proposed methodology

5.1 Data sources

The data for constructing the knowledge graph was retrieved from a large-scale software repository at a telecommunications company. This repository is part of a complex software system involving tens of thousands of developers and multiple releases each year, emphasizing its continuous evolution. It contains several thousand files, hundreds of thousands of lines of code, thousands of functional test cases, and hundreds of configuration objects, underscoring the challenges in understanding and maintaining it. With hundreds of customer issues constantly awaiting resolution, efficient methods for issue resolution and developer assignment are essential, making issue-centric recommendations a key focus of this work.

The knowledge graph was constructed by integrating data from issue-tracking, version control, and documentation sources, using design principles that enable cross-type connections. For example, issues and commits link via referenced issue IDs, as commonly seen in ticketing systems like GitHub or GitLab. This approach connects issues, code changes, files, functions, and developer contributions into a unified view of the software ecosystem. Data not directly tied to specific files or functions was excluded from the recommendation graph to maintain relevance. By merging diverse data, the knowledge graph addresses information fragmentation and supports system-agnostic models for issue resolution and developer recommendation. The data types used for the recommendation system are detailed in Table 1.

Data type	Count (rounded)	Content	Typical length (chars)	Quality of Content
Issue	~ 6000	Title, description	500-1500	Low (free text, unstructured, varying length)
Person	~ 600	Name or ID	0-50	High (concise, consistent format)
File	~ 2000	Filename	0-50 (only filename)	Medium (consistent format, not always expressive enough)
Commit	~ 5200	Commit message	50-500	Low (free text, unstructured, varying length)

Table 1: Overview of data types used in the knowledge graph, including typical content length, count, and structuredness.

5.2 Graph construction

The primary goal of this work is to demonstrate how knowledge graphs, with their rich semantic representations, can improve recommendations and provide deeper insights into data . To showcase this, I constructed a knowledge graph encompassing various modalities: issues, commits, files, and developers, all interconnected through various relationships. As noted in the study by Abu-Salih, determining the structure of a knowledge graph is highly domain-specific and even use-case specific within a domain [5]. This work focuses on recommending competent developers and related files to issues. Constructing a semantically rich knowledge graph requires a deep understanding of the software product; thus, several experienced developers familiar with the system were involved in these steps.

In this solution, a commit-centric approach was employed, where commit nodes bridge files and issues. Commits not only link code changes to the issues they address but also capture the temporal evolution of the codebase. This temporal dimension is essential for representing ongoing modifications, which introduces versioning challenges within the knowledge graph. While commits intuitively connect files to specific issues, maintaining a comprehensive version history requires careful handling to avoid ambiguity when multiple versions of a file exist. Including commit nodes enables tracking historical connections between issues and file changes, providing insights into how specific code modifications relate to various issues and developer contributions over time.

Initially, parsers were written to load data into a graph database, but the unstructured nature of commit messages and issue descriptions - often containing extraneous information like log messages and URLs - posed significant challenges. To address this, an LLM was employed to clean these descriptions, filtering out irrelevant content and retaining only essential information. Constructing the knowledge graph involved carefully defining entities and relationships to capture relevant aspects of the software development process. Since the aim is to recommend developers and files related to specific issues, a subset of the graph focusing on the most relevant nodes and edges was utilized.

Relationships in the graph were represented as triplets in the form (head entity, relation, tail entity). The specific triplets used are listed in Table 2. The knowledge graph captures interactions between issues, commits, files, and developers, using a commit-

centric approach that links code changes to issues. This setup simplifies data extraction from Ericsson's ticketing and version control systems, as commits often reference issue keys directly. The construction of the knowledge graph is use-case specific, focusing on issue-related entities to improve developer and file recommendations.

Head	Relation	Tail
Issue	blocks	Issue
Issue	causes	Issue
Issue	duplicates	Issue
Issue	fixes	Issue
Issue	mentions	Issue
Issue	author	Person
Commit	fixes	Issue
Commit	changes	File
Commit	author	Person

Table 2: Table of the triplets in the knowledge graph.

5.3 Graph embedding

In recommendation systems, model effectiveness is primarily determined by the quality of data sources and the structure of their relationships. Two critical factors influence embedding quality: the richness and processing of node features, and the inherent structure of the graph. In the context of graph embedding models - such as TransE, Graph Auto-Encoder (GAE), Variational Graph Auto-Encoder (VGAE), and Graph Language Model (GLM) - these factors play central roles.

The graph structure dictates how information flows between nodes, influencing the extent to which relationships and dependencies are captured. For example, in models like GAE and VGAE, the graph structure defines the computational graph, shaping the architecture of the neural network and the message-passing mechanisms. In TransE, the focus is on learning embeddings based on relational translations, while GLM integrates both structural and semantic information through advanced language modeling techniques. Node features are equally fundamental across these models, serving as the primary input for embedding generation. The embeddings of each node are based on its own features and, in many cases, the features of its neighbors, underscoring the importance of high-quality, relevant attributes in achieving meaningful embeddings.

Constructing an effective recommendation system thus involves a multi-dimensional optimization, balancing these factors to maximize embedding quality. In this section, I first evaluate the performance of four different graph embedding models on

subgraphs extracted from production data. This evaluation uses standard metrics such as Mean Reciprocal Rank (MRR), Mean Rank (MR), Hits@5, and Hits@10 to compare how each model handles structural and feature information in real-world graphs.

Following this comparative analysis, I go deeper into the influence of graph structure and node features on embedding quality, using a custom-designed Quality of Embedding (QEmb) indicator to highlight the impact and significance of specific adjustments in the graph. Evaluating these factors is particularly challenging, as even slight changes can significantly affect embedding quality in unpredictable ways. To better observe these effects, initial experiments are conducted on a small synthetic dataset depicted in Figure 13, allowing for controlled adjustments. Finally, the approach is applied back to the production repository to assess real-world applicability and impact, focusing on the models that showed the most promise in the initial evaluations.

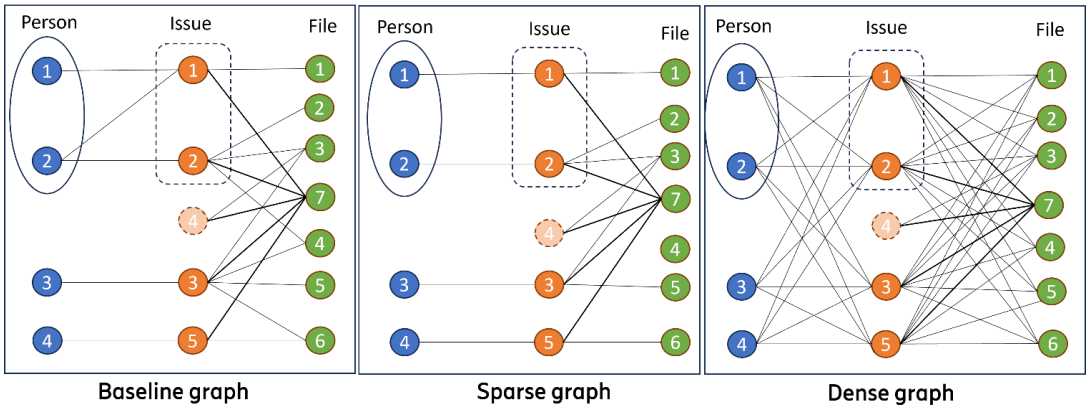

Figure 13: Visual representations of the three synthetic graph structures used for evaluating the impact on embedding quality. Baseline Graph *(left) is a balanced, moderately connected graph simulating a typical small-scale repository.* Sparse Graph *(center) has limited connectivity and isolated sections, testing how disjoint clusters affect information propagation.* Dense Graph *(right) simulates a highly interconnected structure, illustrating the effects of excessive connectivity on the model's ability to capture meaningful relationships.*

5.3.1 Overviewing graph embedding models

To understand the capabilities and learning behaviors of various graph embedding models, I conducted experiments using four distinct approaches: TransE, Graph Autoencoder (GAE), Variational Graph Autoencoder (VGAE), and Graph Language Model (GLM). Each of these models incorporates different aspects of the graph data during training, which are briefly described below.

- **TransE**: TransE is one of the earliest and simplest knowledge graph embedding models. It represents entities as points in a continuous vector space and relations as translation vectors. Training involves learning embeddings based solely on the graph's structure without considering node features.
- **Graph Auto-Encoder**: GAE is a neural network model that uses both the graph's structure and node features to learn low-dimensional representations. It encodes the input graph into a latent space and then reconstructs it, capturing the underlying patterns in the data.
- **Variational Graph Auto-Encoder:** VGAE extends GAE by introducing probabilistic latent variables, allowing for the modeling of uncertainty in the embeddings. It also utilizes both structure and node features but incorporates a variational inference framework during training.
- **Graph Language Model**: GLM is a sophisticated model that combines graph neural networks with language modeling techniques. It considers the graph's structure, node features, and can capture complex relationships by processing sequences of node interactions.

The primary goal of this experiment was to assess how different models learn from graph data and evaluate their performance in various scenarios. To achieve this, subgraphs were extracted directly from the production graph, rather than relying on synthetic graph generation. This approach ensured that node features and connections were representative of real-world data, avoiding potential noise or biases introduced by synthetic graphs. Subgraphs were retrieved by selecting nodes within three or four hops from a central node and including all connections between these nodes. This method was designed to capture local neighborhoods rich in structural information. A large number of such subgraphs were collected, and several graph metrics were computed for each, including clustering coefficient, density, average path length, and the number of edges.

From this collection, ten subgraphs covering a wide range of these metrics were sampled to cover diverse cases. Additionally, two graphs were deliberately altered to create specific conditions: one disconnected graph and one graph lacking relationships between nodes of the same type. This process resulted in a total of fourteen distinct graphs for the experiments. Before training, the edges of each graph were split into training and testing sets, ensuring that all models were evaluated on the same data. TransE, GAE,

VGAE, and GLM were trained on these graphs, and their performance was evaluated using standard metrics: Mean Rank (MR), Mean Reciprocal Rank (MRR), Hits@5, and Hits@10.

Heatmaps of the evaluation metrics were generated to visualize and analyze the results. Each heatmap was sorted according to a different graph metric, enabling observation of how model performance varied with changes in graph properties. One example heatmap is visualized in Figure 14.

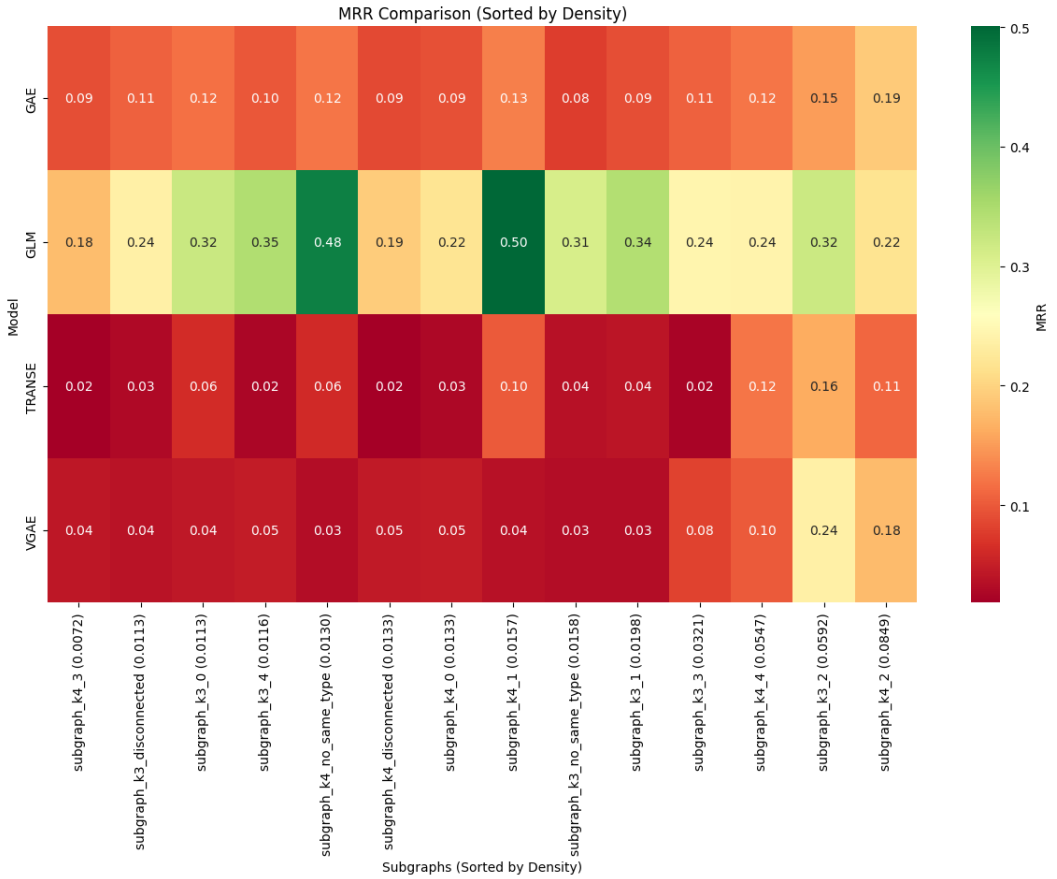

Figure 14 : MRR comparison for the four models (TransE, GAE, VGAE, and GLM) with subgraphs sorted by density. The density value of each subgraph is shown in parentheses beneath the subgraph label. This visualization highlights the variation in model performance as the density of the subgraphs increases.

Graph Language Model (GLM) demonstrated superior performance across most of the evaluation metrics as it can be seen in case of MRR as well in Figure 14. Its ability to capture complex relationships and process rich node features contributed to its effectiveness. However, this performance comes at a cost. GLM requires substantial memory resources, and both training and inference times are significantly longer compared to the other models. Specifically, inference times ranged between 5 to 25

seconds for the subgraphs, whereas the other models achieved inference times between the order of 0.01 and 0.001 seconds. This makes GLM less practical for real-time applications or scenarios with limited computational resources.

Graph Autoencoder (GAE) showed moderate performance but proved to be more cost-efficient and faster than GLM. Interestingly, GAE maintained consistent performance even as the graphs became denser, despite all graphs being inherently sparse depicted in Figure 15. This suggests that GAE effectively utilizes the available structural information without being hindered by increased connectivity.

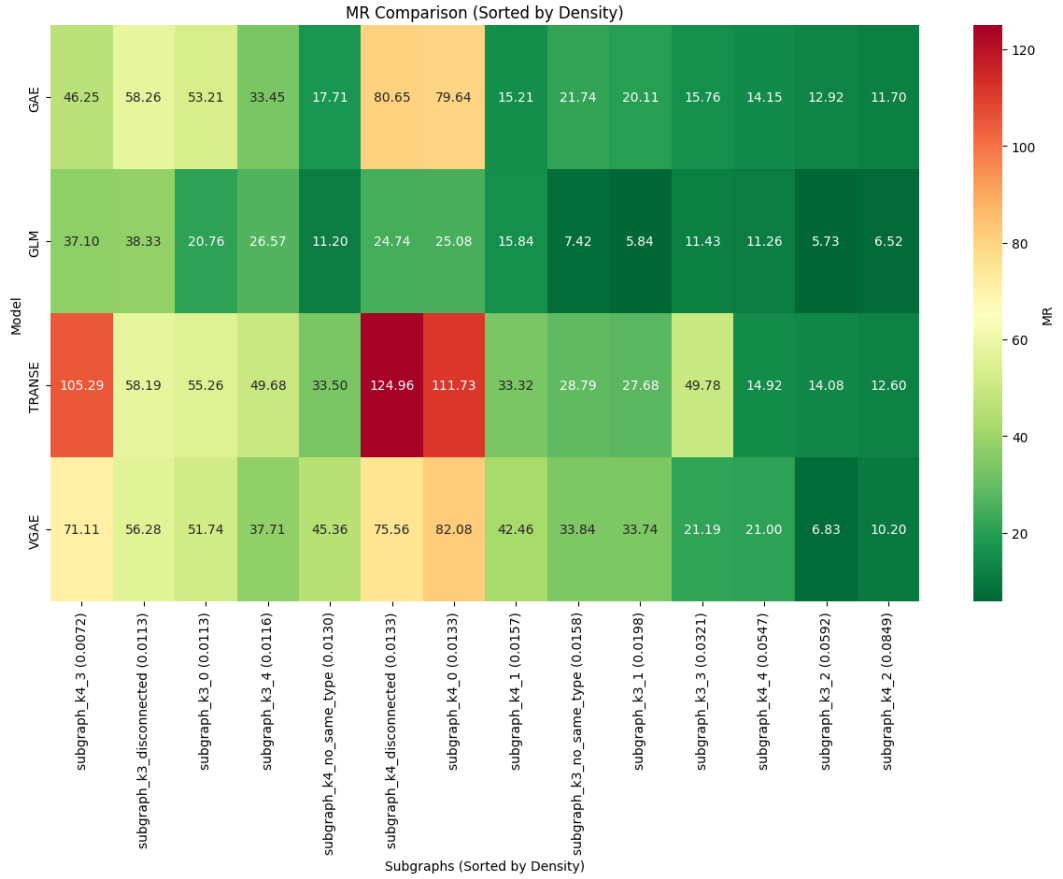

Figure 15: MR comparison for the four models (TransE, GAE, VGAE, and GLM) with subgraphs sorted by density. The density value of each subgraph is displayed in parentheses beneath the subgraph label. Notably, while GAE exhibits slightly higher Mean Rank values compared to GLM, its performance remains competitive, highlighting its efficiency and capability even in denser graphs.

Variational Graph Autoencoder (VGAE) did not yield significant improvements over GAE. In most cases, VGAE's performance was slightly worse. This could be attributed to the complexity introduced by the variational components, which might not provide additional benefits for the types of graphs used in this study. It is possible that

the uncertainty modeling in VGAE did not align well with the graph properties or that the model requires more careful tuning to outperform GAE.

TransE, being the simplest model and lacking a neural network architecture, struggled to capture the complexity of the graphs. It relies solely on structural information and represents relationships as simple translations in the embedding space. This limitation makes it inadequate for modeling 1-to-N, N-to-1, and N-to-N relationships effectively. TransE's performance was inconsistent; it occasionally performed well, possibly due to favorable graph structures, but often yielded poor results as it can also be seen in Figure 16. Its inability to process arbitrary graphs with complex relationships underscores the need for more sophisticated models when dealing with intricate graph data.

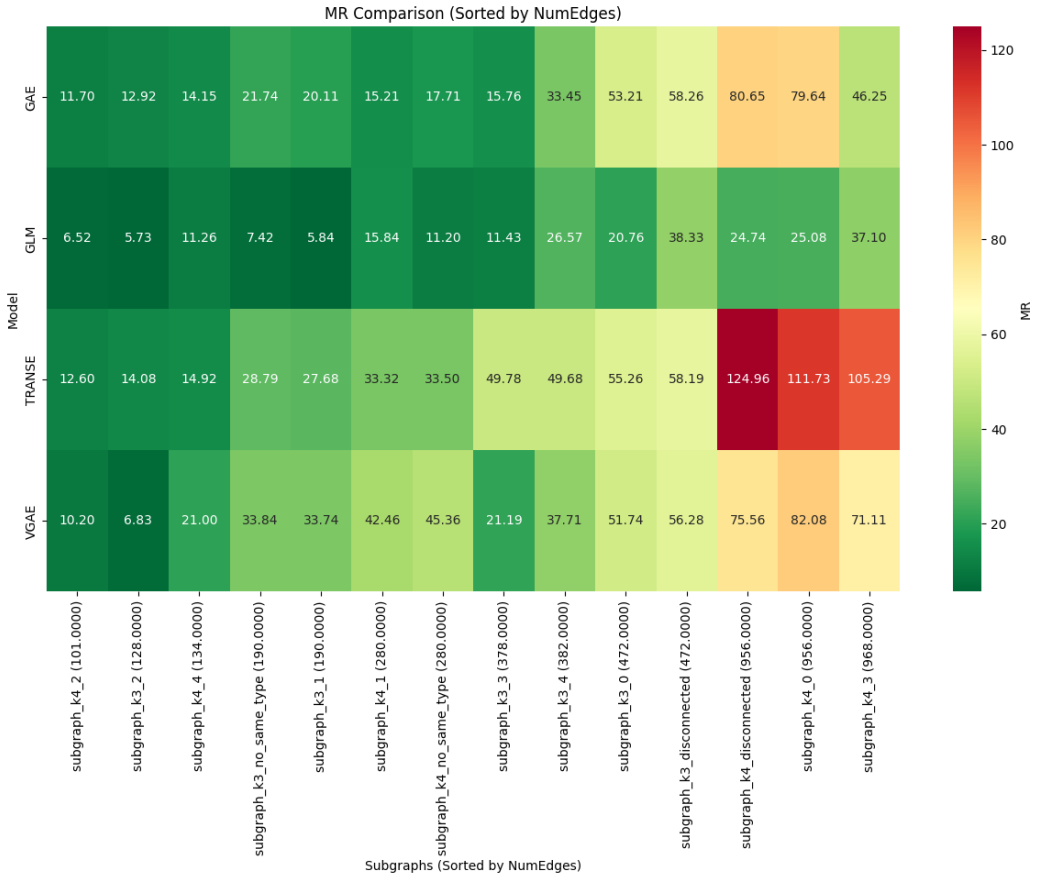

Figure 16: MR comparison for the four models (TransE, GAE, VGAE, and GLM) with subgraphs sorted by the number of edges. The number of edges for each subgraph is displayed in parentheses beneath the subgraph label. TransE struggles significantly as the number of edges increases, likely due to its limited capacity to model complex relationships, particularly when the proportion of 1-to-1 relationships is smaller compared to 1-to-N and N-to-N relationships.

When analyzing the impact of graph metrics on model performance, several trends emerged:

- Number of Edges: As the number of edges increased, the models faced greater challenges in processing the graphs with the same set of parameters as visualized in Figure 16. However, the relationship between the number of edges and performance degradation was sublinear. This indicates that while increased connectivity adds complexity, the models can handle it relatively efficiently up to a point.
- Average Path Length: A similar sublinear relationship was observed with average path length. Models experienced slight decreases in performance as the average path length increased, but the effect was less pronounced than with the number of edges. This suggests that models are robust to changes in the graph's navigational complexity.
- Clustering Coefficient: Within the small range of clustering coefficients present in our subgraphs, this metric did not appear to significantly influence model performance. This could imply that the models are less sensitive to the local cohesiveness of the graph or that the variation in clustering coefficients was insufficient to elicit noticeable effects.
- Density: An increase in graph density correlated with improved performance across all models shown in Figure 15. Denser graphs provide more relational information, which the models can use to learn better representations. This highlights the importance of having sufficient relationships within the graph to facilitate effective learning.

The exploration of different graph embedding models revealed that more complex models like GLM offer superior performance but at the cost of computational efficiency. GAE strikes a balance between performance and resource requirements, making it a viable option for applications where speed and efficiency are crucial. VGAE did not offer the expected advantages over GAE, suggesting that its additional complexity may not be justified in all contexts.

The limitations of TransE underscore the challenges of modeling complex relationships using simple embedding techniques. The inconsistency in its performance indicates that reliance solely on structural information is insufficient for capturing the nuances of real-world graph data.

The influence of graph metrics on model performance highlights the need to consider graph properties when selecting and tuning models. Understanding how factors like edge count, path length, and density affect learning can guide the development of more robust and efficient graph embedding approaches.

Overall, this experiment demonstrates the importance of matching the choice of model to the specific characteristics of the graph data and the requirements of the application. TransE, for instance, struggled significantly with graphs containing many edges, where the proportion of 1-to-1 relationships was smaller relative to 1-to-N and N-to-N relationships, highlighting its limitations in capturing complex graph structures. While GLM has shown superior performance in other scenarios, the computational and memory demands of processing larger graphs exceeded the available resources in this case. Therefore, GAE was used as a practical alternative for these larger graphs, balancing efficiency, and capability. Future work will focus on investigating the feasibility of using GLM for such large-scale graphs to further explore its potential in handling complex graph data. By carefully analyzing both model capabilities and graph properties, I can achieve more effective and efficient graph representations for a wide range of tasks.

5.3.2 Specific evaluation indicator: Quality of Embedding (QEmb)

To evaluate the quality of the embeddings in this specific recommendation task, I developed a heuristic indicator called Quality of Embedding (QEmb). This indicator was designed specifically for this use case and is not intended as a general-purpose measure; however, it effectively illustrates how changes in both node features and graph structure can significantly influence embedding quality. QEmb focuses on evaluating the model's ability to recommend competent developers for a new issue based on similar past issues.

For instance, considering the baseline synthetic graph in Figure 13, consider Issue-4, which is similar to Issue-2, where Issue-1 and Issue-2 also share similarities. Traditional text-based models might recommend developers like Person-2 or Person-1 to Issue-4, based on similarities in issue descriptions alone. However, if these developers are unavailable due to other commitments, text-based recommendations would fail to provide alternatives. The knowledge graph, on the other hand, incorporates additional data modalities that reveal Person-3 - who has previously worked on files related to Issue-2 - making them a suitable candidate. This scenario underscores the value of integrating multiple data sources to achieve better recommendations.

The QEmb indicator, designed for this context, evaluates the embeddings by checking whether Person-2 and Person-3 appear as the top candidates. If they are, the embedding is considered effective, with an ideal scenario showing a low probability gap between these two developers, indicating that both are likely candidates. Additionally, a high gap between the probability of the second and third-ranked candidates shows that the model confidently prefers the top two. Conversely, if Person-2 and Person-3 are not among the top two recommendations, the embedding receives a lower rating. In this case, the indicator accounts for cumulative differences between the probabilities of higher-ranked candidates and the rankings of Person-2 and Person-3, with a larger sum indicating a less effective embedding. Therefore, QEmb indicator can be calculated as seen in (10) and (11).

$If\ P2\ and\ P3\ were\ ranked\ first\ two$:

$$QEmb = 1 + |\min(Prob_{P2}, Prob_{P3}) - Prob_{third-ranked}| - 0.2 * |Prob_{P2} - Prob_{P3}| \quad (10)$$

$If\ P2\ and\ P3\ were\ NOT\ ranked\ first\ two$:

$$QEmb = -\sum_{i=1}^{N} (max(0, Prob_i - Prob_{P2}) + (max(0, Prob_i - Prob_{P3}) \quad (11)$$

Using QEmb across experiments provided quantitative insights into how modifications in graph structure and node features affect model recommendations. This analysis enabled the optimization of graph properties and node attributes to enhance the GNN's recommendation accuracy. Observing how these elements interact also allows for better predictions of model performance and proactive adjustments to achieve optimal embedding quality. By analyzing graph statistics and their influence on embedding quality, the QEmb indicator enables the assessment of how graph changes may impact embeddings in any software project's knowledge graph, supporting customized, effective GNN-based recommendations for software development contexts.

5.3.3 Overviewing the influence of graph structure and node features

To investigate how structural properties and node features affect embedding quality in GAE-based recommendation systems, three distinct dummy graphs were created. Each graph simulates a specific repository structure to highlight potential variations in graph metrics like connectivity and density. Definitions of these metrics are provided in

Table 3 for reference.

1. **Baseline Graph:** This graph represents a typical small-scale repository, balanced in the number of nodes and connections across Person, Issue, and File types. It serves as a standard for observing how typical connections in a moderately populated graph affect embedding quality.
2. **Sparse Graph:** This graph is characterized by disjoint sections with limited connectivity. With fewer connections and distinct clusters, it represents scenarios where certain parts of the repository may be isolated, posing challenges to information flow and potentially reducing embedding coherence.
3. **Dense Graph:** This highly connected graph simulates a situation where almost every node links to most others, forming a densely interwoven structure. High density can influence GAE's ability to preserve meaningful relationships, as excessive connections may lead to over-smoothing, resulting in undifferentiated node embeddings.

Metrics	Description
Longest-shortest path	Measures the maximum distance between the closest connected pairs of nodes, indicating the graph's overall spread.
Average degree	Represents the average number of connections per node, reflecting graph connectivity.
Average path length	Calculates the mean shortest path between all pairs of nodes, showing how easily information can travel within the graph.
Average clustering coefficient	Indicates the tendency of nodes to form tightly knit groups, revealing local density patterns.
Graph density	Measures the ratio of actual connections to possible connections, indicating how fully the graph is connected.
Number of disjunct subgraphs	Counts separate, unconnected clusters within the graph, illustrating fragmentation or isolated regions.

Table 3: This table presents key metrics describing the structural characteristics of the knowledge graph. These metrics provide insights into connectivity, clustering, and fragmentation, aiding in predicting the potential embedding quality of the graph.

Each graph's metrics, such as average degree, longest-shortest path, and clustering coefficient, reveal differences in connectivity and clustering that likely influence how information propagates during message passing.

- **Longest-Shortest Path:** High values limit information propagation across distant nodes, reducing overall graph awareness in embeddings.
- **Average Degree:** Higher degrees increase information diversity but can introduce noise, affecting node-specific expressiveness.
- **Average Path Length:** Longer paths hinder connectivity, limiting GAE's ability to integrate information across the graph.
- **Average Clustering Coefficient:** High clustering strengthens local neighborhoods, enhancing context for nodes with interconnected neighbors.
- **Graph Density:** Moderate density balances connectivity and specificity, making it ideal for expressive embeddings without over-smoothing.
- **Number of Disjoint Subgraphs:** More disjoint subgraphs reduce coherence, limiting information flow and weakening the embedding's global context.

The most critical of these metrics is graph density, as it optimally balances information flow and specificity, which are essential for creating effective and distinctive embeddings in GAE. High density leads to over-smoothing, while low density can cause overly sparse embeddings.

Determining which node features to assign for each node type is not a straightforward task. Node types in the knowledge graph vary significantly in their expressiveness and descriptiveness, which can define the model's recommendation strengths. Some node types, like issues, are highly descriptive, while others, like files or people, offer limited information. As shown in Table 1, I selected appropriate features for each node type, noting the variability in feature quality and structuredness.

Even preprocessing the features posed unique challenges, as certain node types contained unstructured data, such as issue descriptions. To standardize and improve these features, I employed a prompting an LLM to clean and refine them, removing irrelevant content such as URLs and logs to create more coherent node representations. This step, while necessary for enhancing feature quality, introduces complexity into the preprocessing phase and can significantly impact model performance.

To understand how different configurations of node features influence embedding quality, I tested the following scenarios on synthetic graphs:

- **Random Node Features:** In this setup, each node received randomly assigned features, allowing observation of the power of graph structure alone to propagate information. This test helps evaluate the role of connections in the graph and demonstrates the potential for certain connections to overshadow others, as well as the effects of oversmoothing, where node embeddings become indistinguishable.
- **Issue Features Only:** Here, Issue nodes were assigned high-quality features from the production graph, while all other nodes were given unified, type-specific features. This configuration is relevant because SBERT-based fine-tuning was performed specifically for Issue nodes, providing quality embeddings that include titles and the initial tokens of descriptions. Other node types, lacking similar feature quality, offer insights into how well the model can recommend developers and files based solely on Issue data.
- **Issue, Person, and File Features:** In this case, all nodes had features derived from the production graph: issue nodes retained their descriptive embeddings; person nodes included less expressive inferred expertise and IDs; and file nodes used SBERT-embedded filenames, which also lack richness but may correlate with expertise. This configuration helps assess the impact of using real-world data across different node types, allowing observation of correlations such as between developer expertise and file blocks.

5.3.4 Effects of structural and feature configurations on embedding quality

Subsequently, I present the results from evaluating the impact of different node feature and structural configurations on the quality of embeddings. The key metrics for the synthetic graphs, including graph density, average degree, longest-shortest path, and number of disjunct subgraphs, are outlined in Table 4. These metrics provide a comparative baseline for understanding the structural properties of each synthetic graph and their potential influence on embedding quality. By comparing the QEmb and MRR values across these graph structures, I aim to draw insights into how these factors influence the effectiveness of the recommendation system. The QEmb and MRR matrices for each graph structure are displayed in Figure 17, Figure 18, and Figure 19. These results will guide the analysis on optimizing graph structures and node features for improved performance in real-world applications.

	Graphs		
Metrics	*Baseline graph*	*Sparse graph*	*Dense graph*
Longest-shortest path	5	5	3
Average degree	2.375	1.875	6.250
Average path length	2.658	2.838	1.625
Average clustering coefficient	0.000	0.000	0.000
Graph density	0.158	0.125	0.416
Number of disjunct subgraphs	1	2	1

Table 4: Descriptive metrics for the synthetic graphs.

As expected, the synthetic graphs have similar values for metrics like the average path length and longest-shortest path due to their equal number of nodes. The average clustering coefficient is 0 in all cases, as these graphs are Issue-centric; however, in non-Issue-centric scenarios, this metric could be significant in analyzing local connectivity and clustering tendencies. This simplification does not diminish the value of graph density and average degree, where we observe clear differences across the three graph types. Graph density and average degree effectively capture the structural variation between the synthetic graphs, even though they contain the same set of nodes, underscoring their importance as indicators of connectivity and overall structure.

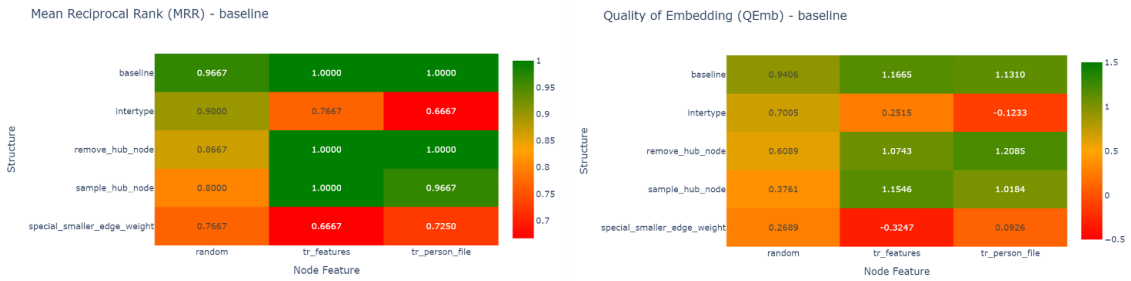

Figure 17: Heatmaps of MRR and QEmb metrics for various structural and node feature configurations on the baseline graph. Rows represent structural adjustments; columns show node feature setups. Higher values indicate better performance, illustrating the impact of structure and features on embedding quality in GAE-based recommendations.

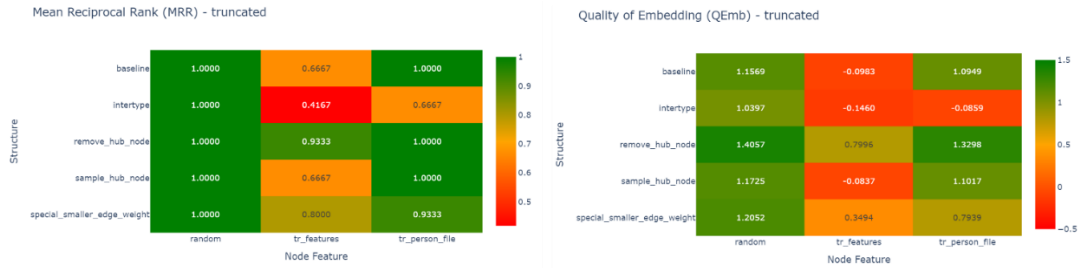

Figure 18: Heatmaps of MRR and QEmb metrics for various structural and node feature configurations on the sparse graph. Rows represent structural adjustments; columns show node feature setups. Higher values indicate better performance, illustrating the impact of structure and features on embedding quality in GAE-based recommendations. The best performance is achieved with random node features, as the model relies solely on structural information, while meaningful node features lower MRR due to missing reasonable connections in the graph.

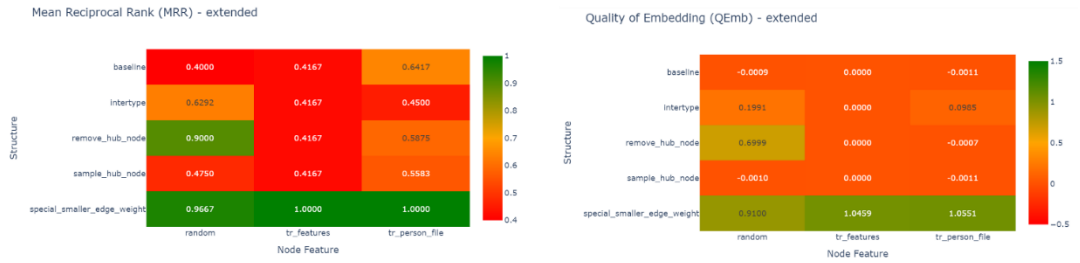

Figure 19: Heatmaps of MRR and QEmb metrics for various structural and node feature configurations on the dense graph. Rows represent structural adjustments; columns show node feature setups. Higher values indicate better performance, illustrating the impact of structure and features on embedding quality in GAE-based recommendations.

The synthetic graph was trained using a 2-layer GCN encoder and a dot product decoder for 90 epochs with a learning rate of 0.0001. The Adam optimizer was employed, alongside a small weight decay of 1e-5 to enhance model generalization. Based on the results observed across the baseline, sparse, and dense graph configurations, several key conclusions can be drawn regarding the effects of structure and node features on embedding quality in GAE-based recommendation systems.

The baseline graph configuration consistently shows stability across different node feature setups. This balance between structure and connectivity allows for moderate performance even with random node features, as sufficient relationships facilitate meaningful message passing. When higher-quality node features are introduced, the embeddings improve significantly, indicating an effective balance between graph structure and node features. This configuration proves to be the most robust, aligning with expectations for a well-connected graph without excessive complexity.

In the dense graph, oversmoothing is evident, particularly when the QEmb value reaches around 0.0, meaning the embeddings become indistinguishable. This result suggests that excessive connections lead to an averaging effect across node embeddings, diminishing the model's ability to differentiate among nodes. This problem is particularly pronounced in graphs where nearly all nodes are interconnected, and as a result, the embeddings lose their specificity. Thus, dense graphs can impede performance by diluting the unique characteristics of each node.

While initially, the sparse graph appears to perform reasonably well, this structure shows considerable instability when different node features are applied. Due to limited connectivity, the information each node learns is highly dependent on its few neighbors. If those neighbors have non-descriptive or irrelevant features, it can heavily impact the embedding quality. This scenario underlines the risk of sparse connectivity, where a lack of redundancy in relationships makes the model overly sensitive to feature quality, potentially degrading the recommendation system's robustness.

The production graph is issue-centric, containing various inter-issue relationships. These issue-to-issue connections strongly influence the embedding process, as can be observed in the results. This setup indicates that in practical applications, emphasizing and fine-tuning these specific connections is crucial for enhancing recommendation quality. The abundance of issue connections in the real-world graph could either facilitate or hinder performance, depending on how well these relationships are weighted and integrated.

Assigning higher weights to critical relationships, such as between issues and their responsible developers, proved beneficial, especially in denser graphs. When numerous connections are present, adjusting edge weights allows the model to prioritize important relationships, improving the specificity of embeddings despite the potential oversmoothing effect of high connectivity.

In sparse graphs, the removal of non-essential high-degree nodes (hub nodes) shows positive effects, particularly when these nodes are overly general (e.g., configuration files). The alternative approach, sampling from these hubs rather than complete removal, was less effective, as it only partially mitigated the overwhelming influence of these nodes. Hub nodes with specific relevance (e.g., experienced developers) should remain, while general hubs may need adjustment.

In summary, these findings underscore the need for balanced graph structure and well-curated node features in recommendation systems. A middle ground between sparse and dense structures offers stability, as dense graphs risk oversmoothing and sparse graphs lack robustness. Additionally, strategic edge weighting and hub node management further enhance embedding quality.

6 Experimental results

In this section, I present the results from applying the graph embedding methodology on the production graph. I first collected key structural metrics and compared them to the synthetic graphs to identify differences and areas for improvement. Based on these comparisons, I applied targeted modifications to optimize embedding quality, following insights from the synthetic graph experiments.

Despite these adjustments, initial results were not fully satisfactory. The lack of high-quality embeddings for certain node types - Commits, Person, and File nodes - significantly impacted overall embedding quality. To address this, I augmented features for these nodes, leading to substantial improvements.

With the augmented features, the results improved markedly. Finally, I evaluated this approach against traditional methods like BM25 and the SBERT-based approach, assessing the benefits of graph-based embeddings in terms of recommendation accuracy and embedding quality.

6.1 Production graph structure

In Table 5, the graph metrics for the production graph compared to those of the synthetic graphs.

	Graphs			
Metrics	*Baseline graph*	*Sparse graph*	*Dense graph*	***Real graph***
Longest-shortest path	5	5	3	**27**
Average degree	2.375	1.875	6.250	**3.2040**
Average path length	2.658	2.838	1.625	**7.7063**
Average clustering coefficient	0.000	0.000	0.000	**0.0056**
Graph density	0.158	0.125	0.416	**0.0002**
Number of disjunct subgraphs	1	2	1	**490**

Table 5: The graph descriptive metrics for the real graph compared to the values for the synthetic graphs.

From this data, several conclusions can be drawn. First, the production graph is significantly sparser than any of the synthetic graphs, as shown by its extremely low graph density. Additionally, the presence of 490 disjunct subgraphs suggests potential limitations in information flow, which could hinder the spread of knowledge across the entire graph. Due to the larger graph size, the average path length and longest-shortest path metrics are much higher, indicating that information must traverse significantly longer paths, which could dilute the message-passing effectiveness.

When examining the relationships between these metrics, it is evident that the production graph does not closely resemble the sparse or dense synthetic graphs. In some ways, it resembles the baseline graph, as its average degree is closer to the baseline (and even the dense graph to some extent) due to the presence of hub nodes. These hub nodes contribute to maintaining a higher average degree, despite the overall sparse structure, which could still help in balancing information flow in certain areas of the graph.

In summary, the production graph combines aspects of all the synthetic graphs, meaning we must draw on insights from each of the synthetic cases to fully understand how to optimize this graph. This combination of sparseness with significant hubs will require careful adjustments that account for both structural complexity and node feature quality to achieve optimal embedding performance.

6.2 Utilizing experiments from the synthetic graph

Based on insights gained from the synthetic graph experiments, I carefully applied several adjustments to the production graph to optimize the training process and improve the quality of embeddings. One major finding from the synthetic graphs was that inner-type connections, particularly among Issue nodes, could dominate the relationships within the graph, potentially overshadowing other important connections. To address this, I assigned a lower edge weight to Issue-Issue links, allowing more diverse relationships to have a balanced influence during training.

Additionally, the experiments highlighted the challenges presented by high-degree nodes, which can introduce noise or oversimplify the embeddings. Therefore, I selectively removed certain high-degree File nodes that were too general, as they could dilute the specificity of other features and relationships. This step helped maintain more meaningful connections and improved the graph’s expressiveness.

Finally, the synthetic graph results confirmed the importance of high-quality node features in achieving robust embeddings. Consequently, I prioritized using the highest quality node features available, especially for key node types like Issues. This focus on quality ensured that the embeddings captured nuanced and relevant information, which was essential for producing effective recommendations. Through these targeted adaptations, I applied insights from the synthetic graph to adjust the production graph setup for optimal training performance.

In the production graph, high-quality node features were primarily available only for Issue nodes. These were constructed using a fine-tuned SBERT model, enabling embeddings that captured semantic similarity between Issues; similar Issues were represented by closely related features, while dissimilar Issues had distinct embeddings. However, the quality of node features for other types was less robust. For Commits, I could only use SBERT embeddings without fine-tuning, as identifying meaningful positive and hard negative examples for this node type posed a challenge. Consequently, the Commit embeddings were less effective in capturing the nuanced relationships between commits and issues.

For Person nodes, creating descriptive features was particularly difficult. While I attempted to represent each person's field of expertise, these features did not capture enough detailed information to meaningfully differentiate between individuals. File nodes faced similar challenges; using only file names as features proved insufficient for effective embeddings, as they lacked context and semantic depth.

To improve the embeddings of these less descriptive nodes, I applied a feature augmentation technique. This approach involved propagating features from the closest Issue nodes to other node types. For each non-Issue node, relevant features were aggregated from nearby Issue nodes, enhancing the expressiveness of their embeddings. This augmentation significantly improved the quality of node features for Commits, Persons, and Files, resulting in a notable improvement in the overall quality of embeddings. This strengthened the graph's utility in recommendation tasks by aligning all node types with the semantic richness initially available only for Issues.

6.3 Results of training

The training process yielded promising results, aligning well with our unique recommendation goals. Unlike typical GAE models, which focus on reconstructing the

graph by learning existing connections, our aim is to recommend relevant entities that may not have explicit links in the graph. This distinction is important, as we prioritize the ability to identify potential connections and recommendations over the strict reconstruction of existing relationships. The sparsity of the production graph adds to the challenge, making it difficult to train a model that correctly classifies every positive example while also reliably distinguishing negative edges. However, for recommendation purposes, our main concern is ensuring that the top-ranked results are relevant rather than capturing all possible links.

To evaluate the general performance of the model, I used the ROC AUC (Area Under the Receiver Operating Characteristic Curve) metric, which measures the model's ability to rank positive samples higher than negative ones without relying on a specific classification threshold. A higher ROC AUC indicates better distinction between positive and negative examples across various thresholds. **The model achieved a test ROC AUC of 0.7344**, which suggests it is reasonably effective in identifying relevant links. While this result alone does not define the final recommendation quality, it indicates we are heading in the right direction.

For a more targeted evaluation of the recommendation quality, I used a labeled dataset where experienced developers, who have been working on the software for several years, rated the relevance of recommended entities on a scale from 1 to 5. This dataset, though small (167 recommendations), was highly reliable due to the expertise of the evaluators. A rating of 1 indicated no relevance, while a 5 meant the recommendation was highly relevant. I used these annotations to fine-tune the training, including setting the learning rate and number of epochs, and to assess the effectiveness of the recommendations.

To further gauge the quality of recommendations, I selected a set of query issues and monitored the ranks of their associated nodes (those rated 4 or 5 by experts) during training. This approach provided insights into how well the model prioritized relevant nodes over time. Using a learning rate of 0.0001, I observed that the sum of the ranks of these annotated nodes steadily decreased over 1000 epochs, as shown in Figure 20. This reduction in rank signifies an improvement in recommendation quality, as relevant nodes were ranked higher as training progressed. Examining links that were not part of the training, validation, or test sets, the model continued to improve its ranking of relevant recommendations, indicating generalization beyond the seen data depicted in Figure 21.

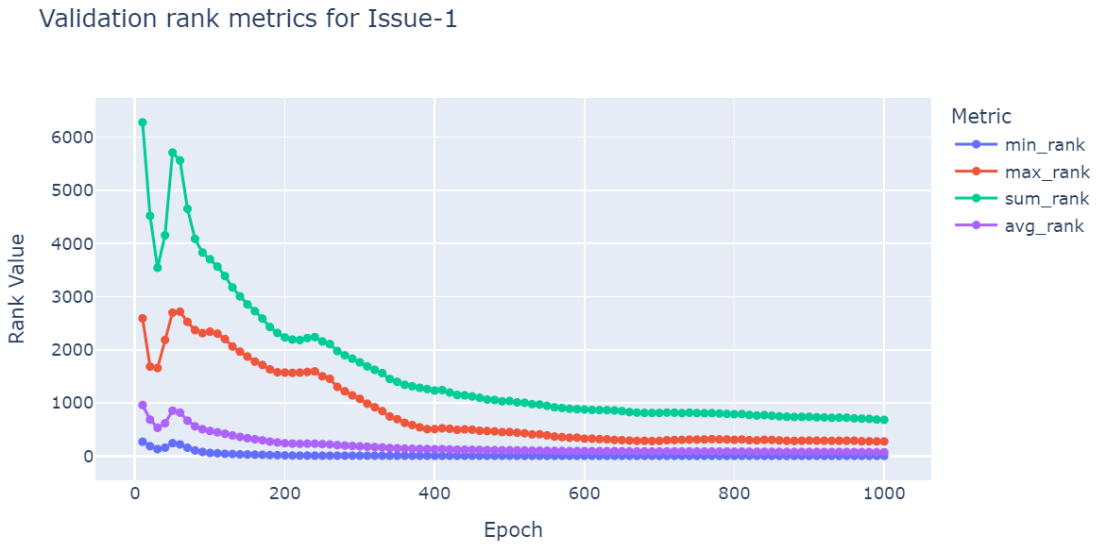

Figure 20: Validation rank metrics for a node with an edge in the validation set, showing the sum, average, minimum, and maximum ranks of relevant recommendations decreasing over training epochs, indicating improved relevance over time.

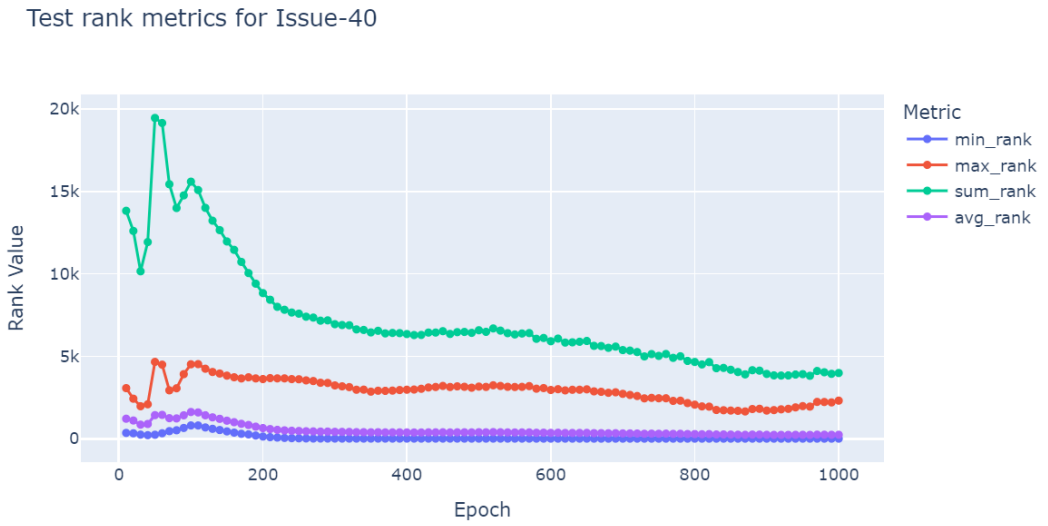

Figure 21: Test rank metrics for a node without any edges in the training, validation, or test sets, illustrating the decline in the ranks of relevant recommendations across epochs as the model generalizes to predict connections even for unobserved nodes.

To further analyze the recommendations, I visualized the distribution of nodes based on their probability of forming a link with a specific node. This distribution reflects the likelihood of each node being relevant to the target node, effectively showing how well the model predicts related entities. In the histogram displayed in Figure 22**,** Figure 23 and Figure 24, real neighbors from the graph are marked in red, while recommendations from traditional approaches - SBERT-based recommendations in green

and BM-25 in blue - are also highlighted. The closer these real or highly rated nodes are to the top, the better our model's performance.

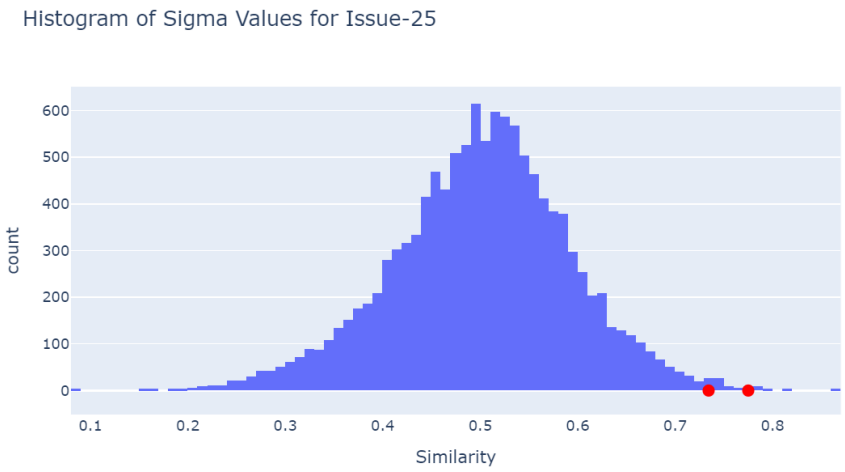

Figure 22: Histogram showing the distribution of nodes ranked by their relatedness to a query node, with real neighbors (red) appearing as the top-ranked nodes, indicating accurate recognition of existing graph connections.

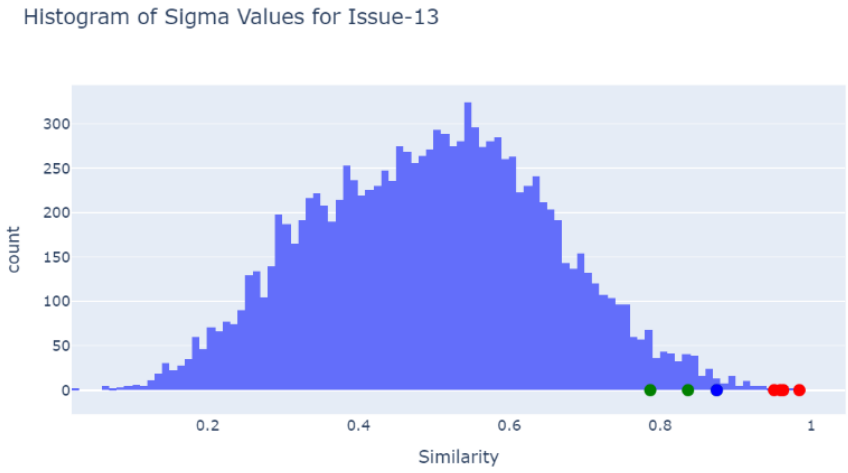

Figure 23: Histogram displaying ranked nodes, where real neighbors (red) are followed closely by good recommendations from traditional methods (BM25 in blue, SBERT in green), demonstrating that common approaches can also identify relevant nodes with high accuracy.

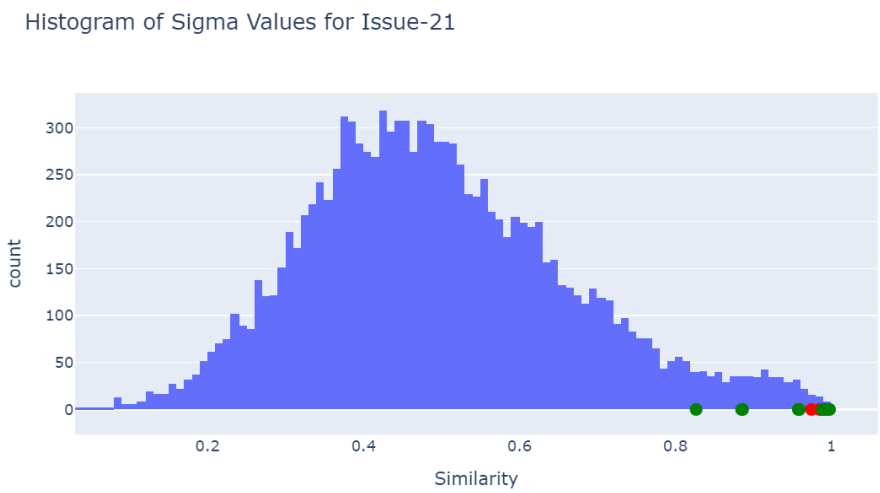

Figure 24: Histogram illustrating a scenario where some relevant recommendations (blue and green) are ranked higher than actual neighbors, underscoring the knowledge graph's incomplete edge problem, where certain valuable connections may be missing but inferred by the model.

The model ranks many real neighbors and highly rated recommended nodes within the top 5 of their respective types, underscoring the effectiveness of our approach. In Figure 22, we see a histogram where real neighbors (highlighted in red) are ranked highest, showing that the model correctly identifies nodes with explicit connections to the query. Figure 23 expands on this by introducing recommendations from common approaches - BM25 (blue) and SBERT (green) - which also achieve high ranks close to the real neighbors. This demonstrates the utility of traditional methods in identifying relevant nodes, even when they are not directly connected in the graph. In Figure 24, certain recommendations surpass actual neighbors in ranking, reflecting the knowledge graph's inherent limitations due to missing edges. This outcome shows that high-quality recommendations might sometimes represent more meaningful connections than those explicitly present in the knowledge graph, highlighting the model's capacity to infer relationships effectively. This outcome confirms that our model successfully identifies not only existing links but also potential relationships, which the traditional methods may overlook. The ability to consistently rank these relevant nodes highly shows the strength of our GAE-based framework in providing meaningful, context-aware recommendations within a complex software environment.

In summary, these results demonstrate the practical success of our approach. By gaining the insights from synthetic graph experiments, optimizing edge weights, removing certain high-degree nodes, and augmenting node features, we were able to train

a model that performs well in a recommendation setting. To further investigate the relevance of this approach a comparison between GAE and common approaches would be necessary.

6.4 Comparison with common approaches

To assess the performance of my Graph Auto-Encoder approach in recommending appropriate developers and relevant files for software issues, I conducted a comparison with two baseline methods: an SBERT-based approach and the BM25 Elasticsearch method. I used the small labeled dataset availeable by the developers. The SBERT-based and BM25 approaches served as my benchmarks. These methods yielded the following distribution of ratings for their recommendations listed in Table 6.

Distribution of ratings for recommendations of SBERT-based approach and BM-25	
Ratings of 4 or 5 (Highly Related)	40 recommendations
Rating of 3 (Moderately Related)	40 recommendations
Rating of 2 (Slightly Related)	20 recommendations
Rating of 1 (Not Related at All)	67 recommendations

Table 6: Distribution of ratings for recommendations of SBERT-based approach and BM-25.

Notably, in more than a third of the cases (67 out of 167), these baseline methods recommended entities that were completely irrelevant to the issues at hand. Applying my GAE-based approach to the same set of issues, I evaluated the ranks assigned to recommended entities according to expert ratings. Table 7 shows the average rank positions that my model assigned to entities based on their expert ratings. Importantly, although the mean rank for highly relevant recommendations (ratings of 4 or 5) was 28.7, this number is somewhat misleading due to the presence of a few outliers; the median rank for these highly relevant recommendations was a much more competitive 8.

Average ranks with GAE for recommendations of SBERT-based approach and BM-25	
For Recommendations Rated 4 or 5	28.7 (median: 8)
For Recommendations Rated 3	83.375
For Recommendations Rated 2	227.45
For Recommendations Rated 1	733.08

Table 7: Average ranks with GAE for recommendations of SBERT-based approach and BM-25.

These findings demonstrate that my GAE approach effectively distinguishes between highly relevant and irrelevant entities. Recommendations rated as highly

relevant by experts (ratings of 4 or 5) were consistently ranked higher by my model, while the average rank increased as the relevance of the recommendation decreased.

Additionally, in nearly half of the (19 out of 40) cases rated 4 or 5 by experts, the GAE model ranked the relevant entity within the top 10 positions, significantly improving the efficiency of locating critical recommendations. In contrast, the SBERT-based and BM25 methods often struggled to prioritize relevant entities accurately, as evidenced by the large number of recommendations rated as completely irrelevant. By utilizing the rich semantic and structural information in the knowledge graph, my GAE approach captures nuanced relationships within the software context, offering more precise and contextually meaningful recommendations.

Another key finding from my analysis was the effectiveness of the Graph Auto-Encoder (GAE) approach compared to the SBERT-based model in the top 5 recommendations. I reviewed the top 5 predictions generated by both methods and found that 42% of the top 5 recommendations from my GAE-based approach received a rating of 4 or 5 (highly relevant), whereas only 30% of SBERT's top 5 predictions achieved similar ratings as depicted in Figure 25. Additionally, when focusing on file recommendations for issues, my model achieved a 64% success rate in recommending files that were rated as highly relevant (4 or 5).

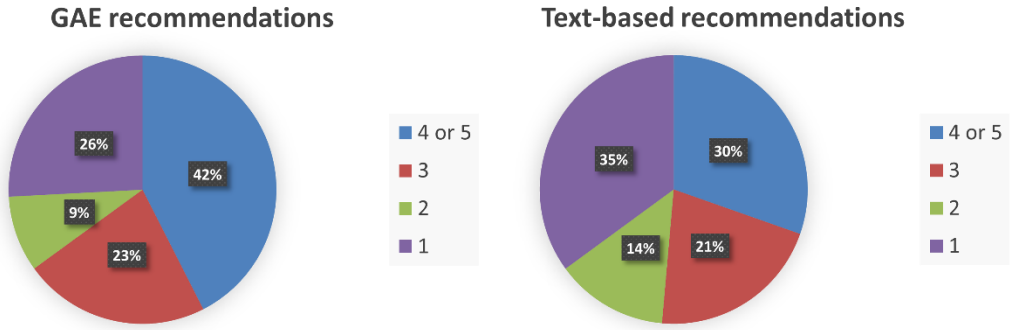

Figure 25: Comparison of relevance ratings for recommendations generated by the GAE-based model and the SBERT-based model. The pie charts show that the GAE-based model outperformed the SBERT-based model by generating a higher proportion of highly relevant recommendations (4 or 5). This highlights the GAE's ability to consider the knowledge graph's structural and relational information more effectively than SBERT.

A common pattern emerged where my model not only identified the relevant entities also recommended by SBERT but also uncovered additional, related entities, thanks to the rich relational information embedded in the knowledge graph. This suggests

that my approach could function effectively as an extension to the SBERT model, enhancing its recommendations with added relational depth and relevance. By using the structural connections in the knowledge graph, my model offers a broader and more semantically relevant recommendation set, potentially making it a more comprehensive tool for developer and file suggestions.

Some experts attached textual feedbacks to the annotated data as well. Based on these, it became evident that certain limitations in the knowledge graph affected the recommendations. For instance, similar modules running on different platforms (e.g., cloud-native vs. virtualized environments) were sometimes suggested as related, despite being unrelated to the specific context of the query. This issue likely arose because platform-specific information was not explicitly included in the knowledge graph. Additionally, the evaluation process revealed that different experts had varying interpretations of relevance - some prioritized exact matches to the query element, while others valued slightly different but contextually useful components. This highlights the subjective nature of relevance and the importance of tailoring recommendations to diverse user expectations.

However, based on this evaluation, I can assert with high confidence that my GAE-based approach offers a superior method for recommending competent developers and relevant files in response to software issues. The ability of my model to align closely with expert assessments demonstrates its practical applicability and potential to enhance efficiency in software development workflows. By effectively ranking relevant entities higher and deprioritizing less relevant ones, my approach improves the accuracy of recommendations and aids in reducing the time and effort required to resolve issues.

7 Applicability

The use of Graph Auto-Encoders in constructing recommendation systems for software development environments offers several significant advantages. Firstly, GAEs are fast to train, making them suitable for iterative development processes where quick updates are essential. Their training efficiency stems from their ability to learn representations without requiring labeled data, operating in an unsupervised or self-supervised manner. This characteristic is particularly beneficial in domains where labeled data is scarce or expensive to obtain.

GAEs exhibit high scalability, as their computational complexity is not directly determined by the number of nodes in the graph but rather by the number of edges and the size of the neighborhood considered during message passing. This means that even for large graphs with millions of nodes, the model remains computationally feasible if the average degree of nodes is manageable. The use of techniques like neighborhood sampling further enhances scalability by limiting the number of neighboring nodes each node aggregates information from during training.

An essential feature of GAEs is their ability to handle dynamic graphs efficiently. When a new node is introduced, there is no need to retrain the entire model. Instead, one can perform a forward pass to compute the embedding for the new node using the existing model parameters. This involves aggregating messages from the new node's neighbors, a process that is significantly faster than retraining. This property is invaluable in rapidly evolving software projects where new issues, files, or developers are continually added.

The embeddings learned by GAEs reside in a semantically rich latent space, capturing both the structural relationships and feature information of the nodes. This latent space facilitates more accurate and meaningful recommendations, as it reflects the complex interactions within the software development ecosystem. By integrating multiple data modalities and relationships, the GAE provides a holistic view that is often unattainable with traditional recommendation approaches.

Despite these advantages, there are certain limitations to consider. Achieving exceptional results with GAEs often requires careful graph construction, including the right amount and types of relationships. Proper handling of structural phenomena such as hub nodes and inner-type relationships is crucial. For instance, hub nodes may dominate

the message-passing process, leading to oversmoothing and loss of distinctive node features. Similarly, inner-type relationships might skew the embeddings if not appropriately weighted or filtered. Therefore, domain expertise is necessary to effectively adapt the graph structure.

The quality of node features plays a central role in the performance of GAEs. Nodes with rich, descriptive features enable the model to learn more meaningful embeddings. In scenarios where certain nodes lack informative features, techniques like feature augmentation - aggregating features from neighboring nodes - become essential. However, this reliance on high-quality node features can be a limitation in contexts where such data is unavailable or difficult to obtain.

Another challenge lies in the evaluation of recommendation systems. Measuring the effectiveness of recommendations is inherently complex, as it often involves subjective judgments about relevance and usefulness. Standard metrics may not capture all dimensions of performance, and there is no one-size-fits-all solution. This complexity requires the development of customized evaluation strategies that align with specific project goals and user needs.

Finally, the GAE model requires that the node for which a recommendation is being made has at least one relationship in the graph. Nodes without any connections cannot participate in the message-passing process, rendering the model incapable of computing embeddings for them. This limitation highlights the importance of ensuring that all relevant entities are adequately connected within the knowledge graph.

In summary, while GAEs offer a powerful and efficient approach for building recommendation systems in software development, their success depends on thoughtful graph construction, quality node features, and appropriate evaluation methods. Recognizing and addressing these limitations is key to utilizing the full potential of GAEs in practical applications.

8 Conclusions

In this work, I explored the application of knowledge graphs and Graph Neural Networks (GNNs) for improving recommendation tasks within complex software development environments. By integrating diverse data modalities - such as code, documentation, issues, and developer interactions - into a semantically rich, unified graph representation, I demonstrated a powerful framework that offers valuable insights into developer assignments and issue resolution. My approach utilizes the strengths of GNNs, particularly Graph Auto-Encoders (GAE), to create embeddings that capture complex relationships within the software ecosystem, improving the quality and relevance of recommendations over traditional unimodal approaches like text embeddings.

The results highlight the performance of GAE-based recommendations compared to other models and baselines. Specifically, the GAE outperformed SBERT-based approaches, achieving 42% of highly relevant (4 or 5) recommendations compared to 30% for SBERT. Additionally, for file recommendations related to issues, my method achieved a relevance accuracy of 64%. Beyond this, I conducted a comprehensive comparison of four models: TransE, GAE, Variational Graph Auto-Encoders (VGAE), and Graph Language Models (GLM). While TransE struggled with the complexity of the graphs and failed to provide consistent results, VGAE offered limited improvements over GAE, with similar or worse performance in most cases. The GLM, despite showing strong potential due to its ability to unify structural reasoning and semantic encoding, was not used for larger graphs due to resource constraints. However, GLM remains a promising direction for future investigations.

These findings demonstrate that incorporating the relational depth and structural context of knowledge graphs leads to significantly more effective recommendations, particularly for complex software projects. The comparison of models underscores the importance of selecting approaches that align with the specific characteristics of the data and application requirements. This study establishes a foundation for future work, where using models like GLM could further enhance recommendation quality, especially in large-scale environments. Overall, this approach holds considerable promise for improving efficiency in navigating and managing complex, long-lived software ecosystems.

Acknowledgements

I would like to sincerely thank my advisors for their guidance and support throughout this project. I am profoundly grateful to my family for their constant encouragement, with special appreciation to my brother, Gergő, as he embarks on an exciting new chapter of his life with the construction of his future home. I wish him all the best in this endeavor and in the years ahead.

Bibliography

[1] R. Minelli, A. Mocci and M. Lanza, "*I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time*" Florence, 2015.

[2] K. Damevski, D. Shepherd and L. Pollock, "*A field study of how developers locate features in source code*" *Empirical Software Engineering,* vol. 21, p. 724–747, 2016.

[3] Falkor, "*Code Graph*" 22 July 2024. [Online]. Available: https://www.falkordb.com/blog/code-graph/. [Accessed 2 11 2024].

[4] W. Lu, S. Chenhan, Z. Chongyang, N. Weikun and H. Kaiyuan, "*Application of knowledge graph in software engineering field: A systematic literature review*" *Information and Software Technology,* vol. 164, 2023.

[5] B. Abu-Salih, "*Domain-specific knowledge graphs: A survey*" *Journal of Network and Computer Applications,* vol. 185, p. 103076, 2021.

[6] F. Qiu, Z. Liu, X. Hu, X. Xia, G. Chen and X. Wang, "*Vulnerability Detection via Multiple-Graph-Based Code Representation*" *IEEE Transactions on Software Engineering,* vol. 50, pp. 2178-2199, 2024.

[7] A. Nayak, V. Kesri and R. K. Dubey, "*Knowledge Graph based Automated Generation of Test Cases in Software Engineering*" in *Proceedings of the 7th ACM IKDD CoDS and 25th COMAD*, Bangalore, 2020.

[8] J. Xu, J. Ai, J. Liu and T. Shi, "*ACGDP: An Augmented Code Graph-Based System for Software Defect Prediction*" *IEEE Transactions on Reliability,* vol. 71, pp. 850-864, 2022.

[9] R. Xie, L. Chen, W. Ye, Z. Li, T. Hu, D. Du and S. Zhang, "*DeepLink: A Code Knowledge Graph Based Deep Learning Approach for Issue-Commit Link Recovery*" *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER),* pp. 434-444, 2019.

[10] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang and L. Zhang, "*Boosting coverage-based fault localization via graph-based representation learning*" in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE),* Athens, 2021.

[11] B. Abu-Salih, H. Alsawalqah, E. Basima, I. Tomayess and W. Pornpit, "*Toward a Knowledge-based Personalised Recommender System for Mobile App Development*" *Journal of Universal Computer Science,* vol. 27, no. 2, p. 208–229, 2021.

[12] X. Wang, X. Liu and J. Liu, "*A novel knowledge graph embedding based API recommendation method for Mashup development*" *World Wide Web,* vol. 24, p. 869–894, 2021.

[13] S. Emre, T. Eray and D. Uğur, "*RSTrace+: Reviewer suggestion using software artifact traceability graphs*" *Information and Software Technology,* vol. 130, p. 106455, 2021.

[14] S. Wu, W. Zhang, F. Sun and B. Cui, "*Graph Neural Networks in Recommender Systems: A Survey*" *CoRR,* vol. abs/2011.02260, 2020.

[15] L. Yang, Z. Liu, Y. Dou, J. Ma and P. S. Yu, "*ConsisRec: Enhancing GNN for Social Recommendation via Consistent*" *CoRR,* vol. abs/2105.02254, 2021.

[16] T. N. Kipf and M. Welling, "*Variational Graph Auto-Encoders*" in *Bayesian Deep Learning Workshop, 30th Conference on Neural Information Processing Systems (NeurIPS),* 2016.

[17] N. Reimers and I. Gurevych, "*Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*" *CoRR,* vol. abs/1908.10084, 2019.

[18] S. Robertson and H. Zaragoza, "*The Probabilistic Relevance Framework: BM25 and Beyond*" *Foundations and Trends in Information Retrieval,* vol. 3, pp. 333-389, 2009.

[19] Google, "*Official Google Blog*" Google, 16 May 2012. [Online]. Available: https://blog.google/products/search/introducing-knowledge-graph-things-not/. [Accessed 27 October 2024].

[20] L. Jure, "*Knowledge Graphs*" CS224W: Machine Learning with Graphs, Stanford University, 2021. [Online]. Available: https://snap.stanford.edu/class/cs224w-2021/slides/10-kg.pdf. [Accessed 2 November 2024].

[21] W. Quan, M. Zhendong, W. Bin and G. Li, "*Knowledge Graph Embedding: A Survey of Approaches and Applications*" *IEEE Transactions on Knowledge and Data Engineering,* vol. 29, pp. 2724-2743, 2017.

[22] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston and O. Yakhnenko, "*Translating Embeddings for Modeling Multi-relational Data*" in *Advances in Neural Information Processing Systems 26 (NeurIPS 2013)*, Nevada, 2013.

[23] Z. Wang, J. Zhang, J. Feng and Z. Chen, "*Knowledge Graph Embedding by Translating on Hyperplanes*" *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, 2014.

[24] Y. Lin, Z. Liu, M. Sun, Y. Liu and X. Zhu, "*Learning Entity and Relation Embeddings for Knowledge Graph Completion*" *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 2015.

[25] D. Bank, N. Koenigstein and R. Giryes, "*Autoencoders*" *CoRR*, vol. abs/2003.05991, 2020.

[26] F. Scarselli, M. Gori, C. A. Tsoi, M. Hagenbuchner and G. Monfardini, "*The Graph Neural Network Model*" *IEEE Transactions on Neural Networks*, vol. 20, pp. 61-80, 2009.

[27] T. N. Kipf and M. Welling, "*Semi-Supervised Classification with Graph Convolutional Networks*" in *5th International Conference on Learning Representations (ICLR)*, Toulon, 2016.

[28] K. O'Shea and R. Nash, "*An Introduction to Convolutional Neural Networks*" *CoRR*, vol. abs/1511.08458, 2015.

[29] "*CS220, Graph Neural Networks*" Stanford University, [Online]. Available: https://snap.stanford.edu/class/cs224w-2021/slides/07-GNN2.pdf. [Accessed 2 November 2024].

[30] W. L. Hamilton, R. Ying and J. Leskovec, "I*nductive Representation Learning on Large Graphs*" in *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, Long Beach, 2017.

[31] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò and Y. Bengio, "*Graph Attention Networks*" in *6th International Conference on Learning Representations (ICLR)*, Vancouver, 2018.

[32] D. P. Kingma and M. Welling, "*Auto-Encoding Variational Bayes*" in *2nd International Conference on Learning Representations (ICLR)*, Scottsdale, 2013.

[33] M. Plenz and A. Frank, "*Graph Language Models*" in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL), Volume 1: Long Papers*, Bangkok, 2024.

[34] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li and P. J. Lie, "*Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*" *Journal of Machine Learning Research*, vol. 21, p. 1–67, 2019.

Appendix

GAE scalability proof

GAE's scalability arises from using sparse matrix operations, shown in the following derivation, where node embeddings are aggregated and updated through sparse adjacency and degree matrices.

$$Let\ H^{(l)} = [h_1^{(l)} \dots\ h_{|V|}^{(l)}]^T \quad (12)$$

$$Then \sum_{u \in N(v)} h_u^{(l)} = A_{v,:} H^{(l)} \quad (13)$$

$$Let\ D\ be\ diagonal\ matrix\ where\ D_{v,v} = Deg(v) = |N(v)| \quad (14)$$

$$Then\ D_{v,v}^{-1} = \frac{1}{|N(v)|} \quad (15)$$

$$Therefore \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} \rightarrow H^{(l+1)} = D^{-1} A H^{(l)} \quad (16)$$

$$Finally\ H^{(l+1)} = \sigma\left(\tilde{A} H^{(l)} W_l^T + H^{(l)} B_l^T\right), where\ \tilde{A} = D^{-1} A \quad (17)$$

In Graph Auto-Encoders, node embeddings across layers are aggregated using sparse matrix operations, making the training process highly efficient. As in *(12)*, let $H^{(l)}$ be the matrix of node embeddings at layer l, where each column represents a node embedding. For each layer, the aggregation operation for neighbors is computed as in *(13)*, where *A* is the adjacency matrix, and $H^{(l)}$ represents the hidden embeddings matrix. To normalize this aggregation, we use a diagonal degree matrix *D* such that $D_{v,v} = |N(v)|$, resulting in *(16)*. In this way, the final update formula, as seen in *(17)*, efficiently combines neighborhood and self-information, enabling scalable training through sparse matrix operations.

Evaluation metric definitions

Metric	Description
Mean Rank (MR)	The average rank of the true positive entity across all predictions. A lower MR indicates better performance.
Mean Reciprocal Rank (MRR)	The average reciprocal of the rank of the true positive entity. Higher values indicate better ranking accuracy.
ROC AUC	The area under the Receiver Operating Characteristic (ROC) curve, which shows the model's ability to distinguish between positive and negative classes. A higher AUC value indicates better discriminatory power across all classification thresholds.
Hits@k	The proportion of cases where the true positive entity appears in the top k predictions. Common values for k are 1, 3, and 10. A higher Hits@k indicates better performance.
Accuracy	The ratio of correctly predicted results to the total number of predictions. Used for binary or multi-class classification tasks.
Precision	The proportion of relevant instances among the retrieved instances. Useful for measuring the relevance of predictions.
Recall	The proportion of relevant instances that were successfully retrieved. Reflects the completeness of predictions.
F1 Score	The harmonic mean of precision and recall. Balances both precision and recall in a single metric, useful for imbalanced datasets.

Table 8: This table provides an overview of the evaluation metrics used to assess model performance. Metrics include Mean Rank (MR), Mean Reciprocal Rank (MRR), ROC AUC, Hits@k, Accuracy, Precision, Recall, and F1 Score, each offering unique insights into ranking accuracy, relevance, completeness, and classification performance.