UNIVERSITÄT KOBLENZ · LANDAU

Leveraging Graph Representation Learning for Software Engineering

Masterarbeit

zur Erlangung des Grades eines Master of Science
im Studiengang Web and Data Science

vorgelegt von

Aditya Mehta

Erstgutachter:	Prof. Dr. Ralf Lämmel Institut für Informatik
Zweitgutachter:	Johannes Härtel Institut für Informatik

Koblenz, im August 2022

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet–Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☑	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☑	☐

Koblenz, 02.08.2022 .... AMehta
(Ort, Datum) (Aditya Mehta)

Zusammenfassung

Deep-Learning-Techniken (DL) können moderne Software-Engineering-Prozesse (SE) verändern, um datengesteuerte, selbstlernende und intelligente Softwaresysteme zu entwickeln. Techniken des Graphenrepräsentationslernens (GRL) helfen bei der Erstellung graphenbasierter Darstellungen von Daten (Code) für die automatische Merkmalsextraktion. Ziel dieser Studie ist es, eine systematische Literaturübersicht zum Thema "GRL für SE" zu erstellen, die der üblichen empirischen SE-Forschungsmethodik folgt. Die Studie bietet Einblicke in die Arten von Softwarecode, deren grafische Repräsentationen, DL-Modelle, Vorteile und Herausforderungen bei der Anwendung von GRL für verschiedene SE-Aufgaben. Darüber hinaus identifiziert diese Studie Arten von Datensätzen, Sprachen und Bewertungsmetriken, die in Experimenten verwendet werden, und stellt potenzielle Anwendungsfälle für zukünftige Arbeiten vor.

Abstract

Deep learning (DL) techniques can transform modern software engineering (SE) processes to make data-driven, self-learning, and intelligent software systems. Graph representation learning (GRL) techniques help to create graph-based representations of data (code) for automated feature extraction. This study aims to conduct a systematic literature review on the topic "GRL for SE", following standard empirical SE research methodology. This study provides insights about types of software code, their graphical representations, DL models, benefits, and challenges of adopting GRL for various SE tasks. Additionally, this study identifies types of datasets, languages and evaluation metrics used in experiments and presents potential future work use cases.

Acknowledgements

I am grateful to my supervisors Prof. Ralf Lämmel and Mr. Johannes Härtel for sharing this research opportunity. The discussions with them helped to shape and evolve the work with feedback at various stages. I am thankful to my deep learner friends Bhupender and Praveen, who helped with technical innovative ideas to conduct research effectively, and proofreading. On personal note, I am thankful to my family, who is supportive to let me go abroad and study here. This is a work in service to the Lord Krishna, and I would dedicate this to my mother, who could not finish her Ph.D. thesis due to marriage. For a joyful learning journey, I'm thankful to science community as this work stands on the shoulders of giants.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Modern engineering systems are increasingly adopting deep learning (DL) techniques, with the geometry of the underlying data becoming increasingly important in the DL field. Geometric deep learning (GDL) is a heavily-researched area with data representations which go beyond Euclidean space [1]. The basic premise is that elements of any system are connected in a network structure. This premise helps to approach problems, which are sub-optimally solved using traditional approaches [2]. For instance, the GDL has contributed to understanding structure-based protein unfolding [3], using the graph neural networks (GNN) in unprecedented fast drug discovery for SARS-CoV-2 (Covid-19) disease [4].

Adoption of DL techniques in SE is leading a paradigm shift toward intelligent software systems. This approach handles fuzzy information, coding conventions, and grammar of languages without relying on manual heuristics as involved in traditional analysis [5]. Code repositories contain a large amount of software artifacts data (code), which can help machines learn automatically to reason about the code [6]. DL offers transformation methods for SE due to reasons including (i) scale of analyzable software data, (ii) automated feature engineering, (iii) transfer learning capabilities of DL to SE use cases, and (iv) robustness and scalability. These factors indicate a high potential for DL to improve the traditional SE process [7].

However, how can various SE tasks leverage GDL effectively, remains still not adequately addressed. In the interplay between DL and SE fields, this study focuses on the theme "deep learning for software engineering" (DL4SE): the use of deep learning techniques across various SE use cases. This study attempts to answer the question: is it possible to treat software as "data" for deep neural networks? Based on the insights from this literature survey, we posit the answer as a confident "yes". In other words, we present neural software analysis as an approach to treat program analysis to learn to reason about software code [5].

1.2 Problem Description

The advent of deep learning led to a paradigm shift from user-defined, heuristics-based feature extraction to automatic feature learning [8]. The set of techniques to implement automated feature learning based on the underlying code treated as graphs is known as graph representation learning (GRL). Software code analysis has been initially approached using manual, heuristics-based methods and later by using natural language processing (NLP) techniques representing code as a sequence of text tokens [6], or as parse trees [9]. Code can be represented as a graph with nodes as elements of the program while edges illustrate relationships among them [10]. Graph representation can help capture syntactic and semantic attributes, which could also reduce large training cycles [10], and capture the dependency structure of the flow in a program [11]. Deep learning frameworks can learn to reason on code with graph representation as input for various downstream SE tasks, such as code classification, summarization, vulnerability detection, and duplicate detection [11]. Automated programming systems offer services from simple use cases such as syntax and code style checking to advanced complex use cases such as proposing names of variables, methods, and even code generation [12].

1.3 Aim of the Thesis

This thesis aims to identify, collect, and analyze the code represented as graph-based formats and their corresponding applications in the SE domain using DL techniques [13]. We follow the empirical systematic literature review (SLR) methodology to conduct this study.

1.4 Research Questions

1. Which types of graph-based representations of code are used in empirical research?

2. Which SE tasks leverage graph representation learning?

   Rationale:

   - To determine applications of GRL in SE

3. Which types of learning types, models and techniques of GRL are widely used?

   Rationale:

   - To identify the role of GRL, benefits and implications
   - To determine non-GRL or traditional methods being outperformed

4. Which datasets, languages, and evaluation metrics do experiments use?

   Rationale:

   - To summarize available datasets, programming languages, and evaluation metrics used in GRL4SE applications

1.5 Contributions

DL techniques have proven to be effective in a variety of applications. Since a wide range of data representations, architectures, models, and techniques are available, selecting the most appropriate configuration for any specific SE task remains a challenge [14]. For instance, for the SE task program classification, what is the suitable graph-based representation, model, technique, and evaluation metric?

This thesis work helps the research community by offering a comprehensive overview of the graph-based deep learning approaches for SE. As per the best of the authors' knowledge, existing surveys are limited to the topic of DL for SE or similar domains but are not focused on graph-based DL. This work is the first to conduct a systematic literature review (SLR) on GRL for SE. Specifically, our work provides information enabling the reader with underlying theoretical concepts and practical implementation techniques to apply GRL to various SE tasks.

1.6 Structure of the thesis

Chapter 2 provides supporting material to enable the reader to understand the fundamental concepts and terminologies.

Chapter 3 provides existing research literature and empirical studies, which try to answer the research questions in related settings.

Chapter 4 gives insights into the research approach, specifically for conducting the systematic literature review.

Chapter 5 provides a detailed analysis of various studies attempting to find answers to proposed research questions.

Chapter 6 discusses the overall concluding remarks, limitations and threats to validity. This chapter also provides potential future work possibilities.

Chapter 2

Background and Fundamentals

2.1 Representation Learning (RL)

Underlying data representation or features heavily determine the performance of DL algorithms [15]. Traditional ML techniques depended on expert human intervention for manually extracting features, and hence unable to be fed with raw data to learn representations [16]. Representation learning enables a machine to process the raw data to learn and discover data representations automatically, i.e. implementing automatic feature extraction and learning [16].

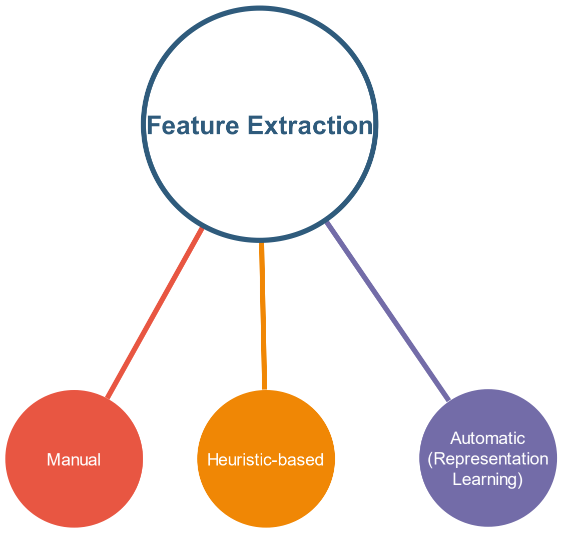


Figure 2.1 Overview: Feature Extraction

2.1.1 Deep Learning (DL)

Deep learning methods are a specific group of techniques in representation learning, with the notion of "deep" implying a large number of intermediate processing layers [15]. DL systems use various transformations on data with a set of learnable parameters to learn to reason from data. The transformations are based on mathematical, linear and non-linear computation methods, with increasingly complex levels of abstractions to create representations of data usable for DL algorithms [15], [16]. DL helps to create computational systems to learn data representations in multi-layered or hierarchical levels of abstraction [16]. Such systems can be "trained" for specific tasks by updating the parameters based on model performance on a labelled dataset. The key idea is to use many examples to train the model to make predictions about unseen data [7].

2.1.2 Graph Representation

Traditional Euclidean data formats cannot capture underlying geometric information in data for many real-world objects [1]. A graph representation helps to store the relational knowledge of interacting entities and make predictions by efficiently accessing the stored knowledge [17].

Graphs have the characteristics of permutation invariance or equivariance [18]. Invariance is a property of function not dependent on the arbitrary ordering of the elements in adjacency matrix [18]. Equivariance property means that the function output follows and changes per permutations in the adjacency matrix. This property is suitable for code-as-graph representations since multiple syntactic variations of written code can implement the same programming logic.

2.1.3 Geometric Deep Learning (GDL)

Deep learning models have been proven successful with Euclidean data types, such as speech, image and video. Recently, the trend of applying this to non-Euclidean data is on a rise [2]. Geometric deep learning is a set of techniques trying to generalize and extend the deep neural models to non-Euclidean domains, such as graphs and manifolds (arising from very different fields of mathematics as graph theory and differential geometry, respectively) [1].

One query that may arise here is why we need to extend DL on graphs. The answer lies in the limitations of traditional models not being able to understand the information in complex structures such as graphs, which contain two types of information for underlying nodes and edges, i.e. relationships among nodes. Naive neural networks such as Multi-Layer Perceptron (MLP) fail on the graph, so it becomes necessary to build novel neural network architectures to process graphs [19].

2.1.4 Embedding Techniques

Embedding techniques convert the raw data into a high-dimensional vector, along with preserving underlying properties of data representation [17]. Based on a trade-off between the benefits and limitations of various embedding techniques, the choice depends heavily on the suitability and needs of the target SE task. Some empirically popular techniques in DL4SE are Kernel methods, DeepWalk, Node2Vec [17], [20].

2.1.5 Graph Representation Learning (GRL)

Graph representation learning (GRL) is the group of techniques learning to extract features automatically by understanding the underlying data. DL models can use such learned graph representation to perform downstream tasks effectively [17]. Figure 2.2 shows the graph representation learning in ML ecosystem.

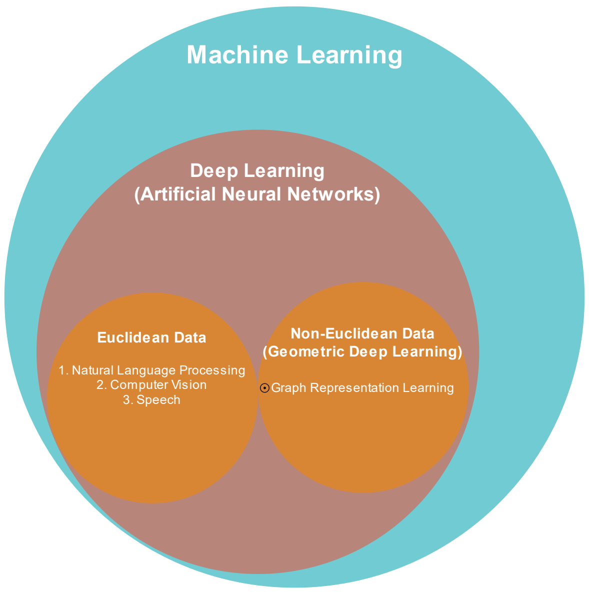

Machine Learning
Deep Learning
(Artificial Neural Networks)
Euclidean Data
1. Natural Language Processing
2. Computer Vision
3. Speech
Non-Euclidean Data
(Geometric Deep Learning)
Graph Representation Learning

Figure 2.2 Overview: GRL in Machine Learning Ecosystem

2.1.6 Graph Neural Networks

A graph neural network (GNN) learns an embedding representation for all nodes, with information about its neighbourhood, to learn the embedding of a whole graph [21], [22]. Most existing GNN models follow the message passing neural networks (MPNN) framework, with nodes exchanging information to learn in neural network architectures with the process collectively called message passing and neighbourhood aggregation [19]. Aggregation function can be simple as sum, average or complex, e.g. based on attention mechanism [23], [24]. A gated graph neural network (GGNN) is a GNN with information transfer between nodes through a gated recurrent unit (GRU) mechanism to effectively capture dependencies of different time scales [10].

A recent survey organizes GNNs into four groups [25], [26]:

1. Recurrent Graph Neural Networks (RecGNNs)
2. Convolutional Graph Neural Networks (ConvGNNs)
3. Graph Autoencoders (GAEs)
4. Spatial-temporal Graph Neural Networks (STGNNs)

2.1.7 Attention Mechanism

Introduced by [27], the attention mechanism captures relevant parts of the input with more importance than others to create a better representation suitable to the target task [28], [29]. Natural language processing (NLP) techniques use this for text data representation, while computer vision (CV) methods use it for image data representation [28]. GRL field extends this concept to generate embedding to learn better graphical representations of data [29].

2.1.8 Transformer

A transformer is a model architecture, dependent on the attention mechanism, used to identify global dependencies between input and output in sequence-to-sequence tasks. It does not depend on recurrence, as was previously used in recurrent neural network (RNN), and allows significant parallelization [27]. This model extracts features of each element to determine the importance of other elements with respect to that particular element. Such architectures have helped solve numerous challenging tasks in NLP and CV effectively and efficiently, and the GRL field has increasingly adopted them in recent years.

2.1.9 Long-range dependency problem and solutions

The vanishing gradient (approaching zero value) in DNNs leads to a long-range dependency problem, due to which DNNs face issues capturing dependency in data over a long-range distance. Long short-term memory (LSTM) processes single data points or their entire sequences based on feedback connections and hence helps to solve the problem. Cells remember values at arbitrary time intervals, while gates control the flow of information incoming or outgoing from cells. The Gated recurrent unit (GRU) approach uses a gated mechanism to capture dependencies from different time scales effectively. Recurrent neural networks (RNNs) with LSTM or GRU are de facto standards to model sequential data. Traditional approaches to treating code as text faced challenges from the limitations of RNNs. As they are designed to process sequences smoothly and are not suitable to capture the control and data flow, the graph should be a better representation [30].

2.1.10 Programming Language Processing (PLP)

The program representation problem, i.e. how to convert code into suitable representation, is essential to exploit the benefits of deep learning [31]. Programming language processing (PLP) is the set of techniques which process the code input from programming languages and analyze it to learn various downstream SE tasks [32].

2.2 Code representations

Code representation captures syntactic, semantic, control and flow information in the code. Approaches can be categorized as sequence-, tree-, and graph-based [31], [33].

Sequence-based

Initially, the code was considered similar to a natural language due to inherent naturalness in programming languages, and NLP techniques were applied [6]. A sequence-based representation such as a natural code sequence (NCS) captures the sequential order to connect neighbouring tokens to preserve the logic [21]. However, various problems arise with treating code as natural language [34]. The syntactic structure differs from natural languages because of strict syntactical limitations, use of delimiters and long-distance references [34]. Source code usually works with an open, ever-expanding vocabulary, and treating code as NL implies out-of-vocabulary problems occurring too often [34]. In sum, sequence-based approaches treat code only as a sequence of tokens and are limited to capturing only syntactic structures, leading to the need to develop novel techniques [30], [33], [35].

Tree-based

Tree-based approaches represent the code to capture structural and content information [33], [36]. Abstract syntax tree (AST) can be defined as a representation of the grammatical structure of code, using tree form with nodes illustrating the structure [37].

Graph-based

Tree-based representation cannot capture relationships such as control flow and data flow. Graph-based approaches augment ASTs with additional edges to capture such flow information in the code, enabling it to capture syntactic as well as semantic information comprehensively [12], [33]. Graph-based representation possesses good generalization ability, however, requires a larger dataset for training [33], [36]. Extending code to a non-Euclidean space, [10] first represented code as a graph. Based on the notion of code as a graph, [38] developed path-based representation using abstract syntax trees.

2.2.1 Representation Types

Here, we explain a few essential graph-based representation types of code:

Abstract Syntax Tree (AST)

AST is an ordered tree representation of the context-free grammar structure of the code, which can capture lexical (such as tokens) and syntactic information of code [37], [39]. Usually, it is the first step used by a parser to check syntax errors in the structure of the program [21].

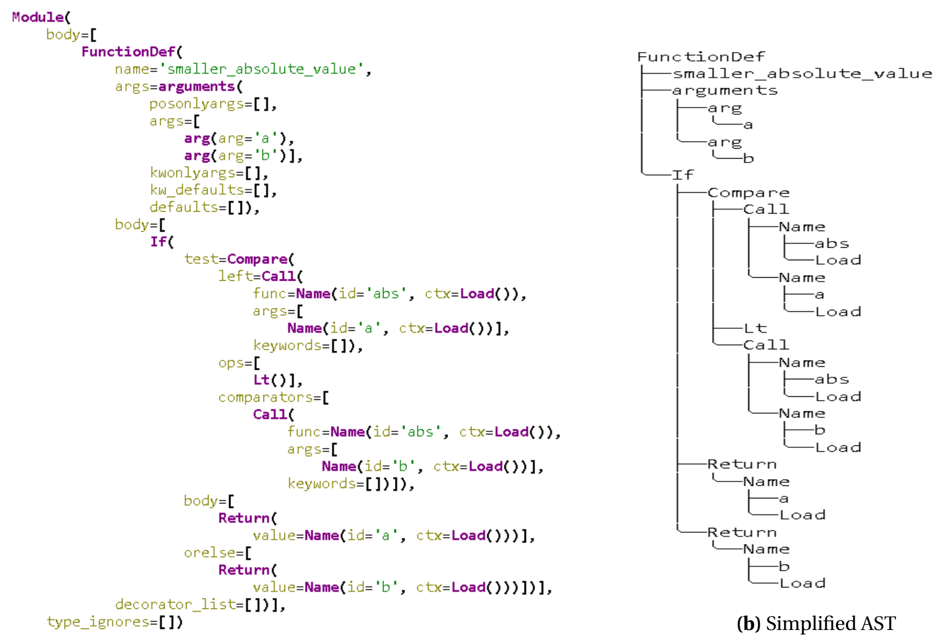
AST has been historically used to analyze and optimize compilers. However, they can be too dense for large programs. Such high density can mislead the model to memorize the syntax instead of it becoming able to learn the associated semantics [40]. AST starts from the root node, the downstream tree contains nodes showing language constructs such as functions, blocks, statements, declarations, and expressions, and finally up to the leaf nodes such as variable names (primary tokens) [21], [34].

For instance, the Python source code to calculate the smaller absolute value between two arguments is shown here.

```python
def smaller_absolute_value(a, b):
    if abs(a) < abs(b):
        return a
    else:
        return b
```

Its corresponding AST is shown in Figure 2.3a, and simplified AST in 2.3b, generated using module AST[1], and the library rich[2] respectively. Various tree elements are illustrated in different colours to differentiate primary tokens, statements, and conditions in the program code.

[1]https://docs.python.org/3/library/ast.html
[2]https://pypi.org/project/rich/



```
Module(
    body=[
        FunctionDef(
            name='smaller_absolute_value',
            args=arguments(
                posonlyargs=[],
                args=[
                    arg(arg='a'),
                    arg(arg='b')],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                If(
                    test=Compare(
                        left=Call(
                            func=Name(id='abs', ctx=Load()),
                            args=[
                                Name(id='a', ctx=Load())],
                            keywords=[]),
                        ops=[
                            Lt()],
                        comparators=[
                            Call(
                                func=Name(id='abs', ctx=Load()),
                                args=[
                                    Name(id='b', ctx=Load())],
                                keywords=[])]),
                    body=[
                        Return(
                            value=Name(id='a', ctx=Load()))],
                    orelse=[
                        Return(
                            value=Name(id='b', ctx=Load()))])],
            decorator_list=[])],
    type_ignores=[])
```

(**a**) Abstract Syntax Tree

```
FunctionDef
├──smaller_absolute_value
├──arguments
│  ├──arg
│  │  └──a
│  └──arg
│     └──b
└──If
   ├──Compare
   │  ├──Call
   │  │  ├──Name
   │  │  │  ├──abs
   │  │  │  └──Load
   │  │  └──Name
   │  │     ├──a
   │  │     └──Load
   │  ├──Lt
   │  └──Call
   │     ├──Name
   │     │  ├──abs
   │     │  └──Load
   │     └──Name
   │        ├──b
   │        └──Load
   ├──Return
   │  └──Name
   │     ├──a
   │     └──Load
   └──Return
      └──Name
         ├──b
         └──Load
```

(**b**) Simplified AST

Figure 2.3 Example Code Representation: Abstract Syntax Trees

AST can be converted into graph-based representation so that it can be fed as an input to DL models to process downstream tasks. The same code's AST is shown in graph-based format in Figure 2.4, generated using the library VAST[3]. The depth or density of graph-based AST is less than that in tree-based AST structures.

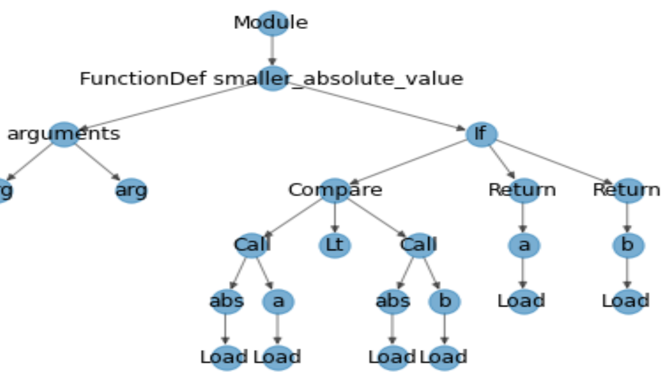


Figure 2.4 Example Code Representation: Graphical Abstract Syntax Tree

[3]https://pypi.org/project/VisAST/

Control Flow Graph (CFG)

A control flow graph is a directed graph representing the flow of control, i.e. various paths traversed by the program during execution based on conditional statements [21], [22]. Statements are nodes and conditions are connected by edges to show the path for the control flow. Traditionally, the static analysis used CFG due to its ability to capture the flow within a program accurately [22]. The reachability analysis identifies loop structures and locates inaccessible parts of code in the programs [22].

The CFG for example code snippet has been shown in Figure 2.5. The left block shows the function definition block, and the centre block shows the control flow in the code. For instance, the "if construct" calls the "abs" module to calculate the absolute value of the variable. At the end of the program execution, variable "a" or "b" is returned based on value comparison. The right block shows various possible keys(symbols), and basic colour-coded elements of the code. This CFG is generated using library py2cfg[4].

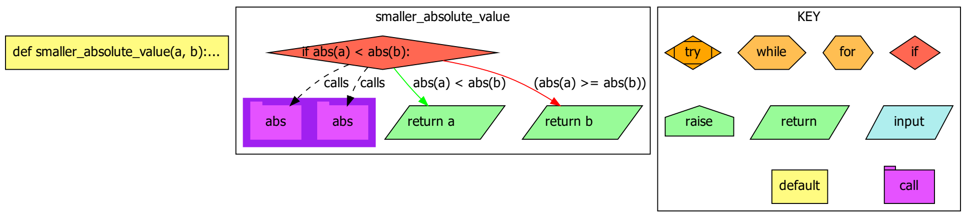


Figure 2.5 Example Code Representation: Control Flow Graph

Data Flow Graph (DFG)

A data flow graph (DFG) keeps track of variables used in a CFG [21]. Each node is usually an element such as a variable or operator, whereas an edge shows the data transfer between two entities. Data flow implies accessing or updating certain variables in program execution. DFG can be used to understand the logic and functionality of a program, suitable for debugging tasks [41], [42].

Miscellaneous representations

Based on the intended purpose to capture syntactic, semantic, control, data flow and dependency information, previously discussed base representations can derive other

[4]https://pypi.org/project/py2cfg/

complex ones. Conditional random field (CRF) is a probabilistic (log-linear) model for the conditional probability of labels y when observation x is given [43]. CRFs can be used to model relations among variables, AST elements and types without considering the data flow [10]. This representation was used by [43] to predict variable names and types.

A function call graph (FCG) models the semantic information associated with functions in code. Each node shows a function with an edge showing the call relation between functions [41], [42]. A program dependency graph (PDG) is a combination of AST and CFG [22]. A code property graph (CPG) is created by augmenting AST with additional edges to capture control-flow, data-flow and dependency information [37], used as a combination of AST, CFG and PDG [44].

2.2.2 GRL for SE Framework

GRL for SE can be described in the framework as shown in Figure 2.6. The code is first converted to a graph representation and then to a vector format so that a DL model can process this input to learn various downstream tasks, ranging from classification to prediction and generation in code-related contexts.

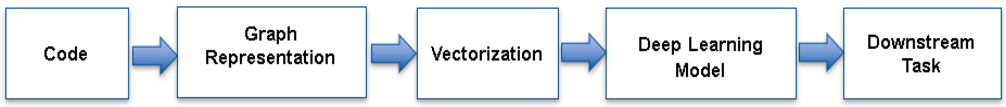


Figure 2.6 Framework: GRL for SE

2.3 Text Mining

As part of information retrieval techniques, text mining is an algorithmic approach to extracting the required information from a collection of text documents (often known as corpus) [45]. For instance, this can help to discover underlying relationships among different terms in the text corpus to find the highest important, also known as weighted terms [46]. One common technique is to extract relevant terms from a text document, using frequencies to give a 'relevance value' to terms. This technique could be extended to a text corpus to calculate the term's relevance in all documents [47].

Term Frequency

Term frequency *tf(t, d)* is the frequency value of the term *t* within document *d*, counted as raw count, relative or also as logarithmically scaled values [45].

Inverse Document Frequency

Inverse document frequency *idf(t)* represents the amount of information carried by the word, based on checking if the word is common or rare in all documents. It is calculated as a logarithmically scaled inverse fraction of the documents containing a term.

$$idf(t,d) = log(\frac{N}{n_t})$$

N is total number of documents in the corpus, while n_t is the number of documents where the term *t* appears (i.e. *cannot be zero*). To avoid divide by zero conditions for terms absent in corpus, it is common practice to smooth the denominator by adding 1, i.e. $(1+n_t)$.

TF–IDF (Term Frequency – Inverse Document Frequency)

TF – IDF *tfidf(t, d)* is a statistical approach involving frequency of a term in a document, and number of documents in corpus with same term present [45], [47]. TF-IDF value is simply multiplication of *tf(t, d)* and *idf(t)* as shown in equation 2.1:

$$tfidf(t,d) = tf(t,d) * idf(t) \tag{2.1}$$

where *tf(t, d)* is the frequency of term *t* in document *d*, and *idf(t)* is number of documents containing the term *t*, with logarithmically scaled values as inverse fraction of total number of documents in corpus.

Terms in large text documents may have a significantly higher frequency value than that of smaller text documents. It results in a high TF-IDF value without necessarily indicating a higher relevance, hence a normalization is recommended [47]. With normalization, the formula becomes

$$tfidf(t,d) = (\frac{tf(t,d)}{tfmax(d)}) * idf(t) \tag{2.2}$$

In the equation 2.2, *tfmax(d)* is the frequency of the most-frequent term in document *d*.

Chapter 3

Related work

3.1 Deep Learning and Software Engineering

A large amount of software data is available in the repositories, and DL techniques can use to learn to assist in SE tasks. This research area is defined as "Big Code" [6], which extends to other data such as requirements and issue trackers. The interaction between the DL and SE fields emerges into two themes. The first theme is DL techniques viewed as a new form of software development, usually defined as software engineering for deep learning (SE4DL) [7]. It represents opportunities to automatically learn the program from large datasets instead of specifying it explicitly by humans. Another theme is to leverage DL techniques to help existing software engineering tasks, defined as deep learning for software engineering (DL4SE) [7]. Both fields offer exciting research opportunities as well as technical challenges. [6] proposed the naturalness of a programming language, which led to techniques treating programming language similar to a natural language to apply NLP techniques.

On the one hand, there has been a lack of research in the SE4DL field. Specifically, [48] conducted a case study based on seven industry projects to identify tools, best practices, and challenges for DL systems using SE processes. [7] presented a workshop paper and discussed research opportunities and diverse applications of SE4DL and DL4SE fields.

On the other hand, the DL4SE field contains a large number of research papers, and we will discuss their strengths and weaknesses next. A few papers discussed research trends in specific focus areas. For instance, [49] conducted a SLR and discussed the lifecycle of federated learning systems from a SE perspective. [50] conducted a SLR to determine 8 ML techniques for ML-based software development effort estimation (SDEE).

Studies without systematic literature review

[7], [51]–[53] presented technical briefings, workshops or tutorial papers where they discussed approaches to adopt DL for SE. [54] researched the changes in software development practices due to ML. They surveyed 14 in-person interviewees, and 342 online respondents and described aspects of SE tasks and knowledge work characteristics. [55], [56] conducted a bibliography analysis and described industry contribution and practicability of DL as a hindering problem in its adoption. [14], [57] conducted a mapping study and identified ML environment for specific SE tasks, with [14] using software engineering body of knowledge (SWEBOK) knowledge areas [58].

Studies with non-SLR style literature review

[6], [59] conducted a literature review describing the naturalness of code. [60] provides insights about recent advances of DL in SE. [61] discussed replicability and reproducibility of DL as issues while using DL in SE. [13] explored the relationship between ML tools and SDLC stages. [5] describes three dimensions: the amount of learnable data, the fuzziness of available information, and well-defined correctness criteria to help in decision making: whether to use neural software analysis. [62] explained pre-trained models of source code (CodePTM) and its impact on adopting DL for SE. However, these papers did not conduct a systematic literature review and did not often provide information about the process of paper collection and documentation about analysis approaches.

Studies with systematic literature review

Now, we discuss SLR studies in the DL4SE with a similar focus as our work, however, have distinctive additional findings. [63] identified challenges of dataset features, resources, and network configuration. [64] uses SWEBOK to map DL applications in SE to its 12 knowledge areas. [65] discussed additional optimization algorithms and overfitting tackling techniques, and [66] conducted SLR based on the components of learning and created concept maps. They identified 23 SE tasks and explored data types, pre-processing techniques, overfitting and underfitting for SE tasks, impact measured, and non-reproducibility factors. [67] is the most comprehensive SLR, consisting of analysis from 1428 papers. They described the complexity of using DL4SE, specifically highlighting differences between ML and DL techniques across 77 SE tasks. They enable the community to measure the changes in SE due to ML/DL, improvements and well-informative discussion on whether to select DL/ML for a SE task.

3.2 Deep Learning on Graphs & Software Engineering

All previously discussed studies were at the intersection of ML/DL and SE. Our work focuses on GRL in the SE context. As GRL is a new research area, this has been not studied well enough in the SE context. The closest works to our work (considering the main research context, focused on GRL) are [68], [69] however, they discussed DL on graphs in the NLP field and also not focused on the SE applications. Another work [70] is closest to our work in its design and main research context. They explored GRL in bio-informatics, discussing various trends, methods and applications. To the best of our knowledge, this is the first study focused on the GRL field and its applications, especially in the SE context.

Chapter 4

Methodology

4.1 Systematic Literature Review

Evidence-based research review has gained traction in recent decades, which was later adapted to SE [71], known as the practice of evidence-based software engineering (EBSE). EBSE closes the gap between research and practice, and follows a process based on a specific set of research questions [71].

On the one hand, a traditional review contains approaches to survey existing research, but the process is not well-documented and hard to reproduce and has a limited objective assessment [72]. On the other hand, a Scientific Literature Review (SLR) follows a well-defined set of procedures and protocols to collect, evaluate and interpret all relevant records in an unbiased manner [71]. SLR serves to find evidence-based answers to a specific set of research questions, along with recording the process in a detailed fashion, hence is reproducible [71].

Also, such a review helps to identify gaps in research areas, and further research as a future work [72]. The relevant articles are considered primary studies, and the review itself is a secondary study. Among a set of different guidelines, the most authoritative and practised in literature is the guideline by Kitchenham [72]. Activities in the review are organized into three major phases: planning, conducting and reporting [73]. The activities are sequential but often contain iterations with the possibility to adapt and refine the approach.

4.1.1 Planning

GRL has been a heavily researched area in recent years [8]. The need for the SLR arises because we would like to identify possibilities of exploiting and extending the benefits of GRL to the SE domain. Also, it would help us identify existing research gaps, and summarize existing literature, to produce evidence objectively and scientifically.

A well-documented protocol should be defined, which contains details such as background, research questions, search strategy, selection criteria, relevant methodology, quality assessment, data extraction approach, evidence synthesis and dissemination for the review [72]. This protocol can also help to reduce the researcher bias [73].

4.1.1.1 Study Selection

Study selection is a process of selecting the papers, i.e. primary studies for data extraction and further analysis. The SE field often contains poor-quality abstracts, so the conclusion should be reviewed in combination with abstract [73]. The selection procedure consists of multiple phases: the first phase contains reviewing based on reading the title, abstract, and conclusion [74], [73]. The next phase is implemented based on the inclusion and exclusion criteria as per the research context [72], mentioned below:

Inclusion Criteria

- ✓ Study published in the last 10 years, i.e. 2012-2022
- ✓ Study contains code representation in graph-based format
- ✓ Qualitative or quantitative empirical research about GRL in SE

Exclusion Criteria

- × Study not answering at least one research question with empirical evidence
- × Study focused on graph theory/ML/DL, however not in GRL for SE context
- × Study not from conferences or journals, such as Ph.D. dissertations, tutorials, and magazines

× Study length less than or equal to 2 pages

× Study not available in full-text format

× Study not in English language

4.1.1.2 Quality Assessment

The next phase of study selection relies on a set of specific quality assessment questions with defining a measurement scale to assess the overall quality of primary studies [72]. We assign a value such as 1 (Yes), 0 (No), and 0.5 (Partially) based on the study's ability to answer a particular question. Aggregated scores for all questions are compared with a threshold score of 4 (out of 5) to consider for further analysis. A summary of different questions covering aspects of quality is in the below table 4.1.

ID	Question	Answers
Design		
QA1	Are the study aim and contribution clearly described?	Yes/No/Partially
Conduct		
QA2	Does the study describe inputs, outputs and methodology in detail?	Yes/No/Partially
QA3	Does the study describe approach for preparation of dataset?	Yes/No/Partially
Data Synthesis		
QA4	Are the approach of analysis and evaluation well-described?	Yes/No/Partially
Credibility		
QA5	Is the study sufficiently referenced?	Yes/No/Partially

Table 4.1 Overview: Quality Assessment Criteria

4.1.1.3 Data Extraction

The data extraction form should be created while defining the study protocol [72]. This step is required to collect information to aggregate and synthesize the evidence to answer research questions. Information such as title, authors, publication details, study type and elements about research questions are considered [72]. Data extraction form for this study have been summarized in below table 4.2.

Data Item	**Information to capture**	**RQ**
ID	Identifier of the research study in SLR	
Title	Title of the study	
Authors	Authors of the study	
Year	Year of the study	
Publication Venue & Name	Journal or conference name	
Citations	Number of citations of the study	
Main Contributions & Summary	Main contributions summary of the study	
Code Type	Assembly/binary/compiler/source code	RQ1
Representation Type	code-as-graph representation type	RQ1
Representation Characteristics	Attributes of representation	RQ1
Embedding Type & Level	Type and level of graph embedding	RQ1
Software Engineering Task	SE task type and details	RQ2
Extensibility	Language/framework-specific or agnostic	RQ2
Platform Support	Supported platforms	RQ2
Learning Type	Machine Learning type (Supervised/Unsupervised/Semi-Supervised)	RQ3
Traditional Methods	Traditional methods for the SE task	RQ3
Role of GRL	Role of the GRL in the SE task	RQ3
Neural Network Type/Model	Type and characteristics of the neural network	RQ3
Dataset Type	Type of dataset and names	RQ4
Programming Language	Target programming language of the experiment	RQ4
Evaluation Metrics	Metrics for evaluation of the results	RQ4
Research Gaps/Future Work	Current gaps in research, and possible future work	RQ4

Table 4.2 Overview: Data Extraction Form

4.1.1.4 Data Synthesis

We collect and summarize the results in this step to create a descriptive synthesis based on qualitative data. The extracted data is next analyzed to demonstrate the homogeneity or heterogeneity of results [72].

4.1.2 Conducting

4.1.2.1 Search Strategy

In an academic research ecosystem, the search types, goals, and their corresponding heuristics could be summarized [75] as shown in below table 4.3:

Search Type	Goals	Dominant Heuristics
Lookup	To identify research articles to fulfill information gaps with quick targeted search	* Straightforward search, navigation * Most specific first
Exploratory	To learn about a body of research, with goals becoming clearer iteratively.	* Wayfinding (with little prior knowledge) * Most specific first * Snowballing/pearl growing (association) * Post-query filtering (limit as per meta-info)
Systematic	To identify all research artifacts with transparent and reproducible search	* Snowballing/pearl growing (association) * Post-query filtering (limit as per meta-info) * Building blocks (with Boolean operators) * Handsearching (manual screening) * Successive fraction (as per exclusion list)

Table 4.3 Overview: Academic Search Types

In our study, we first employ heuristics of post-query filtering and building blocks while implementing TF-IDF to improve the query string. Once we collect research articles, we use the heuristic of hand-searching for manual screening, with successive fraction heuristic to eliminate artefacts following the selection criteria.

4.1.2.2 Bibliographic Search Systems

Based on a comprehensive study by [76], academic search systems could be classified broadly into two groups: principal and supplementary sources. Few principal systems are ACM Digital Library, ScienceDirect, and Scopus, whereas arXiv, DBLP, Google Scholar, and IEEE Xplore are supplementary system [76], [73].

For evidence synthesis in systematic reviews and meta-analysis, available search systems cannot directly be used [76]. It is advisable to use multiple databases because electronic databases and digital libraries cannot fulfil the requirements to identify relevant articles for a review due to the lack of standardized keywords [74], [73], [72].

Based on criteria such as coverage, the relevance of results and their results export features, we choose ACM digital library as our principal search system, along with IEEE Xplore and ArXiv as supplementary systems. Also, we consider using the pioneer repository "machine learning for big code and naturalness" (ML4Code)[1] as another secondary system to identify relevant articles.

4.1.2.3 Query String for Extraction

Guidelines

Instead of performing a search directly in the database, a log document must record the complete search process. It helps to register the different steps, make it reproducible and accountable, and allows the searcher to control the whole process [77].

The PICO (Population, Intervention, Comparison and Outcomes) framework can be used to determine keywords and derive underlying concepts from research questions [72]. Key concepts, Boolean operators, thesaurus terms, and various synonyms and variations help to formulate the query phrase as part of single line search strategy [77].

Pilot Search

Initially, it is advised to conduct a broad search, and later make it more sensitive, and check if new relevant articles are found by comparing the results [77]. After extracting key concepts and elements from the research questions, we create a search query, used to conduct a pilot search on the principal system (ACM digital library). The purpose is to determine the patterns of information and knowledge representation in relevant papers.

Pilot Search Query String

```
(Source OR Code OR Software OR Module OR Program*
OR Syntax OR Semantic OR Langu* OR Represent*)
AND (Graph OR Tree)
AND (Neural OR Network OR Machine OR Deep
OR Learn* OR data* OR Engine*)
```

We obtain 306 search results on ACM database as on February 25, 2022 .

[1]https://ml4code.github.io/papers.html

4.1.2.4 Improvement of Query String

We implement the TF-IDF technique to identify the relevant terms for improving the search results. The algorithm calculates the frequency of different terms in all documents. The prerequisite step involves extracting text documents in a format which supports text mining techniques to read from them. We pre-process the full text of documents by extracting only alphabetical letters and discarding other characters, converting everything to lower case, and splitting the sentences into a list of words. We also remove stop words (e.g. 'is', 'the') from text with a list of words pulled from a standard SPACY library [2]. We perform data processing in Python using the TfidfVectorizer method of the Scikit-Learn library [3].

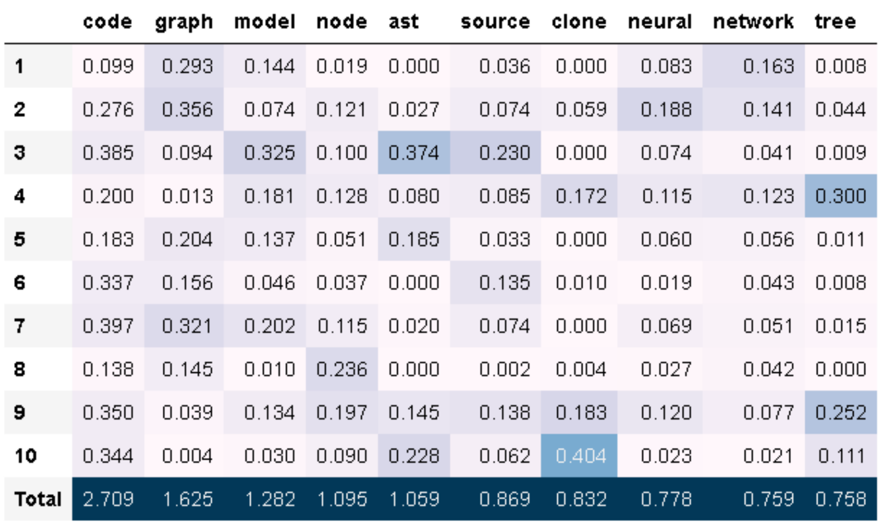
Based on screening by reading the title and abstract of results, a number of documents (with a threshold heuristic: in our study as ten) are marked as relevant and similarly other ten documents are noted as irrelevant. The weighing technique is applied to calculate a global TF-IDF value for each term from all documents in the text corpus [47].

The steps of the algorithm are as below [47], [78]:

1. Mark a number of documents(for instance, 10 as threshold) as relevant, i.e. group *R*.
2. Mark a number of documents (10) as irrelevant, i.e. group *IR*.
3. Get TF-IDF values for the terms in studies marked as relevant (*R*), calculated by summing TF-IDF values, i.e. $\sum_{d_i \epsilon R} tfidf(t, d_i)$ where d_i is a document in *R*, having term *t*.

   A overview of TF-IDF values from relevant documents is shown below in Figure 4.1, with color intensity mapped to the magnitude of value.

[2]https://spacy.io/
[3]https://scikit-learn.org/stable

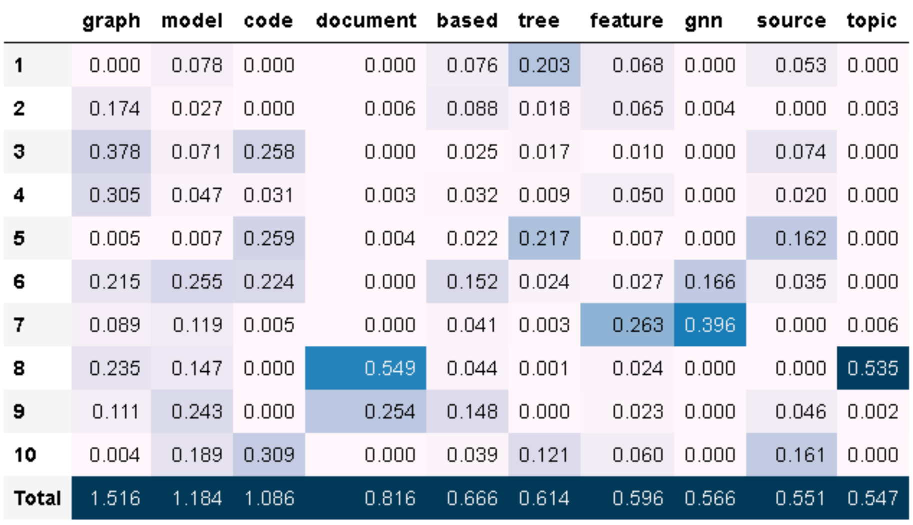


	code	graph	model	node	ast	source	clone	neural	network	tree
1	0.099	0.293	0.144	0.019	0.000	0.036	0.000	0.083	0.163	0.008
2	0.276	0.356	0.074	0.121	0.027	0.074	0.059	0.188	0.141	0.044
3	0.385	0.094	0.325	0.100	0.374	0.230	0.000	0.074	0.041	0.009
4	0.200	0.013	0.181	0.128	0.080	0.085	0.172	0.115	0.123	0.300
5	0.183	0.204	0.137	0.051	0.185	0.033	0.000	0.060	0.056	0.011
6	0.337	0.156	0.046	0.037	0.000	0.135	0.010	0.019	0.043	0.008
7	0.397	0.321	0.202	0.115	0.020	0.074	0.000	0.069	0.051	0.015
8	0.138	0.145	0.010	0.236	0.000	0.002	0.004	0.027	0.042	0.000
9	0.350	0.039	0.134	0.197	0.145	0.138	0.183	0.120	0.077	0.252
10	0.344	0.004	0.030	0.090	0.228	0.062	0.404	0.023	0.021	0.111
Total	2.709	1.625	1.282	1.095	1.059	0.869	0.832	0.778	0.759	0.758

Figure 4.1 Analysis: TF-IDF values from relevant documents

4. Get TF-IDF values for the terms in studies marked as irrelevant *IR*, calculated by summing TF-IDF values, i.e. $\sum_{d_j \epsilon IR} tfidf(t, d_j)$ where d_j is a document in *IR*, having term t.

   A overview of TF-IDF values from irrelevant documents is as shown below in Figure 4.2.



	graph	model	code	document	based	tree	feature	gnn	source	topic
1	0.000	0.078	0.000	0.000	0.076	0.203	0.068	0.000	0.053	0.000
2	0.174	0.027	0.000	0.006	0.088	0.018	0.065	0.004	0.000	0.003
3	0.378	0.071	0.258	0.000	0.025	0.017	0.010	0.000	0.074	0.000
4	0.305	0.047	0.031	0.003	0.032	0.009	0.050	0.000	0.020	0.000
5	0.005	0.007	0.259	0.004	0.022	0.217	0.007	0.000	0.162	0.000
6	0.215	0.255	0.224	0.000	0.152	0.024	0.027	0.166	0.035	0.000
7	0.089	0.119	0.005	0.000	0.041	0.003	0.263	0.396	0.000	0.006
8	0.235	0.147	0.000	0.549	0.044	0.001	0.024	0.000	0.000	0.535
9	0.111	0.243	0.000	0.254	0.148	0.000	0.023	0.000	0.046	0.002
10	0.004	0.189	0.309	0.000	0.039	0.121	0.060	0.000	0.161	0.000
Total	1.516	1.184	1.086	0.816	0.666	0.614	0.596	0.566	0.551	0.547

Figure 4.2 Analysis: TF-IDF values from irrelevant documents

5. Calculate the global TF-IDF by subtracting the *IR* values from *R* values, dividing the resultant by number of documents in *R*, i.e. cardinal value of group *R*.

$$tfidf(t,D) = \frac{\sum_{d_i \epsilon R} tfidf(t,d_i) - \sum_{d_j \epsilon IR} tfidf(t,d_j)}{|R|} \tag{4.1}$$

6. The outcome is a list of global TF-IDF values, which provides the top key terms after sorting (in our study top 10 key terms). These terms are recommended by the method for improving search results by modifying the query string. The key terms have been shown in the Figure 4.3.

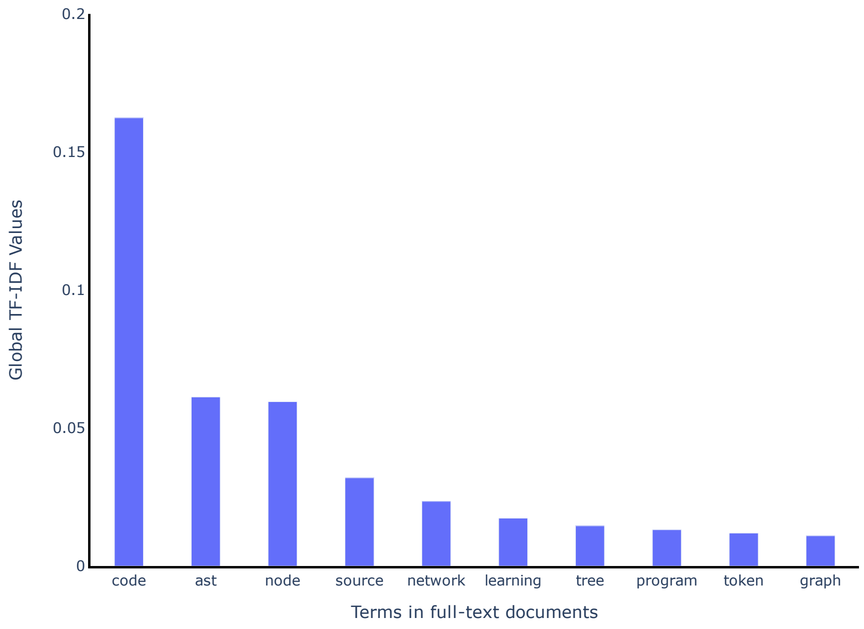


Figure 4.3 Analysis: Identification of relevant terms

7. The relevant terms are included in final search string, while irrelevant terms are removed.

4.1.2.5 Final Query String

Based on the result of the TF-IDF technique, we identify the most important terms to update the query string. Sometimes due to variation in terminologies used by different databases and limitations of the search query string, few relevant papers could be missing [79]. To mitigate this, we use the database guides[4] to extract relevant information seamlessly and accurately. We identify related terms and synonyms by following the IEEE thesaurus[5], which is a vocabulary of scientific terms. Also, we add abbreviations such as "NN" for "neural network". We extract the articles based on these keywords search in specific parts, such as title, abstract and metadata.

Along with the use of Boolean operators such as AND & OR, we create the final query string, adapted as per the principal search system, i.e. ACM and other supplementary sources (IEEE Xplore and ArXiv). The query string for ACM, IEEE Xplore, and ArXiv produce 109, 362 and 142 results as of March 16, 2022 and have been mentioned in the appendix at A.1, A.2, and A.3 respectively. We extract 420 results from the ML4Code repository as of March 25, 2022 . Thus after merging and de-duplicating, we collected 881 articles from all sources, which we will process in further steps.

4.1.2.6 Screening

The overall screening procedure has been as per the research protocol mentioned previously. The high-level overview of different steps with the number of articles selected at different phases is illustrated in Figure 4.4. Thus, at the end of the process, we analyze 53 research studies with a full-text review. The selected research studies (abbreviated RS) are mentioned in the appendix A.5. The different phases and associated information have been recorded and are parked digitally on this webpage[6].

[4]https://libguides.oulu.fi/databaseguides/acm
[5]https://www.ieee.org/publications/services/thesaurus.html
[6]https://gitlab.uni-koblenz.de/adityamehta/master-thesis

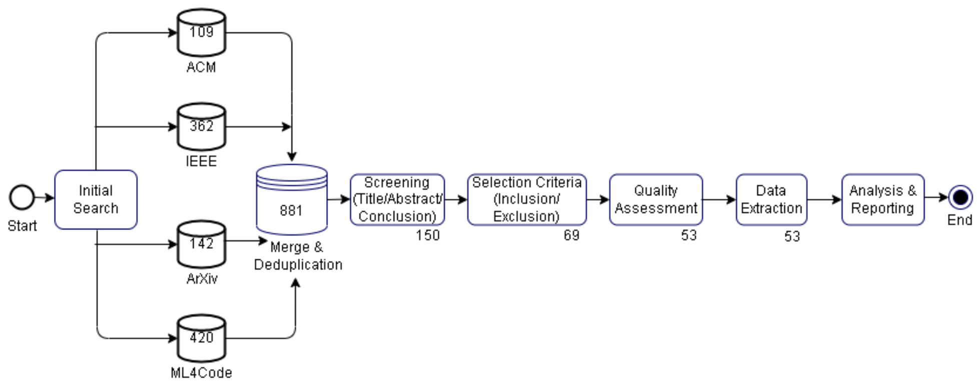


Figure 4.4 Workflow: Systematic Literature Review

SLR involves manual efforts and activities at various stages making it a time-consuming task. Tools supporting different phases help reduce the time and effort. It does not compromise the quality because the researcher takes the final decision [80]. Exploiting benefits, we utilize tools such as Parsifal, Rayyan [80], and Thoth [81]. We used these tools to shortlist the papers by reading through the title and abstract in the first screening stage, marking them as accepted or rejected before quality assessment. All software tools used in this study are listed at A.4.

4.1.2.7 Data Analysis

We follow the data extraction form 4.2 to analyze the selected articles for a full-text review to identify the answers to research questions.

4.1.3 Reporting

The results of data analysis and evidence synthesis has been reported and discussed comprehensively in the next chapter 5.

Chapter 5

Analysis and Results

This chapter discusses the analysis and results from extracted data to find research trends and answers to stated research questions. Insights enable us to synthesize evidence based on empirical research and contribute to creating the taxonomy of representations, DL models, and the GRL4SE field.

5.1 Systematic Literature Review

5.1.1 Publication trends

First, we start by measuring the research activity based on the distribution of publications over the last few years and shown in Figure 5.1. Though we considered identifying studies from the last ten years, however, we found studies only since 2018 in papers selected in the SLR. The pioneering work done by [10] represented code-as-graph, which inspired the GRL field with graphical representations of code.

This trend from the analysis shows a continuous growth in the papers published each year. Openreview [1] is a platform to promote openness in scientific communication, specifically the peer review process enabled with web-based interfaces, databases and APIs. [82] surveyed the research papers on Openreview in 2017 to identify trends across various fields of deep learning research. They find GRL as one of the top actively researched areas, along with other equivalent active research areas such as

[1]https://openreview.net/about

reinforcement learning, adversarial ML, natural language processing and computer vision. Our results confirm the similar trend found in their results, i.e. number of publications in the GRL field is growing continuously in recent years.

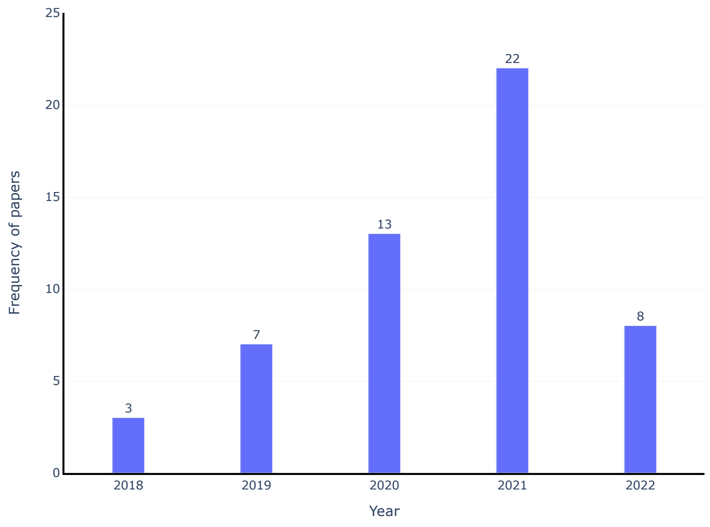


Figure 5.1 Analysis: Publications distribution over years

Next, we analyze the source of these publications. Specifically, we analyze the distribution of the publication venues, which can be either a journal or a conference. Abbreviations and full names for these venues have been mentioned adequately at A.6. Communities in SE and AI fields have been increasingly providing attention to conducting research on GRL in SE. Premier SE venues such as ICSE, TSE, TOSEM, ESEM, SANER, ICST and some renowned AI venues such as ICLR, NIPS, AAAI, ICML, ICTAI, and ICPC are identified as publication venues in our SLR. As analysis is shown in Figure 5.2, the top 5 publication venues are ArXiv, ICLR, IEEE Access, NIPS, and AAAI, having the high number of contributions to studies considered in our SLR study.

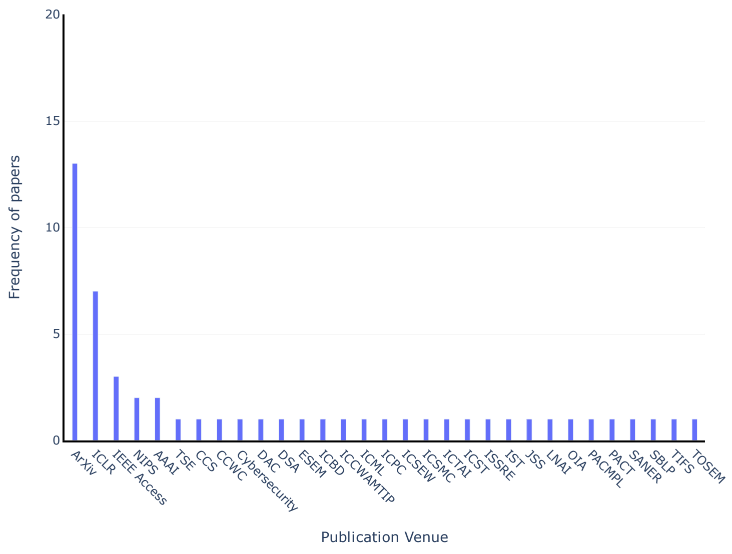


Figure 5.2 Analysis: Publication venue for studies

Next, we measure and analyze the citations for studies. The rationale is to identify seminal papers introducing pioneer techniques in the field which are influential in promoting further research activity. This data has been recorded on May 24, 2022 from Google Scholar. The citation frequency distribution for all papers follows approximately power law, with the most cited paper having citations count more than twice of the second most-cited paper, and so on.

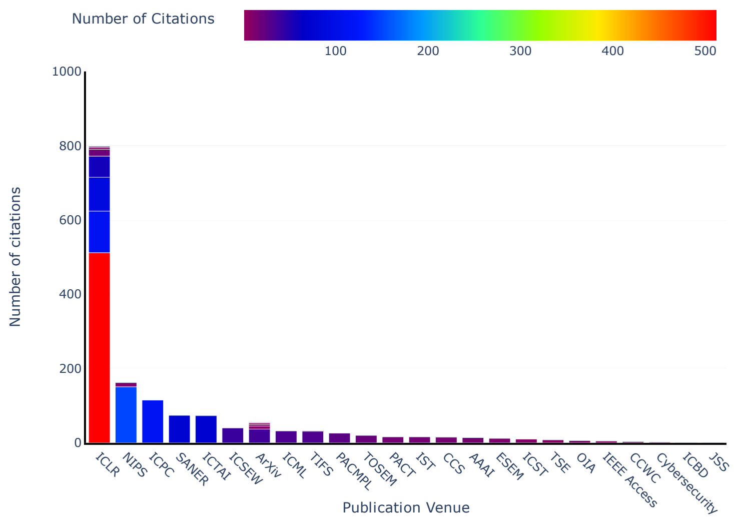


Figure 5.3 Analysis: Citations count for studies as per publication venue

To dig deeper, we group these highly cited papers with their respective publication venues. This analysis has been illustrated in Figure 5.3. The top 5 publication venues with the most influential work are ICLR, NIPS, ICPC, SANER, and ICTAI. Also, ArXiv is another great resource since it contains preprints of papers, enabling a researcher to identify the latest research activities. We considered ArXiv because many high-cited papers in full-text format are easily available here. As stated previously and observed from the graph, [10] is the pioneering work in the field, with an exceptionally high number of citations, which led to the birth and subsequent evolution of the GRL field.

5.1.2 Research Question 1

5.1.2.1 Code Type

First, we begin to identify different types of code considered in the studies. This has been analyzed and plotted in Figure 5.4. We can see that source code is the most empirically used code type, followed by other rarely analyzed code types such as binary code, compiler code, and assembly code.

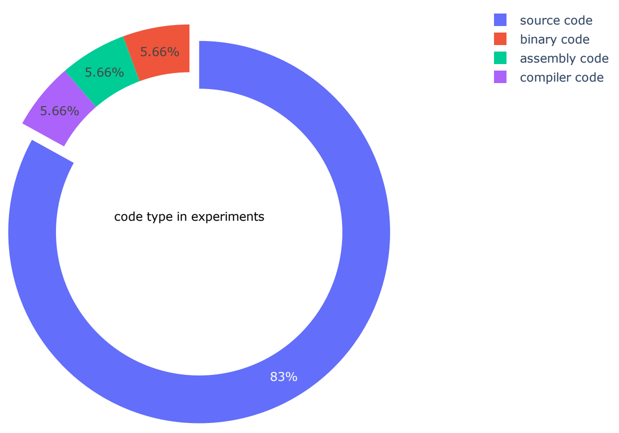


Figure 5.4 Analysis: Code Types

5.1.2.2 Code as Graph Representation Types

AST, IR and syntax code graph (SCG) are a few widely used formats, but they capture information related only to the syntactic structure [83], [84].

To capture semantic information, ASTs are augmented with additional information by introducing new nodes and edges. It is done to capture the flow of information by developing the control flow, data flow, and call flow graphs, with derived representations known as control flow graph (CFG), data flow graph (DFG), and CDFG. To capture information and flow for control-dependence and data-dependence, program dependence graph (PDG) is developed and used [33], [85], [86], [87].

The various representation types have been shown diagrammatically in the Figure 5.5. Most studies (a total of 28) use AST as the representation type, and CFG is the second most widely used format.

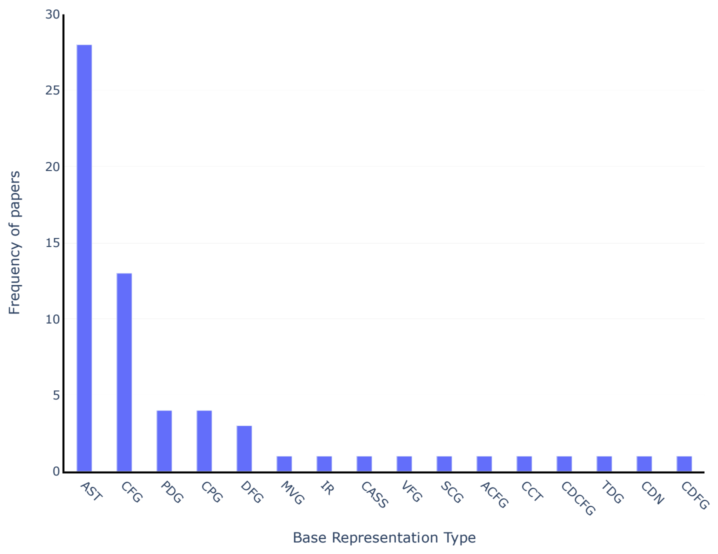


Figure 5.5 Analysis: Code Representation Types

Other representations

Code property graph (CPG), a representation derived from the combination of AST, CFG and PDG, was introduced by [88] and used by [89], [37], [90], [44] focused on vulnerabilities detection. Inspired by the NLP field with the premise that context is essential to capture the semantics, [40] created context-aware semantics structure (CASS) representation. Few studies created dependency graphs, e.g. [20] created class dependency network (CDN) based on dependencies among classes. Similarly, [91] developed a type dependency graph (TDG) of the types used in the code. The taxonomy of various representation types with their respective study reference is given in table 5.1.

Base Representation Type	**Count of studies**	**Studies reference**
Abstract Syntax Tree (AST)	28	[30], [19], [86], [26], [34], [92], [12], [93], [94], [95], [39], [35], [21], [96], [97], [24], [98], [99], [10], [100], [101], [41], [102], [103], [42], [104], [105], [106]
Control Flow Graph (CFG)	14	[107], [30], [86], [93], [92] [108], [23], [28], [36], [109], [110], [111], [22], [29]
Code Property Graph (CPG)	4	[89], [37], [90], [44]
Program Dependency Graph (PDG)	4	[33], [85], [86], [87]
Data Flow Graph (DFG)	3	[92], [30], [93]
Control Data Flow Graph (CDFG)	1	[108]
Control Data Call Flow Graph (CDCFG)	1	[112]
Variable-based Flow Graph (VFG)	1	[113]
Class Dependency Network (CDN)	1	[20]
Type Dependency Graph (TDG)	1	[91]
Calling Context Tree (CCT)	1	[107]
Syntax Code Graph (SCG)	1	[84]
Context-Aware Semantics Structure (CASS)	1	[40]
Intermediate Representation (IR)	1	[83]

Table 5.1 Taxonomy: Code Representation Types

5.1.2.3 Graph Representation Attributes

The graph representation is usually a combination of attributes such as direction-orientation, weighted edges consideration, and types of nodes in the graph, i.e. homogenous or heterogeneous as per same or different types of nodes. Weights give special attention to specific relations in graphs such as the edges of importance. The attention mechanism helps to include weighted edges in graphs [27]. [103] used bidirectional graphs, with weights shared across nodes. [34] first introduced and used multi-graphs, and this led to future studies such as [99] and [35] also using the same.

[100] was introduced the notion of heterogeneous graph, and developing on the similar structures, [102] and [33] used heterogeneous graphs. [91] first introduced the hypergraph structure (graph about graphs), which helped subsequent studies as [24] to develop typed and qualified hypergraphs.

5.1.2.4 Embedding Level

Embedding converts the representation into vectorized form, and DL algorithms can consume them. The graph embedding can be simple as node-based, edge-based or complex as graph-based [10]. The node-based embedding is the most basic and essential representing tokens or words and associated data types. Initially, code is converted into a node-level embedding, and various nodes exchange the information through edges using message passing neural networks (MPNN) framework. It can help develop the edge and graph level embedding [102].

5.1.2.5 Extensibility and platform support

The majority of studies consider the analysis of single-platform and specific programming languages. [20] created class dependency network (CDN), which is multi-platform and used for defect prediction in their study. Similarly, [83] developed a compiler-independent, language-independent representation used as a multi-platform format. The representations used by [106], [19], [30], [29], [108], [35], [98] are extensible to other languages than ones mentioned in studies, and hence can be made language-agnostic. Multiple studies proposed to extend their work as an additional plug-in or a feature in a typical integrated development environment (IDE) [39], [94].

5.1.3 Research Question 2

5.1.3.1 Software Engineering Tasks

Software defects

A defect occurs if the software is not behaving as per intended requirements and produces unexpected results [20]. It is also known as ’bug’, and the process of removing defects is usually known as debugging, and the research area is called software defect prediction (SDP) [20]. SDP could be used to detect defects within a single project (within project defect prediction WPDP) or among a collection of projects (cross-project defect prediction CPDP), [20]. Vulnerabilities are a subset of defects considered as weaknesses of the system, which can be attacked for malicious purposes [114]. A vulnerability is not always necessarily a defect, and it can also be caused by configuration or deployment parameters, or integration challenges. Common vulnerabilities and exposures (CVE) and common weakness enumeration

(CWS) are a few of the standard databases for vulnerabilities in software [21], [35], [44], [87].

Miscellaneous SE tasks

The defect and vulnerability detection task predicts a defect or vulnerability in the code and suggests possible fixes [36]. Code classification tasks can involve identifying the class of code, for instance, to check and compare the code submission by students with expected code solution at the university [83]. Analyzing the previous text tokens, DL algorithms can predict the next token such as the name of a program element such as variable or method [10]. Code summarization and review task includes generating rich text summaries such as docstring (usage description) of a function [100]. Code similarity detects clone of a code, for instance, to identify plagiarism in code [109]. Code search, generation and completion capabilities help to create and maintain documentation of software systems and assist in the development of software systems [96].

With gradual typing becoming popular in languages such as Python and TypeScript, the need to infer type annotations automatically arises [91]. Type inference capabilities can help to detect type based on the underlying data. The user can control the verbosity level of logs, which will be printed during program execution [106]. SE task build repair involves identifying and locating the errors in the build and suggesting the fix to create a successful build [99]. DL techniques can help to simulate the program behaviour and can even predict workload [103].

Address pre-fetching helps to model data flow and branch prediction helps to model control flow during execution [110]. Given an OpenCL kernel and a choice of two devices to run it on (CPU or GPU), a heterogeneous device mapping (DevMap) task is used in compiler analysis to predict the device which will give the best performance [83]. The thread coarsening task involves identifying the number of parallel threads which should be merged to obtain faster execution time [30]. Similarly, loop vectorization is an optimization technique for compilers to identify the number of instructions to pack together from different loop iterations [30].

5.1.3.2 Analysis

This analysis attempts to identify trends of GRL4SE task types and is shown in Figure 5.6.

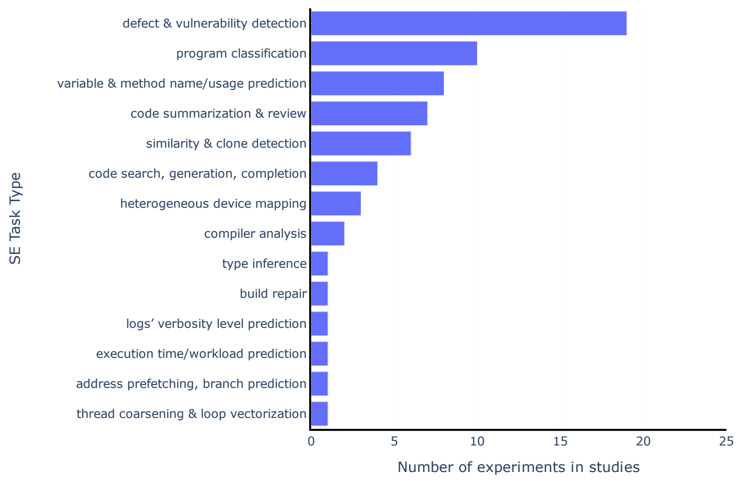


Figure 5.6 Analysis: SE task types

GRL4SE is the most widely used in defect and vulnerability detection tasks. Code classification, prediction of variable and method name, code summarization and review, similarity and clone detection are some other common applications of GRL4SE tasks, while other tasks such as type inference, and build repair among others are used seldom and listed in the table in the bottom area. We create the taxonomy for SE tasks and reference studies as given in table 5.2.

Task type	**References**
defect and vulnerability detection	[20], [22], [24], [36], [90], [97], [98], [102], [111], [21], [44], [92], [89], [35], [29], [87], [23], [30], [37]
code classification	[41], [83], [108], [42], [23], [31], [94], [12], [22], [110]
variable and method name/usage prediction	[10], [34], [93], [100] [111], [101], [33], [107]
code summarization and review	[26], [84], [86], [95], [100], [104], [105]
similarity and clone detection	[109], [28], [40], [22], [85], [19]
code search, generation, completion	[113], [96], [39], [93]
heterogeneous device mapping	[30], [83], [112]
compiler analysis	[83], [112]
address prefetching, branch prediction	[110]
type inference	[91]
logs' verbosity level prediction	[106]
build repair	[99]
execution time simulation and workload prediction	[103]
thread coarsening and loop vectorization	[30]

Table 5.2 Taxonomy: SE Tasks

5.1.3.3 Applications of code representations

Several studies used a graph-based representation of code, which we discuss here briefly. [115] used code represented in a graph-based format with a combination of token-based sequence and structure-based graphs. They train the model using a graph neural network and the gated recurrent unit used to fix and correct syntactical errors in the code. [116] created a program-derived semantics graph (PSG), a new graphical structure to capture the semantics of code, but they do not use it further for ML applications in SE. [117] performed a comparison between graph embedding techniques node2vec and Bag of Graphs. These graph embedding techniques helped to detect cryptography misuse in code. [118] converts code to graphs and stores this representation in a graph database. It is accessed to predict the class name recommendations by performing graph mining.

[119] uses graph-based code representation and GRL along with NLP for vulnerability analysis. [120] uses NLP techniques along with graph mining to generate a graph of inter-related requirements (IRR) to find clusters of code. [121] uses AST subtrees along with topic modelling (a NLP technique) to summarize functional concepts of defects, used further for defect prediction. [122] uses NLP query text and code represented as a combined graph-based format, which is used in graph matching and searching model to find the best matching code snippet.

[123] creates program interaction dependency graph (PIDG) for similarity-based plagiarism identification in code. [124] creates control flow graphs used to grade assignments by comparing the similarity between code submissions based on graph mining. [125] performs similarity detection by using TF-IDF and cosine similarity techniques to identify the cryptography dataset of CFG in C#.

[126] creates a graph from commit patches for just-in-time bug prediction across stages of code in the SLDC lifecycle. [127] uses meta-modelling and graph-based verification process using graph transformation in the DL programs to detect faults and design inefficiencies.

5.1.4 Research Question 3

5.1.4.1 Learning Types

Machine Learning can be classified as supervised, unsupervised, or semi-supervised, as per the availability, usage of the target variable and the end goal of the task. We analyze the learning types in the studies shown in Figure 5.7. We observe that most studies train the ML model in a supervised manner. Only two studies [12] and [33] utilized unsupervised ML, while only [102] used a self-supervised manner.

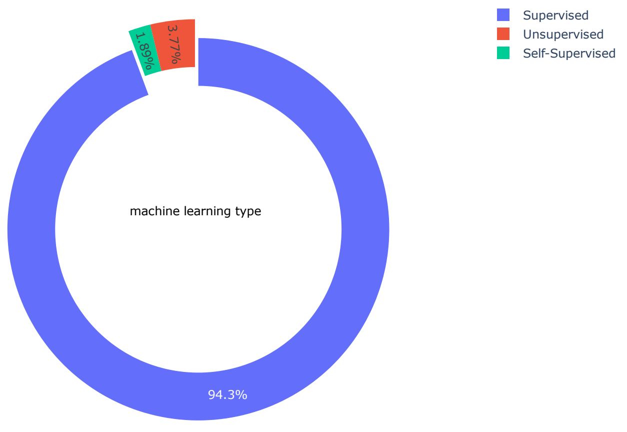


Figure 5.7 Analysis: (Machine) Learning types

5.1.4.2 Neural Network Types

Various types of neural network models and architectures have evolved in the GRL4SE. The initial models depended on the graph mining algorithms. Initially used in computer vision, convolution neural networks (CNN) derive multiple architectures based on convolution operations among graphs [26]. With the advent of message passing approaches, newer families of models included GNN-based models [25]. The attention mechanism has inspired GRL researchers to develop attention-based model architectures, which is an active research area today. The complete taxonomy of various model architectures is summarized and shown in the table 5.3.

Graph Networks	**Convolution-based and Recurrent Models**	**Graph Neural Network-based Models**	**Attention-based Models**
Graph Matching Network (GMN)	Graph Convolution Network (GCN)	Graph Neural Network (GNN)	Graph Attention Network (GAT)
Graph Isomorphism Network (GIN)	Convolutional Graph Neural Network (ConvGNN)	k-dimensional GNN (k-GNN)	Graph Relational Embedding Attention Transformer (GREAT)
	Crystal Graph Convolutional Network (CGCN)	Multi-Flow Graph Neural Network (MFGNN)	Gated Graph Attention Neural Network (GGANN)
	Deep Graph Convolutional Neural Network (DGCNN)	Gated Graph Neural Network (GGNN)	Attention-based Heterogeneous GNN (AHG)
	Directed Graph Convolutional Neural Network (DGCNN)	GraphSAGE (SAmple and aggreGatE)	Multi-hop Attention Graph Neural Network (MAGNA)
	Simplified Graph Convolution Network (SGC)	Graph Interval Neural Network (GINN)	Hetereogeneous Graph Transformer (HGT)
	Relational Graph Convolutional Network (RGCN)		GN-Transformer (GNT)
	Recurrent Graph Network (RGN)		

Table 5.3 Taxonomy: Neural network model families

5.1.4.3 Traditional methods

Traditional approaches consist of manual feature engineering [23] based on statistical metrics of source code such as Halstead, CK, McCabe's [36]. Traditional representations did not focus on basic blocks, which is the basis of contextual dependencies in the source code [22].

For bug detection, formal reasoning and combinatorial search are traditionally done manually by experts, which is a time-consuming process [102]. For vulnerability detection, traditionally used methods included static analysis, dynamic analysis, and symbolic execution. Static analysis, driven by rule-based reasoning, resulted in a high number of false positives [21], can often detect a limited set of bugs [98]. Dynamic analysis (fuzzing, dynamic taint analysis or symbolic execution) is often resource-hungry, and tedious, yet it does not have a comprehensive coverage [44], [92]. Also, these techniques work at the component or file level, not at the function level in the source code [89]. NLP-based methods for vulnerability detection often do not have a deep granularity, cannot handle cross-function vulnerabilities, and rely heavily on expert knowledge [37].

For clone detection, traditionally done using static analysis were able to target only type 1, 2, 3 clones (out of 4 clone types) [85], [19]. For code similarity, conventional methods have less precision and scalability, and similarity based on syntactic features is too restrictive [85]. Rule-based approaches based on predefined rules and templates were limited in types of summaries that could be generated [95].

In the task variable name or usage prediction, traditional methods include static analyzers [10] and NLP techniques. However they often have problems of out-of-vocabulary predictions, as vocabulary cannot be fixed for code words [34], [91], [106]. Traditional approaches often faced problems in generalization due to training on synthetic data [35], long-range dependency capture issues [104], [42].

5.1.4.4 GRL Methods

Role of GRL

Here, we discuss how the introduction of GRL methods has helped to overcome challenges with traditional methods. Syntactic and semantic properties of code can be best retained with a graphical representation [10], using the embedding layer of code [21], [101]. [110] suggested utilizing GNN to learn fused representations of source code and its execution together. [104], [89], [106] utilized GNNs to create code representations.

[104] developed the Transformer-XL model to alleviate problems of long-range dependency. [86] used a graph diffusion mechanism to model longer-range token dependencies adequately. [34] developed a graph-structured cache for out-of-vocabulary problems in NLP-based techniques. [20] used SMOTETomek sampling to handle a class imbalance in defect prediction approaches.

[28] suggested using capsules to reduce mean square error for similarity detection. [103] suggested using a graph neural network as a surrogate model of a compiler. Therefore, instead of a large search space, a surrogate model is trained to predict the runtime behaviour of programs. As all the relationships in code are not equal, the attention mechanism helps to learn the most important tokens in code [26].

Benefits and impact

GRL reduced the requirements on amounts of training data, model capacity, and training regime [10]. Use of GRL improved performance of models on various metrics such as accuracy, precision, recall, specifically reducing the false positive rates [34], [21], [41], [110], [106], [19], [108], [29], [107], [42], [23]. GRL-based architectures are highly scalable, generalize well on unseen data [109], [111], [86].

5.1.5 Research Question 4

5.1.5.1 Dataset Types

We analyze the different datasets to identify the most common ones for various applications in experiments. This is shown in the Figure 5.8. Here we show only the top 10 most frequently analyzed datasets. Programming online judge (POJ), Juliet test suite, PROMISE and SARD are the most commonly used datasets. Dataset choice is heavily dependent on the SE task under consideration. Each study in SLR did not necessarily experiment, while few studies conducted multiple experiments.

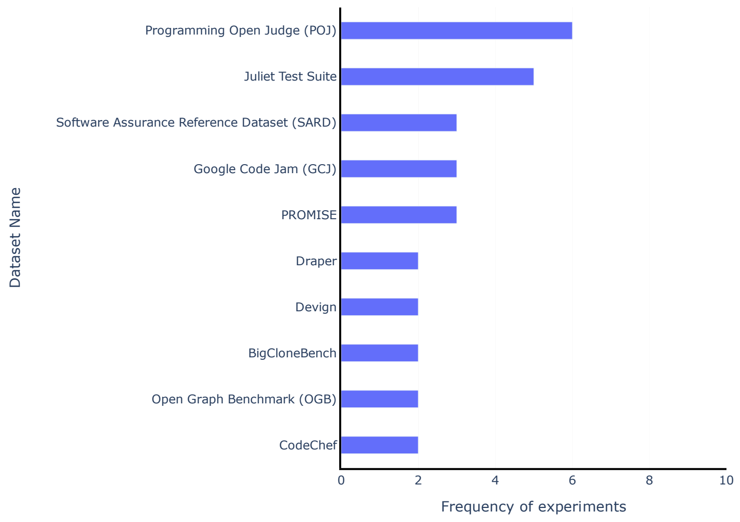


Figure 5.8 Analysis: Number of experiments for dataset types

5.1.5.2 Languages

Here, we attempt to identify the common programming languages analyzed in the empirical experimental studies. This analysis is in Figure 5.9 showing the top 10 most frequent languages. We observe that Java is mostly empirically analyzed. Other frequently used languages are C, C++ and Python. Decision choice of language can be based on the background knowledge and familiarity of the researcher and the availability of suitable technical tools.

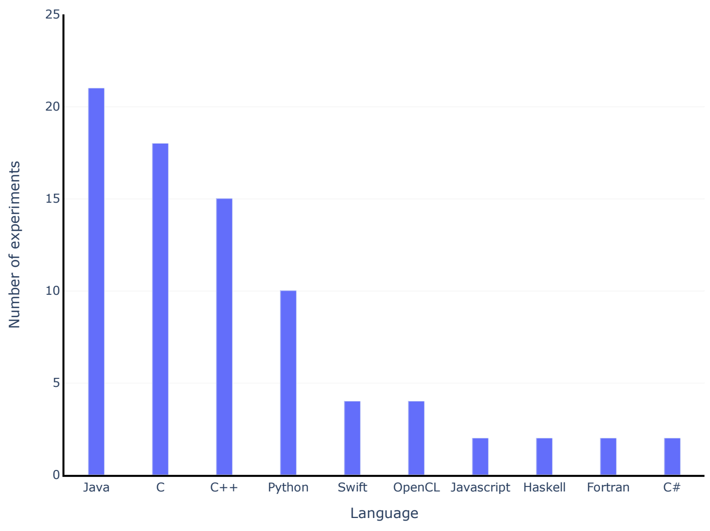


Figure 5.9 Analysis: Dataset Languages

5.1.5.3 Evaluation Metrics

A summary of various metrics and their description is depicted in the table 5.4.

Metric Name	**Description**
Accuracy	number of correct predictions out of all predictions in a dataset
Precision	number of correctly predicted relevant data out of all retrieved data
Recall	number of the correctly predicted relevant data out of all relevant data
F1 Score	harmonic mean of precision and recall, a combined measure of effectiveness
Area Under Curve **AUC Score**	relationship between true positive rate (TPR) and false positive rate(FPR)
Recall-Oriented Understudy for Gisting Evaluation **ROUGE Score**	evaluation of machine summary/translation against human-produced references
Bilingual Evaluation Understudy **BLEU Score**	similarity between two sentences in evaluation of machine translation systems

Table 5.4 Summary: Evaluation Metrics

Usually, for evaluation purposes, ML models utilize metrics such as accuracy, elements of confusion matrix or derived metrics such as precision, recall and F1 score [128]. Since the training set is usually class imbalanced, i.e. contains a large number of negative examples with rare positive examples, a ML model can offer high accuracy just by predicting the negative class irrespective of the input, hence accuracy is not a reliable metric [128], [129].

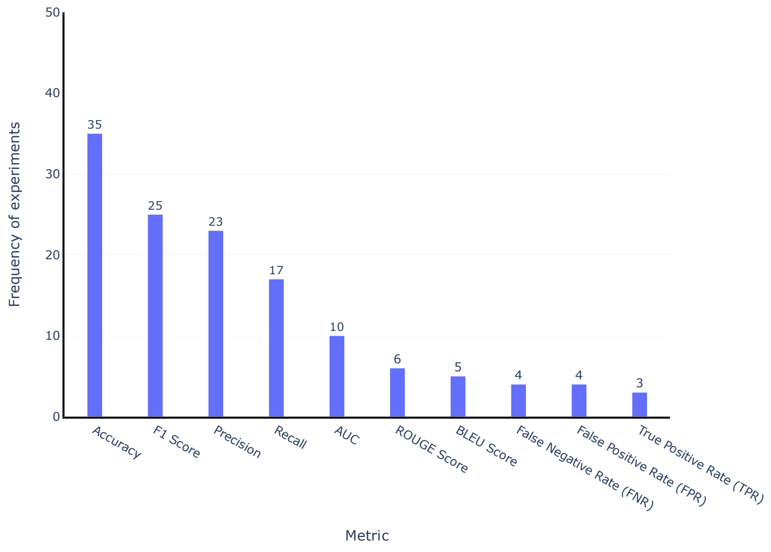


Figure 5.10 Analysis: Evaluation Metrics

The analysis for the top 10 evaluation metrics is in Figure 5.10. We observe that even though not a reliable metric, accuracy is the most commonly used, followed by F1 score, precision, recall and AUC. ROUGE-L, METEOR, and BLEU scores are other evaluation metrics for specific tasks such as machine summary or translation.

5.2 Discussion

5.2.1 Combined Taxonomy of GRL4SE field

We aim to create a combined taxonomy with hierarchy-based levels to get a high-level overview of GRL4SE (Figure 5.11). The first level groups different SE tasks, and the next layers provides information about the representation type, DL model, and example study. For example, we consider the pioneer study [10] for the task variable name prediction. Next, we observe that the representation type is AST, and the model architecture used is a gated graph neural network (GGNN). At the leaf nodes, we see that multiple studies (RS1, RS3, RS43) conducted similar experimental task.

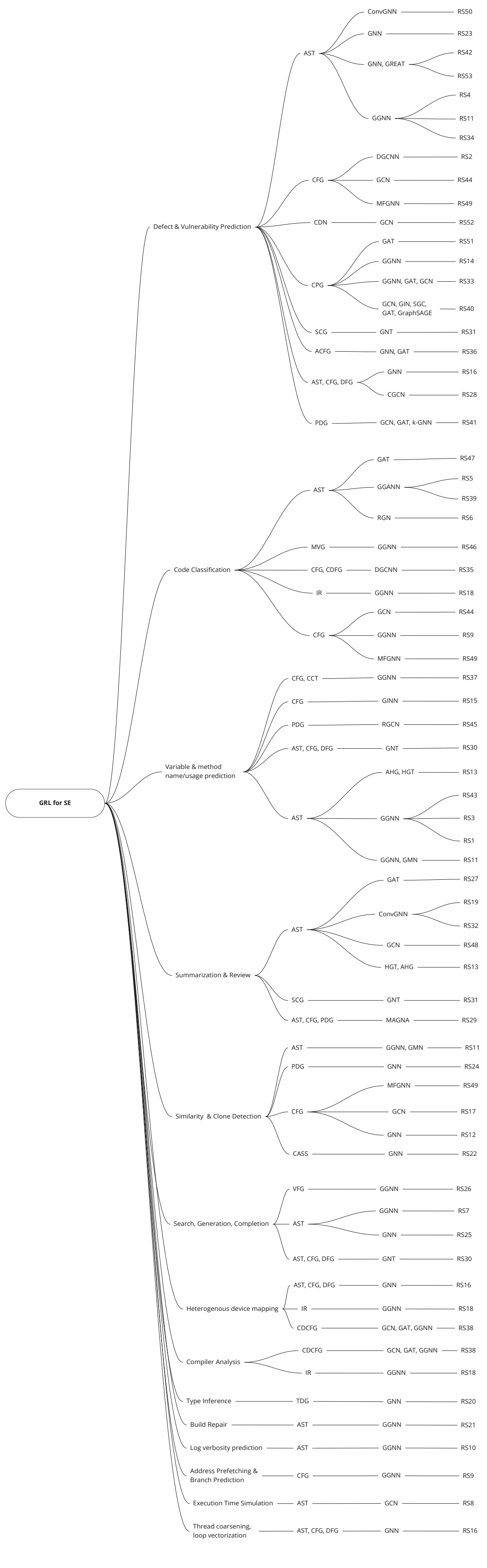


Figure 5.11 Taxonomy: GRL for SE (Zoom-in for the best view)

5.2.2 Evaluation: Graph-based and traditional approaches

As per the SE task in context, we evaluate different approaches based on their effectiveness in terms of performance (considering evaluation metrics), limitations, and threat to validity.

Defect and vulnerability detection

Graph-based approaches provide better results than traditional approaches such as FlawFinder [37], [87], logistic regression classifier, random forest classifier [97], support vector machines (SVM) [23], [37] - Bag Of Words [22], [36], CNN [21], [44], [89], [92], [97], RNN [44], [89], TBCNN [23], [36], LSTM [23], Bi-LSTM [21], [37], [92].

Limitations include input code expected to be stripped, compile-able [29], model trained on synthetic rather than real datasets [89], generalization based on chosen dataset [97], missing projects in datasets [22], or data annotation issues [44], [87].

Program classification

Graph representation methods are found to give better performance than non-graph approaches such as static mapping [83], TBCNN [23], [31], [41], [42], [83], LSTM [23], [83], [110], SVM [22], [23], Weisfeiler-Lehman Test, ideal static selector (ISS) [94].

Challenges include limited dataset with missing projects, training data only in one language with possible outliers [22].

Prediction of name/usage for variable and methods

Approaches using graph representation are found to provide better results than alternative conventional approaches such as BERT [107], word2vec, bi-directional RNN [10], code2seq, code2vec [101], TransCodeSum [100].

The main challenges are limited data evaluation scope, coverage of transformations, and manual processes of bug review and inspection [101].

Code summarization and review

Graph-based approaches performed better than vanilla transformers [86], tree2seq [84], graph2seq, code2seq, Bi-LSTM [26], [95], [104], Tree-based RNN[105].

Some challenge include impact of neighboring classes for code summary generation [95], inter-rater bias among human evaluators [26], [95], large training time [105], possibility of data leaks [104].

Similarity and clone detection

Graphical methods outperform traditional approaches such as tree-based convolution [85], RtvNN (RNN-based clone detector), CDLH (Clone Detection with Learning to Hash), [19], code2vec, code2seq, seq-RNN, seq-transformer, Bag-of-features (BoF), recurrent neural network (RNN) [40], SVM [22].

The main challenges are the requirement of compile-able code, static analysis to generate PDG, inability to extend to incomplete programs [85], and limited program size processing [109].

Code search, generation, completion

Graph-based representation methods are found to perform better than LSTM [39], plain transformer [39], [93]. Major challenges are the small dataset and scalability because of LLVM IR, which can only extract the program with complete dependencies [113].

Heterogeneous device mapping and compiler analysis

Graphical approaches outperform traditional ones as inst2vec [112], LSTM, TBCNN, static mapping [83]. Limitations include simplifying representation vocabulary of instruction and operand types, addressing class imbalance [83].

Miscellaneous tasks

For the task of address pre-fetching and branch prediction, graphical methods outperform traditional baselines such as stride data prefetcher, address correlation (AC) prefetcher, LSTM-delta [110]. For type inference, graphical approaches outperform conventional methods such as inference by TypeScript compiler [91], and the limitation is the simplified treatment of function types and generic types (i.e., collapsing them into their non-generic counterparts) [91].

For logs' verbosity level prediction task, graphical approaches outperform other baselines such as RNN - LSTM model [106]. A challenge is a poor generalization on unseen data because of differences in logging styles of projects. For the build repair task, graph-based approaches outperform other methods such as sequence-to-sequence [99] with the ability to improve performance by including more context and performing more propagation steps. A limitation is to understand the type signatures of rare methods. For the task of prediction of execution time and workload, graphical approaches generalize better than those without graph convolution component [103].

5.2.3 Adversarial attacks

In DL, neural networks are often vulnerable to adversarial inputs, which intentionally lead the model to make a specific (incorrect) prediction [130]. The basic idea is to add intentionally-created noise to the correctly labelled inputs so that model will give desired incorrect output. In a non-targeted attack, the incorrect output is caused by reasons tracing back not only to changes in input but instead to systemic errors in the ecosystem. Since a program code is a discrete object having certain syntactic and semantic properties, adding specific noise types such as those used for the attack on image input is not possible [32]. [130] developed the Discrete Adversarial Manipulation of Programs (DAMP) framework to generate targeted adversarial attack examples.

The defence against such an attack includes two types of techniques. The first group of methods contains a gatekeeper element to check the incoming inputs and helps to use existing models without retraining needs. The second set of methods requires retraining the model using a modified training dataset or with a different loss function than the original. Some techniques can drop the success rates of attacks significantly, with a small cost of 2% degradation in accuracy [32], [130]. Such methods involve studying various transformations in input and their impact on the outputs, so that robustness of models can be improved [101]. Such rewrite of code can help to enhance generalization capabilities as well [35], [102].

5.2.4 Challenges and implications

Major challenges are explainability and interpretability of the model [33], [35], [92], limited generalization to unseen datasets [22], [92], [101], extensibility to other languages, SE tasks and data types [23], [41], [91], [97], [100], [113]. Another major challenge is the lack of distributed representations for compiled code [23]. Investigation of false positives, training on program slices to reduce noise in data, and cross-project prediction [21], [89] are a few open challenges [44], [102]. Reducing the computation time by increasing training speed [41], or using pre-trained data [31], are also some open research challenges.

Implications

Computational costs could be high for large graphs, increasing training time [34], [107]. The major implication is the requirement of sophisticated hardware workstations with high computational capabilities, a large amount of memory, and multiple GPUs [10], [12], [19], [26], [29], [30], [35], [44], [83], [87], [95], [98], [102]–[104], [108], [109], [111].

Chapter 6

Conclusion

6.1 Conclusion

In this study, we present results from a SLR analysis from top research venues, with a methodology based on guidelines by [72]. The process began by defining a set of research questions about the applications of graph-based DL models to SE tasks. A pilot search using a base query string helped refine the query string as per relevant studies found in the results. We conducted a literature search on ACM, IEEE Xplore, ArXiv, and ML4Code repositories. By following quality assessment protocols, our study surveys 53 primary studies. As all studies are published within the last five years, it is evident that machine learning and software engineering communities are conducting high research activity in this area. Based on the information extracted, we identified answers to research questions and created taxonomies on various aspects of the GRL approach to SE tasks. This SLR provides future researchers with the necessary insights to apply GRL for applications within the SE field.

The first research question was to identify types of code and graphical representations. As per the survey results, source code is the most widely used and represented as syntactic structures in AST, IR, or as flow graphs such as CFG, DFG, or as dependency graphs such as PDG, CDN, or TDG. The second RQ aimed to determine applications of GRL in SE tasks. We identified various SE tasks with defect and vulnerability prediction, program classification, prediction of name and usage of methods and variables, code summarization, and clone detection as prevalent tasks.

The third RQ was about finding learning types, GRL models, benefits and implications. Supervised is the most common learning type, attention-based deep neural networks are increasingly gaining traction, and GRL techniques outperform traditional techniques, such as those from non-graph methods such as code-as-text approaches of NLP. The fourth research question aimed to identify datasets, languages, and evaluation metrics. POJ, Juliet Suite, SARD, GCJ, and PROMISE are a few most commonly used datasets, while Java, C, C++, and Python as commonly tested languages. Accuracy, F1 score, precision, recall, and AUC are a few top evaluation metrics, along with ROUGE and BLEU scores used in machine summary or translation tasks. Also, we provided a combined taxonomy of the field, which helps to identify specific representation and DL model architecture and an example reference study to follow. The implications include high computational costs, explainability, interpretability, and extensibility to other languages and projects.

6.2 Threats to validity

Threats to validity could be analyzed from two aspects: bias and validity. We consider study selection bias, researcher bias, publication bias and internal and external validity.

Bias

Study selection relies on the search strategy, research repositories, selection criteria and quality analysis. As the search query string is created with terms matching to research questions, the query string may have missed some studies on the borderline between SE, DL, GRL, and ML fields. Some studies may use other terms in their title, abstract, and keywords, hence we might miss finding them. To address this, we chose a combination of principal and supplementary research sources and modified query strings as per common terminologies and guidelines of databases. The study selection and quality assessment in screening studies were conducted and reviewed by a single author, posing a threat to the reliability aspect.

The study does not address systemic biases that occurred due to publication bias, as the positive results are likely to be published as compared to negative ones. Also, researchers tend to claim their methods are the best and outperform others on chosen datasets. This may lead to an overestimation of performance. We tried to address this by creating quality assessment questions to avoid bias toward some particular models. Also, as we did not consider grey literature, publication bias is likely to exist to some extent.

Validity

Internal validity is a prerequisite for external validity. Threats to internal validity could be due to errors in data extraction. To decrease the inaccurate data extraction threat, we improved the data assessment form iteratively to address the research questions correctly. External validity is the extent of effects observed in the study to be applicable in unknown environments, hence generalization and direct applicability to a new SE task are limited. This work does not include an experiment-based approach and relies on the claims of other research articles about results so possesses a limited external validity.

6.3 Summary

Though GRL presents solutions to a wide range of SE tasks, however, traditional methods are still cost-effective for simple and repeatable use cases because of little resource and computation needs. GRL4SE is complementary to traditional approaches and helps to overcome challenges in solving complex problems for advanced use cases.

6.4 Future work

Based on a large number of research articles collected, hierarchical clusters with mapping of SE task to SDLC stages can be created [14]. As the study provides the reader with relevant information to apply GRL in the SE context, we propose two potential research possibilities. The first possibility could be: given a system with language and framework of choice, can we use GRL to predict the input software module as one of the model, view and controller (MVC) components? In our research, we found that MVC components overlap in terms of underlying code, and defining well-defined boundaries to separate them is an open research challenge. Another potential research avenue can be: given a system with language and framework of choice, can we use GRL to predict a software module as either a front-end or back-end component?

Appendix A

Appendix

A.1 ACM Search Query String

```
[[Title: code] OR [Title: software]
OR [Title: program]] AND [Title: graph]
AND [[Abstract: source] OR [Abstract: code]
OR [Abstract: software] OR [Abstract: program*]
OR [Abstract: synta*] OR [Abstract: semantic]
OR [Abstract: represent*] OR [Abstract: engine*]
OR [Abstract: "se"]] AND [[Abstract: graph]
OR [Abstract: tree]] AND [[Abstract: neural]
OR [Abstract: net*] OR [Abstract: "?nn"]
OR [Abstract: learn*] OR [Abstract: machine]
OR [Abstract: deep] OR [Abstract: "ml"]
OR [Abstract: "dl"] OR [Abstract: data*]]
AND [Publication Date: (03/01/2012 TO 03/31/2022)]
```

A.2 IEEE Xplore Search Query String

```
("Document Title":Code OR "Document Title":Software
OR "Document Title":Program) AND ("Document Title":Graph)
AND ("All Metadata":Source OR "All Metadata":Code
OR "All Metadata":Software OR "All Metadata":Program*
OR "All Metadata":Synta* OR "All Metadata":Semantic
OR "All Metadata":Represent* OR "All Metadata":Engine*
OR "All Metadata":"SE") AND ("All Metadata":Graph
OR "All Metadata":Tree) AND ("All Metadata":Neural
OR "All Metadata":Net* OR "All Metadata":"?NN"
OR "All Metadata":Machine OR "All Metadata":Deep
OR "All Metadata":Learn* OR "All Metadata":"ML"
OR "All Metadata":"DL" OR "All Metadata":data*)
Filters Applied: 2012 - 2022
```

A.3 ArXiv Search Query String

```
size: 100; date_range: from 2012-01-01 to 2022-12-31;
classification: Computer Science (cs);
include_cross_list: True; terms:
AND title=Code OR Software OR Program;
AND title=Graph; AND abstract=Source OR Code
OR Software OR Program* OR Synta*
OR Semantic OR Represent* OR Engine* OR "SE";
AND abstract=Graph OR Tree;
AND abstract=Neural OR Net* OR "?NN"
OR Machine OR Deep OR Learn* OR "ML" OR "DL" OR data*
```

A.4 Software Tools

Programming	Python, Jupyter Notebook
SLR Automation	Parsifal, ASReview, Rayyan, Thoth
Reference Manager	Mendeley Desktop, Mendeley Reference Manager (complementary features), JabRef, Bibtool
Diagrams	Draw.io Diagrams, Miro, Inkscape
TeX	TeXstudio, Overleaf
VPN	WireGuard
PDF Reader	Sumatra PDF
Data Collection	Microsoft Excel
Markdown Files	Zettlr

Table A.1 Software tools used in this study

A.5 Selected Research Studies for SLR

ID	Title	Authors	Year
RS1	Learning to represent programs with graphs	Allamanis, Miltiadis; Brockschmidt, Marc; Khademi, Mahmoud	2018
RS2	Convolutional neural networks over control flow graphs for software defect prediction	Phan, Anh Viet; Nguyen, Minh Le; Bui, Lam Thu	2018
RS3	Open Vocabulary Learning on Source Code with a Graph-Structured Cache	Cvitkovic, Milan; Singh, Badal; Anandkumar, Anima	2018
RS4	Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks	Zhou, Yaqin; Liu, Shangqing; Siow, Jingkai; et al.	2019
RS5	Program Classification Using Gated Graph Attention Neural Network for Online Programming Service	Lu, Mingming; Tan, Dingwu; Xiong, Naixue; et al.	2019
RS6	Unsupervised Classifying of Software Source Code Using Graph Neural Networks	Vytovtov, Petr; Chuvilin, Kirill	2019
RS7	Generative Code Modeling with Graphs	Brockschmidt, Marc; Allamanis, Miltiadis; Gaunt, Alexander	2019
RS8	Simulating Execution Time of Tensor Programs using Graph Neural Networks	Tomczak, Jakub M.; Lepert, Romain; Wiggers, Auke	2019
RS9	Learning Execution through Neural Code Fusion	Shi, Zhan; Swersky, Kevin; Tarlow, Daniel; et al.	2019
RS10	Using GGNN to recommend log statement level	Li, Mingzhe; Pei, Jianrui; He, Jin; et al.	2019
RS11	Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree	Wang, Wenhan; Li, Ge; Ma, Bo; et al.	2020
RS12	FuncGNN: A graph neural network approach to program similarity	Nair, Aravind; Roy, Avijit; Meinke, Karl	2020
RS13	Learning to Represent Programs with Heterogeneous Graphs	Wang, Wenhan; Zhang, Kechi; Li, Ge; et al.	2020
RS14	Learning to map source code to software vulnerability using code-as-a-graph	Suneja, Sahil; Zheng, Yunhui; Zhuang, Yufan; et al.	2020
RS15	Learning semantic program embeddings with graph interval neural network	Wang, Yu; Wang, Ke; Gao, Fengjuan; et al.	2020
RS16	Deep Program Structure Modeling Through Multi-Relational Graph-based Learning	Ye, Guixin; Tang, Zhanyong; Wang, Huanting; et al.	2020
RS17	Code Characterization with Graph Convolutions and Capsule Networks	Haridas, Poornima; Chennupati, Gopinath; Santhi, Nandakishore; et al.	2020
RS18	ProGraML: Graph-based Deep Learning for Program Optimization and Analysis	Cummins, Chris; Fisches, Zacharias V.; Ben-Nun, Tal; et al.	2020
RS19	Improved code summarization via a graph neural network	LeClair; Alexander Haque; Sakib Wu; et al.	2020
RS20	LambdaNet: Probabilistic Type Inference using Graph Neural Networks	Wei, Jiayi; Goyal, Maruth; Durrett, Greg; et al.	2020
RS21	Learning to Fix Build Errors with Graph2Diff Neural Networks	Tarlow, Daniel; Moitra, Subhodeep; Rice, Andrew; et al.	2020
RS22	MISIM: A Neural Code Semantics Similarity System Using the Context-Aware Semantics Structure	Ye, Fangke; Zhou, Shengtian; Venkat, Anand; et al.	2020
RS23	HOppity : LEarning GRaph TRansformations DEtect and FIx BUgs in PRograms	Tech, Georgia; Wang, Ke	2020
RS24	Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks	Mehrotra, Nikita; Agarwal, Navdha; Gupta, Piyush; et al.	2021
RS25	Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs	Wang, Yanlin; Li, Hui	2021
RS26	deGraphCS: Embedding Variable-based Flow Graph for Neural Code Search	Zeng, Chen; Yu, Yue; Li, Shanshan; et al.	2021
RS27	CoCoSum: Contextual Code Summarization with Multi-Relational Graph Neural Network	Wang, Yanlin; Shi, Ensheng; Du, Lun; et al.	2021
RS28	Software Vulnerability Detection via Deep Learning over Disaggregated Code Graph Representation	Zhuang, Yufan; Suneja, Sahil; Thost, Veronika; et al.	2021
RS29	Graph Conditioned Sparse-Attention for Improved Source Code Understanding	Cheng, Junyan; Fostiropoulos, Iordanis; Boehm, Barry	2021
RS30	Learning to Extend Program Graphs to Work-in-Progress Code	Li, Xuechen; Maddison, Chris J.; Tarlow, Daniel	2021
RS31	GN-Transformer: Fusing Sequence and Graph Representation for Improved Code Summarization	Cheng, Junyan; Fostiropoulos, Iordanis; Boehm, Barry	2021
RS32	Transformer-XL with Graph Neural Network for Source Code Summarization	Zhang, Xiaoling; Yang, Shouguo; Duan, Luqian; et al.	2021
RS33	Vulnerability Detection in C/C++ Source Code With Graph Representation Learning	Wu, Yuelong; Lu, Jintian; Zhang, Yunyi; et al.	2021
RS34	Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection	Wang, Huanting; Ye, Guixin; Tang, Zhanyong; et al.	2021
RS35	Graphs based on IR as Representation of Code	Faustino, Anderson	2021
RS36	BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network	Ji, Yuede; Cui, Lei; Huang, H. Howie	2021
RS37	GRAPHSPY: Fused Program Semantic Embedding through Graph Neural Networks for Memory Efficiency	Guo, Yixin; Li, Pengcheng; Luo, Yingwei; et al.	2021
RS38	Learning Code Representations Using Multifractal-based Graph Networks	Ma, Guixiang; Xiao, Yao; Capota, Mihai; et al.	2021
RS39	Student Program Classification Using Gated Graph Attention Neural Network	Lu, M.; Wang, Y.; Tan, D.; et al.	2021
RS40	Software Defect Prediction for Specific Defect Types based on Augmented Code Graph Representation	Xu, Jiaxi; Ai, Jun; Shi, Tao	2021
RS41	DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network	Cheng, Xiao; Wang, Haoyu; Hua, Jiayi; et al.	2021
RS42	Self-Supervised Bug Detection and Repair	Allamanis, Miltiadis; Jackson-Flux, Henry; Brockschmidt, Marc	2021
RS43	On the generalizability of Neural Program Models with respect to semantic-preserving program transformations	Rabin, Md Rafiqul Islam; Bui, Nghi D. Q.; Wang, Ke; et al.	2021
RS44	Bin2vec: learning representations of binary executable programs for security tasks	Arakelyan, Shushan; Arasteh, Sima; Hauser, Christophe; et al.	2021
RS45	Universal Representation for Code	Liu, Linfeng; Nguyen, Hoan; Karypis, George; et al.	2021
RS46	Multi-View Graph Representation for Programming Language Processing: An Investigation into Algorithm Detection	Long, Ting; Xie, Yutong; Chen, Xianyu; et al.	2022
RS47	Algorithm Selection for Software Verification using Graph Attention Networks	Leeson, Will; Dwyer, Matthew B.	2022
RS48	Turn Tree into Graph: Automatic Code Review via Simplified AST Driven Graph Convolutional Network	Wu, B.; Liang, B.; Zhang, X.	2022
RS49	Precise Learning of Source Code Contextual Semantics via Hierarchical Dependence Structure and Graph Attention Networks	Zhao, Zhehao; Yang, Bo; Li, Ge; et al.	2022
RS50	Graph Neural Network for Source Code Defect Prediction	Sikic, Lucija; Kurdija, Adrian Satja; Vladimir, Klemo; et al.	2022
RS51	Vulmg: A Static Detection Solution For Source Code Vulnerabilities Based On Code Property Graph and Graph Attention Network	Haojie, Zhang; Yujun, Li; Yiwei, Liu; et al.	2022
RS52	GCN2defect : Graph Convolutional Networks for SMOTETomek-based Software Defect Prediction	Zeng, Cheng; Zhou, Chun Ying; Lv, Sheng Kai; et al.	2022
RS53	HEAT: Hyperedge Attention Networks	Georgiev, Dobrik; Brockschmidt, Marc; Allamanis, Miltiadis	2022

Table A.2 Selected research studies for SLR

A.6 Journals and conferences names of studies

Journal or Conference Name	Full Name
AAAI	Association for the Advancement of Artificial Intelligence
ArXiv	ArXiv
CCS	Computer and Communications Security
CCWC	Computing and Communication Workshop and Conference
Cybersecurity	Cybersecurity
DAC	Design Automation Conference
DSA	Dependable Systems and Their Applications
ESEM	Empirical Software Engineering and Measurement
ICBD	International Conference on Big Data
ICCWAMTIP	International Computer Conference on Wavelet Active Media Technology and Information Processing
ICLR	International Conference on Learning Representations
ICML	International Conference on Machine Learning
ICPC	International Conference on Program Comprehension
ICSEW	International Conference on Software Engineering Workshops
ICST	International Conference on Software Testing, Verification and Validation
ICTAI	International Conference on Tools with Artificial Intelligence
IEEE Access	IEEE Access
ISSRE	International Symposium on Software Reliability Engineering
IST	Information and Software Technology
JSS	Journal of Systems and Software
LNAI	Lecture Notes in Artificial Intelligence
NIPS	Neural Information Processing Systems
OIA	Open Innovations Association
PACMPL	Proceedings of the ACM on Programming Languages
PACT	Parallel Architectures and Compilation Techniques
SANER	Software Analysis, Evolution, and Reengineering
SBLP	Brazilian Symposium on Programming Languages
SMC	International Conference on Systems, Man, and Cybernetics
TIFS	IEEE Transactions on Information Forensics and Security
TOSEM	Transactions on Software Engineering and Methodology
TSE	IEEE Transactions on Software Engineering

Table A.3 Journals and Conferences Names

Bibliography

[1] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[2] M. M. Bronstein, J. Bruna, T. Cohen, and P. Veličković, "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges," *arXiv preprint arXiv:2104.13478*, 2021.

[3] D. S.-D. Jérôme Tubiana Haim J. Wolfson, "Scannet: An interpretable geometric deep learning model for structure-based protein binding site prediction," Nature, 2022. DOI: 10.1038/s41592-022-01490-7.

[4] M. Cheung and J. M. F. Moura, "Graph neural networks for covid-19 drug discovery," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 5646–5648. DOI: 10.1109/BigData50022.2020.9378164.

[5] M. Pradel and S. Chandra, "Neural software analysis," *Communications of the ACM*, vol. 65, pp. 86–96, 1 Nov. 2022, ISSN: 15577317. DOI: 10.1145/3460348. [Online]. Available: http://arxiv.org/abs/2011.07986.

[6] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–36, 2018, ISSN: 15577341. DOI: 10.1145/3212695. arXiv: 1709.06182.

[7] P. Devanbu, M. Dwyer, S. Elbaum, *et al.*, "Deep learning software engineering: State of research and future directions," April 2020 Sep. 2020. [Online]. Available: http://arxiv.org/abs/2009.08525.

[8] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *arXiv preprint arXiv:1709.05584*, 2017.

[9] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning*, PMLR, 2014, pp. 649–657.

[10] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, Nov. 2018. arXiv: 1711.00740. [Online]. Available: http://arxiv.org/abs/1711.00740.

[11] V. Romanov, V. Ivanov, and G. Succi, "Approaches for Representing Software as Graphs for Machine Learning Applications," in *Proceedings - 2020 International Computer Symposium, ICS*, Institute of Electrical and Electronics Engineers Inc., Dec. 2020, pp. 529–534, ISBN: 9781728192550. DOI: 10.1109/ICS51289.2020.00109.

[12] P. Vytovtov and K. Chuvilin, "Unsupervised classifying of software source code using graph neural networks," vol. 2019-April, IEEE, Apr. 2019, pp. 518–524, ISBN: 978-952-68653-8-6. DOI: 10.23919/FRUCT.2019.8711909. [Online]. Available: https://ieeexplore.ieee.org/document/8711909/.

[13] S. Shafiq, A. Mashkoor, C. Mayr-Dorn, and A. Egyed, "A literature review of using machine learning in software development life cycle stages," *IEEE Access*, vol. 9, pp. 140 896–140 920, 2021, ISSN: 21693536. DOI: 10.1109/ACCESS.2021.3119746. [Online]. Available: https://ieeexplore.ieee.org/document/9568959/.

[14] O. T. Borges, J. C. Couto, D. D. A. Ruiz, and R. Prikladnicki, "How machine learning has been applied in software engineering?," vol. 2, SCITEPRESS - Science and Technology Publications, 2020, pp. 306–313, ISBN: 9789897584237. DOI: 10.5220/0009417703060313. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0009417703060313.

[15] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013, ISSN: 01628828. DOI: 10.1109/TPAMI.2013.50. arXiv: 1206.5538. [Online]. Available: http://ieeexplore.ieee.org/document/6472238/.

[16] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015, ISSN: 14764687. DOI: 10.1038/nature14539.

[17] F. Chen, Y. C. Wang, B. Wang, and C. C. Kuo, "Graph representation learning: A survey," *APSIPA Trans. Signal Inf. Process.*, vol. 9, 2020, ISSN: 20487703. DOI: 10.1017/ATSIP.2020.13. arXiv: 1909.00958.

[18] W. L. Hamilton, *Graph Representation Learning*. 2020.

[19] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," IEEE, Feb. 2020, pp. 261–271, ISBN: 9781728151434. DOI: 10.1109/SANER48275.2020.9054857. [Online]. Available: https://ieeexplore.ieee.org/document/9054857/.

[20] C. Zeng, C. Y. Zhou, S. K. Lv, P. He, and J. Huang, "Gcn2defect : Graph convolutional networks for smotetomek-based software defect prediction," IEEE, Oct. 2022, pp. 69–79, ISBN: 978-1-6654-2587-2. DOI: 10.1109/issre52982.2021.00020. [Online]. Available: https://ieeexplore.ieee.org/document/9700305/.

[21] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in Neural Information Processing Systems*, vol. 32, pp. 1–11, NeurIPS Sep. 2019, ISSN: 10495258. [Online]. Available: http://arxiv.org/abs/1909.03496.

[22] Z. Zhao, B. Yang, G. Li, H. Liu, and Z. Jin, "Precise learning of source code contextual semantics via hierarchical dependence structure and graph attention networks," *Journal of Systems and Software*, vol. 184, p. 111 108, Feb. 2022, ISSN: 01641212. DOI: 10.1016/j.jss.2021.111108. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121221002053.

[23] S. Arakelyan, S. Arasteh, C. Hauser, E. Kline, and A. Galstyan, "Bin2vec: Learning representations of binary executable programs for security tasks," *Cybersecurity*, vol. 4, p. 26, 1 Dec. 2021, ISSN: 2523-3246. DOI: 10.1186/s42400-021-00088-4. [Online]. Available: https://cybersecurity.springeropen.com/articles/10.1186/s42400-021-00088-4.

[24] D. Georgiev, M. Brockschmidt, and M. Allamanis, "Heat: Hyperedge attention networks," Jan. 2022. [Online]. Available: http://arxiv.org/abs/2201.12113.

[25] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[26] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," ACM, Jul. 2020, pp. 184–195, ISBN: 9781450379588. DOI: 10.1145/3387904.3389268. [Online]. Available: https://doi.org/10.1145/3387904.3389268%20https://dl.acm.org/doi/10.1145/3387904.3389268.

[27] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[28] P. Haridas, G. Chennupati, N. Santhi, P. Romero, and S. Eidenbenz, "Code characterization with graph convolutions and capsule networks," *IEEE Access*, vol. 8, pp. 136 307–136 315, 2020, ISSN: 21693536. DOI: 10.1109/ACCESS.2020.3011909. [Online]. Available: https://ieeexplore.ieee.org/document/9149622/.

[29] Y. Ji, L. Cui, and H. H. Huang, "Buggraph: Differentiating source-binary code similarity with graph triplet-loss network," ACM, May 2021, pp. 702–715, ISBN: 9781450382878. DOI: 10.1145/3433210.3437533. [Online]. Available: https://doi.org/10.1145/3433210.3437533%20https://dl.acm.org/doi/10.1145/3433210.3437533.

[30] G. Ye, Z. Tang, H. Wang, *et al.*, "Deep program structure modeling through multi-relational graph-based learning," ACM, Sep. 2020, pp. 111–123, ISBN: 9781450380751. DOI: 10.1145/3410463.3414670. [Online]. Available: https://doi.org/10.1145/3410463.3414670%20https://dl.acm.org/doi/10.1145/3410463.3414670.

[31] T. Long, Y. Xie, X. Chen, W. Zhang, Q. Cao, and Y. Yu, "Multi-view graph representation for programming language processing: An investigation into algorithm detection," 1 Feb. 2022. [Online]. Available: http://arxiv.org/abs/2202.12481.

[32] U. Alon, "Machine Learning for Programming Language Processing," Ph.D. dissertation, 2021.

[33] L. Liu, H. Nguyen, G. Karypis, and S. Sengamedu, "Universal representation for code," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12714 LNAI, pp. 16–28, Mar. 2021, ISSN: 16113349. DOI: 10.1007/978-3-030-75768-7_2. [Online]. Available:

http://dx.doi.org/10.1007/978-3-030-75768-7_2%20https://link.springer.com/10.1007/978-3-030-75768-7_2%20http://arxiv.org/abs/2103.03116.

[34] "Open vocabulary learning on source code with a graph-structured cache," *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June, pp. 2662–2674, Oct. 2018. [Online]. Available: http://arxiv.org/abs/1810.08305.

[35] H. Wang, G. Ye, Z. Tang, *et al.*, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, Mar. 2021, ISSN: 1556-6013. DOI: 10.1109/TIFS.2020.3044773. [Online]. Available: https://ieeexplore.ieee.org/document/9293321/.

[36] A. V. Phan, M. L. Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," vol. 2017-Novem, IEEE, Nov. 2018, pp. 45–52, ISBN: 9781538638767. DOI: 10.1109/ICTAI.2017.00019. [Online]. Available: https://ieeexplore.ieee.org/document/8371922/.

[37] Z. Haojie, L. Yujun, L. Yiwei, and Z. Nanxin, "Vulmg: A static detection solution for source code vulnerabilities based on code property graph and graph attention network," IEEE, Dec. 2022, pp. 250–255, ISBN: 9781665413640. DOI: 10.1109/iccwamtip53232.2021.9674145. [Online]. Available: https://ieeexplore.ieee.org/document/9674145/.

[38] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018, ISSN: 15232867. DOI: 10.1145/3192366.3192412. arXiv: 1803.09544.

[39] Y. Wang and H. Li, "Code completion by modeling flattened abstract syntax trees as graphs," Mar. 2021. [Online]. Available: http://arxiv.org/abs/2103.09499.

[40] F. Ye, S. Zhou, A. Venkat, *et al.*, “Misim: A neural code semantics similarity system using the context-aware semantics structure,” Jun. 2020, Machine Inferred Code Similarity (MISIM)<br/>1. Context-aware semantics structure (CASS)<br/>2. Neural code similarity scoring algorithm. [Online]. Available: http://arxiv.org/abs/2006.05265.

[41] M. Lu, D. Tan, N. Xiong, Z. Chen, and H. Li, “Program classification using gated graph attention neural network for online programming service,” vol. 14, pp. 1–12, 8 Mar. 2019. [Online]. Available: http://arxiv.org/abs/1903.03804.

[42] M. Lu, Y. Wang, D. Tan, and L. Zhao, “Student program classification using gated graph attention neural network,” *IEEE Access*, vol. 9, pp. 87 857–87 868, 2021, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3063475. [Online]. Available: https://ieeexplore.ieee.org/document/9367198/.

[43] V. Raychev, M. Vechev, and A. Krause, “Predicting program properties from "Big Code",” in *ACM SIGPLAN Not.*, vol. 50, New York, NY, USA: ACM, Jan. 2015, pp. 111–124, ISBN: 9781450333009. DOI: 10.1145/2676726.2677009. [Online]. Available: https://dl.acm.org/doi/10.1145/3306204%20https://dl.acm.org/doi/10.1145/2676726.2677009.

[44] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and Y. Heights, “Learning to map source code to software vulnerability using code-as-a-graph,” Tech. Rep., 2020.

[45] M. A. Hearst, “Untangling text data mining,” in *Proc. 37th Annu. Meet. Assoc. Comput. Linguist. Comput. Linguist.* -, Morristown, NJ, USA: Association for Computational Linguistics, 1999, pp. 3–10. DOI: 10.3115/1034678.1034679. [Online]. Available: www.aaai.org/%20http://portal.acm.org/citation.cfm?doid=1034678.1034679.

[46] R. Garg and Heena, “Study of text based mining,” in *Proc. Int. Conf. Adv. Comput. Artif. Intell. ACAI 2011*, New York, New York, USA: ACM Press, 2011, pp. 5–8, ISBN: 9781450306355. DOI: 10.1145/2007052.2007054. [Online]. Available: http://portal.acm.org/citation.cfm?doid=2007052.2007054.

[47] G. D. Mergel, M. S. Silveira, and T. S. da Silva, “A method to support search string building in systematic literature reviews through visual text mining,” *Proc. 30th Annu. ACM Symp. Appl. Comput.*, vol. 13-17-Apri, pp. 1594–1601, Apr. 2015. DOI: 10.1145/2695664.2695902. [Online]. Available: https://dl.acm.org/doi/10.1145/2695664.2695902.

[48] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch, “Software engineering challenges of deep learning,” IEEE, Aug. 2018, pp. 50–59, ISBN: 9781538673829. DOI: 10.1109/SEAA.2018.00018. [Online]. Available: https://ieeexplore.ieee.org/document/8498185/%20http://arxiv.org/abs/1810.12034%20http://dx.doi.org/10.1109/SEAA.2018.00018.

[49] S. K. Lo, Q. Lu, C. Wang, H. Y. Paik, and L. Zhu, “A systematic literature review on federated machine learning: From a sofware engineering perspective,” *ACM Computing Surveys*, vol. 54, pp. 1–39, 5 Jun. 2021, ISSN: 15577341. DOI: 10.1145/3450288. [Online]. Available: https://dl.acm.org/doi/10.1145/3450288.

[50] J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang, “Systematic literature review of machine learning based software development effort estimation models,” *Information and Software Technology*, vol. 54, pp. 41–59, 1 Jan. 2012, ISSN: 09505849. DOI: 10.1016/j.infsof.2011.09.002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584911001832.

[51] K. Meinke and A. Bennaceur, “Machine learning for software engineering: Models, methods, and applications,” ACM, May 2018,

pp. 548–549, ISBN: 9781450356633. DOI: 10.1145/3183440.3183461. [Online]. Available: https://dl.acm.org/doi/10.1145/3183440.3183461.

[52] M. Fan, A. Jia, J. Liu, T. Liu, and W. Chen, "When representation learning meets software analysis," *RL+SE and PL 2020 - Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages, Co-located with ESEC/FSE 2020*, pp. 17–18, 2020. DOI: 10.1145/3416506.3423578.

[53] T. Lin, F. Chen, and X. Fu, "Methodological principles for deep learning in software engineering," vol. 2021-Octob, IEEE, Oct. 2021, pp. 1–3, ISBN: 9781665443319. DOI: 10.1109/IPCCC51483.2021.9679405. [Online]. Available: https://ieeexplore.ieee.org/document/9679405/.

[54] Z. Wan, X. Xia, D. Lo, and G. C. Murphy, "How does machine learning change software development practices?" *IEEE Transactions on Software Engineering*, vol. 47, pp. 1857–1871, 9 Sep. 2021, ISSN: 19393520. DOI: 10.1109/TSE.2019.2937083. [Online]. Available: https://ieeexplore.ieee.org/document/8812912/.

[55] X. Li, H. Jiang, Z. Ren, G. Li, and J. Zhang, "Deep learning in software engineering," pp. 1–10, May 2018. [Online]. Available: http://arxiv.org/abs/1805.04825.

[56] V. Pranathi, G. R. Reddy, K. S. Kumar, G. Jhansi, and B. Rajitha, "Analysis of deep learning in software engineering," vol. 020042, 2022, p. 020 042, ISBN: 9780735443686. DOI: 10.1063/5.0083699. [Online]. Available: http://aip.scitation.org/doi/abs/10.1063/5.0083699.

[57] F. Ferreira, L. L. Silva, and M. T. Valente, "Software engineering meets deep learning: A mapping study," ACM, Mar. 2021, pp. 1542–1549, ISBN: 9781450381048. DOI: 10.1145/3412841.3442029. [Online].

Available:
https://dl.acm.org/doi/10.1145/3412841.3442029.

[58] I. C. Society, *Guide to the Software Engineering Body of Knowledge Version 3.0 (SWEBOK Guide V3.0)*, ISBN: 9780769551661.

[59] P. Muenchaisri, "Literature reviews on applying artificial intelligence/machine learning to software engineering research problems: Preliminary," *CEUR Workshop Proceedings*, vol. 2506, pp. 30–35, Seed 2019, ISSN: 16130073.

[60] S. K. Pani and A. K. Mishra, "Machine learning applications in software engineering: Recent advances and future research directions," *International Journal of Engineering Research and Technology (IJERT)*, vol. 8, pp. 1–4, 1 2020. [Online]. Available: www.ijert.org.

[61] C. Liu, C. Gao, X. Xia, D. Lo, J. Grundy, and X. Yang, "On the replicability and reproducibility of deep learning in software engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 31, pp. 1–46, 1 Jan. 2020, ISSN: 1049-331X. DOI: 10.1145/3477535. [Online]. Available:
http://arxiv.org/abs/2006.14244%0Ahttp://dx.doi.org/10.1145/3477535.

[62] C. Niu, C. Li, B. Luo, and V. Ng, "Deep learning meets software engineering: A survey on pre-trained models of source code," May 2022. [Online]. Available: http://arxiv.org/abs/2205.11739.

[63] A. F. D. Carpio and L. B. Angarita, "Trends in software engineering processes using deep learning: A systematic literature review," *Proceedings - 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020*, pp. 445–454, 2020. DOI: 10.1109/SEAA51224.2020.00077.

[64] F. Kumeno, "Sofware engneering challenges for machine learning applications: A literature review," *Intelligent Decision Technologies*, vol. 13, pp. 463–476, 4 Feb. 2020, ISSN: 18724981. DOI: 10.3233/idt-190160. [Online]. Available:

https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/IDT-190160.

[65] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Computing Surveys*, vol. 1, p. 3 505 243, Dec. 2021, ISSN: 0360-0300. DOI: 10.1145/3505243. [Online]. Available: https://dl.acm.org/doi/10.1145/3505243.

[66] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research," *ACM Transactions on Software Engineering and Methodology*, vol. 31, pp. 1–58, 2 Apr. 2022, ISSN: 1049-331X. DOI: 10.1145/3485275. [Online]. Available: http://arxiv.org/abs/2009.06520%20https://dl.acm.org/doi/10.1145/3485275.

[67] S. Wang, L. Huang, A. Gao, *et al.*, "Machine/deep learning for software engineering: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 5589, pp. 1–1, c 2022, ISSN: 0098-5589. DOI: 10.1109/TSE.2022.3173346. [Online]. Available: https://ieeexplore.ieee.org/document/9772253/.

[68] S. Vashishth, N. Yadati, and P. Talukdar, "Graph-based deep learning in natural language processing," ACM, Jan. 2020, pp. 371–372, ISBN: 9781450377386. DOI: 10.1145/3371158.3371232. [Online]. Available: https://dl.acm.org/doi/10.1145/3371158.3371232.

[69] L. Wu, Y. Chen, H. Ji, and B. Liu, "Deep learning on graphs for natural language processing," ACM, Aug. 2021, pp. 4084–4085, ISBN: 9781450383325. DOI: 10.1145/3447548.3470820. [Online]. Available: https://dl.acm.org/doi/10.1145/3447548.3470820.

[70] H. C. Yi, Z. H. You, D. S. Huang, and C. K. Kwoh, "Graph representation learning in bioinformatics: Trends, methods and applications," *Brief.*

Bioinform., vol. 23, no. 1, pp. 1–16, 2022, ISSN: 14774054. DOI: 10.1093/bib/bbab340.

[71] B. A. Kitchenham, T. Dybå, and M. Jørgensen, “Evidence-based software engineering,” vol. 26, IEEE Comput. Soc, 2004, pp. 273–281, ISBN: 0-7695-2163-0. DOI: 10.1109/icse.2004.1317449. [Online]. Available: http://ieeexplore.ieee.org/document/1317449/.

[72] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering,” *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*, vol. 1, pp. 1–54, 2007.

[73] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, “Lessons from applying the systematic literature review process within the software engineering domain,” *Journal of Systems and Software*, vol. 80, pp. 571–583, 4 2007, ISSN: 01641212. DOI: 10.1016/j.jss.2006.07.009. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2006.07.009.

[74] Y. Xiao and M. Watson, “Guidance on conducting a systematic literature review,” *Journal of Planning Education and Research*, vol. 39, pp. 93–112, 1 2019, ISSN: 0739456X. DOI: 10.1177/0739456X17723971.

[75] M. Gusenbauer and N. R. Haddaway, “What every researcher should know about searching – clarified concepts, search advice, and an agenda to improve finding in academia,” *Res. Synth. Methods*, vol. 12, no. 2, pp. 136–147, 2021, ISSN: 17592887. DOI: 10.1002/jrsm.1457.

[76] M. Gusenbauer and N. R. Haddaway, “Which academic search systems are suitable for systematic reviews or meta-analyses? Evaluating retrieval qualities of Google Scholar, PubMed, and 26 other resources,” *Res. Synth. Methods*, vol. 11, no. 2, pp. 181–217, 2020, ISSN: 17592887. DOI: 10.1002/jrsm.1378.

[77] “A systematic approach to searching: An efficient and complete method to develop literature searches,” *Journal of the Medical Library Association*, vol. 106, pp. 531–541, 4 2018, ISSN: 15589439. DOI: 10.5195/jmla.2018.283.

[78] S. Marcos-Pablos and F. J. García-Peñalvo, “Decision support tools for SLR search string construction,” in *ACM Int. Conf. Proceeding Ser.*, New York, NY, USA: ACM, Oct. 2018, pp. 660–667, ISBN: 9781450365185. DOI: 10.1145/3284179.3284292. [Online]. Available: https://dl.acm.org/doi/10.1145/3284179.3284292.

[79] B. Kitchenham, R. Pretorius, D. Budgen, *et al.*, “Systematic literature reviews in software engineering-a tertiary study,” *Information and Software Technology*, vol. 52, pp. 792–805, 8 2010, ISSN: 09505849. DOI: 10.1016/j.infsof.2010.03.006. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2010.03.006.

[80] B. Karakan, “Tool support for systematic literature reviews: Analyzing existing solutions and the potential for automation,” 2021. [Online]. Available: http://elib.uni-stuttgart.de/handle/11682/11459.

[81] A. Hinderks, F. Jose, D. Mayo, J. Thomaschewski, and M. J. Escalona, “An slr-tool: Search process in practice : To conduct and manage systematic literature review (slr),” *Proceedings - 2020 ACM/IEEE 42nd International Conference on Software Engineering: Companion, ICSE-Companion 2020*, pp. 81–84, 2020, ISSN: 02705257. DOI: 10.1145/3377812.3382137.

[82] G. Wang, Q. Peng, Y. Zhang, and M. Zhang, “What Have We Learned from OpenReview?” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 12858 LNCS, pp. 63–79, 2021, ISSN: 16113349. DOI: 10.1007/978-3-030-85896-4_6. arXiv: 2103.05885.

[83] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, “ProGraML: Graph-based Deep Learning for Program Optimization

and Analysis,” Mar. 2020. arXiv: 2003.10536. [Online]. Available: http://arxiv.org/abs/2003.10536.

[84] J. Cheng, I. Fostiropoulos, and B. Boehm, “Gn-transformer: Fusing sequence and graph representation for improved code summarization,” vol. 1, 1 Nov. 2021. [Online]. Available: http://arxiv.org/abs/2111.08874.

[85] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, “Modeling functional similarity in source code with graph-based siamese networks,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2021, ISSN: 0098-5589. DOI: 10.1109/TSE.2021.3105556. [Online]. Available: https://ieeexplore.ieee.org/document/9516896/.

[86] J. Cheng, I. Fostiropoulos, and B. Boehm, “Graph conditioned sparse-attention for improved source code understanding,” Dec. 2021. [Online]. Available: http://arxiv.org/abs/2112.00663.

[87] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, “Deepwukong: Statically detecting software vulnerabilities using deep graph neural network,” *ACM Transactions on Software Engineering and Methodology*, vol. 30, pp. 1–33, 3 May 2021, ISSN: 15577392. DOI: 10.1145/3436877. [Online]. Available: https://dl.acm.org/doi/10.1145/3436877.

[88] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” *Proceedings - IEEE Symposium on Security and Privacy*, pp. 590–604, 2014, ISSN: 10816011. DOI: 10.1109/SP.2014.44.

[89] Y. Wu, J. Lu, Y. Zhang, and S. Jin, “Vulnerability detection in c/c++ source code with graph representation learning,” IEEE, Jan. 2021, pp. 1519–1524, ISBN: 978-1-6654-1490-6. DOI: 10.1109/CCWC51732.2021.9376145. [Online]. Available: https://ieeexplore.ieee.org/document/9376145/.

[90] J. Xu, J. Ai, and T. Shi, “Software defect prediction for specific defect types based on augmented code graph representation,” IEEE, Aug. 2021, pp. 669–678, ISBN: 978-1-6654-4391-3. DOI: 10.1109/DSA52907.2021.00097. [Online]. Available: https://ieeexplore.ieee.org/document/9622967/.

[91] J. Wei, M. Goyal, G. Durrett, and I. Dillig, “Lambdanet: Probabilistic type inference using graph neural networks,” pp. 1–11, Apr. 2020, Type dependency graph –gt; GNN and pointer-network –gt; predict user-defined types. [Online]. Available: http://arxiv.org/abs/2005.02161.

[92] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari, and J. Laredo, “Software vulnerability detection via deep learning over disaggregated code graph representation,” Sep. 2021. [Online]. Available: http://arxiv.org/abs/2109.03341.

[93] X. Li, C. J. Maddison, and D. Tarlow, “Learning to extend program graphs to work-in-progress code,” May 2021. [Online]. Available: http://arxiv.org/abs/2105.14038.

[94] W. Leeson and M. B. Dwyer, “Algorithm selection for software verification using graph attention networks,” vol. 1, 1 Jan. 2022. [Online]. Available: http://arxiv.org/abs/2201.11711.

[95] Y. Wang, E. Shi, L. Du, *et al.*, “Cocosum: Contextual code summarization with multi-relational graph neural network,” *Journal of the ACM*, vol. 1, 1 Jul. 2021. [Online]. Available: http://arxiv.org/abs/2107.01933.

[96] M. Brockschmidt, M. Allamanis, and A. Gaunt, “G c m g,” *Iclr*, pp. 1–24, 2019.

[97] L. Sikic, A. S. Kurdija, K. Vladimir, and M. Silic, “Graph neural network for source code defect prediction,” *IEEE Access*, vol. 10, pp. 10 402–10 415, Mar. 2022, ISSN: 21693536. DOI: 10.1109/ACCESS.2022.3144598.

[98] G. Tech and K. Wang, “Hoppity : Learning graph transformations detect and fix bugs in programs,” *Iclr 2020*, pp. 1–17, 2020.

[99] D. Tarlow, S. Moitra, A. Rice, *et al.*, “Learning to fix build errors with graph2diff neural networks,” ACM, Jun. 2020, pp. 19–20, ISBN: 9781450379632. DOI: 10.1145/3387940.3392181. [Online]. Available: https://dl.acm.org/doi/10.1145/3387940.3392181.

[100] W. Wang, K. Zhang, G. Li, and Z. Jin, “Learning to Represent Programs with Heterogeneous Graphs,” pp. 1–10, 2020. arXiv: 2012.04188. [Online]. Available: http://arxiv.org/abs/2012.04188.

[101] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, “On the generalizability of neural program models with respect to semantic-preserving program transformations,” *Information and Software Technology*, vol. 135, p. 106 552, Jul. 2021, ISSN: 09505849. DOI: 10.1016/j.infsof.2021.106552. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584921000379.

[102] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, “Self-supervised bug detection and repair,” NeurIPS May 2021. [Online]. Available: http://arxiv.org/abs/2105.12787.

[103] J. M. Tomczak, R. Lepert, and A. Wiggers, “Simulating execution time of tensor programs using graph neural networks,” pp. 1–8, Apr. 2019, Dataset for learning runtime of tensor program configuration as graphs: 1. NN with mean as aggregation function 2. GNN to propagate info among nodes before aggregation 3. NN representing whole AST. [Online]. Available: http://arxiv.org/abs/1904.11876.

[104] X. Zhang, S. Yang, L. Duan, Z. Lang, Z. Shi, and L. Sun, “Transformer-xl with graph neural network for source code summarization,” IEEE, Oct. 2021, pp. 3436–3441, ISBN: 9781665442077. DOI: 10.1109/SMC52423.2021.9658619. [Online]. Available: https://ieeexplore.ieee.org/document/9658619/.

[105] B. Wu, B. Liang, and X. Zhang, “Turn tree into graph: Automatic code review via simplified ast driven graph convolutional network,” Feb. 2022. [Online]. Available: `http://arxiv.org/abs/2202.07946`.

[106] M. Li, J. Pei, J. He, *et al.*, “Using ggnn to recommend log statement level,” pp. 1–10, Dec. 2019. [Online]. Available: `http://arxiv.org/abs/1912.05097`.

[107] Y. Guo, P. Li, Y. Luo, X. Wang, and Z. Wang, “Graphspy: Fused program semantic embedding through graph neural networks for memory efficiency,” vol. 2021-Decem, IEEE, Dec. 2021, pp. 1045–1050, ISBN: 978-1-6654-3274-0. DOI: `10.1109/DAC18074.2021.9586120`. [Online]. Available: `https://ieeexplore.ieee.org/document/9586120/`.

[108] A. Faustino, “Graphs based on ir as representation of code,” ACM, Sep. 2021, pp. 75–82, ISBN: 9781450390620. DOI: `10.1145/3475061.3475063`. [Online]. Available: `https://doi.org/10.1145/3475061.3475063%20https://dl.acm.org/doi/10.1145/3475061.3475063`.

[109] A. Nair, A. Roy, and K. Meinke, “FuncGNN: A graph neural network approach to program similarity,” *International Symposium on Empirical Software Engineering and Measurement*, 2020, ISSN: 19493789. DOI: `10.1145/3382494.3410675`. arXiv: `2007.13239`.

[110] Z. Shi, K. Swersky, D. Tarlow, P. Ranganathan, and M. Hashemi, “Learning execution through neural code fusion,” pp. 1–13, Jun. 2019. [Online]. Available: `http://arxiv.org/abs/1906.07181`.

[111] Y. Wang, K. Wang, F. Gao, and L. Wang, “Learning semantic program embeddings with graph interval neural network,” *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–27, OOPSLA Nov. 2020, ISSN: 24751421. DOI: `10.1145/3428205`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3428205`.

[112] G. Ma, Y. Xiao, M. Capota, *et al.*, "Learning code representations using multifractal-based graph networks," IEEE, Dec. 2021, pp. 1858–1866, ISBN: 978-1-6654-3902-2. DOI: 10.1109/BigData52589.2021.9671685. [Online]. Available: https://ieeexplore.ieee.org/document/9671685/.

[113] C. Zeng, Y. Yu, S. Li, *et al.*, "Degraphcs: Embedding variable-based flow graph for neural code search," Mar. 2021. [Online]. Available: http://arxiv.org/abs/2103.13020.

[114] M. K. Thota, F. H. Shajin, and P. Rajesh, "Survey on software defect prediction techniques," *International Journal of Applied Science and Engineering*, vol. 17, pp. 331–344, 4 Dec. 2020, ISSN: 1727-7841. DOI: 10.6703/IJASE.202012_17(4).331. [Online]. Available: https://doi.org/10.6703/IJASE.202012_17(4).331.

[115] L. Wu, F. Li, Y. Wu, and T. Zheng, "Ggf: A graph-based method for programming language syntax error correction," ACM, Jul. 2020, pp. 139–148, ISBN: 9781450379588. DOI: 10.1145/3387904.3389252. [Online]. Available: https://doi.org/10.1145/3387904.3389252.

[116] R. G. Iyer, Y. Sun, W. Wang, and J. Gottschlich, "Software language comprehension using a program-derived semantics graph," VL Apr. 2020. [Online]. Available: http://arxiv.org/abs/2004.00768.

[117] G. E. D. P. Rodrigues, A. M. Braga, and R. Dahab, "Using graph embeddings and machine learning to detect cryptography misuse in source code," IEEE, Dec. 2020, pp. 1059–1066, ISBN: 9781728184708. DOI: 10.1109/ICMLA51294.2020.00171. [Online]. Available: https://ieeexplore.ieee.org/document/9356194/.

[118] S. Kurimoto, Y. Hayase, H. Yonai, H. Ito, and H. Kitagawa, "Class name recommendation based on graph embedding of program elements," vol. 2019-Decem, IEEE, Dec. 2019, pp. 498–505, ISBN: 978-1-7281-4648-5. DOI: 10.1109/APSEC48747.2019.00073.

[Online]. Available: https://ieeexplore.ieee.org/document/8946106/.

[119] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, "Cpgva: Code property graph based vulnerability analysis by deep learning," Aug. 2018, pp. 184–188. DOI: 10.1109/ICAIT.2018.8686548.

[120] M. Singh and G. S. Walia, "Using semantic analysis and graph mining approaches to support software fault fixation," IEEE, Oct. 2020, pp. 43–48, ISBN: 9781728198705. DOI: 10.1109/ISSREW51248.2020.00035. [Online]. Available: https://ieeexplore.ieee.org/document/9307741/.

[121] J. Xu, F. Wang, and J. Ai, "Defect prediction with semantics and context features of codes based on graph representation learning," *IEEE Transactions on Reliability*, vol. 70, pp. 613–625, 2 Jun. 2021, ISSN: 0018-9529. DOI: 10.1109/TR.2020.3040191. [Online]. Available: https://ieeexplore.ieee.org/document/9290043/.

[122] X. Ling, L. Wu, S. Wang, *et al.*, "Deep graph matching and searching for semantic code retrieval," *ACM Transactions on Knowledge Discovery from Data*, vol. 15, pp. 1–21, 5 Oct. 2021, ISSN: 1556-4681. DOI: 10.1145/3447571. [Online]. Available: https://doi.org/10.1145/3447571.

[123] H. Cheers and Y. Lin, "A novel graph-based program representation for java code plagiarism detection," ACM, Jan. 2020, pp. 115–122, ISBN: 9781450376907. DOI: 10.1145/3378936.3378960. [Online]. Available: https://doi.org/10.1145/3378936.3378960%20https://dl.acm.org/doi/10.1145/3378936.3378960.

[124] K. Sendjaja, S. A. Rukmono, and R. S. Perdana, "Evaluating control-flow graph similarity for grading programming exercises," IEEE, Nov. 2021, pp. 1–6, ISBN: 978-1-6654-9453-3. DOI: 10.1109/ICoDSE53690.2021.9648464. [Online]. Available: https://ieeexplore.ieee.org/document/9648464/.

[125] A. Desku, B. Raufi, A. Luma, and B. Selimi, “Cosine similarity through control flow graphs for secure software engineering,” IEEE, Oct. 2021, pp. 1–4, ISBN: 978-1-6654-2714-2. DOI: 10.1109/ICEET53442.2021.9659648. [Online]. Available: https://ieeexplore.ieee.org/document/9659648/.

[126] M. Nadim, D. Mondal, and C. K. Roy, “Leveraging structural properties of source code graphs for just-in-time bug prediction,” 2022, ISSN: 15737535. DOI: 10.1007/s10515-022-00326-0. [Online]. Available: http://arxiv.org/abs/2201.10137.

[127] A. Nikanjam, H. B. Braiek, M. M. Morovati, and F. Khomh, “Automatic fault detection for deep learning programs using graph transformations,” *ACM Transactions on Software Engineering and Methodology*, vol. 31, pp. 1–27, 1 May 2022, ISSN: 1049-331X. DOI: 10.1145/3470006. [Online]. Available: https://dl.acm.org/doi/10.1145/3470006.

[128] F. Provost and T. Fawcett, *Data Science for Business: What you need to know about data mining and data-analytic thinking.* " O’Reilly Media, Inc.", 2013.

[129] J. Zhou, A. H. Gandomi, F. Chen, and A. Holzinger, “Evaluating the quality of machine learning explanations: A survey on methods and metrics,” *Electronics*, vol. 10, no. 5, p. 593, 2021.

[130] N. Yefet, U. Alon, and E. Yahav, “Adversarial examples for models of code,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020, ISSN: 24751421. DOI: 10.1145/3428230. arXiv: 1910.07517.