Masters's Thesis

Predicting Contributions in Open-Source Software Projects

A Graph Neural Network Approach

Emily Ries

April 2, 2026

Advisor:

Alisa Welter	Chair of Software Engineering
Tobias Dick	Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel	Chair of Software Engineering
Prof. Dr. Verena Wolf	Chair of Modelling and Simulation

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

Emily Ries: *Predicting Contributions in Open-Source Software Projects*, © April 2026

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

Open-Source Software (OSS) has become a fundamental part of modern software development. While its beneficial properties, such as public and license-free access, drive innovation, the accompanying uncoordinated contributions can negatively impact software quality, leading to inefficiencies, erroneous code, or inconsistent implementation styles. These collaborations can pose risks to project sustainability and long-term maintenance.

To mitigate these issues, this thesis presents the **Graph**-based **C**ontribution **P**redictor (GraphCP), a novel approach to predict future contributions by learning from past collaboration patterns. Furthermore, we provide an extensive discussion on the practical application of such an approach. We demonstrate the effectiveness of link prediction in the OSS domain and leverage GraphSAGE to capture the complex dynamics of developer-artifact interactions.

Methodologically, we model OSS projects as developer networks by extracting historical contribution data from GitHub. We then apply GraphSAGE, a well-established Graph Neural Network (GNN) architecture, to learn the latent collaborative representations of developers and use a small neural network as a classification head. This approach provides a structured framework for evaluating the applicability of GNNs in developer networks.

Our evaluation demonstrates promising results in predicting project contributions. Consequently, these findings indicate that modeling developer networks with GNNs is a viable approach to support OSS development. In practice, such a model can serve as a foundation for contribution recommendation systems, anomaly detection, or change impact analysis. Hence, enhancing project coordination and supporting the sustainability of OSS ecosystems.

Contents

List of Figures

List of Tables

Acronyms

OSS Open-Source Software
AI Artifical Intelligence
ML Machine Learning
DL Deep Learning
DNN Deep Neural Network
GNN Graph Neural Network
VCS Version Control System

1 Introduction

The history of open-source software started in 1985, when the *Free Software Foundation*[1] was founded. In 1991, the first release of the most prominent open-source software (OSS) Linux[2] marked a milestone in the development of OSS. Finally, the launch of GitHub[3], a platform hosting OSS projects, in 2008, opened a new dimension for collaboration. Since then, contributions to open-source projects have increased [22]. Open-source ecosystems, such as Linux, Apache, or Kubernetes, provide building blocks for applications and services. As a result, they have become foundational to modern IT infrastructure [6].

With OSS, the dynamics of software development have shifted from strictly closed-source competition towards a paradigm of global, collective effort to drive technological advancement [6]. Nowadays, OSS is a critical infrastructure and according to the 2025 *State of Open Source Report* [4], 96% of the consulted organizations either maintained or increased their use of OSS in the past year [53]. They are primarily driven by cost efficiency and the ability to leverage the expertise of developers from around the world [53].

Despite its advantages, OSS also faces some challenges. The freedom of open collaboration is accompanied by uncoordinated contributions, making it more difficult to maintain an overview of who is working on which artifact. These unmanaged contributions can lead to mistakes that are not immediately apparent, for example, hidden bugs from less experienced developers [2]. This particularly affects the software security. Furthermore, as different developers from around the world can contribute to OSS projects independently from each other, they might introduce varying code styles or even duplicate work. This can negatively impact the software quality and efficiency as it requires more reviews to maintain the project [31]. A further problem can emerge when developers leave the project and have not documented their contribution properly. This creates knowledge gaps, which also leads to increased effort for maintaining the project. In general, OSS projects are highly dependent on their community and contributors. This results in potential risks, as unstructured community work can jeopardize the project.

The goal of this thesis is to explore the potential of link prediction and propose a method that helps to support project coordination, quality, and security. To achieve this, we introduce the **Graph**-based **C**ontribution **P**redictor (GraphCP), which utilizes developer networks that capture the evolving dynamics between project contributors and artifacts. There exists research using developer networks to classify core/persistent and peripheral/fluctuating

1 https://www.fsf.org/de
2 https://www.linux.org/
3 https://github.com/
4 https://www.openlogic.com/resources/state-of-open-source-report?utm_source=osi&utm_medium=website-referral&utm_campaign=opl-glb-2025q1-con-2025stateofopensourcereport&utm_content=resources

developers [25] or detecting factors that contribute to project successes or failures [24]. Yet, to the best of our knowledge, the potential of link prediction for OSS-based developer networks remains largely unexplored.

To this end, we first provide an extensive discussion on the practical applications of link prediction in OSS development. These range from automated contribution recommendations to preventive risk assessment through methods such as change impact analyses or anomaly detection. Furthermore, exploring the different scenarios underscores the importance of a deep learning approach to capture the complexity of developer networks, which static rule-based systems typically fail to address.

Second, we develop and implement GraphCP to demonstrate the technical feasibility of these applications. A core aspect of our research is exploring the applicability of the well-established GraphSAGE [19] architecture to OSS developer networks. By strategically focusing on the bipartite collaboration between authors and files, we establish a foundational proof of concept that fulfills the structural assumptions of GraphSAGE. We develop a model that transforms a raw graph into a suitable representation for the downstream task link prediction. Finally, we independently evaluate GraphCP on 7 different OSS projects of varying sizes, and compare the performances against traditional baselines.

Our findings reveal that GraphCP outperforms two out of three baselines and remains highly competitive with the third, i.e., preferential attachment (PA). Notably, GraphCP performs especially well when predicting interactions between peripheral nodes, where the baseline method PA tends to perform poorly. Furthermore, our evaluation demonstrates that focusing on more recent contribution data significantly enhances GraphCP's performance, highlighting the importance of temporal dynamics in OSS collaboration. While GraphCP already shows promising results, further incorporating semantic data, such as source code of files or issue contents, could enhance the model's expressiveness. This would bridge the gap between our technical proof of concept and practical applications of link prediction, such as contribution recommendation or risk assessment systems.

1.1 Overview

The remainder of this thesis is structured as follows: Chapter 2 provides the necessary background knowledge, introducing the fundamental concepts of graphs in computer science, complex graph structures, and the nature of developer networks. The chapter further establishes a foundation in deep learning to subsequently introduce graph representation learning as a method to transform graph-structured data for traditional Machine Learning (ML) tasks. Chapter 3 focuses on research on developer networks and proposing approaches for complex graph embeddings. In Chapter 4, we explore different perspectives and application scenarios for link prediction models within developer networks. Chapter 5 details our methodology, providing a comprehensive overview of the dataset collection and the subsequent classification pipeline designed to answer our research question. Furthermore, it outlines the experimental setup, presents a detailed description of the selected open-source projects, and the evaluation metrics used to assess GraphCP. The experimental results are summarized, discussed, and qualitatively analyzed in Chapter 6. Finally, Chapter 7 concludes the thesis by summarizing the key findings.

2

Background

In this thesis, we want to apply a deep learning approach on developer networks to develop a model that predicts links between node pairs present in the network. This chapter provides essential background information that is required to ease a deeper understanding of the main topic. This includes the concept of graphs and developer networks, but also a brief description of deep learning and deep learning in the context of graphs.

2.1 Foundation of Graphs

Graphs are mathematical structures that model pairwise relations between objects. A simple graph is a collection of vertices that are connected by edges [47].

It is formally defined as a tuple $G = (V, E)$, where $V = \{v_1, ..., v_N\}$ is a set of $N = |V|$ nodes, and $E = \{e_1, ..., e_M\}$ is a set of $M = |E|$ edges.

In directed graphs, an edge $e \in E$ is directed from one node v_i to another node v_j. This can be represented as (v_i, v_j). In undirected graphs, the order of the tuple elements does not matter. In both scenarios, we say that v_i and v_j are adjacent.

In graph-theoretical terms, adjacent nodes are also called neighbors. More specifically, we define the neighbor set of node v as follows:

$$N(v) = \{u \in V | v \neq u, \exists e \in E : e = (u, v)\}$$

Besides representing graphs by explicitly drawing the nodes and connecting adjacent nodes with lines, another common data structure is the adjacency matrix, which leverages the information of neighbor sets. This representation is more convenient, especially for large graphs [47]. Given a graph $G = (V, E)$, an adjacency matrix has $N = |V|$ rows and columns and is formally defined as $A \in \{0, 1\}^{NxN}$ [35]. Its entries reflect the existence of an edge between two node pairs v_i and v_j. That is:

$$A_{i,j} = \begin{cases} 0 & \text{if } v_i \text{ and } v_j \text{ are not adjacent} \\ 1 & \text{if } v_i \text{ and } v_j \text{ are adjacent} \end{cases}$$

2.1.1 Complex Graphs

In the earlier section, we introduced simple graphs that can also be described as homogeneous. This term is derived from the property that the graph only has a single type of nodes

and edges. However, real-world applications cannot be represented by simple homogeneous graphs but require more advanced representations [35]. For example, a graph that illustrates an academic network describes entities, i.e., nodes, including authors, papers, or venues. The connections between those entities can describe citation relations or authorship relations [35]. Thus, the graph has different node and edge types and is not considered homogeneous but rather heterogeneous.

2.1.2 Heterogeneous Graphs

For heterogeneous graphs, the nodes and edges are additionally associated with a type. T_n and T_e indicate the sets of node and edge types, and the mapping functions $\phi_n : V \rightarrow T_n$ and $\phi_e : E \rightarrow T_e$ assign a type to each node and edge in the graph, respectively. In our case, we will work on a more specialized type of heterogeneous graphs, i.e., a knowledge graph, which will be described next.

2.1.3 Knowledge Graphs

A knowledge graph's primary purpose is to structure relevant knowledge to a specific domain and is formally defined as $G = (V, E, R)$, where R denotes a set of relations. The nodes and edges are of different types, and the edges additionally describe different relations between two entities within a specific domain. Specifically, an edge $e \in E$ that connects two nodes $v_i, v_j \in V$ can be represented as a triplet (v_i, r, v_j), where $r \in R$.

For example, in an e-commerce knowledge graph, the nodes can represent entities such as customers, products, categories, and orders. The links between two nodes include relations such as `placed_order` between a customer and an order entity or `type_of` between product and category entities [48].

As we can see, graphs are essential representations for real-world data [35], including developer networks that capture the collaborations of software projects.

2.2 Developer Networks

The evolution and collaboration of software projects can be represented in developer networks. They are able to illustrate the relationships and interactions between different developers and their activities in projects [5]. A general developer network of one specific project is modeled as a simple graph $G = (V, E)$, where V is the set of all developers that contributed to the project and the edges model different relationships between the developers. These relationships can be of different types [5]:

- **cochange:** Linked developers have both edited a common source-code artifact (e.g., file or function)
- **mail:** Linked developers sent an e-mail to the same e-mail thread
- **issue:** Linked developers contributed to the same issue (e.g., commenting or reviewing)

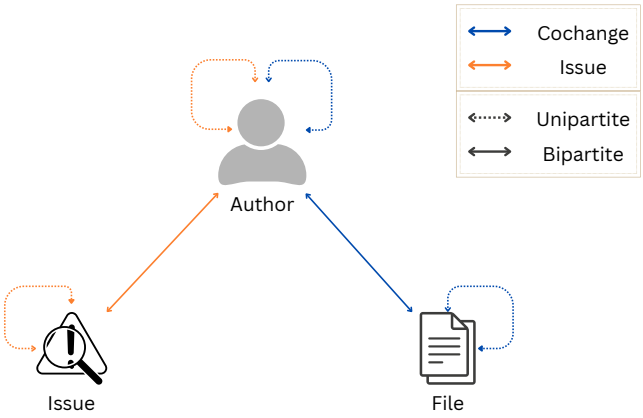


Figure 2.1: Schema Network of our Graph Data

Based on these networks, one can apply different tasks, such as finding core developers who have contributed the most to maintaining the project or, more generally, understanding the coordination of developers within one software project.

In this thesis, we assume that we are given a combined network, consisting of *cochange* and *issue* edges. This kind of developer network is considered a knowledge graph, as it has different types of nodes and edges, and the edges describe different relations between two entities. In our case, the nodes represent either authors or artifacts. The artifacts are again sub-grouped into files and issues. Edges represent a cochange - between authors and authors or authors and files - or issue relations between issues and issues or authors and issues. Our graph data is also illustrated in figure 2.1, which serves as an abstraction of all entities and their connections.

2.3 ML: An Overview

Artifical Intelligence (AI), Machine Learning (ML), and Deep Learning (DL) are terms that are often used interchangeably, yet they are by definition not the same. AI is a broad field within computer science. It aims to create algorithms that are able to perform tasks that usually require human intelligence [36].

ML is a subfield of AI. Without explicitly programming but with a large amount of data, ML uses algorithms that analyze and learn from the data to make predictions, e.g., about unknown variables or future outcomes [36]. A common example is the spam filter for emails. In an AI approach, a developer writes a program in which he or she sets predefined conditions for what is considered spam. Incoming emails are then automatically filtered by the program. Different from this, in the ML approach, a model is fed with spam and non-spam mail and learns its own rules or conditions for classification.

In general, data serves as the foundation of ML models, enabling them to recognize patterns, classify information, and generate predictions. It can appear in different formats. Common types are structured data in tabular format, or unstructured text or image data. The latter often requires preprocessing steps to be a suitable input for ML methods. Depending on the data and the task, users have to decide which type of ML approach fits best. These

categories include supervised learning and unsupervised learning [36]. In supervised settings, the model is trained with labeled data, i.e., the input is mapped to a true value, aiming for the model to learn the underlying relationship between input and output. Referring to our example, emails are either labeled as spam or not spam. Common models are linear regression, logistic regression, or neural networks. These techniques rely on the assumption that the data are independently and identically distributed (i.i.d.). This is essential to guarantee generalization on new and unknown data [35]. In unsupervised settings, the data is unlabeled and the true target values are unknown. The algorithm uses the data to identify patterns in order to cluster it or find a representative version in a new dimension [36].

DL is an advancement of ML and employs a specific form of neural networks, namely Deep Neural Network (DNN). These models are intended to imitate the human brain and are built as a consecutive series of parameter layers. Unlike ML, which requires domain experts to manually select and fine-tune relevant input features, these models can understand complex patterns in the data and automatically learn important features [36].

2.3.1 Deep Neural Networks

Formally, a DNN aims to approximate a function f^* that explains the underlying relationship between the input $\boldsymbol{x}$ and its corresponding output y. For example, for a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y [17]. Figure 2.2 shows a simple example of a network

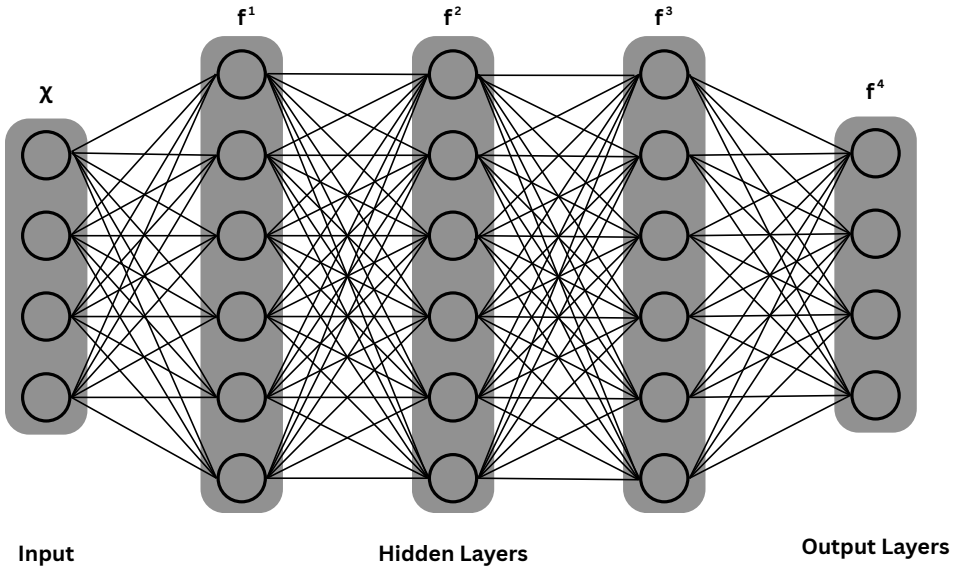


Figure 2.2: An illustrative example of feedforward networks [35]

with four functions f^1, f^2, f^3, f^4, each representing one layer, connected in a chain, such that $f(\boldsymbol{x}) = f^4(f^3(f^2(f^1(\boldsymbol{x}))))$. The number of layers indicates the depth of the network, and within each layer, there are multiple units [35]. The actual computation happens neuron-wise. Each function unit retrieves elements from the previous layer. During one operation, the elements, each having individual weights, are linearly combined. Next, they

are passed through an activation function to introduce non-linearity, which improves their approximation capability. Mathematically, the computation of one unit for the first layer can be represented as:

$$h = \alpha(b + \sum_{i=1}^{n} \boldsymbol{w_i} \cdot \boldsymbol{x_i}),$$

where n denotes the length of the input vector $\boldsymbol{x}$, b is a bias term, and $\alpha()$ is an activation function. Assume that in the k-th layer of the neural network, we have N^k nodes and the layer's output can be represented as $\boldsymbol{h}^k$, with $\boldsymbol{h}_i^k$ denoting its i-th element. Then, we can generalize the operation to compute $\boldsymbol{h}_j^{k+1}$ in the $(k+1)$-th layer as follows [35]:

$$\boldsymbol{h}_j^{k+1} = \alpha(\boldsymbol{b}_j^k + \sum_{i=1}^{N^k} \boldsymbol{W}_{ji}^k \boldsymbol{h}_i^k).$$

Here, $\boldsymbol{W}_{ji}^k$ denotes the weight corresponding to the connection between $\boldsymbol{h}_i^k$ and $\boldsymbol{h}_j^{k+1}$.

During the training process, we optimize $f(\boldsymbol{x})$, such that it approximates the ideal function, i.e., $f^*(\boldsymbol{x})$. This includes adjusting the weights such that an appropriate loss function is minimized. The loss function should capture the discrepancy between the true target value and the estimated output from our network. Depending on the tasks, the loss functions can vary. For example, in binary tasks, where a sample $\boldsymbol{x}$ is either 0 or 1, the output $\hat{y}$ of the neural network should be a single value indicating the probability of the sample being predicted as label 1. The most common loss in that case is the binary cross-entropy loss that measures the difference between the ground truth and the prediction as follows [35]:

$$BCE = -\frac{1}{N} \sum_{i=1}^{N} [y_i \cdot log(\hat{y}_i) - (1 - y_i) \cdot log(1 - \hat{y}_i)], \tag{2.1}$$

where N is the number of observations, y_i is the actual binary label (0 or 1) for i^{th} observation and $\hat{y}_i$ is the predicted probability of the i^{th} observation being in class 1. Although there exist several predefined loss functions for standard tasks, they can also be customized for individual purposes.

To minimize the loss, a general approach, called gradient descent, is to apply an optimization algorithm that updates the parameters by taking a step towards the direction of the negative gradient [35]. The Backpropagation algorithm is an essential part and provides an efficient way to realize this computation. It encompasses two phases [35]:

1. Forward Phase: The input is passed through the layers to obtain an output with the current set of weights. Given the output, the loss function is evaluated.

2. Backward Phase: Calculate the gradients of the loss function with respect to the parameters (weights), starting from the last layer and moving backwards to the first layer.

The introduced concepts are just fundamental basics of neural networks. Based on these concepts, a lot of research innovated more techniques to harness their full potential, including convolutional neural networks, recurrent neural networks, or autoencoders. Ultimately, the potential of neural networks lies in their ability to perform complex calculations that would be impossible for a human to achieve. As we will work on complex, unstructured

real-world data, i.e., developer networks, we will approach our task using advanced DL-based models. Typical ML techniques are constructed assuming easier data structures, such as tabular or image data. However, our data structure is more complex and requires a certain transformation to be useful in ML. One way to achieve this transformation will be explained in the following section.

2.4 Graph Representation Learning

The nodes in graphs that represent real-world applications are inherently connected, implying dependencies between them. However, this contradicts traditional ML techniques, which often assume i.i.d. data [35]. Consequently, we need a new approach to apply computational tasks on graphs. The most popular technique creates a flattened representation of the graph to further apply traditional ML methods, having the graph representation vector as input. The method, also known as graph representation learning or graph embedding, automatically learns node features with minimal human intervention. Unlike feature selection, which removes irrelevant and redundant node features, representation learning generates a set of new node features preserving key information of the node in the original graph [35]. As a result, one can map nodes from the original graph domain to a corresponding embedding in the embedding domain. The learned features facilitate downstream tasks, including link-prediction.

As DNNs have achieved major successes in the last two decades, researchers began to transfer and generalize DNNs to graphs in the form of GNNs, to realize graph representation learning [35]. GNN-based techniques can be divided into spatial and spectral approaches [35]. Spectral approaches take advantage of the spectral view of graphs using the Fourier graph transform and the inverse graph Fourier transform [7]. Spatial approaches leverage the input node features but also the graph structure, that is, the information from locally close neighbors [35].

Depending on the provided dataset, graph representation learning can also be differently categorized. Datasets that consist of a set of graphs where one graph is considered one datapoint are usually utilized for graph-focused tasks, such as graph classification or graph generation [35]. On the other hand, as in our case, if the entire data is reflected by one graph and the nodes correspond to datapoints, we operate in the node-focused tasks domain. This includes node classification or link prediction.

2.4.1 GNN for Node-focused Tasks

A general GNN framework for node-focused tasks typically consists of an alternating sequence of graph filtering operations and non-linear activation layers [35]. The output of the last layer is the new set of feature nodes, i.e., the node embedding, that can be leveraged as input for the downstream node-focused task. The simplified Cheby filter (GCN-filter) is a very common spatial-based filter. In its process, each node aggregates information from its 1-hop neighbors. This process is conceptually derived from the spectral-based Cheby filter, which leverages k-hop neighbors when creating a new node feature. Similar

to GCN-filters, there also exist GAT-filters. These filters build graph attention networks (GAT) using self-attention mechanisms. While the aggregation in GCN-filters is based on the graph structure, GAT-filters first attend to all its neighbors to generate an importance score for each neighbor [35]. These scores are then used in the aggregation process. Both mentioned filters always consider the entire 1-hop neighborhood of nodes; thus, they use the entire adjacency matrix, unable to generalize to unseen nodes.

2.4.2 GraphSAGE

The key idea of GCN is message passing. That is, each node updates its feature representation by aggregating the information from all its neighbors. While this is a transductive approach, there also exists another, very popular graph filter known as GraphSAGE (SAmple and aggreGatE) [19]. Instead of using the entire neighborhood sets, GraphSAGE inductively tries to generate node embeddings by uniformly sampling a fixed-size set from a node's local neighborhood $\mathcal{N}(v)$ and aggregating the features [19]. By adding additional layers, one can include nodes from the multi-hop neighborhood, i.e., the second layer aggregates over the features from the two-hop neighborhood. By only considering a subset of the total neighborhood, GraphSAGE is able to generalize to unseen nodes and is more scalable compared to other approaches, including GCN.

2.4.2.1 *Forward Propagation*

As GraphSAGE is a GNN-based algorithm, it encompasses the two stages of backpropagation as well. During the forward pass, the algorithm samples a subset of the neighborhood of one node and aggregates the features to create a new embedding of the current node. More specifically, for a single node v_i, the process to generate its new features can be formulated as follows [35]:

$$\mathcal{N}_S(v_i) = \text{SAMPLE}(\mathcal{N}(v_i), S) \tag{2.2}$$

$$\boldsymbol{f}'_{\mathcal{N}_S(v_i)} = \text{AGGREGATE}(\{\boldsymbol{F}_j \mid \forall v_j \in \mathcal{N}_S(v_i)\}) \tag{2.3}$$

$$\boldsymbol{F}'_i = \sigma\left([\boldsymbol{F}_i, \boldsymbol{f}'_{\mathcal{N}_S(v_i)}]\,\boldsymbol{\Theta}\right) \tag{2.4}$$

Here, SAMPLE() is a function which samples S nodes from the nodes neighborhood set $\mathcal{N}(v_i)$. AGGREGATE() combines the information from the sampled neighboring nodes, and $[\cdot,\cdot]$ is the operation that concatenates the old features of node v_i with the new features from the aggregation function. The last function that is applied, i.e., σ is an activation function that adds non-linearity. This process can be expanded to deeper levels, i.e., the multi-hop neighborhood. Then, for each depth layer k we have a separate aggregation function and separate weights. In general, the process would be defined as presented in Algorithm 1. First, an initial feature vector for each node in the graph is created. For each layer k, new node features are computed based on the procedure described in Equations 2.2, 2.3, and 2.4. Specifically, the k-hop neighborhood $\mathcal{N}^k$ is considered and the layer-specific aggregation function AGGREGATE_k is applied. The resulting node feature vectors are then assigned to z.

Algorithm 1 GraphSAGE forward propagation [19, 35]

$F_i^0 \leftarrow x_i,\ \forall i \in V$
for $k \in K$ **do**
  for $v_i \in V$ **do**
    $\mathcal{N}_S^k(v_i) = \text{SAMPLE}(\mathcal{N}^k(v_i), S)$
    $\boldsymbol{f}^{k'}_{\mathcal{N}_S^k(v_i)} = \text{AGGREGATE}_k(\{\boldsymbol{F}_j^{k-1} \mid \forall j \in \mathcal{N}_S^k(v_i)\})$
    $\boldsymbol{F}_i^k = \sigma\left([\boldsymbol{F}_i^{k-1}, \boldsymbol{f}^{k'}_{\mathcal{N}_S^k(v_i)}]\,\boldsymbol{\Theta}^k\right)$
  end for
end for
$z_i \leftarrow F_i^K,\ \forall i \in V$

Hamilton et al. [19] introduced various suitable AGGREGATE() functions:

- **Mean aggregator.** Taking the element-wise mean of the neighboring vectors $\{\boldsymbol{F}_j \mid \forall v_j \in \mathcal{N}_S(v_i)\}$. This is very similar to GCN, however GCN treats the current node v_i equally as its neighbors, consequently it is part of the aggregation process [35]
- **LSTM aggregator.** This function treats $\mathcal{N}(v_i)$ as a sequence and processes it using the long short-term memory (LSTM) architecture. As there is no natural order among the neighbors in [19] they adopted a random order.
- **Pooling operator.** The input features are passed through a layer of a neural network, including a non-linear activation function. The output is then summarized using the max pooling operation [19].

As the mean aggregator is the standard choice in many GNN architectures, we use this as our aggregation function as well.

2.4.2.2 *Backward Propagation*

In order to learn useful node representations, the parameters $\boldsymbol{\Theta}^k$ for each layer $k \in K$ need to be tuned via a gradient descent-based approach during the backward phase [19]. As we have no target node representation that we aim to achieve, the objective of training a GraphSAGE embedding model is to minimize a loss in a self-supervised setting. By sampling node pairs that are not connected in the original graph, we can assign them to negative labels and the true node pairs to positive labels. According to Hamilton et al. [19], the unsupervised loss is then defined as follows:

$$J_G(z_u) = -\log\sigma(z_u^\top z_v) - Q\,\mathbb{E}_{v_n \sim P_n(v)}\left[\log\sigma(-z_u^\top z_{v_n})\right] \tag{2.5}$$

Here, $u, v \in V$ are neighboring nodes and z_u, z_v are their corresponding output representations, σ is the sigmoid function, $P_n(v)$ is a negative sampling distribution, and Q defines the number of negative samples [19]. This way, the loss function ensures to maximize the similarity between nodes that co-occur together and minimize the similarity between distant nodes [19].

3 Related Work

As we want to deploy a model that is able to predict links in a developer network, it is important to review related work. Our approach encompasses two main areas: developer networks and deep learning on graphs. Thus, we address both topics individually, as well as in combination.

3.1 Developer Networks

The research on developer networks started at the beginning of the 2000s. Depending on the source of data used to mine networks and the objectives, e.g., gaining knowledge or using them for classification tasks, there exist various approaches. Most of this work primarily focused on applying analytical techniques to gain more insights into the organizational structure behind software projects.

3.1.1 Graph Understanding and Insights

López-Fernández et al. [33] use a rather descriptive approach and apply a social network analysis for characterizing software projects, their evolution over time, and their internal structure. In their case, they build two networks per project using the commit information of the code versioning repository system. One is the commiter network, where vertices correspond to developers, and edges reveal whether they contributed to at least one common module. The other is the module network, where the nodes are software modules of the project, which are linked when at least one committer has contributed to both of them. For their analysis, they consider common property measurements of the network topology, such as node degree, weighted node degree, closeness centrality, betweenness centrality, or clustering coefficients of nodes. One of their findings reveals that only a handful of developers have a direct relationship with more than 200 other developers, i.e., they are likely to be core developers. Additionally, by visualizing the distribution of the connection degree of four time snapshots of one module network, they find a significant growth in the connection degree of the most connected module, from early to later stages of the project.

Sureka et al. [50] also utilize social network analysis, but with the intention to investigate risks and vulnerabilities in the network. As OSS project settings are unstructured, it is important to have an overview of critical employees, core team, or subject matter experts, and perform risk analysis to gain insights [50]. Consequently, they develop tools and techniques to support the performance of risk, threat, and vulnerability analysis with respect

to a software development team. In general, they investigate different patterns, cohesive subgroups and clusters, or centralities. They mine a defect tracking system, specifically the bug reports from Mozilla Firefox, to create developer networks. The nodes represent developers, and these nodes are linked by an edge if they collaborated with each other in resolving a bug. With their tools, one can answer important questions like: "who are the developers working on critical bugs", "are there developers working on a variety of bug severities and are there developers working only on bug reports belonging to a certain bug severity", or "who is the most connected developer" [50].

Joblin et al. [25] provide an analytical approach using network-based operationalizations to distinguish between core and peripheral developers. In particular, they find that commonly used count-based operationalizations do not suffice for that task. Count-based metrics include the commit count as the number of commits a developer has authored or the lines of code (LOC) count as the sum of added and deleted lines of code a developer has authored. According to prior research, core developers should achieve a higher commit count and a higher LOC count than peripheral developers. Joblin et al. use fine-grained developer networks capturing data from mailing lists and VCSs. They analyze projects using overlapping analysis time windows of three months. That is, each network reflects the project for three months. On these networks, Joblin et al. compute network-based metrics, including degree centrality, eigenvector centrality, or hierarchy, expecting core developers to be placed in the upper region of the hierarchy, to study developer roles. In an online survey, they asked developers to report the roles of developers in their project according to their perception, to obtain a ground-truth classification. In an empirical study of ten OSS projects, Joblin et al. [25] reveal that their network perspective offers more valuable insights regarding developer roles than traditional count-based operationalizations.

Another approach by Yan et al. [56] uses network analysis in order to profile the expertise of developers. As previous research only considered a single community in which the developer was active or simply summed up the expertise in individual communities, they want to profile the expertise across multiple communities. After collecting, filtering, and preprocessing data from Stack Overflow and GitHub, they construct a heterogeneous information network [56]. This network comprises four node types: developers, ability terms, questions, and projects. Ability terms include tags from Stack Overflow and topics in GitHub. The questions are selected from Stack Overflow and annotated with suitable ability terms. Similarly, the projects that are collected from GitHub are also assigned to ability terms. Depending on the node pairs, there exist nine different edge types [56]. This includes, among others, a follow connection between developers, a commit edge from developer to project, or an answered connection from question to developer. The edges are additionally weighted by scores that indicate the intensity of knowledge that was propagated, e.g., by committing to a project or a question that is answered. To evaluate the developer expertise, a method called random walk with restart (RWR) is adopted. This method iteratively estimates the proximity between developer nodes and ability term nodes, and the outcome should reflect the developer expertise. As there is no ground truth data indicating developer expertise, Yan et al. [56] estimate the true expertise by the answer scores of developers, assuming the higher the developer's answer scores, the higher their expertise [56]. To evaluate the effectiveness of their approach, they compare it to a related one, called CPDScorer [21], and reveal that the novel approach performs significantly better.

While the CPDScorer has an accuracy of 55.57%, the proposed method by Yan et al. [56] achieves an accuracy of 67.88%.

3.1.2 Graphs for Downstream Tasks

The approaches so far, focus on different analytical techniques to gain insights on developer social networks. However, other researches leverage developer networks for prediction or classification tasks.

Meneely et al. [39], Wolf et al. [55], and Joblin et al. [24] utilize developer networks to find early indicators of future project success or failure. All three approaches initially adopt manual feature engineering. That is, they use network analysis techniques to calculate node-specific metrics. By combining these metrics per node, they create a feature vector. These vectors are then used as input for the predictive models. Meenely et al. [39] first calculate metrics such as different degrees, closeness, and betweenness centralities of nodes and then apply the negative binomial regression that resulted from a model selection process. Wolf et al. [55] use a similar approach but train a Bayesian classifier. Out of these three approaches, Joblin et al. [24] differ the most as they additionally aggregate the newly created node features to retrieve a graph-level vector representation and then train a logistic regression model. As they are all differently motivated, use different networks, models, and evaluation metrics, it is unclear which approach performs best.

Only recently, researchers began to use more advanced ML and DL techniques on developer networks. As defect-prone changes are a serious problem in software development and lead to software faults, Bryan et al. [8] address this issue by leveraging contribution graphs to improve just-in-time (JIT) defect prediction. The contribution graph is similar to our graph, i.e., an undirected bipartite graph consisting of developer and file nodes. These nodes are connected if the developer committed a change to the file. The edges are additionally labeled by an algorithm, called SZZ [45], indicating whether the change is defect-prone or not. The contribution graph is further projected into a one-mode projection graph. Here, the nodes only consist of developers, who are linked together if they made changes to the same file. From there on, they derive graph structural properties in the contribution graph by extracting centrality metrics, including degree, betweenness, or closeness, from the nodes in the one-mode projection graph. These features are used for their first setting. For the second setting, they further identify communities and assign group identifiers to each node in the developer-only graph using an algorithm that is not further detailed in their work. Additionally, they apply node2vec to create graph node embeddings. For each setting, they use three widely used classifiers for JIT prediction and compare them with each other, but also with the state-of-the-art model [8]. They test the proposed method on 14 well-known, open-source projects, including ActiveMQ, Camel, or OpenJPA, focusing only on the middle part of the time axis, i.e., the period with the highest development activity [8]. They find that both settings produce similar results in terms of classification results and that the relative performance increases by 152% for the F1 score. While the baseline model reaches only 0.33, the best models in each setting achieve approximately 0.77 as F1 score. Although their work shows promising results, they mention examining the effectiveness using inductive embedding frameworks, such as GraphSAGE,

in the future [8]. Just like us, they want to efficiently generate node embeddings with the ability to also generate them from previously unseen graph data.

All these approaches utilize developer networks. However, they mainly compute characteristic features manually. The next section primarily focuss on research that adopts deep graph representation learning on complex graphs to automatically generate features. Yet, it is important to note that the graphs are not necessarily developer networks.

3.2 Graph Representation Learning

Representation learning on complex graphs has seen significant developments in recent years. Most of the revolutionary GNN approaches were developed using homogeneous graphs. However, real-life applications and especially our case, are not homogeneous but rather heterogeneous. The graph nodes and edges are of different types and have different attributes. Moreover, our graph captures the interaction and collaboration of an OSS project over a longer time span. As a result, our data are more complex and specified to a specific application domain, which limits the availability of comparable studies. Yet, there still exist related studies that offer valuable insights influencing our approach.

3.2.1 Dynamic Approaches in Different Domains

Yang et al. [57] propose a novel technique to embed dynamic heterogeneous graphs using hierarchical attentions, called DyHAN. They sample different user behaviors from three different datasets with 7 to 11 time steps. This includes `click`, `collect`, `add_to_cart`, and `buy`, regarding the EComm [1] or `retweet`, `reply`, and `mention` for the Twitter dataset [2]. DyHAN encompasses three steps: node-level attention, edge-level attention, and temporal attention. Initially, for each timestep, the input graph is divided into edge-type-specific subgraphs. Then, for each subgraph, a self-attention is employed to aggregate the node embedding. As a result, there exists a node embedding for each time step and each edge type. In the second step, an attention layer aims to learn the importance of different edge types. This layer creates a fused embedding of each node at each time step. Last, all node embeddings across the time series are aggregated in the temporal attention layer, resulting in one embedding per node [57]. Similar to DyHAN, Fan et al. [14] propose a new method that integrates the spatial and temporal dependencies to create node embeddings, called heterogeneous temporal graph neural network (HTGNN). It is composed of multiple aggregation layers, intra-relation, inter-relation, and across-time aggregation. These layers are comparable to those in the previous paper. However, they reveal that DyHAN [57] processes the spatial and temporal dependencies in a serialized manner, and the performance heavily depends on the characteristics of the graph. Therefore, Fan et al. [14] deploy a graph-agnostic framework to learn node-embeddings.

Unlike [14, 57], Hu et al. [20] do not split the heterogeneous graph into subgraphs for each relation type present in the dataset. They introduce the node- and edge-type dependent

1 https://tianchi.aliyun.com/competition/entrance/231719/introduction
2 https://snap.stanford.edu/data/higgs-twitter.html

attention mechanism. By extracting all linked node pairs, their method aims to aggregate information from a source node s to obtain a contextualized representation for a target node t [20]. This process is decomposed into three steps: Heterogeneous Mutual Attention, Heterogeneous Message Passing, and Target-Specific Aggregation. Attention mechanisms in transformers usually work with query, key, and value matrices. Based on this, Hu et al. [20] map the target node t into a query vector and the source node s into a key vector. Their dot product indicates the attention. Different from the vanilla version, which uses a single set of projection weights for all cases, their version provides a distinct set of projection weights per relation. In parallel to the attention calculation, the meta relations of edges are incorporated into the message passing process, which passes information from source nodes to target nodes [20]. The last step is to aggregate information to the target node t from all its source nodes s, which were computed in the previous attention and message passing steps. Hu et al. [20] evaluate their model using the Open Academic Graph [60] dataset. This dataset contains five node types: papers, authors, institutions, venues, and fields, as well as different relations between them. For evaluation, they select different real-world downstream tasks, including a link prediction task between papers and candidate authors [20]. As baselines, they choose designs for homogeneous graphs [28] but also heterogeneous graphs [44, 59] and specifically [54], which is the static version of HTGNN [57]. Overall, they observe that on average, their method outperforms the baseline models. Although our graph data contains temporal information, it is not the primary focus of this research.

3.2.2 Software Development Domain

The previous approaches do not address knowledge graphs within the context of software development. As an example, Ruenin et al. [42] confront the challenge of selecting qualified developers to enable successful software development. According to Ruenin et al. [42], they are the first to provide this recommendation application in the software engineering domain [43]. They build a knowledge graph using data from Moodle, which is an open-source software project housed in the JIRA platform [42]. It captures different factors, including developers' collaboration, skills, or task dependencies. For the graph embedding, they compare different embedding models that use different scoring functions to guide the embedding model. The embedding model using convolutional layers as a score function, i.e. ConvKB [40], outperforms the other models and captures best the semantics of the knowledge graph [42]. Consequently, to build their recommendation algorithm, they apply the ConvKB and are able to surpass the baseline models such as RECAST [52] and Liu et al. [32].

Similar to [42], another very recent paper by Sun et al. [49] proposes a novel approach to express developers' technical expertise. Likewise to us, they use GitHub data to construct a meaningful social network that encompasses entities such as repositories, developers, PRs, and issues, as well as their social relationships. To learn technical expertise, the method is divided into four main components. As a first step, the graph is created and preprocessed by removing bot developers, noise data, and simplifying the relationship to a minimum such that only the follow relationship between developer nodes is considered [49]. The second step is to obtain the initial technical expertise representation of GitHub entities.

Code data, natural language text, and discrete data are individually embedded using GraphCodeBERT [18], xml-roberta-base [11] for entities such as repository, pull request, or issue, and one-hot encoding, respectively. Finally, each entity retrieves the concatenation of all the embeddings from its different parts. Next, the initial technical expertise representation is refined through a GNN-based training process leveraging information from the structure and attributes of the social activity graph. For this, they partition the graph into edge-type-based subgraphs and employ separate GraphSAGE [19] models within each graph. The final node embeddings result from aggregating the corresponding node representations from all subgraphs. The training process is more complex, since two learning objectives are applied: graph structure learning and graph attribute learning. However, to effectively handle both, they design a unified model architecture used for each task. Last, to showcase the developer's technical expertise and enhance collaborative efforts within the GitHub ecosystem, a series of social relationship recommendation tasks is implemented. This includes recommending developers with similar expertise or recommending repositories that align with developers' expertise. As a result of the first task, the process of finding a suitable organization is facilitated while at the same time the communication among developers is promoted [49]. The second task serves as support for new, meaningful contributions to the repositories. For training and realizing these tasks, the representations from the previous steps are leveraged. Compared to five baseline models on three GitHub social relationship recommendation tasks, the proposed method by Sun et al. [49] outperforms and achieves improvements on hit ratio@10 and F1 score.

A related study to ours by Thakrar et al. [51], also investigates GitHub data, but focuses on the Stargazers dataset [3]. This dataset contains more than 12,000 developer networks. The present developers have starred GitHub repositories and are connected by edges if they follow each other. For analysis reasons, they first preprocess the data to find basic properties such as the number of nodes and edges, as well as the average degree and density [51]. Next, they prepare the dataset for graph-based modeling and extract features, including the degree, clustering coefficient, and betweenness centrality. Besides developing a model that classifies a developer network into either web development or machine learning communities, another objective is to develop a recommendation algorithm. This recommendation system aims to predict potential node connections within the network. That is, future follower relationships between GitHub users. Therefore, they first use GraphSAGE to learn the node features and generate appropriate meaningful embeddings that are then used for the downstream task [51]. The initial 5-dimensional feature space of a node is transformed into a more expressive 100-dimensional embedding. To estimate the likelihood of connections, they compute the dot product between two node embeddings to create scores. Then they apply a margin loss function that maximizes the score of node pairs with existing edges and minimizes the score of node pairs with non-existent edges. Their model shows promising results considering the **a**rea **u**nder the **c**urve (AUC) scores as a primary metric. However, as there are still misclassifications, they suggest to further refine the approach by applying a more sophisticated negative sampling strategy or incorporating temporal evolution [51]. This research is similar to ours, however, we use more complex data that represents the collaboration of developers with artifacts within one project. That is, we have a heterogeneous graph capturing interactions over a longer period of time.

3 https://snap.stanford.edu/data/github_stargazers.html

While Thakrar et al. [51] can find potential new follower relations between developers, our approach aims to find new contributions between developers and files. Additionally, we choose a different prediction head coupled with a distinct evaluation strategy. This is further explained in Chapter 5.

A very recent study by Luo et al. [34] proposes a method to detect code vulnerabilities. They represent source code in an Inter-Procedural Abstract Graph (IPAG) and train a heterogeneous attention GNN (HAGNN) on it. The initial graph is split into multiple subgraphs capturing the different features of the source code. On a high-level perspective, they create new node feature vectors that can be used for vulnerability detection. As a first step, the graph is sliced into subgraphs and the node names and edges are encoded into a numeral format [34]. To create new node vectors, they build as many message passing units as there are edge types. Each message passing is based on the GraphSAGE approach using the mean aggregation method. The resulting node features - dependent on the edge relation - are then aggregated again into their original node set types. Our approach will take a similar direction, however, for the time being, we focus only on one edge type, namely, author-file co-change relations.

In general, many studies for heterogeneous graph representation learning leverage the GraphSAGE model to learn node representation. By building edge-specific subgraphs, they circumvent the problem of heterogeneity and utilize methods that were invented using homogeneous graphs. At the end, they typically aggregate the intermediate results to create a summarized representation of the graph and its nodes.

4

Operational Applications of Link Prediction

The following chapter discusses various perspectives on applying link prediction to developer networks that capture open-source software project activities and interactions. To demonstrate the importance of a deep learning-based approach, first, we start with negative examples, i.e., cases in which we do not need a deep learning link prediction model but simple rule-based solutions would suffice. Next, we present more realistic examples, in which rule-based approaches reach their limit, and the strengths of a deep learning approach are underlined. These examples are complemented by small graph visualizations. Some of the examples that we discuss in this chapter go beyond the capabilities of our current approach. Nevertheless, they motivate the applicability of link prediction when considering different perspectives.

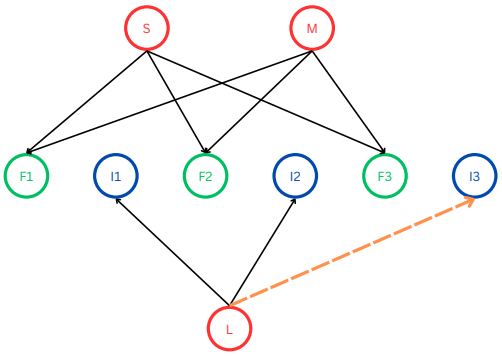


Figure 4.1: A simple graph illustrating Sara (S), Mary (M), and Lea (L) as developers. Sara and Mary always worked on the same files and Lea always handled issues regarding the files they modified. A new issue regarding file F3 came in. Based on the consistent previous activities, the rule-based approach recommends that Lea should handle that issue (dashed orange line).

4.1 Applicability of a Rule-Based Approach

In small and stable settings with limited entities that always behave the same, rule-based solutions are perfectly applicable [61]. To better understand the following example, see Figure 4.1. Sara (S in the graph) and Mary (M) always work on the same files (F1, F2, F3). In the past, Lea (L) always handled issues (I1, I2) regarding the files that Sara and Mary

modified. If a new issue corresponding to F3 is created, the rule-based approach would rely on the experience from the past and would simply recommend Lea in that case.

Another example could be a small team of ten developers in a software company. There exists one project manager who assigns work to each team member. And each member is also specified for concrete topics, such as frontend and backend implementations. Then, it is easy to create rule-based decisions to assign the next file or issue to a developer. Take a similar setting as Figure 4.1, Sara always works on the same file as Mary. Now, Mary creates a new file. A rule-based approach would recommend Sara to work on that file as well. One can even add a time variable, e.g., on Mondays and Tuesdays, Sara is handling the issues regarding the frontend, and on every other day in the week, Mary works on them.

However, graphs illustrating open-source software collaborations are larger-scale. There are hundreds of developers contributing to hundreds of files or issues in a self-organized way. Unlike in a company with a predefined hierarchical structure, in open-source software projects, structures evolve dynamically over time. Additionally, open-source software projects are characterized by dynamic changes and latent developer behavior and therefore create more complex networks [26]. Simple rule-based methods are not sufficient, as they cannot predict complex and context-sensitive characteristics. It is rather time-consuming or simply not feasible to reliably create decision thresholds or rules that recommend interactions or detect unusual interactions [61].

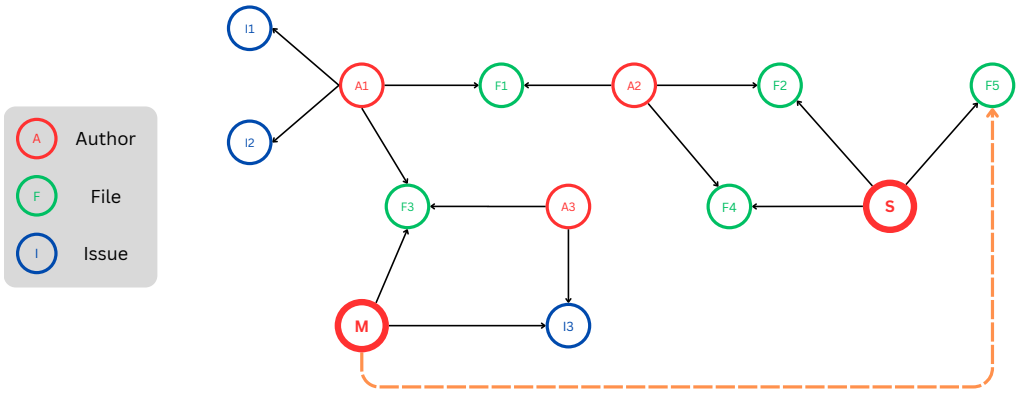
4.2 Perspectives of Link Prediction Models

Link prediction, at its core, is the prediction of missing connections or future connections between entities in a graph [29]. However, it can be interpreted as more than these simple tasks, as it can also serve as a tool for learning underlying relationships and patterns in a network. A very common use case of link prediction on social networks is a recommendation system [29]. Taking the node features and the graph topology into consideration, the model is able to recommend connections between two entities. Edge prediction can also serve as a viable tool for contextual updates by identifying the impact of new links or changes to nodes. It is aware of indirect connections and can see what is influenced by a new event [13]. A last, also very common application of link prediction, is anomaly detection [29]. A graph-based model learns the typical structure and interactions between entities from the past and is sensitive to connections that deviate significantly from the norm. In the following, we further discuss in detail the different perspectives of link prediction, by stating the problem, introducing small examples, and highlighting the strength of a deep learning approach by discussing the limitations of rule-based solutions.

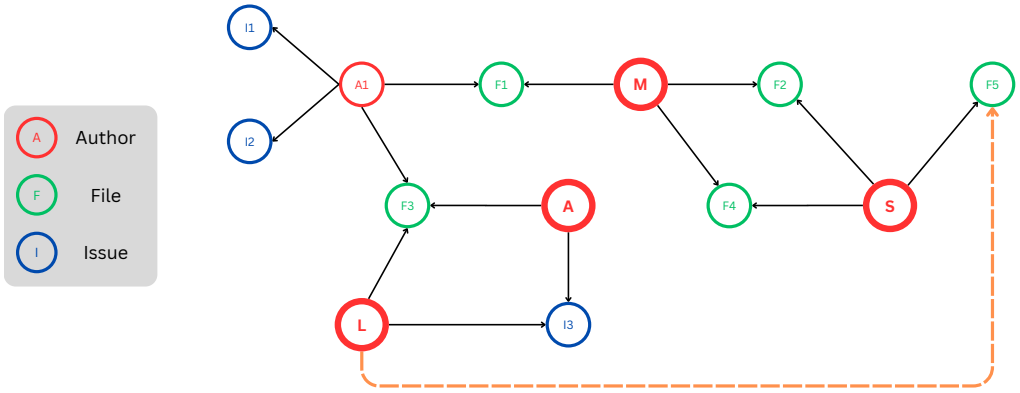
4.2.1 Recommendation

Contributions in open-source software projects are typically self-organized by developers and therefore rarely planned in advance [15]. Developers typically work independently from each other, and considering that their work is voluntary, they allocate their own time to tasks that they select, i.e., they do not have a static work behavior but are rather dynamic

in contributing to the project [15]. Consequently, it is rarely specified in advance which developer will work on which file or issue next.



(a) A small subgraph from a larger developer network illustrating interactions of different developers with several files or issues. Sara and Mary, both highlighted here, never worked on the same file or issue before, however, based on the context, the model predicts a link between Mary and the file that Sara created.



(b) A small subgraph from a larger developer network. Sara and Mary have worked on the same files before. Lea and Alexa handled the same issue and file. Both developer groups never have worked directly on the same file or issue, however a model would recommend that Lea works on Saras file next.

Figure 4.2: Two examples for "recommendation" scenario

Taking Figure 4.2a for a better understanding, imagine Sara (S), as a frontend developer with little knowledge about backend development, copies database query code from Mary (M) to an existing file (F5) on which she dominates the commit history, i.e., only she has a link to it. Now the new version introduces performance issues. The task is then to predict which developer will fix the problem that arises from the copy-paste action, which is not explicitly represented in the edges, e.g., the next developer-file interaction.

Simple rule-based decisions would probably look at who committed the most to this file, who fixed bugs in this file the most in the past, or who is the file owner. So these heuristics rely on local behavior. It is possible to include more global information of the graph, however, this would also induce significantly higher computational costs [13]. A

deep learning approach, however, could automatically combine multiple input variables, such as code content embeddings, issue resolutions in the past, or co-change patterns.

A second example illustrated on a high level in 4.2b could be as follows: Sara (S) usually created and changed files, which Mary (M) then reviewed afterwards. They mainly focused on data mining and loading from external sources. Alexa (A) focused on model training in the past. But has never worked on data mining or loading files before. A fourth contributor, Lea (L), typically reviewed Alexa's code. Now, for the first time, Sara created a file (F5) in which she trains a model on the data she also mined previously. The task of a model is then to determine which developer is most likely to review that file.

Simple rule-based methods would probably recommend Mary, as this was the pattern in the past. However, based on the semantic code similarity and the multi-hop knowledge that Lea usually reviewed files about model training, a link prediction model would probably consider Lea to work on that file next.

4.2.2 Change Impact Analysis

As mentioned before, open-source implementations and requirements are rarely defined in advance. If the implementation was planned in diagrams, it would provide substantial support for understanding components before use, for peer reviews at the design level, and for sharing guidelines within the development community [15]. Moreover, with increasing contributions to open-source projects, the network increases in connections and complexity [26]. This makes it even more difficult to obtain such substantial support. Additionally, with increasing complexity, it is also challenging to maintain an overview of the implicit effects that modifications of one file might have on other non-adjacent files [10]. These challenges motivate the application of link prediction on complex developer networks to better understand the effects of changes.

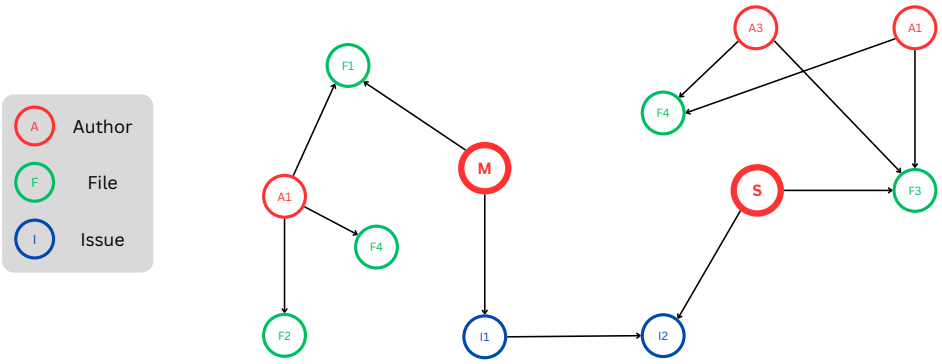


Figure 4.3: First example for the "change impact analysis" scenario. A small subgraph from a larger developer network. Mary only worked on one file and the corresponding issue to it. Sara only worked on another file and its corresponding issue. The two issues are linked and build a bridge between the two files. A change in one file might then affect the other.

Take Figure 4.3 as an illustrative example. Imagine Mary (M) worked on file F1 and on the corresponding issue I1, and Sara (S) worked on file F3 and the corresponding issue I2. Both issues are connected to each other. According to the graph, F1 and F3 are not linked directly, i.e., they have not been changed in the same commit. However, the issues

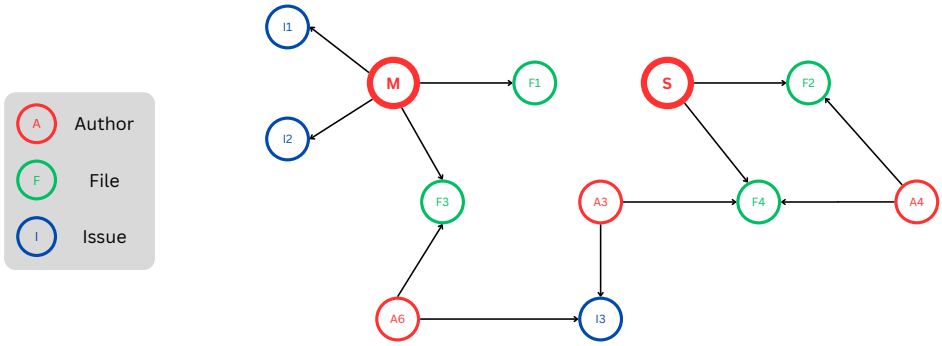


Figure 4.4: Second example for the "change impact analysis" scenario. A small subgraph from a larger developer network. Sara works on backend files, while Mary works on frontend files. Mary accesses an attribute from a returned value from the backend logic, i.e., there is a semantic dependency, even though they are not directly linked together. Now, Sara slightly changes the name of the attribute. This affects, among others, Mary's implementation.

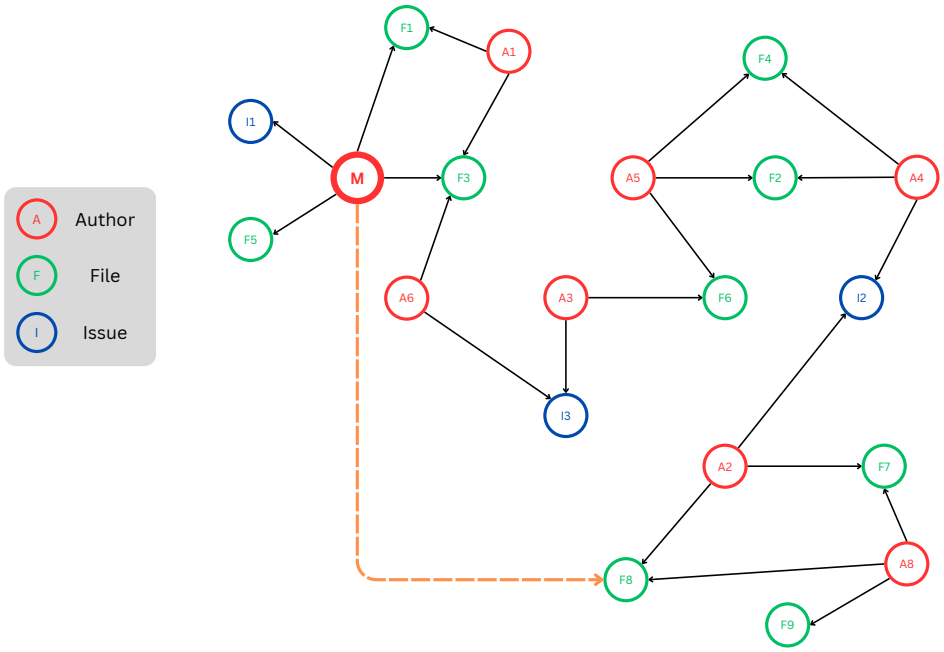
are directly linked, indicating that they belong to the same topic. As Sara and Mary are the only ones working on the issues and also solely worked on files F3 and F1, respectively, they share an implicit dependency. If Mary now changes F1, does this have an impact on the issues or on F3? A rule-based solution would probably only look at files that were committed with the modified file F1. As this is not the case for F3, the rules would not consider F3. A deep learning approach, however, could traverse the links from F1 to F3 via message passing [35], and could detect that these files share a similar structural context. So, although the files have no history of co-changes, a link prediction model might predict a potential link.

Alternatively, imagine the following example, which is also illustrated in Figure 4.4. Sara (S) is working on the backend and Mary (M) is working on the frontend code. Mary previously wrote an implementation in file F1, which queries the backend and then accesses the attribute *user_name* from the response. As a result, there exists a latent dependency between the two implementations. Sara now changed the attribute from *user_name* to *username* in file F2. This also affects Mary's implementation (F1), which is not immediately apparent. A rule-based approach mimics human reasoning and needs specific thresholds and logic that are predefined [10]. Regarding the semantic or content of files, this is not feasible, as natural language cannot be generalized or known in advance what to look for. Consequently, rules do not detect this connection. However, a deep learning approach might identify it and see which interdependent files are impacted by Sara's modification.

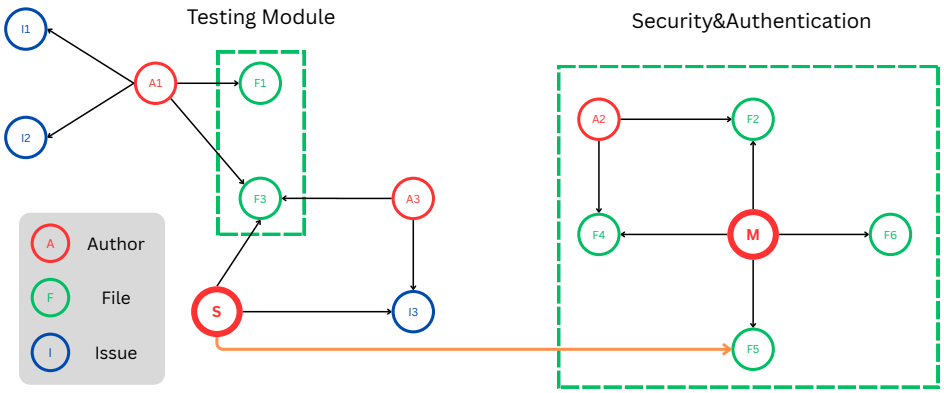
4.2.3 Anomaly Detection

Like any software, open-source software can contain vulnerable code that inexperienced developers may trigger unintentionally or attackers can exploit [37]. This may arise from a lack of support that would help, particularly with security issues. But also from the fact that issues about vulnerabilities that need to be fixed are published. This way, bad actors can immediately act on them and execute targeted attacks [37]. Besides actual vulnerabilities,

sudden interaction changes or deviations from typical patterns can hint at anomalous behavior of developers as well [12]. While it is not possible to prevent everything, one can remain vigilant and establish early warning systems that may help minimize some of the risks.



(a) A small subgraph from a larger developer network. Assume Mary is a hub node regarding one module (files 1,3,5). There exists a second module cluster (files 2,4,6) and a third one (files 7,8,9). Mary now modifies a file (F8) from a different module she typically works on. This should be flagged as anomalous behavior.



(b) A small subgraph from a larger developer network. Here, the explicit modules are marked to better understand the difference. Sara who hasn't contributed a lot so far, changes a critical file from the security and authentication module. What impact has that change?

Figure 4.5: Two examples for the "anomaly detection" scenario

For example, as illustrated in Figure 4.5a, imagine there exist three module clusters about the user interface (UI), analysis, and database (DB). Mary (M) has only worked on UI files, but suddenly modifies code in a file within the DB cluster. A link prediction model could

see the structural distance and might flag this interaction as anomalous. On the other hand, rule-based methods need to be defined in advance and would need some kind of blacklist of developers who should not work on files within this module. However, this is practically infeasible, as a human would manually have to set these rules based on some heuristic that assesses the qualification of each developer [16]. Additionally, this should be updated regularly to reflect changes over time.

Another example that not only considers structural information but also the content of files, is shown in Figure 4.5b. Imagine Sara (S) rarely contributes to the project, and if she does, she is only creating test suites. However, now, she changed a file that is beyond her experience, i.e., a config file that stores authentication credentials. Sara appends an additional personal token to the existing ones. As mentioned before, rule-based methods need to predefine a blacklist of developers who should not work on files within this module. This is again not practical, as the qualification assessment must be reviewed by a human regularly during software evolution. A deep learning approach, however, might detect that Sara is inexperienced in that area and has never worked on that file before, and it might know that the touched file is rather critical. Consequently, it detects this connection as anomalous.

4.3 Methodological Opportunities and Limitations

In the previous section, we discuss several potential link prediction applications in OSS projects. As these examples were not limited by any assumptions but were solely theoretical, they do not account for practical constraints issued by data availability or representation learning techniques. In this section, we discuss which of the examples can, in principle, be realized using GraphSAGE and our provided data. As discussed in the related work Chapter 3, GraphSAGE serves as a representative inductive graph representation learning technique, that is well-established in prior work and suitable for large-scale networks. To this end, we identify methodological limitations and motivate the design decisions that will be addressed in the subsequent chapter.

4.3.1 Capabilities and Data Availability

As we can see in the examples, a link prediction model on developer networks can be used as an approach to detect potential relations between entities (e.g., files, authors, and issues) by focusing on structural and contextual patterns from the activities observed in the past. Link prediction models can interpret the graph in a way that they are able to recognize latent structural proximity [13]. As a result, one can apply link prediction as a base method for many different applications [13].

In this thesis, as shown in Figure 2.1 and described in Chapter 2, we are given a heterogeneous multi-graph. Developers, defined as authors, are either linked to a file when they modified the file or to an issue when they contributed (e.g., comment, create, closed, etc.) to that issue. Furthermore, authors can be connected to other authors, issues can be connected to other issues, or files can be linked to other files. That is, if two authors changed the same

file or worked on the same issue, they are linked together. Or, regarding files, if both files were committed in the same commit, they are connected by an edge. Consequently, the links do not imply semantic dependency, e.g., *file a* imports a function from *file b*, but cochange dependency.

As an embedding model, we use GraphSAGE, which learns a useful representation of our network entities. A key characteristic of GraphSAGE is its capability of inductively embedding previously unknown nodes. As it does not use the entire neighborhood of each node during training, it learns how to generalize to unseen nodes [19]. This property is especially useful in our case, given that new files and issues are regularly introduced in open-source projects. Another notable property of GraphSAGE is that it learns the structure and patterns of the graph, including hops and clusters, that emerge naturally over time [19]. By incorporating and aggregating adjacent node features, the model is also able to learn even more informative representations for each node.

However, GraphSAGE is designed to operate on a homogeneous graph. [19]. It assumes a unified node type and attribute list for every entity in the network. In our thesis, nevertheless, we are provided with heterogeneous data without content information. Consequently, our approach considers author file connections to provide a form of homogeneity and relies on structural topology only.

4.4 Functional Boundaries

Table 4.1: Analysis of GraphSAGE Capabilities across Scenarios

Scenario/Problem	**Node Feature Requirements**	**Edge Requirements**
Recommendation (Figure 4.2a)	file content	author-file interaction
Recommendation (Figure 4.2b)	file content and issue discussion/content	all interactions
Impact of Change (Figure 4.3)	metainformation	all interactions
Impact of Change (Figure 4.4)	file content	author-file interactions
Anomaly Detection (Figure 4.5a)	metainformation	author-file interactions
Anomaly Detection (Figure 4.5b)	file content and module information	author-file interactions

In this subsection, we further discuss each example from this chapter in more detail. This includes stating the requirements and assessing whether our approach is capable of

meeting them. For a better overview, Table 4.1 provides a summary of the examples and their requirements at a high level.

For the first recommendation example, it would suffice to only have author-file connections, as it primarily focuses on the cochange interaction. The primary focus here is the content of the affected files. Only if we can use the content of the files, a deep learning model would be able to detect the copied database query implementation from Mary in the frontend file modified by Sara. Even if we have the file content embedded in file node features, standard GraphSAGE would expect a unified node feature set. This would only hold if author files also contain some content that can be embedded. Then, GraphSAGE would be applicable, as the embedded content could be passed on to the neighboring nodes via message passing.

The second recommendation example, both illustrated in Figure 4.2, considers all three entities, i.e., authors, files, and issues. Furthermore, it assumes that we have access to the issue discussion (natural language text) and the actual code implementation. Even if such semantic information were available, a standard graphSAGE model approach would not suffice. This approach traditionally only works for homogeneous graphs. Including a third node type, i.e., issues, and the accompanying second edge type, the embedding model could not differentiate appropriately between the entities, but rather treats each edge and node the same. As a result, the complex semantics would not be reflected in the node embeddings. GraphSAGE, however, is adaptable but would introduce more complexity, as already described in the related work Chapter 3. On a high level, adding a third node and a second edge type would require separate GraphSAGE models per edge type and then afterward an aggregation of their results.

The first example, focusing on the perspective using link prediction to see the impact of a change in a software project, requires all three entities. The addressed files were not changed within the same commit before, so they are not directly connected. The issue nodes in combination with the authors, however, are fundamentally important in this case, as they build a bridge to the two files. Consequently, the specific code or issue content is not necessary, but the network topology is sufficient for an embedding model. Nevertheless, the standard GraphSAGE implementation cannot handle all three types, as discussed before.

For the second example that shows the impact of change, both illustrated in Figures 4.3 and 4.4, it suffices to only consider the author-file connection. Yet, this example focuses on the modification of source code. Consequently, the actual implementation of a file must be taken into account. To this end, the code content should be incorporated into the node feature embedding, such that modifications are explicitly represented and propagated to other entities through message passing. This way, a deep learning approach is aware of the similarities, but also of the minimal changes after a small modification in a file.

In the last two rows in Table 4.1, we sum up the requirements and model capability for the anomaly detection examples, on a high level. For the first example, we focus only on the cochange interaction between authors and files. Thus, we can exclude the issue nodes and corresponding connections. Furthermore, we do not necessarily require any content of the files, their metadata suffices. A standard GraphSAGE can be applied here, as we only have one edge type in the network. Although we do not explicitly know the file-module affiliation, we can assume that GraphSAGE identifies subgroups or clusters which can be seen as modules. From the perspective of a link prediction model, a contribution to a file

that is structurally distant from a developer's typical working space leads to significant dissimilar embeddings. The model might therefore interpret the new link as a potential anomaly.

The last example regarding anomaly detection, both illustrated in Figure 4.5, requires at least the author-file interactions. Additionally, the data should provide information to differentiate between file types, such as test files or security-critical files. This can be achieved by including file content, but also through additional metadata, such as annotations or specific naming conventions. As we only focus on one edge type, the standard GraphSAGE approach could be applied under the condition that every node shares the same feature set.

In this chapter, we analyzed the general opportunities and limitations of applying a deep learning link prediction model, in particular using GraphSAGE as an embedding model, on developer networks under minimal constraints. This includes having enough insightful information, such as content or having different entities and their interactions. A more realistic setting and our practical opportunities, however, are accompanied by some constraints. This is further discussed in the next chapter.

5 Methodology

After presenting different perspectives of applying deep learning-based link prediction on unconstrained settings in the previous Chapter 4, we now present our approach to develop a link prediction model, which we refer to as GraphCP. Given our dataset and the standard definition of GraphSAGE, we intentionally reduce the complexity and introduce some constraints in order to provide a proof-of-concept which also serves as a controlled baseline for evaluating the applicability of our approach. In this chapter, we first present and discuss our research question that helps us evaluate our method. Next, we provide some preliminary information. This includes the data collection and a description of how to build a developer network. Afterward, we outline the methodological approach and explain the steps we take to answer the research question. At the end of this chapter, we describe the experimental setup on how to evaluate the approach.

5.1 Research Question

As we are working on OSS projects, we can retrieve our data from GitHub, which is a commonly used platform to host source code. Our goal is to utilize the emerging developer networks to learn connections between entities, maybe to even predict the possibility of future connections. As we have seen in the related work section, there exists research of applying graph representation learning techniques on different versions of developer networks. However, they are rather limited. In particular, only a few of the developer networks considered in these works stem from GitHub data. Thus, we want to explore and investigate whether a common DL graph representation learning technique is suitable to create embeddings that can be used as input for a link prediction model. More specifically, we want to answer the following question:

> **RQ 1:** To what extend can GraphSAGE in combination with a neural network be applied to perform link prediction on developer networks?

This research question examines the applicability of a standard GraphSAGE-based link prediction model, i.e., GraphCP, on developer networks. Once a pipeline is built, we further evaluate and compare the approach. To this end, we want to investigate whether our method significantly differs according to different projects, i.e., their corresponding developer networks. To answer the research question properly, we design a structured approach that provides us with representative and analyzable measures. But before doing that, it is important to familiarize oneself with the provided dataset.

5.2 Preliminaries

This section provides the necessary preliminaries required to present our approach. After discussing the concept of collecting essential information from GitHub, we describe how to build a corresponding developer network and what information it holds on a high level.

5.2.1 Data Collection

Table 5.1: Overview of all elements that are contained in a commit of a VCS.

Attribute	Description
Commit Hash	A unique identifier
Author	Name and e-mail address of the individual responsible for authoring the changes
Author Date	Date and time of day when the author wrote the code in the commit
Committer	Developer that commits the changes after checking the quality and whether specified project guidelines are satisfied
Commit Date	The timestamp when a change was committed to the software
Commit Message	A description of the change provided by the author
Lines Changed	A measurement that expresses all added and deleted lines between two revisions

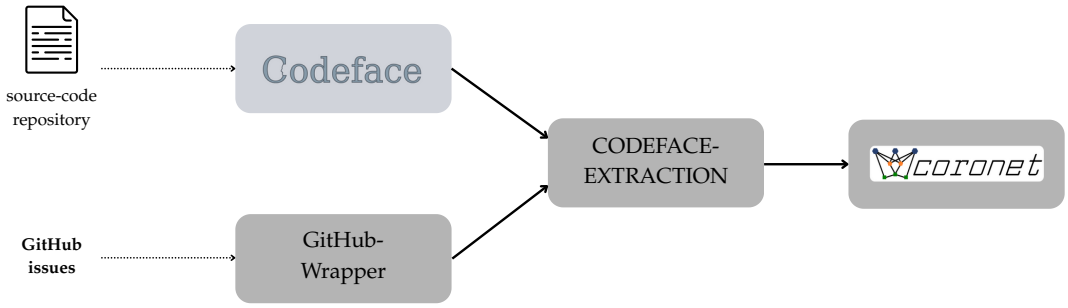


Figure 5.1: Overview of data-extraction and data-processing toolchain

To create developer networks, we use the tool Codeface [1] to extract collaboration and communication data from the VCS GitHub that hosts multiple different OSS projects [5]. The VCS allows to reconstruct the state of the software at different points in time along its development history with the help of commits [23]. A commit can be considered as a unit of change and includes elements that are presented in Table 5.1 [23]. With another tool

1 https://github.com/se-sic/codeface

called GitHubWrapper [2], we can, among other things, extract data for issue events. These data contain information on which GitHub user has triggered which event. Additionally, it documents at what point in time the corresponding issue or pull request was triggered [5]. The metadata also includes issue states (e.g., open, closed, reopened) or event names (e.g., commented, mentioned, renamed, etc.). Codeface also provides the functionality to extract email data, however, this is outside the scope of this thesis. The outputs of CodeFace and GitHubWrapper are combined and post-processed in a unified format using codeface-extraction [3].

5.2.2 Building Developer Networks

Table 5.2: Overview of shared node attributes

Attribute	Explanation	Used in our approach
id	unique identifier of node	✓
type	either author or artifact	✓
kind	either author, file, or issue	–
first activity	timestamp of first activity in project	–
last activity	timestamp of last activity in project	–

To create graph-structured data out of the information we have collected from GitHub OSS projects, we use the network library coronet [4]. coronet removes all bot-triggered events and can build various kinds of developer networks [5]. Figure 5.1 is an adapted illustration of the entire process based on [5]. At the end, we have graph-structured data, containing information about all nodes and edges. That is, author, file, and issue nodes, as well as cochange and issue links between the entities. The nodes and edges of the graph are described by individual sets of features. In general and as can be seen in Table 5.2, the nodes share a subset of attributes, such as `id`, `type`, and `kind`. The `type` attribute assigns a node to a high-level entity class of the graph, i.e., `Author` or `Artifact`, and the `kind` attribute specifies the category. That is, artifact nodes can be specified in either issue or file nodes. Furthermore, each link is described by a set of different attributes containing insightful meta information. That includes, among others, the source and target node ids, the timestamp when the action occurred, or the relation type, i.e., issue or cochange. The edge attributes are described in more detail in Table A.1. The resulting graph enables a further analysis of it, as in our case, training representative node embeddings and a link prediction model.

2 https://github.com/se-sic/GitHubWrapper
3 https://github.com/se-sic/codeface-extraction
4 https://se-sic.github.io/coronet/

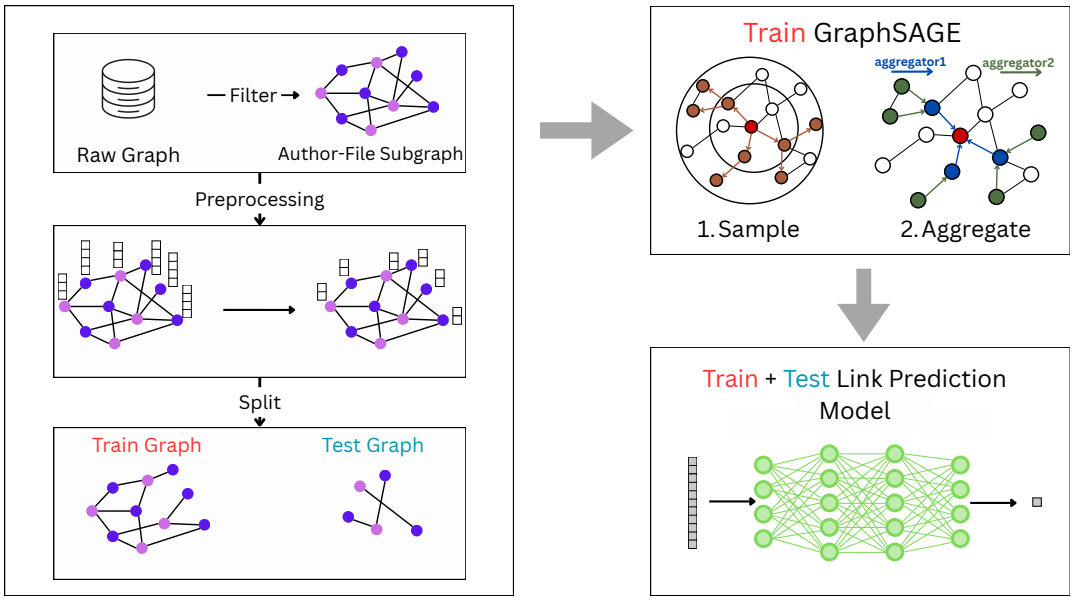


Figure 5.2: High-level process overview. The first step, i.e., data preparation, is visualized on the left hand side. As a second step, this figure illustrates the training process of GraphSAGE. Finally, we train and test the link prediction model, which is visualized in the lower-right part.

5.3 Operationalization

Many heterogeneous graph learning approaches first partition the graph into edge-specific subgraphs, apply distinct techniques separately, and then use an aggregation function to combine all intermediate results into a total embedding and a new representation. As this introduces high complexity, we isolate the cochange signal and focus only on one edge relation, i.e., the author-to-file connection. This controlled boundary allows us to apply GraphSAGE without any adjustments and answer our research question. Expanding the graph to include further entities and links would require an adaptation of GraphSAGE. For example, including issue nodes and links would mix different topologies, while GraphSAGE treats every node and edge the same. As a result, it would not capture the complex semantics reflected in the network [54].

Figure 5.2 illustrates the entire process to build GraphCP, which the following subsections explain in more detail. To answer our research question, we implement a pipeline that takes one developer network built from one project as input. Within the pipeline, the input graph is first preprocessed to obtain a suitable input format for the second step, i.e., learning the node embeddings. This is realized by training the GraphSAGE model. As we have no ground truth of how the embeddings should look, we cannot directly evaluate this model. This is implicitly done in the third step, where we leverage the model to obtain node embeddings of node pairs, which are used as input for our link prediction model. For the

link prediction model, we can self-supervise and use node pairs with existing connections, but also sample node pairs with non-existing connections to create a dataset with positive and negative links. Our model's performance is evaluated by comparing it to multiple baseline methods.

5.3.1 Data Preparation

Given the entire input graph, we first filter out the required nodes and edges to create the author-file subgraph. Here, the edge attribute `relation` is beneficial to directly filter out issue-related edges. The resulting subgraph is still a complex heterogeneous multi-graph, so we further need to preprocess the network to be suitable for the GraphSAGE model.

As the entities slightly differ, we first need to unify the attributes, encode them into a numeric format, and select the attributes for the initial feature vector. Each node contains information about the first activity or occurrence in the project. These timestamp entries seem to be a benign feature, however, they reveal a lot about the connections between authors and files. For example, the first occurrence of a file indicates when it was committed to the project by an author. And the last occurrence would already reveal the impossibility of a connection with an author who became active at a later time. This implies that the attribute reveals too much information and thereby influences our link prediction model. Thus, we decide to only consider the `id` and `type` attributes as they differentiate the nodes the best and are the intersection of both node attributes. The `kind` attribute is either always `Author` for author nodes or `File` for file nodes, consequently, it is rather meaningless, and we do not necessarily require it for our embedding.

The last step for our data preparation is the split into a training and a test graph. We sort the edges by their `date` property and take the most recent edges for our test graph. As a result, the train graph contains the majority of the earlier interactions within the observation window. For testing, the most recent edges are retained from the end of the observation time. As the original graph is a multi-graph, i.e., there exist multiple edges between the same node pair, we remove duplicate edges in both sets. Then, we remove edges in the test set that already appeared in the training set and edges that have at least one node that was not apparent in the training graph before. These preparation steps were guided by the work of Kalyani et al. [27].

5.3.2 Learn Node Embedding

The second step of our approach is to learn a new graph representation. Therefore, we leverage GraphSAGE to convert our raw input graph into a suitable embedding, which in turn serves as input for our link prediction model. This embedding should be informative enough to capture spatial and structural information about each node. GraphSAGE is a well-known approach that was already implemented and is publicly accessible [5]. Thus, we

5 https://github.com/pyg-team/pytorch_geometric/blob/4e70e5d159ef0ee0f2a5c9129b560ef227e9147f/examples/graph_sage_unsup.py#L43

can simply use it and do not have to manually implement the architecture and training process.

We train a GraphSAGE model with preset hyperparameters and use early stopping without patience, once the loss is small enough to reduce overfitting. As we have an unsupervised setting with only knowing existing edges between node pairs, we use a predefined Neighbor Sampler Class, that, given the existing (positive) edges, also samples the same amount of non-existing (negative) edges. Additionally, regarding the unsupervised fact, the loss (Eq. 2.5) needs to ensure that the model learns to optimize the mapping, such that nearby nodes in the original network should also be close in the embedding space, whereas unconnected nodes should be far apart. This is also realized in the implementation.

5.3.3 Learn Edges

Once a GraphSAGE model is trained, we utilize it to train our link prediction model. The model is fed with a concatenation of the embeddings of a node pair and predicts whether the nodes are linked by an edge. As the inputs are less complex now, it suffices to choose a simple neural network architecture, having only two layers, which outputs one scalar. This scalar is passed to a sigmoid layer to retrieve a probability of whether the link exists or not. We use the standard binary cross-entropy loss (Eq. 2.1) to optimize our model. Again, we do not have a labeled dataset but need to self-supervise. Therefore, we sample for both the training and the test set negative edges. The model selection is performed based on test performance using a ranking-based metric. In particular, we train the link prediction model and choose the model with the highest average precision score on our test set.

5.4 Experimental Setup

To evaluate the resulting link prediction model GraphCP, we run the previously proposed pipeline, illustrated in Figure 5.2, on multiple graph datasets. To this end, we compare the performances of the link prediction models trained independently on each dataset and investigate whether our approach can be directly applied to projects that experienced similar, but also different evolutions. This includes projects of different sizes, i.e., observing projects for a longer or shorter period of time, but also containing more nodes. All the projects are compared by initially discussing their structure to clarify what may set them apart from other graphs. Then, their performances resulting from the pipeline are presented and analyzed later in Chapter 6. The following section provides an overview of the datasets considered, followed by a detailed description of the evaluation procedure, including the metrics used and the process for identifying the best combination of embedding and link prediction model.

5.4.1 Datasets

In this section we describe each dataset that we use to evaluate GraphCP. This includes descriptive statistical information such as node count, edge count, or average shortest path length of the entire graph and per graph split, as well as the observation period they capture in activities.

Table 5.3: Total Dataset Overview for Evaluation. Here, **#A** and**#F** are abbreviations for the number of author and file nodes respectively. **#UEdges** denotes the number od unique edges, after deduplicating and **APL** describes the average shortest path length in the graph.

Dataset	Observation Time	#A	#F	#Edges	#UEdges	APL
Keras	2015-2019	716	351	7 088	2 787	3.4
Atom	2011-2020	299	3 874	35 610	8 877	3.2
Deno	2018–2020	348	2 912	19 116	9 905	3.1
OpenSSL	1998-2020	418	2 509	47 350	14 899	2.7
VSCode	2015-2020	1 001	7 955	154 979	29 475	3.1
Moby	2013-2020	1 154	11 213	64 509	38 052	3.6
Nextcloud	2010-2020	669	14 938	112 084	38 588	3.5

To evaluate our approach under a realistic setting, we run the process, which is depicted in Figure 5.2, on 7 different OSS projects as heterogeneous multi-graphs. The observed activities span from their individual beginning until the end of 2020, except for one project, which ended in 2019. A primary design decision is the temporal distribution of the data. As described before, we split our graph into train and test graphs. In our case, we take the most recent 10% activities until the end of the observation window and use these edges as test graph. The previous activities are used to train the embedding and link prediction models. For the train-test split, we still use all edges, including the multi-edges. After that, we deduplicate the edges for both the train and test graphs. Furthermore, we constrain our test graph to only hold edges that the models did not see before, i.e., multi-edges from the train graph, as well as only edges with nodes that the models know. This results, as illustrated in Table 5.4, to significantly smaller test sets. To ensure a statistically significant evaluation, we sample 10 times more negative samples than positive ones. This ensures that the test set is large enough to robustly measure the model's discriminative power.

The described networks capture the OSS projects Keras [6], Atom [7], Deno [8], OpenSSL [9], VSCode [10], Moby [11], and Nextcloud [12]. Besides Atom, which was deprecated as the community involvement declined significantly [46], every other project is still active to this day. Based on Table 5.3, one can say that we cover a wide range of network topologies. This

6 https://keras.io
7 https://atom-editor.cc
8 https://docs.deno.com/runtime/
9 https://www.openssl.org/
10 https://code.visualstudio.com
11 https://mobyproject.org/
12 https://nextcloud.com/

Table 5.4: Train and test graph overview per dataset for evaluation. Here **#A** and **#F** are abbreviations for the number of author and file nodes, respectively. **APL** denotes the average shortest path length in the graph.

Dataset	Split	#A	#F	#Edges	APL
Keras	Train	638	319	2 487	3.3
	Test	103	151	101	–
Atom	Train	235	3 594	7 677	3.1
	Test	86	497	215	–
Deno	Train	298	2 608	9 000	3
	Test	85	672	269	–
OpenSSL	Train	370	2 117	12 835	2.6
	Test	88	1 422	1 176	–
VSCode	Train	884	7 408	26 877	3.1
	Test	177	2 618	1 289	–
Moby	Train	1 073	10 315	34 505	3.7
	Test	121	3 153	1 839	–
Nextcloud	Train	590	14 060	33 115	3.5
	Test	111	3 653	3 696	–

diversity allows us to properly examine the generalizability of our approach to different project structures. The largest networks, in terms of node and unique edge size, are Moby and Nextcloud. Despite having the longest observation window, OpenSSL surprisingly involves only 418 contributors and contains just 2 509 files, so 2 927 nodes in total. Additionally, it has the smallest average shortest path length of 2.7, i.e., to reach one node from another, it only takes 2 to 3 hops on average. This indicates that the graph is very centralized and might show a hub-dominated structure where nodes are highly connected to many other nodes. This characteristic poses a challenge for link prediction, as the nodes share the same neighborhood and might be very similar in terms of embeddings. Consequently, it is hard to identify negative edges from actual edges. The shortest average path length of all other networks ranges from 3 to 4. Compared to OpenSSL, they are more sparse but still very strongly connected.

Regarding the edges, one can say that each model contains significantly fewer unique edges than total (multi-)edges, as repeated interactions between the same node pairs lead to multiple edges between identical nodes. A higher number in multi-edges indicates a repeated collaboration between the same two entities. Deno and Moby still retain approximately 50% of the total edges after constraining them to unique edges. Keras, Nextcloud and OpenSSL all keep 30% to 40%. The projects that are pruned significantly are Atom and VSCode. The number of unique edges is only 25% and 19% of the total number of

edges, respectively. A significant decline after deduplication, e.g., retaining only 19%, presumably indicates that contributors only engaged with specific files and did not explore other components.

Besides the size and connectivity of the datasets, another important aspect is the node distribution. The smallest project Keras is the most balanced, in terms of the amount of author and file nodes. The imbalance score (defined as $\frac{|\#A-\#F|}{\#A+\#F}$) of this project is 0.34. Additionally, Keras is the only project that contains more author than file nodes. The largest project, i.e., Nextcloud, has the highest imbalance between author and file nodes. Here, the imbalance score is approximately 0.91. Considering the substantial number of edges, this might indicate that authors tend to be linked to large amount of files.

Table 5.4 provides graph-split specific descriptive statistics. That is, statistical information for both the train graph and the test graph for each dataset. The previously described filtering of test edges is clearly visible in Table 5.4. For example, Keras, Deno, and Atom are only left with 101, 269, and 215 actual positive test edges, respectively. We can identify that for all projects except for VSCode, Moby, and Nextcloud, the average shortest path lengths of the train graphs are minimally smaller than for the corresponding total graphs. This indicates that the models are trained on more compact networks, which might make it harder to learn to distinguish between valid and invalid edges, as all nodes are close by. The imbalance between author and file nodes is similar in the total graphs and their corresponding train graphs. Regarding the test graphs, the imbalance decreases for the small graphs Keras, Atom, and Deno, but increases for the larger graphs. Again, the imbalance of author and file nodes indicates a one-to-many relation, that is, authors are connected to multiple files. This potentially facilitates the link prediction task as authors naturally show a high tendency to create links to new files.

5.4.2 Evaluation Metrics

In the following, we introduce the metrics, we use to evaluate GraphCP. This includes their definition but also a justification why we select them.

To comprehensively evaluate the performance of our link prediction model, i.e., GraphCP, we use two threshold-independent metrics: the **a**rea **u**nder the **r**eceiver **o**perating characteristic **c**urve (AUROC) and the **a**rea **u**nder the **p**recision-**r**ecall curve (AUPR). This selection aligns with the recommendation of Bi et al. [4]. They reveal that some common and well-known metrics yield conflicting rankings for the same algorithms and therefore suggest combining the standard AUROC with a second metric, such as AUPR, to ensure a robust evaluation [4]. Using AUPR is also motivated by the sparsity of real-world networks. As mentioned by Yang et al. [58], in typical binary classification tasks, classes are approximately balanced. However, real-world networks are often very large and sparse [58]. That is, the number of edges is significantly less than the maximum number of possible edges. To reflect this reality in our experiment, we conduct negative sampling with a ratio of 1:10 (i.e., for every positive edge, we sample 10 negative edges). While AUROC has historically been the primary metric used to assess the link prediction performance [27], it can be too optimistic in such imbalanced scenarios and favors accurate classification of positive examples, at the cost of significantly misclassifying negative examples. AUPR compensates this weakness

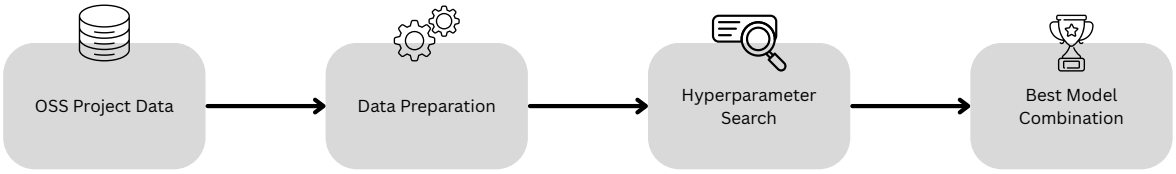


Figure 5.3: This figure provides a high-level overview of the process of finding the best link prediction model. The explicit hyperparameter search is further discussed in the chapter. Given one dataset, i.e., one OSS project, we find the best model combination for our link prediction model.

by addressing the precision of the positive class. It measures how well the model is at classifying true positives without having too many false positives [30].

AUROC measures the ability of the model to discriminate between positive and negative classes across all possible thresholds [4]. The score ranges from 0 to 1, where a higher value indicates better performance. A score of 0.5 indicates that the model has a 50% chance of ranking a positive instance above a negative one. Consequently, it performs no better than a random classifier. The AUROC score can be interpreted as the probability that a randomly sampled positive edge receives a higher score than a randomly sampled negative edge. This interpretation allows us to approximate AUROC via pairwise comparisons between positive and negative samples [4]. If n positive–negative pairs are sampled at random, with the positive edge receiving a higher score in n_1 cases and equal scores occurring in n_2 cases, AUROC can be approximated as [4]:

$$AUROC = \frac{n_1 + 0.5 \cdot n_2}{n}$$

AUPR measures the area under the precision-recall curve, which plots Precision (on the Y-axis) against Recall (on the X-axis) for different thresholds [4]. Precision and Recall are defined as follows [4]:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

A higher AUPR indicates that true missing links are consistently ranked higher than non-existent ones.

5.4.3 Hyperparameter Search

To obtain the best combination of embedding and link prediction model, we run a hyperparameter search. Within this search, we try different parameter configurations for both the GraphSAGE model and the link prediction model. Figure 5.3 provides a high-level overview of this pipeline. It further highlights that for each OSS dataset, we independently run this hyperparameter search to find the best model.

To train GraphSAGE, we need to set the following hyperparameters in advance: the number of epochs, the batch size, the learning rate, and the output dimension, i.e., the dimension of the embedding vector. The parameter selection can be seen in Table 5.5.

Table 5.5: Hyperparameter Search Space

Model	Hyperparameter	Values
GraphSAGE	Number of Epochs	$\{250, 300, 350, 400\}$
	Batch Size	$\{32, 64, 128, 256\}$
	Learning Rate	$\{0.01, 0.001\}$
	Output Dimension	$\{16, 32, 64\}$
Link Prediction	Number of Epochs	$\{10, 20, 30, 40\}$
	Batch Size	$\{8, 16, 32, 64\}$

The number of epochs can be interpreted as a maximum value. We train GraphSAGE with early stopping. That is, we monitor the validation loss and stop as soon as it increases compared to the previous epoch.

The batch size controls how many nodes are processed in a single forward and backward pass. According to Goodfellow [17], batch sizes are commonly a power of 2, which typically range from 32 to 256. A smaller batch size can offer a regularization effect and is therefore better at generalization, yet, it requires more steps and increases the runtime. Larger batch sizes generalize worse than smaller batch sizes, however, the training time is faster and larger batches provide a more accurate estimate of the gradient [17]. We try a variety of sizes to find the optimum.

The learning rate determines the size of the step moving into the negative direction of the gradient, which is computed during the backward phase [17]. To avoid overshooting the optimum, the learning rate should not be too large. Therefore, we select standard, conservative values 0.01 and 0.001.

The last parameter we consider in our search is the output dimension. This dimension determines the size of the feature vector representing a node. A higher dimension increases the representational capacity of the model, allowing it to capture more complex structural relationships and preserve richer semantic information within the graph [35]. However, if it is too high, it might lead to computational inefficiency or overfitting. Thus, we try a range of different values to find the output dimension that best represents the nodes of our graphs.

Similar to GraphSAGE, the link prediction model, a simple two-layer neural network, requires preset hyperparameters. Here, we investigate the number of epochs and the batch size. The values we consider are also summarized in Table 5.5.

Since the prediction model is less complex and contains fewer parameters, it requires fewer iterations to converge. Consequently, the number of epochs only ranges from 10 to 40. Furthermore, there is no computational need for excessively large batch sizes. Thus, we only consider values between 8 to 64.

Once all combinations of GraphSAGE models are trained, we train the link prediction model on the train graph and evaluate on the validation graph. To overcome the unsupervised setting, we self-supervise and create negative edges, similar to GraphSAGE. We ensure that our negative sampling strategy avoids trivial invalid edges that violate the bipartite structure. We post-filter our sampled negative edge index to ensure that at least

80% of the sampled edges are valid bipartite negatives. For efficiency, we sample only 5 times, in a few cases even 10 times, more negatives to obtain a sufficiently large pool of candidate edges, and accept the result if at least 80% are valid. The node pairs are then mapped to labels, i.e., if a link exists, the target value is 1, else it is 0. As the primary metric, we choose the AUPR score as it is robust to class imbalance and a standard for link prediction, as discussed before. Additionally, we report AUROC to assess the performance, as it is commonly used in other research.

Algorithm 2 Hyperparameter Search

```
Input: Set of embedding models E, set of link configurations C
Output: Best embedding model e*, best link prediction model l*

(e*,l*) ← null
best_avg_precision ← 0

for e in E do
    for c in C do
        l ← TrainLinkPredictor(e,c)
        avg_precision ← Evaluate(l)
        if avg_precision > best_avg_precision then
            (e*,l*) ← (e, l)
            best_avg_precision ← avg_precision
        end if
    end for
end for

return (e*,l*)
```

To find the best combination of embedding and link prediction model, we need to iterate over all embedding models that are temporarily saved and train each configuration for the link prediction model, only saving the model with the highest average precision score. The hyperparameter search is also described in Algorithm 2 and is repeated independently for each dataset.

5.5 Baselines

To assess the performance of our proposed approach, we compare it against a set of established baselines. This selection allows us to examine the benefits or disadvantages of learning complex structural embeddings via message passing compared to simpler topological predictions.

Random Classifier: Acts as a lower bound. This classifier randomly assigns scores to the tested edges. Independent of the class imbalance, the random baseline achieves an AUROC score of 0.5. As AUPR, however, is sensitive to class imbalance, the random baseline is not 0.5. Given our test set ratio of 1:10 (ten negative samples per one positive sample), the

probability of a positive class is $\frac{1}{11}$. Consequently, a random classifier baseline reaches an AUPR of $\frac{1}{11} \approx 0.09$.

Logistic Regression (LR): A standard feature-based machine learning baseline. LR assumes i.i.d. data as input and is defined as follows [9]:

$$P(y = 1) = \frac{1}{1 + e^{-\boldsymbol{\beta}\boldsymbol{X}}}$$

where $P(y = 1)$ is the probability of the output variable being 1, e is the base of the natural logarithm, $\boldsymbol{\beta}$ is the coefficient vector for the independent feature input vector $\boldsymbol{X}$. We construct the input $\boldsymbol{X}$ for a link (u, v) by concatenating their features. In our case, we take the resulting features from our embedding model. LR does not create new features via message passing or neighborhood aggregation but strictly relies on the input features to predict the tested links. In our experimental setup, this baseline acts as a direct counterpart to the small non-linear neural network that we use as the classification head in our approach. Therefore, we can examine whether the node representations are simply linearly separable or require a more complex classification method.

Preferential Attachment (PA): A topology-based heuristic that operates on the assumption that nodes with high degrees (hubs) are more likely to form new connections [27]. The PA score between nodes x and y can be expressed as the product of their degrees [27]:

$$PA(x, y) = |\mathcal{N}(x)| \times |\mathcal{N}(y)|$$

where $\mathcal{N}(x)$ denotes the set of neighbors of x. This score helps to determine whether the likelihood of a link is solely determined by the individual popularity of the nodes. In other words, we examine whether it is a trivial task to predict edges or if the link prediction model requires more understanding about local community structures, i.e., via message passing.

5.6 Statistical Tests

To ensure a rigorous evaluation and to verify that any performance differences between our approach and the baselines are not due to random variance, we employ structural and statistical tests. First, to objectively interpret the performance of the preferential attachment baseline, we analyze the topological characteristics of our datasets. Preferential attachment is known to perform well on scale-free networks. A scale-free network has a degree distribution that follows the power-law, that is, there are a few, but a significant number of nodes that have many connections (hubs), while most nodes in the network have significantly fewer connections [41]. Consequently, we test whether our selected graphs follow the power-law distribution. Instead of relying on absolute goodness-of-fit tests that often fail on noisy empirical data and are computationally intensive, we evaluate the scale-free property of our selected graphs using a relative comparison [1]. Therefore, we compare the power-law fit against two specific distributions, i.e., the exponential and the log-normal distribution. The exponential distribution serves as a baseline. If the node degree distribution cannot significantly better fit the power-law than the exponential distribution, the nodes of the graph do not follow a heavy-tailed distribution [1]. In contrast, the log-normal serves as a rigorous alternative, as it is also heavy-tailed [1].

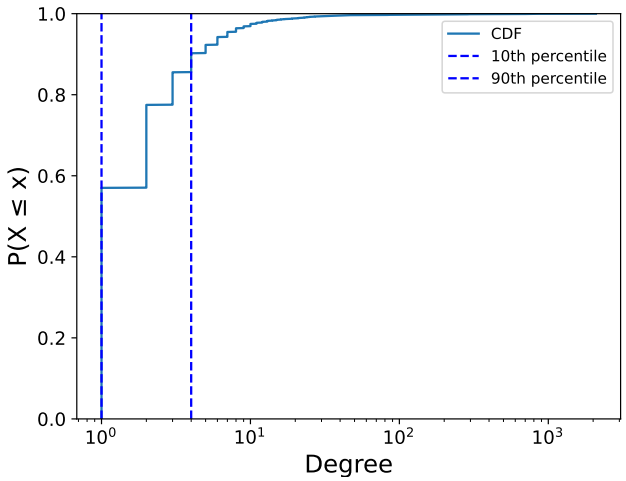


Figure 5.4: This plot illustrates the cumulative distribution function (CDF) for one of our graph projects Atom. Additionally, we highlight the lower and upper 10^{th} percentile of the node degree distribution.

Furthermore, we conduct statistical significance testing to evaluate our model's predictive performance across the datasets. Specifically, we use the Wilcoxon signed-rank test across our datasets. As a non-parametric test for paired samples, the Wilcoxon signed-rank test is appropriate for our setup as it does not assume a normal distribution of the underlying data [38]. We utilize this test for two purposes. One, to compare the performance of our proposed approach against the baselines, and second, to evaluate our model's robustness under varying experimental conditions, such as reduced training set sizes.

Following standard practice, we adopt a significance level of $\alpha = 0.05$ for all statistical tests to determine the rejection of the null hypothesis.

5.7 Controlled Experiments

Instead of solely reporting the performance of GraphCP on the varying datasets and comparing it to the baseline methods, we conduct targeted experiments to investigate the model's behavior under specific, challenging conditions. While the overall performance is evaluated on the entire datasets, we want to further investigate the model performance on edge cases, i.e., predicting links between low-degree nodes. Additionally, we explore whether the training size has an effect on the model performance. These experiments are detailed in the following.

5.7.1 Evaluation on Low Degree Nodes

Networks can contain hub nodes with a large number of connections. In our domain, these represent either developers who work on a large amount of different files, or files that were touched by a large amount of developers (see Figure 2.1). For these cases, the probability for

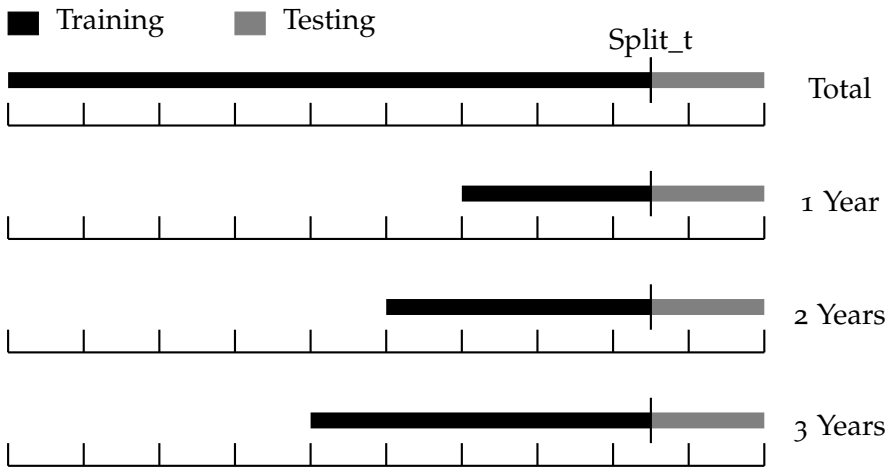


Figure 5.5: Visualization of the training and test split along the timeline. The top bar shows the distribution of our standard approach. The subsequent bars illustrate at a high-level of abstraction, the experiments conducted with progressively larger datasets.

a new link formation is already high. However, predicting connections between less active, low-degree nodes might be a more challenging task. To evaluate this, we determine the degree thresholds for the lower and upper 10% of nodes based on their degree distribution (i.e., the 10^{th} and 90^{th} percentiles). For a better understanding, consider Figure 5.4. This plot visualizes the cumulative distribution function (CDF) for the Atom node degree sequence. The x-axis shows the degree of a node and is sorted from the smallest number on the left to the largest on the right. The y-axis represents the percentage of the nodes that have a degree less than or equal to the corresponding x-value. The left and right dashed lines represent the lower and upper 10^{th} percentiles, respectively. In this example, at least 10% of the nodes in the Atom graph have exactly one connection. The second dashed line reveals that 90% of the nodes have 4 or fewer connections. After computing the percentiles, we filter all test edges for which both nodes have a degree less than or equal to the lower threshold. By predicting these specific edges using GraphCP and the baseline models, this experiment reveals how well GraphCP performs on more challenging cases.

5.7.2 Truncating Training Graph

As explained in Section 5.3.1 and Section 5.4.1, we initially train GraphCP on the entire project history up to the most recent 10%, which are held out for testing. While this provides the model with a comprehensive historical context, it may also not represent the most recent activities in the test set. This way, the graph may contain information that is outdated, noisy, and not relevant anymore. Therefore, we want to experiment on a subset of projects by truncating their training graph. Specifically, we evaluate on the largest graphs Moby and Nextcloud but also on the smallest graph, i.e., Keras. Taking Figure 5.5 for a better understanding, in these experiments, we keep the split timestamp between the test and train graph and restrict the training graph to cover only the past 1 year, 2 years, and 3 years

prior to the split. For each case, we compare the model's performance to the baselines, as well as to the model trained on the entire dataset. It is important to note that the test sizes for the different settings vary as well, as we filter the test set based on the corresponding training set (see Section 5.3.1). Consequently, to ensure fair comparison across different training sizes, we evaluate the performance on the intersection of their respective test edges.

6 Evaluation

After discussing the experimental setup and evaluation process in the previous Chapter 5, we now present the performance results of our proposed approach. First, we analyze the statistical tests on the dataset structure and the hyperparameter search results. This is followed by analyzing the quantitative performance of our model and providing a comparative overview. Afterward, we interpret the results in the context of the specific network characteristics. At the end of this chapter, we reflect on the results by answering our research question but also by considering possible threats to validity. This includes internal and external validity that we have to keep in mind during the process and our evaluation. We finish this chapter by discussing the potential of our methodology and provide an outlook on future research directions to further enhance the capabilities of GraphCP.

6.1 Dataset Analysis

Before providing a thorough analysis of the model and baseline performances, we first examine the structure and distribution of the node degrees for each project we considered in our evaluation. As described in Section 5.6, we investigate whether our networks exhibit a scale-free characteristic, i.e., follow the power-law distribution. To this end, we compare the node degree distribution to the exponential distribution and the log-normal distribution.

To investigate whether our graphs exhibit scale-free properties, we perform two statistical tests that reveal if their degree distributions are better modeled by a power-law or alternative distributions. The exponential distribution serves as a minimum baseline that tests whether our data is heavy-tailed. The log-normal distribution serves as a rigorous alternative as it is also a heavy-tailed distribution similar to power-law [1]. The tests return the log-likelihood ratio R between the two candidate distributions and p as the significance value. R is positive if the data is more likely in the first distribution, that is the power-law distribution, and negative if the data is more likely in the alternative distribution [1]. The p-value indicates whether the observed log-likelihood ratio is statistically significant, or if the result could have occurred by chance.

Table 6.1 summarizes the results for each dataset and test. It is evident that Keras is the only project for which it is inconclusive to say if the degree distribution is heavy-tailed. Both the minimal baseline test as well as the comparison to log-normal result in inconclusive test results. Although its small positive R value hints at a preference for the power-law, the p-value of 0.06 prevents us from stating statistical significance against the exponential baseline. The inconclusiveness in the test statistic is visually reflected in Figure 6.1. This plot shows

Table 6.1: Test results comparing the power-law vs. exponential and vs. log-normal distributions for the degree distributions of our seven dataset projects. A positive R value indicates a better fit for the power-law, while a negative R value favors the alternative distribution. The p-value indicates how likely the result happened by chance. As a significance value, we choose $\alpha < 0.05$.

	vs. Exponential			vs. Log-normal		
Dataset	R	p-**value**	**Best Fit**	R	p-**value**	**Best Fit**
Keras	1.9	0.06	Inconclusive	-0.38	0.71	Inconclusive
Atom	4.25	<0.01	Power-law	1.42	0.16	Inconclusive
Deno	5.34	<0.01	Power-law	-1.00	0.36	Inconclusive
OpenSSL	6.22	<0.01	Power-law	-0.98	0.33	Inconclusive
VSCode	6.87	<0.01	Power-law	-0.97	0.33	Inconclusive
Moby	8.82	<0.01	Power-law	-1.01	0.31	Inconclusive
Nextcloud	7.16	<0.01	Power-law	-0.96	0.34	Inconclusive

the complementary cumulative distribution function (CCDF) of the degree distribution for Keras. It also provides for comparison the best fits for a power-law, exponential, and log-normal distribution. The Keras distribution does not strictly follow one of the investigated distributions. While it initially aligns with the power-law and log-normal fits, the data points drop off notably at higher degrees.

In contrast, for the remaining projects, we always have a large positive R value and very small p-values, indicating that the power-law model significantly outperforms the exponential distribution. This implies that the node degree distributions of these projects are heavy-tailed rather than exponentially bounded. However, when comparing the power-law to the log-normal distribution, we observe inconclusive results for every project. As the p-values are substantially large, we cannot statistically distinguish between a power-law or log-normal distribution. As a representative example, Figure 6.2 visually reflects that the node degree distribution for Atom follows a power-law much more closely than an exponential distribution. However, it remains ambiguous whether the log-normal or the power-law distribution represents a better fit. In this context, it is important to note that the log-normal distribution has two parameters to serve as a degree of freedom for fitting, making it more flexible and potentially leading to overfitting the data [1]. Regardless of which distribution strictly provides the better fit, both are inherently heavy-tailed. This property confirms that the graph contains highly connected hub nodes. Consequently, the underlying assumption of preferential attachment remains valid, making it an appropriate baseline for our approach.

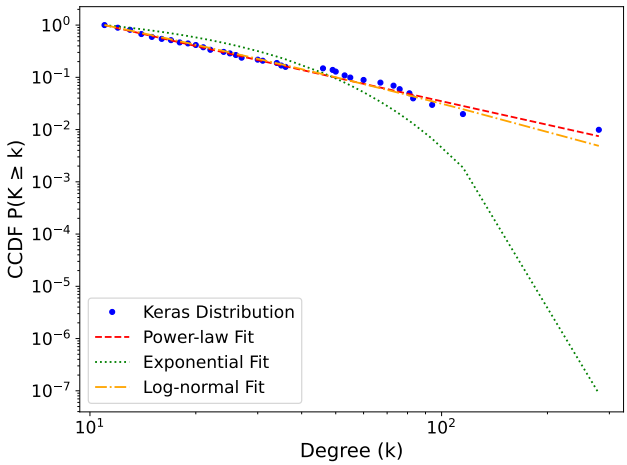


Figure 6.1: Complementary cumulative distribution function (CCDF) of the degree distribution for the KERAS network. The KERAS node degree distribution (blue dots) is plotted alongside the best fits for a power-law, exponential, and log-normal distribution. The plot visually confirms that there is no clear best fit for the investigated distribution.

6.2 Hyperparameter Analysis

Table 6.2: Best Parameter Configuration for GRAPHSAGE per Dataset

Dataset	Epochs	Batch Size	Learning Rate	Output Dimension
KERAS	300	256	0.001	64
ATOM	350	128	0.001	64
DENO	400	64	0.01	64
OPENSSL	250	256	0.01	32
VSCODE	300	128	0.01	16
MOBY	400	64	0.001	64
NEXTCLOUD	350	128	0.001	16

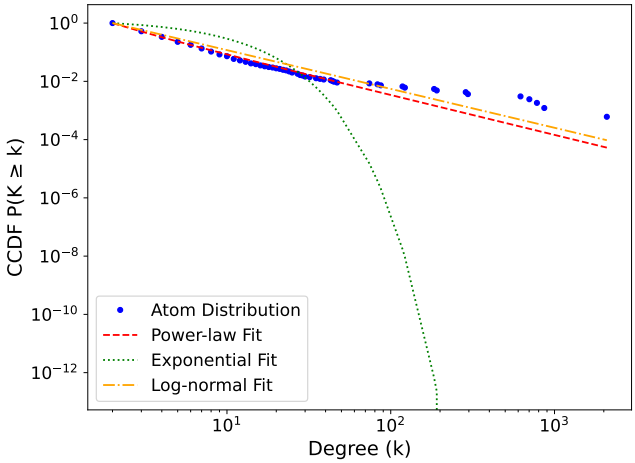


Figure 6.2: Complementary cumulative distribution function (CCDF) of the degree distribution for the Atom network. The Atom node degree distribution (blue dots) is plotted alongside the best fits for a power-law, exponential, and log-normal distribution. The plot visually confirms that, compared to the exponential distribution, the power-law is a better fit for the Atom node distribution. In comparison with the log-normal distribution, however, the difference is less clear.

Table 6.3: Best Parameter Configuration for Prediction Model per Dataset

Dataset	Epochs	Batch Size
Keras	10	64
Atom	20	64
Deno	10	32
OpenSSL	30	16
VSCode	30	16
Moby	20	64
Nextcloud	10	32

To understand how the models achieved different results across different datasets, we will analyze the hyperparameter results, which can be seen in Tables 6.2 and 6.3. The best model performances were achieved using these parameter configurations. In general, we can see in Table 6.2, that the smallest and the second largest projects needed the largest vector space to represent a node. That is, the output dimension for the Keras, Atom, Deno, and Moby projects was always 64. Whereas for VSCode and Nextcloud it was only 16, and for OpenSSL 32. Except for Deno, OpenSSL, and VSCode, the individual models for all other projects required a learning rate of 0.001 to reach the best result. Regarding the epoch number and the batch size, we cannot see a distinguishable pattern. Furthermore, as we also use early stopping, it might be the case that the models did not actually train for that amount of epochs.

In Table 6.3, we can also observe variations in the parameter configuration. The parameters are almost evenly distributed across the different projects, making it hard to detect a pattern or a trend. It is interesting to see that the link prediction on the Nextcloud dataset only required 10 epochs to reach a remarkable performance. For the Keras project, the best metrics were achieved by training the model for the fewest epochs, i.e., 10 iterations. As we also tried more epochs, we can rule out that the bad performance of our approach is due to underfitting. The best models on the VSCode and OpenSSL data were the only models trained with the smallest batch size, i.e., 16 in link prediction. Additionally, these models only needed a small embedding dimension of only 16 and 32, respectively, for the GraphSAGE model.

As described in Chapter 5, for each link prediction configuration, we iterated over all embedding models and recorded the best-performing embedding–link prediction combination. These intermediate results are documented in Tables A.2- A.8. The best performing configuration and the most used embedding configuration IDs are highlighted in bold. It is worth noting that for the top three largest datasets, the embedding parameter configuration corresponding to the best-performing combination appeared multiple times as the optimal choice. That is, for VSCode, the embedding configuration with ID 36 appeared the most, for Moby, it is ID 83, and for Nextcloud embedding configuration ID 63 appeared the most. This suggests that these models are particularly effective in representing the nodes in the graph. For the remaining datasets, no such pattern can be detected.

In general, we can say that there is not one globally best combination of the link prediction and embedding model parameter configuration. However, we identified that smaller networks require higher dimensionality (i.e., larger output dimensions for the feature vectors) to be best described. This indicates that these networks require larger latent space capacity to capture subtle differences or complex relationships within the graph [3].

In larger datasets, the embedding ID associated with the best-performing configuration also occurs multiple times. That is, when testing every link prediction parameter configuration, choosing this embedding ID yielded the best intermediate performance. This indicates that for these networks, the embedding model creates well-structured and strong representations. For the other projects, i.e., Keras, Atom, and Deno, the embedding IDs vary across the different link prediction setups. This suggests that the networks allow different ways of representation while still being almost as good as the best combination. A notable case is OpenSSL, where ID 61 emerged as the most frequent embedding configuration, despite ID 19 achieving the best performance across all combinations. It is important to note that both configurations share the same learning rate and output dimension. This indicates that the training dynamics, specifically the number of epochs or the batch size, were the key factor for a better performance. A similar pattern is observed for VSCode, where configuration IDs 36 and 54 appeared equally often, confirming that multiple strong parameter setups exist for this network.

Overall we observe, that for each project, different configurations seem to work better. Consequently, the results show that no single hyperparameter configuration is globally optimal, but rather strongly depends on the project and dataset.

6.3 Results

This section presents the performance results of our approach. We have run our process, as illustrated in Figure 5.2, on 7 different datasets and compared the best combination, derived from the hyperparameter search, with the baselines. These are the random classifier (Random), logistic regression (LR), and preferential attachment (PA). Table 6.4 summarizes the overall performance of our GNN approach, i.e., GraphCP, compared to the baselines per dataset. The table reports the primary metric AUPR, used to find the best model combination in our hyperparameter search, and the AUROC score. For each dataset, we highlight the best scores in bold. Since the model was retrained for each dataset, the results demonstrate the adaptability of the proposed method to different data distributions rather than cross-dataset generalization.

6.3.1 Overall Performance and Trend

Table 6.4: A detailed summary of the performance for our approach, i.e., GraphCP, and the baseline methods. For each dataset, we independently train GraphCP and report the AUPR and AUROC on the test set. Additionally, we evaluate the performance on the baseline methods.

	Keras		Atom		Deno		OpenSSL		VSCode		Moby		Nextcloud	
Method	AUPR	AUROC	AUPR	AUROC	AUPR	AUROC	AUPR	AUROC	AUPR	AUROC	AUPR	AUROC	AUPR	AUROC
Random	0.09	0.51	0.09	0.51	0.09	0.50	0.09	0.50	0.09	0.48	0.09	0.51	0.09	0.50
LR	0.19	0.77	0.23	0.81	0.24	0.81	0.12	0.65	0.22	0.74	0.45	0.91	0.45	0.92
PA	**0.57**	**0.90**	0.33	0.76	**0.57**	0.81	0.46	0.90	**0.68**	0.81	**0.73**	0.86	0.70	0.91
GraphCP	0.42	0.87	**0.61**	**0.91**	0.55	**0.90**	**0.60**	**0.92**	0.67	**0.87**	0.67	**0.96**	**0.86**	**0.98**

We first provide a total overview of the performance of the baselines and our approach, i.e., GraphCP. A more detailed summary is provided in Table 6.4.

As can be observed, GraphCP outperforms the random and LR baselines across all datasets and the metrics. As expected, the random baseline always reaches an AUPR score of 0.09 and an AUROC score of approximately 0.5. This indicates that the task is not trivial. In three out of seven projects, our method surpasses the popularity-based heuristic (PA) in terms of the average precision score (AUPR). Specifically, on the Deno and VSCode projects, it only slightly underperforms the PA baseline, with a difference of 0.02 and 0.01 AUPR score points, respectively. Furthermore, in terms of AUROC, GraphCP almost always exceeds the performance compared to the baselines. Additionally, as expected, AUROC is less sensitive to dataset imbalance and shows substantially higher scores than AUPR. This trend is consistent throughout the datasets.

6.3.1.1 *Comparison to Baselines*

Comparing our approach to the baselines, distinct patterns emerge. The topological heuristic preferential attachment (PA), which assumes that high-degree nodes are more likely to be

Table 6.5: Statistical results of the Wilcoxon-signed rank test, as described in Section 5.6. We compare the performance results from Table 6.4 of our method against the performance results across all datasets of the baselines. As significance value, we chose $\alpha = 0.05$. If the p-value is smaller than the significance value we can reject the null-hypothesis and state that GraphCP significantly performs better than the corresponding baseline.

Baseline	p-value
Random	0.0078^{**}
LR	0.0078^{**}
PA	0.23

connected to other nodes, shows substantially better performances for Keras and Moby. The differences to GraphCP are 0.15 and 0.05 in terms of AUPR scores, for Keras and Moby, respectively. For Deno and VSCode, PA only marginally outperforms GraphCP, having an AUPR score difference of 0.01 and 0.02, respectively. Conversely, GraphCP performs considerably better than PA on the Atom, OpenSSL, and Nextcloud projects. In general, we can see that in three out of seven projects, GraphCP performs better than PA, nearly matches the PA performance in two, and is substantially outperformed in the remaining two. This suggests that for networks where preferential attachment dominates, the centrality-driven attachment between two nodes may already be sufficient. In such cases, our approach, i.e., GraphCP, may not provide additional benefits, as it focuses on more complex patterns rather than prioritizing simpler connections. However, the strong performance results for Atom, OpenSSL, and Nextcloud highlight GraphCP's ability to capture complex structural patterns where a simple popularity-based heuristic, such as PA, fails.

The LR baseline does not follow any trend and rather shows mixed results. While it performs better on the Nextcloud or Moby datasets, in terms of AUPR score, it performs almost as poorly as the random baseline on the OpenSSL project. Additionally, for Keras, Atom, Deno, or VSCode, its AUPR score only ranges from 0.19 to 0.24. This indicates that for these projects, the non-linear neural network architecture is essential for the link prediction task.

To assess whether the performance difference is statistically significant, we performed the Wilcoxon signed-rank test comparing the AUPR scores of GraphCP against the baselines across our dataset. Table 6.5 summarizes the test outcomes. As expected, GraphCP significantly outperforms the random and LR baselines. As both p-values are substantially smaller than our preset significance level $\alpha = 0.05$, we can confidently reject the null-hypothesis. This confirms that GraphCP achieves statistically significantly higher AUPR scores than these two baselines. Considering the ambiguous PA results, it is not surprising that the statistical comparison between GraphCP and PA yields an inconclusive result. Since for this test the p-value is larger than α, we fail to reject the null hypothesis. As a result, we cannot claim with statistical confidence that our approach performs better than PA across all datasets.

In general, we can see that GraphCP does not significantly perform better than every baseline. Especially PA represents a strong baseline for our approach.

6.3.1.2 *Dataset-Level Analysis*

Regarding individual datasets, we observe considerable performance variations in Table 6.4. While GraphCP performs worst on the smallest dataset, i.e., the Keras project, reaching an AUPR score of only 0.42 (compared to PA with 0.57), our proposed method demonstrates a stable performance on the Atom, OpenSSL, VSCode, and Moby, reaching AUPR scores between 0.60 and 0.67. The corresponding AUROC scores for these projects reveal a slightly higher variance.

Atom and OpenSSL are the networks with the worst performances regarding the PA baseline, only reaching 0.33 and 0.46 as AUPR scores, respectively. This implies that these networks do not primarily follow the simple popularity-based link formation, making it difficult for basic heuristics to achieve meaningful predictions. In particular, the structure of these networks seems to be complex enough and not as predictable using the simple topological heuristic PA. A similar observation can be made for Nextcloud. On this project, GraphCP achieves the highest overall AUPR score of 0.86 and AUROC score of 0.98. While PA also performs relatively well on this dataset, having an AUPR of 0.70, it is not its best performance across all datasets.

Overall, whenever the proposed GNN approach outperforms the PA baseline, it does so by considerably better results. This indicates that for these graphs, the model is able to effectively learn dataset-specific patterns. In cases where it performs similarly to or worse than the PA baseline, it implies that the structural signals are sufficient for prediction, which is a property of scale-free and therefore hub-driven networks.

6.3.2 Controlled Experiment Analysis

The following subsection provides an overview of the smaller controlled experiments as described in Section 5.7. As Table 6.4 summarizes the performance on the entire test dataset, we now want to observe how well the baselines and our approach perform on a filtered test dataset, where we only consider links between nodes with a low degree. Furthermore, as a second experiment, we investigate the behavioral change of the model's performance when truncating the training dataset.

6.3.2.1 *Performance on Low-Degree Nodes*

After analysing GraphCP's performance on the entire dataset, we now present the results on edges between low-degree nodes. As described in Section 5.7, we only consider links between nodes that both have a degree at or below the 10^{th} percentile of the overall degree distribution. As can be seen in Table 6.6, for the projects Keras, OpenSSL, and Nextcloud, there exists no such positive labeled edge in the testdata. Therefore, we cannot report any measurements. Furthermore, we can observe that the amount of positive edges under this constraint is relatively small. While Moby is the only dataset with the most reliable positive sample size of 43 out of 4166 total examined edges, Atom only has two, and Deno and VSCode only have one true positive edge. This makes it hard to make any confident statements. As expected, PA performs poorly in this setting, reaching almost always the same score as the random baseline. Besides the Atom project on which GraphCP performs

substantially better than the baselines, it is evident that the LR baseline always outperforms GraphCP on the remaining datasets. Yet, the scores are not substantially different. This implies that GraphCP is highly competitive.

Table 6.6: Performance comparison between GraphCP and the baselines, similar to Table 6.4. Instead of taking the entire test set, in this experiment we only considered edges between low-degree nodes. Consequently, the amount of test edges is smaller than the entire test set, and more interestingly, the amount of true positive test edges is substantially smaller than all considered edges in this experiment. This can be seen in the last two rows of this table.

	Atom		Deno		VSCode		Moby	
Method	AUPR	AUROC	AUPR	AUROC	AUPR	AUROC	AUPR	AUROC
Random	0.003	0.26	0.003	0.23	0.001	0.75	0.011	0.54
LR	0.008	0.71	**0.011**	0.76	**0.008**	**0.95**	**0.334**	**0.99**
PA	0.003	0.48	0.006	**0.78**	0.001	0.72	0.010	0.15
GraphCP	**0.048**	**0.96**	0.009	0.71	0.004	0.90	0.305	0.97
$\lvert E_{positive} \rvert$		2		1		1		43
$\lvert E_{total} \rvert$		655		389		2538		4166

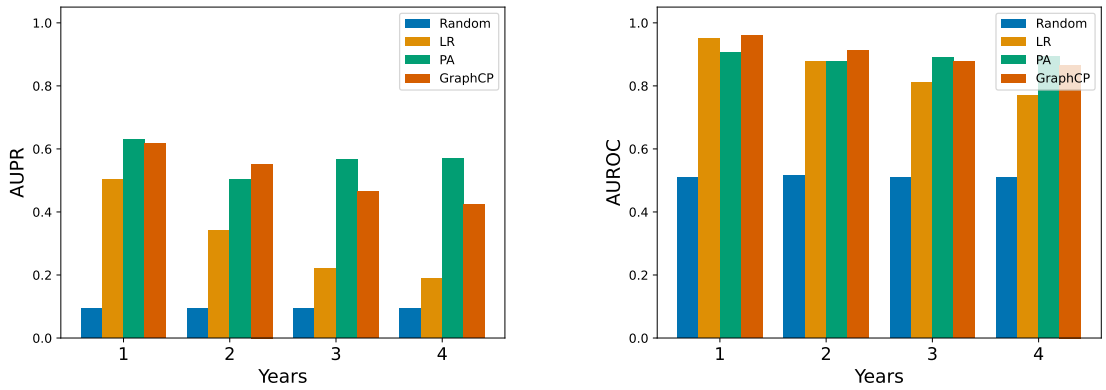
6.3.2.2 *Performance on Truncated Training Graphs*

Table 6.7: Overview of the growing train graph size, in terms of edges. Additionally as the test set is filtered to match with the training nodes and to ensure that no train edge is present in the test set, the test edge size also varies.

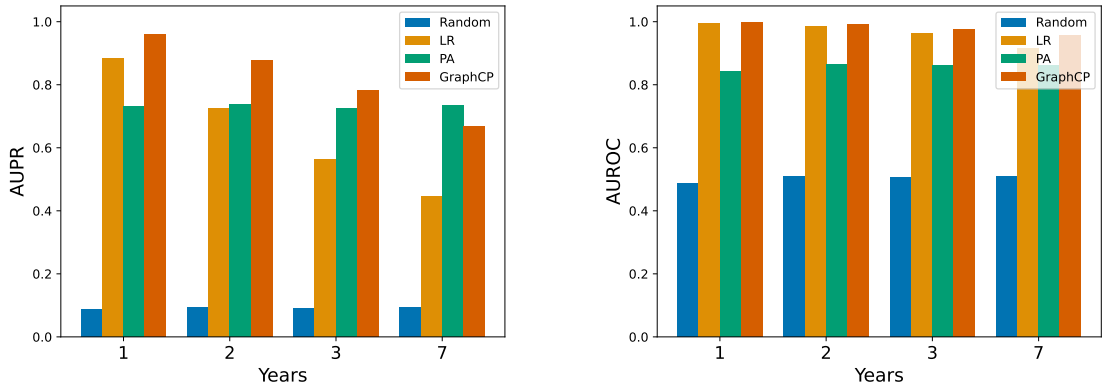
	Keras		Moby		Nextcloud	
Time	$\lvert E_{Train} \rvert$	$\lvert E_{Test} \rvert$	$\lvert E_{Train} \rvert$	$\lvert E_{Test} \rvert$	$\lvert E_{Train} \rvert$	$\lvert E_{Test} \rvert$
1 Year	840	114	4 495	1 437	3 030	2 336
2 Years	1 683	102	11 536	1 865	6 632	3 411
3 Years	2 221	102	20 949	1 852	11 296	3 666
Total*	2 487	101	34 505	1 839	33 115	3 696

*Total interval: Keras (4y), Moby (7y), Nextcloud (10y).

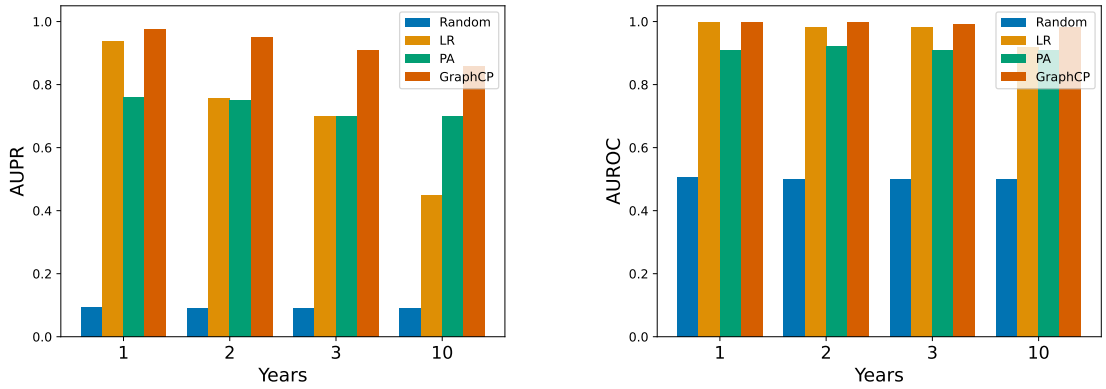
Another constraint we want to investigate is the performance behavior on smaller training data. So far, we have trained the model on the entire project history. For a few datasets, we want to see if truncating the training data positively or negatively impacts the model performance. To this end, we only use one, two, or three years prior to the train-test split. As longer time periods more closely approximate the entire dataset, these specific constraints



(a) KERAS



(b) MOBY



(c) NEXTCLOUD

Figure 6.3: Comparison of the predictive performance across different training sizes and methods for three datasets: Keras (a), Moby (b), and Nextcloud (c). The left column displays the AUPR score, while the right column displays the AUROC score. In each subplot, the rightmost category illustrates the results of the original approach (using the entire training data), while the previous bars demonstrate the performance metrics using more recent subsets of the training data.

serve as a rigorous comparison. For this experiment, we select the smallest dataset, i.e., Keras, alongside the two largest datasets Moby and Nextcloud.

We summarize the predictive performances in Figure 6.3, which illustrates the corresponding trends across the different time constraints. In particular, Figure 6.3 provides a visual overview of the different performances on each dataset, comparing the model that was trained on the entire train graph. Starting from left, the train graph size increases further to the right. The rightmost category in the plots serves as a comparison baseline and illustrates the performance of GraphCP when trained on the entire train graph. The figures in the left column illustrate the AUPR comparisons and the figures on the right present the AUROC results. The varying train and test sizes for these experiments are summarized in Table 6.7.

As shown for Keras in Figure 6.3 (a), GraphCP is almost as good as the PA baseline when trained exclusively on data from one year prior to the test set activities. While PA achieves an AUPR score of 0.63, GraphCP closely follows with 0.62. Interestingly, when increasing the training data by one more year, we can even see that GraphCP outperforms PA, having a difference of 0.05 AUPR points. However, as the training size is further increased to three years — approximating the entire available project history - we can observe that GraphCP's performance drops substantially below that of PA's. This decline also aligns with the performance of the entire dataset. Finally, regarding the second baseline, LR, we see that our non-linear classification approach consistently outperforms the linear classifier across all temporal splits.

On the larger datasets, we observe a different behavioral shift. Although GraphCP's performance differed substantially between the full Moby and Nextcloud datasets, both models perform notably well on the smallest training data. Both reaching an AUPR score of 0.96 and 0.98, respectively. It is important to note the varying testing conditions shown in Table 6.7. While methods are tested on 1 437 test edges for Moby, the Nextcloud test set contains almost 1 000 additional edges.

As the training data increases for both projects, we can observe in Figure 6.3 that GraphCP's performance decreases in terms of AUPR scores. For the Moby project, AUPR scores decrease by approximately 0.1 AUPR points with each timestamp, alongside a substantial increase in the absolute volume of training data. Despite this decline, GraphCP consistently outperforms both baselines. In particular, we see that PA's performance is the least impacted by the varying training sizes, whereas LR mirrors the decreasing trend of GraphCP.

When training GraphCP on the smallest train data of the Nextcloud project, it performs almost perfectly with an AUPR score of 0.98. While LR remains competitively close with an AUPR of 0.94, PA only achieves 0.76. Increasing the training data only has a small impact on GraphCP's and PA's performance. Although LR's performance drops substantially after the first increase, it remains marginally better than PA. The small decline in the performance aligns with the small increase in the training data.

After comparing the differently trained models with the baselines, we also want to draw a comparison to the model that was trained on the entire dataset. In Table 6.8, we can see for each dataset a direct comparison between the models that were trained on the specific temporal splits and the entire project history. As their training data differ, the corresponding test sets also differ. Therefore, we choose to evaluate the performance of the GraphCP models on the intersected test set.

At first glance, it is evident that across all examined time splits, each GraphCP$_{Split}$ model substantially outperforms the GraphCP$_{Total}$ model in terms of AUPR and AUROC score. As already observed previously in this section, with increasing training size across all datasets, GraphCP$_{Split}$'s performance gradually decreases in terms of AUPR scores. Furthermore, as the train split increases, the absolute number of intersected test edges increases as well. Especially for the largest dataset Nextcloud $|E_{test}|$ grows from 1 821 to 3 618 edges. Despite this increase, the AUPR scores fluctuations for GraphCP$_{Total}$ remain marginal. The largest drop occurs on the Keras project, where GraphCP$_{Total}$ achieves an AUPR score of 0.45 on the smallest test set and 0.40 on the largest test set. In contrast, the model's performance consistently reaches 0.64 on the Moby project, and fluctuates only minimally between 0.86 and 0.87 on the Nextcloud dataset. As illustrated in Figure 6.3 and reported in Table 6.8, the predictive performance increases under a truncated training graph. To statistically confirm this observation, we performed a one-sided Wilcoxon signed-rank test. Specifically, we compared the paired AUPR scores across all investigated datasets and GraphCP$_{Split}$ against the corresponding baseline scores of the model trained on the entire graph, i.e., GraphCP$_{Total}$. At a significance level of $\alpha = 0.05$, the results confirm that our approach yields significantly better performance on the smaller training sets compared to being trained on the complete dataset ($p = 0.0020$).

Table 6.8: Performance comparison between the model trained on specific temporal splits (GraphCP$_{Split}$) and the model trained on the entire dataset (GraphCP$_{Total}$). To ensure a fair and direct comparison, both models were evaluated strictly on the intersected test set of the respective split.

		GraphCP$_{Split}$		**GraphCP$_{Total}$**		
Dataset	**Train Split**	AUPR	AUROC	AUPR	AUROC	$\lvert E_{test}\rvert$
Keras	1 Year	**0.60**	**0.95**	0.45	0.87	90
	2 Years	**0.56**	**0.92**	0.40	0.86	101
	3 Years	**0.43**	**0.87**	0.40	0.86	101
Moby	1 Year	**0.96**	**0.99**	0.64	0.96	1 292
	2 Years	**0.86**	**0.99**	0.64	0.96	1 795
	3 Years	**0.75**	**0.97**	0.64	0.95	1 833
Nextcloud	1 Year	**0.98**	**0.99**	0.86	0.98	1 821
	2 Years	**0.95**	**0.99**	0.86	0.98	2 989
	3 Years	**0.91**	**0.99**	0.87	0.98	3 618

6.4 Discussion

After highlighting and analyzing the key findings from our comparative evaluation and the controlled experiments, we now proceed to interpret the results. To achieve this, the

discussion is structured along different topics. We start with an overall discussion of our approach compared to the baselines. Afterward, we further discuss the impact of using a smaller and more recent subset to train GraphCP and examine the performance variations across the different projects. At the end, we emphasize the discrepancy between AUPR and AUROC and finish by answering our research question.

6.4.1 Baseline Comparison

As we can see in the previous section and in Table 6.4, PA performs surprisingly well, particularly on large datasets. This indicates that the collaborative development in these projects is driven by centralized developers who work on large amounts of files, or by popular files that attract many developers. As confirmed by our statistical analysis of the degree distributions in Section 6.1, these examined networks exhibit strong heavy-tailed characteristics. That is, there exist a few nodes with much higher degrees than the majority of the nodes. Consequently, links between strictly low-degree nodes are exceedingly rare. Nodes are classified as low-degree if their degree is at or below the 10^{th} percentile (see Section 5.7). As shown in Table 6.6, we could observe that there are either no edges between nodes in the lowest 10^{th} degree percentile, with the Moby dataset being the only exception. This indicates that almost every true connection in the networks involves at least one node with a higher degree. In such scale-free graphs, we can therefore assume that new connections are predominantly drawn to established hub nodes. This perfectly explains the strong predictive performance of the PA baseline on the entire dataset and the severe performance drop when evaluated strictly on the low-degree subset.

The logistic regression (LR) baseline demonstrates the complexity of the task. Since LR uses the same GraphSAGE embeddings as GraphCP but fits a linear decision boundary in the vector space (see Section 5.5), its consistently lower performance validates the non-linear nature of the link prediction task. The difference between LR and PA is almost always substantially large. However, the Atom project presents a notable exception where both baselines perform poorly: LR and PA only reach an AUPR score of 0.23 and 0.33, respectively. As PA only achieves a low AUPR score for the Atom project and our approach reaches 0.61, we can assume that the node degree is not the primary signal in this network. Alternatively, the observed performance difference may be explained by the project's lifecycle. As mentioned before, unlike the other datasets, Atom was deprecated since the community involvement declined in its final years. It is therefore possible that the most recent activities used for testing, as well as the preceding training data, do not follow popularity-based dynamics. Instead, the underlying interaction patterns may be more complex. As a result, simpler approaches, such as PA and LR, are less effective while GraphCP outperforms them. Furthermore, the substantial performance gap between GraphCP and LR demonstrates that the non-linear classifier of our approach is significantly better than simply fitting a linear decision boundary to our data. This aligns with the observation in Table 6.6, where the baselines similarly fail to compete with GraphCP on the Atom dataset. Since LR uses the same embeddings as the classifier in our approach, but simply fits a linear hyperplane in the vector space, this confirms that the combination of GraphSAGE and a non-linear classification model seems to be the best model approach on this dataset.

A similar pattern emerges for the OpenSSL project. Here, GraphCP again substantially outperforms the LR and PA baselines, indicating that for many samples a higher degree is not the primary signal, particularly regarding PA. The fact that LR performs near random-guessing levels even though utilizing the embedded node features strongly implies that the neural network is required for this kind of graph.

Regarding the LR performance on the low-degree node pairs, we see that a simple linear classifier works marginally better in three out of four cases. However, because this performance difference is minimal and true positive edges in these subsets are exceedingly scarce, we cannot confidently say it outperforms GraphCP.

Overall, we can say the proposed method constantly outperforms LR on the entire dataset and is highly competitive with PA. This is also statistically confirmed by the Wilcoxon signed rank test in Table 6.5. On the low-degree node pairs, GraphCP performs substantially better than PA and similarly well as LR. This proves that the structural characteristic and a non-linear classifier are essential for a good link prediction model. In the specific cases where GraphCP performs slightly worse than PA, we can assume that our model overcomplicates the prediction by searching for deeper latent community structures. This way, it misses the dominant and trivial hub signals. Nevertheless, our model does not decide on who is popular but goes beyond that, therefore it succeeds in complex networks where simple heuristics fail.

6.4.2 Impact of Training Size

In our controlled experiment that investigates the performance of our approach on a smaller training set, we observe that the model performs substantially better when trained on smaller, more recent subsets of the data rather than the complete project history. This suggests that for the examined OSS projects Keras, Moby, and Nextcloud, recent project activities are more representative of the current network state. We can assume that after a longer period of time, some developers naturally worked on a majority of files that eventually lose relevance, introducing noise into our training data. Additionally, using the entire project history might accumulate outdated hub files. Despite their marginal importance at a later point in time, their historically high degree of connectivity exaggerates their structural significance within the graph. This hypothesis is supported by GraphCP's performance on the Moby dataset. As can be seen in Table 6.7 and in Figure 6.3 (b) and (c), the performance differs from training on the one year graph with only 4 495 training edges and training on the three years graph with 20 949 edges causing GraphCP's AUPR score to drop substantially from 0.96 to 0.78. Furthermore, this hypothesis is also statistically confirmed by the one-sided Wilcoxon signed-rank test. The result (mentioned in Section 6.3.2.2) allows us to confidently reject the null hypothesis and confirm that training on more recent activities yields a significantly higher performance than training on the entire project history.

Overall, we can see that the project structure and developer activities evolve, making older data less meaningful. Our findings suggest that it suffices to only use the most recent activities for training. This enables the model to learn meaningful structural patterns and achieve higher predictive precision than the comparison baselines but also than the original GraphCP.

6.4.3 Performance Variations

In our analysis, we observe GraphCP increasingly performed better with increasing graph size, with one exception regarding Atom and Deno. Yet, as the baseline method PA sometimes performs even better than GraphCP, we can assume that the performance strongly correlates with the network topology, particularly with hub-dominant and peripheral connections.

The contrast between Keras, i.e., the lowest performing project, and Nextcloud, i.e., the best performing project, illustrates that network structure is an important signal. Based on the power-law test results in Section 6.1, Nextcloud's node degree distribution follows a heavy-tailed distribution. This is also supported by the high imbalance ratio (0.91) of the author and file nodes. On this network, the high PA AUPR score (0.7) indicates that connectivity is a dominant signal, which GraphCP leverages. Our model's performance for Nextcloud, however, reveals that it additionally learns meaningful features of the nodes, which makes it more precise than simply choosing the most popular node (reaching an AUPR score of 0.86). Conversely, Keras is not only substantially smaller than Nextcloud, in terms of training edges, but also differently structured. Based on the power-law test in Table 6.1, we cannot state that the node degree distribution is heavy-tailed, as the comparison to the exponential distribution already reveals inconclusive results. Furthermore, as we have described in Section 5.4.1, Keras is more compact and balanced with a node imbalance score of 0.34. Furthermore, unlike the other projects, Keras not only holds the most balanced node types but also contains more author than file nodes. This indicates an overall different project dynamic than the other examined project graphs. The absence of edges between low-degree nodes, as revealed in Section 5.7, and the non-heavy-tailed distribution suggest that the node degrees are more evenly distributed. As we have also seen in Figure 6.3 and Table 6.8, although GraphCP performs better on a smaller, more recent subset of the training graph than GraphCP$_{Total}$, its AUPR score is still substantially smaller than for the other graphs. This suggests that for this network, the structural signals are too meaningless, making the more complex GNN approach reach its limits by attempting to detect latent patterns from a small training space. In this case, the degree-based multiplication heuristic PA performs marginally better, as it avoids potential overfitting or hallucination, which can occur in deep learning approaches.

The network structure also impacts the difficulty of the classification task. In large datasets, such as Nextcloud, with a high node type imbalance, a heavy-tailed node degree distribution, and the absence of strictly low-degree edges, the vast majority of randomly sampled negative edges are trivial. That is, it is highly unlikely that they exist, which facilitates GraphCP's ability to distinguish between the actual positive test edges and the random negative edges. Regarding Keras, where the distribution tends to be more evenly distributed in terms of node degrees but also node type imbalance, the sampled negatives are less trivial. This is also visible in Moby. Although Nextcloud and Moby are similar in train size, Moby contains a notable number of edges between low-degree nodes. A stronger presence of low-degree edges might lead to more challenging negative samples, which are not as trivial to classify as those in Nextcloud. Nevertheless, GraphCP performs relatively well in the sparse environment, reaching an AUPR score of 0.31 (see Table 6.6).

Finally, it is important to note that a direct comparison between the performance AUPR scores is challenging as the test-set ratios individually differ across the projects. The Keras test set only made up approximately 4% of the edges, whereas the Nextcloud test set contains 10% of the total observed edges that we use.

Overall, the results suggest that GraphCP's performance variation is influenced less by graph size itself but more by structural properties, such as node degree distribution and node type imbalance. Especially, these properties impact the complexity of the negative samples.

6.4.4 Metric Discrepancy

Although our model did not perform the best on every project, in terms of AUPR, it almost always reached the highest AUROC score, except for the Keras project. As discussed, we sample 10 times more negative test edges than positive test edges, making the entire test set highly imbalanced, but also more realistic, as described in Section 5.4.2. In such scenarios, AUROC favors accurate classification of positive examples, at the cost of significantly misclassifying negative examples, making it less sensitive to class-imbalance [27]. Yet, we can still see that our approach ranks positive and negative samples better compared to our baseline methods. However, besides Nextcloud, the AUPR score is always significantly lower than the AUROC score, particularly regarding GraphCP. This perfectly shows that AUROC can be misleading and too optimistic. The methods are able to classify a large amount of negatives, which are probably very easy, given the random sampling method. Yet, considering the lower AUPR score, we can be sure that the model is not able to precisely classify edges. That is, besides the small amount of positive edges, the models still classified edges between nodes that actually do not exist.

6.4.5 Evaluation of the Research Question

Having analyzed and discussed the structural dependencies, performance variations, and experimental results, we can finally address our research question: To what extent can GraphSAGE in combination with a neural network be applied to perform link prediction on developer networks? The applicability of our approach cannot be summarized by a binary conclusion. Instead, it depends on the network topology and connectivity, as well as the temporal window of the training data.

Globally, our approach, i.e., GraphCP, shows promising results. It consistently outperforms the Random and Logistic Regression (LR) baselines across all investigated datasets. While it remains strongly competitive with Preferential Attachment (PA), it reveals valuable behavior on the sparse part of the network. That is, its predictive performance on test edges between nodes with a low degree. On this set of edges, GraphCP constantly substantially outperforms PA. Although GraphCP significantly outperforms LR globally, it is highly competitive on peripheral links. This suggests that a non-linear neural network as a classification head, like in GraphCP, is better than a linear hyperplane.

GraphCP's applicability is most prominent when it is trained only on a subset of the entire project history. Our experiment on the three datasets Keras, Moby, and Nextcloud demonstrates that training GraphCP on more recent activities from the training graph achieves significantly better results than using the entire training graph. Furthermore, in the largest datasets, i.e., Moby and Nextcloud, this approach even enabled GraphCP to constantly outperform every baseline method. Although this was only tested on a subset of datasets, the results strongly suggest that the model performs the best when it learns from current activities rather than the entire history that sometimes ranges over a decade.

However, there are clear limitations derived from the graph topology, as well as the representation learning technique. As we have seen on the Keras project, GraphCP struggles more on networks that do not contain significant hub-nodes but are more evenly distributed. In such balanced datasets, the structural signal may be too weak for GraphCP to provide an advantage over a simple popularity-based heuristic. Furthermore, our approach is limited by the standard definition of GraphSAGE, which requires us to treat our original heterogeneous bipartite developer network as a homogeneous network. While the results indicate that the model is effective in differentiating node types only through features, we see potential for improvement by using true heterogeneous architectures or using more entities and connections, not only author and file nodes.

6.5 Threats to Validity

Before and during our evaluation, it is important to be aware of several aspects that threaten our validity. As we limited our approach not only in data but also in resources we now emphasize how to put the results right into context. To achieve this, we discuss the threats to internal and external validity in the following section.

6.5.1 Threats to Internal Validity

As our model heavily relies on GitHub data, which initially might contain biases or may be of poor quality, we must be aware of the fact that it underwent a lot of preprocessing steps (see Section 5.2.1 and Section 5.2.2). This includes bot removal, but also removing duplicate users, or merging users and their contributions if they were identified as the same exact person.

Additionally, it is important to note that the edges between authors and files do not necessarily mean that the exact author edited the file, but revisited it, simply taking the role of the committer. However, in the case of the Deno graph, both the name of the source node and the name saved in the edge attribute `author.name` were always the same. This indicates that the Deno graph does not pose a threat in this context, nevertheless, it might be the case for the other graphs we consider for our evaluation.

Another threat to internal validity is that, as explained in Section 5.3.1, we deduplicate our edges, which blurs the actual evolution in the projects. Instead of capturing the entire development, we only retain individual interactions, even though some may occur more

frequently than others. Consequently, we treat all connections equally, although they may carry different significance.

6.5.2 Threats to External Validity

Conducting a comparative analysis with different input graphs to answer our research question validates our findings to a certain degree. The analysis reveals characteristics that might impact the performance and thus, enhance the proposition of whether the approach is generalizable in our context.

However, our research question investigates to what extent we can apply GraphSAGE to developer networks. Thus, we focused only on author-file connections, in order to apply the standard GraphSAGE model. Therefore, we treat our heterogeneous bipartite graph as homogeneous and distinguish the nodes based on their `type` attribute. A potential threat to external validity could arise once we introduce more information to the input graph. That is, adding issue nodes and issue connections to the graph, i.e., adding complexity. This could impact the model behavior and requires adjustments to the approach. Another threat to external validity is the random negative sampling. Although this sampling strategy is easy to implement and fast, it might also create trivial negative cases. Especially on large graphs, it is likely to find two nodes that are further away and only reachable via many hops. This makes it easier for the prediction model to identify the representative distance and to correctly classify them. Hard negatives would be nodes that are 3 hops away.

6.6 Future Work

As we have seen, GraphCP shows promising results. Our approach is able to learn latent patterns between the entities and is highly competitive with the topological heuristic PA. Especially on edges between peripheral nodes, our model substantially outperforms PA in terms of the average precision. Moreover, GraphCP significantly outperforms PA when trained on recent contribution patterns rather than the entire project history. As our approach treats the input as a homogeneous graph with only a two-dimensional initial feature vector to distinguish between authors and files, these results demonstrate that the topological structure already provides a strong signal for contribution prediction. However, to make the theoretical scenarios in Chapter 4 technically more feasible, we need to make GraphCP more context-aware.

In the context of contribution recommendation in OSS projects, identifying the right developer for a specific file requires an understanding of semantic similarity. Incorporating source code as an additional feature would allow GraphCP to identify a similarity between different files that are topologically distant from one another.

Regarding change impact analysis, issues that highlight problems in a specific file can help identify other files that may require corresponding changes. By introducing issue nodes and their discussion content to the input graph, the model could identify the relationships between files that address the same underlying topic.

Finally, for anomaly detection, it is essential to be aware of the different modules and who works in system-critical areas. A context-aware model can alert as soon as a link between an author and an artifact appears unusual. To this end, the model requires semantic metadata regarding the functional role of an artifact, such as its module affiliation or its level of system-criticality. This context would allow the model to distinguish between a typical developer behavior and a potentially risky deviation.

To enable these scenarios, we need to shift towards a more heterogeneous approach. Currently, we utilize GraphSAGE in its standard form, hence GraphCP treats the network as homogeneous. This limits the feature space for author and file nodes to be the same, only containing their `id` and `type` attributes. A heterogeneous transition allows for type-specific feature spaces. That is, author nodes could contain different features than file nodes. Furthermore, a heterogeneous approach enables us to expand our graph. Instead of using only author and file nodes, we could add issue nodes, and their corresponding links, but also connections within the same node kind, such as those between files. So far, the initial node feature vector consisted of their `id` and `type`. To include semantic similarity between files, we could include the source code of the files or their module membership in the project. For issues, we could incorporate information about the discussion and the files they address. All this information can be encoded into a numerical format using specialized encoder architectures (e.g, CodeBERT [18]). As described in related studies [14, 34, 49, 57] in Chapter 3, having heterogeneous graph data as input, we would need to train independent embedding models per edge relation and later aggregate their representation into one single node representation. This way, we can incorporate more information about the entities and enable the model to learn specific semantic relations between entities. Furthermore, based on our findings in Section 6.4.2, future work should prioritize the most recent activities to train GraphCP. This eliminates outdated noise, such as developers who only worked on the project at the beginning of the project's lifetime.

In general, while our results confirm the strength of the topological signal, the logical next step is to close the semantic gap through a heterogeneous approach. Future work could build on our GraphCP approach by incorporating more information as discussed here, and develop a more context-aware link prediction model on developer networks.

7

Conclusion

This thesis provides an extensive discussion of practical application scenarios for link prediction models which can enhance OSS project coordination. Based on these requirements develop and evaluate GraphCP as a novel deep learning-based proof of concept. In particular, we evaluate to what extent a deep learning framework, specifically GraphSAGE, combined with a non-linear neural network as classification head, can predict future contributions. The concept of developer networks enables a graph-structured representation of community activities within a project. This representation is used as input data for our approach. The performance of GraphCP was compared against the random baseline as a trivial comparison, the logistic regression baseline (LR) as an alternative to the non-linear classification in GraphCP, as well as the simple popularity-based heuristic, preferential attachment (PA).

GraphCP establishes a foundational proof of concept by modeling bipartite author-file interactions as a homogeneous graph. The evaluation across seven different OSS projects revealed three notable findings. First, GraphCP consistently outperformed Random and LR baselines and remained highly competitive with PA. Second, an important finding is that GraphCP always outperformed PA on peripheral node edges. This indicates that GraphCP is able to find non-trivial links and does not consistently follow the popularity-based trend. Third, in a targeted experiment, we further demonstrated that GraphCP performs substantially better when trained on a smaller and more recent subset of activities. This suggests that a more current state of the developer network is more meaningful, leading to a better predictor for the future, rather than using the entire history.

In summary, while simpler rule-based or linear models offer a competitive baseline, the more sophisticated non-linear deep learning method, GraphCP, allows for a more explorative and precise understanding of developer networks. While the various application possibilities explored in this thesis, i.e., recommendation systems, change impact analysis, or anomaly detection, require more data integration and a heterogeneous approach, GraphCP serves as a viable technical foundation. Consequently, this thesis presents a technical proof of concept and real-world utility helping to improve the collaboration management and quality of OSS projects.

A

Appendix

A.1 Data Description

Table A.1: Overview of Edge Attributes per Relation

Category	Attribute	Cochange	Issue
Shared Data	source	✓	✓
	target	✓	✓
	issue.id	✓	✓
	issue.state	✓	✓
	relation	✓	✓
	weight	✓	✓
	artifact	✓	✓
	artifact.type	✓	✓
	author.name	✓	✓
	committer.name	✓	✓
	date	✓	✓
	creation.date	✓	✓
	closing.date	✓	✓
	hash	✓	✓
	event.name	✓	✓
	file	✓	✓
	type	✓	✓
Cochange-Specific	changed.files	✓	–
	added.lines	✓	–
	deleted.lines	✓	–
	diff.size	✓	–
	artifact.diff.size	✓	–

A.2 Hyperparameter Search Results

The following tables summarize the intermediate results from the hyperparameter search per project. We highlighted the best combination and the embedding configuration id which appeared the most in bold. **L ID** and **E ID** denote the hyperparameter configuration id for the link prediction model and the embedding, i.e., the GNN, model.

Table A.2: Best Intermediate Results for Keras

L ID	E ID	AUROC	AUPR
0	77	0.88	0.47
1	59	0.85	0.43
2	32	0.86	0.46
3	**47**	**0.88**	**0.51**
4	38	0.86	0.46
5	17	0.88	0.48
6	37	0.87	0.44
7	82	0.87	0.49
8	41	0.87	0.46
9	92	0.85	0.44
10	38	0.86	0.46
11	26	0.85	0.47

Table A.3: Best Intermediate Results for Atom

L ID	E ID	AUROC	AUPR
0	82	0.93	0.67
1	32	0.91	0.62
2	56	0.91	0.64
3	41	0.92	0.68
4	88	0.91	0.64
5	11	0.90	0.62
6	**65**	**0.92**	**0.69**
7	64	0.89	0.63
8	31	0.88	0.63
9	5	0.91	0.65
10	28	0.89	0.65
11	92	0.90	0.64

Table A.4: Best Intermediate Results for Deno

L ID	E ID	AUROC	AUPR
0	53	0.88	0.53
1	82	0.88	0.55
2	**80**	**0.90**	**0.59**
3	76	0.88	0.58
4	36	0.85	0.55
5	77	0.87	0.53
6	29	0.88	0.54
7	2	0.84	0.53
8	29	0.88	0.54
9	2	0.87	0.54
10	14	0.86	0.53
11	7	0.85	0.54

Table A.5: Best Intermediate Results for OpenSSL

L ID	E ID	AUROC	AUPR
0	26	0.91	0.54
1	61	0.96	0.58
2	61	0.96	0.58
3	61	0.96	0.57
4	2	0.93	0.56
5	82	0.96	0.59
6	28	0.92	0.51
7	26	0.91	0.57
8	48	0.95	0.57
9	**19**	**0.93**	**0.61**
10	56	0.93	0.56
11	61	0.95	0.57

Table A.6: Best Intermediate Results for VSCode

L ID	E ID	AUROC	AUPR
0	70	0.87	0.63
1	**36**	0.87	0.65
2	54	0.90	0.67
3	54	0.91	0.66
4	**36**	0.87	0.54
5	**36**	0.86	0.66
6	86	0.90	0.67
7	**36**	0.88	0.65
8	18	0.88	0.65
9	**36**	**0.87**	**0.69**
10	54	0.89	0.67
11	54	0.90	0.66

Table A.7: Best Intermediate Results for Moby

L ID	E ID	AUROC	AUPR
0	6	0.91	0.60
1	64	0.93	0.66
2	81	0.94	0.64
3	65	0.91	0.62
4	11	0.95	0.63
5	**83**	0.94	0.65
6	**83**	0.95	0.64
7	**83**	**0.96**	**0.66**
8	93	0.86	0.63
9	77	0.93	0.63
10	95	0.93	0.65
11	81	0.95	0.63

Table A.8: Best Intermediate Results for Nextcloud. For this network two combinations were limited in our time constraint and did not finish. Therefore, we only have 10 configurations reported.

L ID	E ID	AUROC	AUPR
0	78	0.98	0.78
1	49	0.97	0.84
2	**63**	**0.98**	**0.86**
3	**63**	0.97	0.80
4	11	0.98	0.83
5	49	0.97	0.79
6	**63**	0.97	0.79
7	34	0.98	0.84
8	11	0.98	0.82
9	**63**	0.97	0.83
10	**63**	0.97	0.77
11	95	0.97	0.83

Statement on the Usage of Generative Digital Assistants

For this thesis, we have used Grammarly[1], Chat-GPT[2], and Gemini[3] as assistance for text rewriting and technical consultation. Chat-GPT and Gemini were used to find a better wording by describing the subject or a better articulation of our thoughts. The content was written independently. In addition Chat-GPT and Gemini were used as a supporting tool to verify our understanding of the technical concepts or to generate code snippets, especially for specialized functions. No long AI-generated text or code blocks were included verbatim.

We are aware of the potential dangers of using these tools and have used them sensibly with caution and with critical thinking.

1 https://www.grammarly.com
2 https://chatgpt.com
3 https://gemini.google.com/app

Bibliography

[1] Jeff Alstott, Ed Bullmore, and Dietmar Plenz. "powerlaw: A Python Package for Analysis of Heavy-Tailed Distributions." In: *PLoS ONE* 9.1 (2014), e85777.

[2] Anchore. *Open Source Software Security: Risks, Best Practices & Tools*. Accessed: 2026-28-03. 2024. URL: https://anchore.com/blog/open-source-software-security-in-software-supply-chain/.

[3] ApX Machine Learning. *Vector Dimensionality Understanding - Vector Databases & Semantic Search*. APXML Blog. Accessed: 2026-03-17. 2024. URL: https://apxml.com/courses/vector-databases-semantic-search/chapter-1-embeddings-vector-spaces-foundation/vector-dimensionality-understanding.

[4] Yilin Bi, Xinshan Jiao, Yan-Li Lee, and Tao Zhou. "Inconsistency among evaluation metrics in link prediction." In: *PNAS Nexus* 3.11 (2024), pgae498.

[5] Thomas Bock. "Emerging Organizational Patterns in Evolving Open-Source Software Projects - An Analysis of Developer Activity and Coordination." PhD thesis. Saarland University, 2024.

[6] Emperor Brains. *The Impact of Open Source on Software Development: Collaboration Over Competition*. Medium. Accessed: 2026-28-03. 2024. URL: https://medium.com/@emperorbrains/the-impact-of-open-source-on-software-development-collaboration-over-competition-4d8ce85c3bfa.

[7] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. "Spectral Networks and Locally Connected Networks on Graphs." In: *arXiv preprint arXiv:1312.6203* (2014). arXiv: 1312.6203 [cs.LG].

[8] Jonathan Bryan and Pablo Moriano. "Graph-based machine learning improves just-in-time defect prediction." In: *PLOS ONE* 18.4 (2023), e0284077.

[9] Nikolaj Buhl. *Logistic Regression: Definition, Use Cases, Implementation*. Encord Blog. Accessed: 2026-01-15. 2023. URL: https://encord.com/blog/what-is-logistic-regression/.

[10] ClearObject. *Rules-based vs. Deep Learning: A Powerful Synergy in Modern AI*. Clearobject Blog. Accessed: 2025-12-23. URL: https://www.clearobject.com/rules-based-vs-deep-learning/.

[11] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. "Unsupervised Cross-lingual Representation Learning at Scale." In: *CoRR* abs/1911.02116 (2019).

[12] RocketMe Up Cybersecurity. *Using Behavioral Analytics to Identify Anomalous User Activity*. Medium. Accessed: 2026-01-12. 2024. URL: https://medium.com/@RocketMeUpCybersecurity/using-behavioral-analytics-to-identify-anomalous-user-activity-6788db431f71.

[13] Nur Nasuha Daud, Siti Hafizah Ab Hamid, Muntadher Saadoon, Firdaus Sahran, and Nor Badrul Anuar. "Applications of link prediction in social networks: A review." In: *Journal of Network and Computer Applications* 166 (2020), p. 102716.

[14] Yujie Fan, Mingxuan Ju, Chuxu Zhang, Liang Zhao, and Yanfang Ye. "Heterogeneous Temporal Graph Neural Network." In: *CoRR* abs/2110.13889 (2021).

[15] Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. huff. "Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE Tools." In: *Perspectives on Free and Open Source Software*. 2007, pp. 245–264.

[16] GeeksforGeeks. *Rule-Based System vs. Machine Learning System*. GeeksforGeeks Blog. Accessed: 2025-12-23. 2025. URL: https://www.geeksforgeeks.org/machine-learning/rule-based-system-vs-machine-learning-system/.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[18] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. "GraphCodeBERT: Pre-training Code Representations with Data Flow." In: *CoRR* abs/2009.08366 (2020).

[19] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive representation learning on large graphs." In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2017, 1025–1035.

[20] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. "Heterogeneous Graph Transformer." In: *Proceedings of The Web Conference 2020*. Association for Computing Machinery, 2020, 2704–2710.

[21] Weizhi Huang, Wenkai Mo, Beijun Shen, Yu Yang, and Ning Li. "CPDScorer: Modeling and Evaluating Developer Programming Ability across Software Communities." In: 2016.

[22] Alpana Jain. *The Role of Open Source in Modern Software Development*. Accessed: 2026-21-03. 2024. URL: https://medium.com/@alpanajain/the-role-of-open-source-in-modern-software-development-efc9812ab4b0.

[23] Mitchell Joblin. "Structural and Evolutionary Analysis of Developer Networks." PhD thesis. University of Passau, 2017.

[24] Mitchell Joblin and Sven Apel. "How Do Successful and Failed Projects Differ? A Socio-Technical Analysis." In: *ACM Trans. Softw. Eng. Methodol.* 31.4 (2022), pp. 1–24.

[25] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. "Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics." In: IEEE Press, 2017, pp. 164–174.

[26] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. "Evolutionary Trends of Developer Coordination: A Network Approach." In: *CoRR* abs/1510.06988 (2015).

[27] I. Bhargavi Kalyani, A. Rama Prasad Mathi, and Niladri Sett. "Evaluating link prediction: new perspectives and recommendations." In: *International Journal of Data Science and Analytics* 20.7 (2025), 6855–6886.

[28] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks." In: *CoRR* abs/1609.02907 (2016).

[29] Ajay Kumar, Shashank Sheshar Singh, Kuldeep Singh, and Bhaskar Biswas. "Link prediction techniques, applications, and performance: A survey." In: *Physica A: Statistical Mechanics and its Applications* 553 (2020), p. 124289.

[30] Yuki Li. *AUROC and AUPRC*. Medium. Accessed: 2026-01-26. 2025. URL: https://medium.com/@yukims19/auroc-and-auprc-b25183827e5a.

[31] Zhixing Li, Yue Yu, Minghui Zhou, Tao Wang, Gang Yin, Long Lan, and Huaimin Wang. "Redundancy, Context, and Preference: An Empirical Study of Duplicate Pull Requests in OSS Projects." In: *IEEE Transactions on Software Engineering* 48.4 (2022), pp. 1309–1335.

[32] Haibin Liu, Mu Qiao, Daniel Greenia, Rama Akkiraju, Stephen Dill, Taiga Nakamura, Yang Song, and Hamid Motahari Nezhad. "A Machine Learning Approach to Combining Individual Strength and Team Features for Team Recommendation." In: 2014, pp. 213–218.

[33] Luis Lopez-Fernandez, Gregorio Robles, and Jesus Gonzalez-Barahona. "Applying Social Network Analysis to the Information in CVS Repositories." In: *International Workshop on Mining Software Repositories* (2004), pp. 101–105.

[34] Yu Luo, Weifeng Xu, and Dianxiang Xu. "Detecting code vulnerabilities with heterogeneous GNN training." In: *Int. J. Inf. Secur.* 24.5 (2025).

[35] Yao Ma and Jiliang Tang. *Deep Learning on Graphs*. Cambridge University Press, 2021.

[36] Bernardo Costa e Marcelo Trojahan. *What are the differences between AI, ML, DL, and Gen AI?* e-core. Accessed: 2025-09-11. 2024. URL: https://www.e-core.com/na-en/articles/differences-between-ai-ml-dl-gen-ai.

[37] Ninad Mathpati. *Risks of Open-Source Software*. Cobalt Blog. Accessed: 2025-12-11. 2023. URL: https://www.cobalt.io/blog/risks-of-open-source-software.

[38] Elliot McClenaghan. *The Wilcoxon Signed-Rank Test*. Blog-Technology Networks Informatics. Accessed: 2026-02-22. 2024. URL: https://www.technologynetworks.com/informatics/articles/the-wilcoxon-signed-rank-test-370384#D2.

[39] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. "Predicting failures with developer networks and social network analysis." In: Association for Computing Machinery, 2008, 13–23.

[40] Dai Quoc Nguyen, Tu Dinh Nguyen, Dat Quoc Nguyen, and Dinh Phung. "A Novel Embedding Model for Knowledge Base Completion Based on Convolutional Neural Network." In: Association for Computational Linguistics, 2018, 327–333.

[41] Dmitry Paranyushkin. *Types of Networks: Scale-Free, Power Law, and Degree Distribution*. Blog-NodusLabs. Accessed: 2026-01-30. 2023. URL: https://support.noduslabs.com/hc/en-us/articles/4402048501266-Types-of-Networks-Scale-Free-Power-Law-and-Degree-Distribution.

[42] Pisol Ruenin, Morakot Choetkiertikul, Akara Supratak, and Suppawong Tuarob. "Automatic recommendation of developers for open-source software tasks using knowledge graph embedding." In: *Science, Engineering and Health Studies* 16.— (2022), p. 22020006.

[43] Pisol Ruenin, Morakot Choetkiertikul, Akara Supratak, and Suppawong Tuarob. "TeReKG: A temporal collaborative knowledge graph framework for software team recommendation." In: *Know.-Based Syst.* 289.C (2024).

[44] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. "Modeling Relational Data with Graph Convolutional Networks." In: Springer-Verlag, 2018, 593–607.

[45] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. "When do changes induce fixes?" In: *SIGSOFT Softw. Eng. Notes* 30.4 (2005), 1–5.

[46] GitHub Staff. *Sunsetting Atom*. GitHub. Accessed: 2026-02-10. 2022. URL: https://github.blog/news-insights/product-news/sunsetting-atom/.

[47] Maarten can Steen. *Graph Theory and Complex Networks: An Introduction*. Maarten can Steen, 2010.

[48] John Stegeman. *What Is a Knowledge Graph?* neo4j. Accessed: 2025-09-11. 2024. URL: https://neo4j.com/blog/knowledge-graph/what-is-knowledge-graph/.

[49] Yanchun Sun, Jiawei Wu, Xiaohan Zhao, Haizhou Xu, Ye Zhu, Zhenpeng Chen, Sihan Wang, Huizhen Jiang, and Gang Huang. "Automatically Deriving Developers' Technical Expertise from the GitHub Social Network." In: *ACM Trans. Softw. Eng. Methodol.* 35.4 (2025).

[50] Ashish Sureka, Atul Goyal, and Ayushi Rastogi. "Using social network analysis for mining collaboration data in a defect tracking system for risk and vulnerability analysis." In: Association for Computing Machinery, 2011, 195–204.

[51] Karishma Thakrar and Aniket Chauhan. *GitHub Stargazers | Building Graph- and Edge-level Prediction Algorithms for Developer Social Networks*. 2025.

[52] Suppawong Tuarob, Noppadol Assavakamhaenghan, Waralee Tanaphantaruk, Ponlakit Suwanworaboon, Saeed-Ul Hassan, and Morakot Choetkiertikul. "Automatic team recommendation for collaborative software development." In: *Empirical Software Engineering* 26.4 (2021), p. 64.

[53] Nick Vidal. *Key insights from the 2025 State of Open Source Report*. Accessed: 2026-25-03. 2025. URL: https://opensource.org/blog/key-insights-from-the-2025-state-of-open-source-report.

[54] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. "Heterogeneous Graph Attention Network." In: Association for Computing Machinery, 2019, 2022–2032.

[55] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. "Predicting build failures using social network analysis on developer communication." In: IEEE Computer Society, 2009, pp. 1–11.

[56] Jiafei Yan, Hailong Sun, Xu Wang, Xudong Liu, and Xiaotao Song. "Profiling Developer Expertise across Software Communities with Heterogeneous Information Network Analysis." In: Association for Computing Machinery, 2018.

[57] Luwei Yang, Zhibo Xiao, Wen Jiang, Yi Wei, Yi Hu, and Hao Wang. "Dynamic Heterogeneous Graph Embedding Using Hierarchical Attentions." In: *Advances in Information Retrieval*. Springer International Publishing, 2020, pp. 425–432.

[58] Yang Yang, Ryan N. Lichtenwalter, and Nitesh V. Chawla. "Evaluating link prediction methods." In: *Knowledge and Information Systems* 45.3 (2014), 751–782.

[59] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. "Heterogeneous Graph Neural Network." In: Association for Computing Machinery, 2019, 793–803.

[60] Fanjin Zhang, Xiao Liu, Jie Tang, Yuxiao Dong, Peiran Yao, Jie Zhang, Xiaotao Gu, Yan Wang, Bin Shao, Rui Li, and Kuansan Wang. "OAG: Toward Linking Large-scale Heterogeneous Entity Graphs." In: Association for Computing Machinery, 2019, 2585–2595.

[61] sabrepc. *Rule-Based Systems vs Machine Learning and AI*. sabrepc Blog. Accessed: 2025-12-11. 2025. URL: https://www.sabrepc.com/blog/Deep-Learning-and-AI/machine-learning-system-vs-rule-based-system.