

From Trees to Graphs: Advancing Regression Analysis through Model- Centric AI, Data-Centric AI, and Active Learning

PETER SAMOAA

Department of Computer Science and Engineering

Division of Data Science and AI

CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden 2024

THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

From Trees to Graphs: Advancing Regression
Analysis through Model-Centric AI, Data-Centric
AI, and Active Learning

Peter Samoaa



Department of Computer Science and Engineering
Division of Data Science and AI
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2024

From Trees to Graphs: Advancing Regression Analysis through Model-Centric AI,
Data-Centric AI, and Active Learning

PETER SAMOAA

ISBN 978-91-8103-101-0

© PETER SAMOAA, 2024.

Doktorsavhandlingar vid Chalmers tekniska högskola

Ny serie nr 5559

ISSN 0346-718X

Department of Computer Science and Engineering

Chalmers University of Technology

SE-412 96 Göteborg, Sweden

Telephone + 46 (0) 31 - 772 1000

Typeset by the author using L^AT_EX.

Printed by Chalmers Reproservice

Göteborg, Sweden 2024

To my parents, sisters, and brother, I am your permanent ambassador to the future and your hope that you will never be disappointed.

I want to dedicate this to my countries, England and Italy, where I always feel at home. These countries gave me the right to live. I will always be grateful for that.

I also want to thank the British Royal Family Institute, which supported me for 15 years through my school studies.

I would dedicate this to the memory of our gracious queen, Her Majesty Queen Elizabeth II. There can simply be no finer example of dignified public duty and unstinting service. We all owe our sincere gratitude for her continued devotion, living every day by the pledge she made on her 21st birthday. Her dedication to our country has been incomparable, and, as such, she leaves an enduring legacy.

Special appreciation goes to my early line manager, Ivica Crnkovic, for the kindness, inspiration, unlimited support, and for being like a father. I will always remember you.

Abstract

Context: Trees and graphs are fundamental data structures that are extensively utilized for modelling relationships and facilitating efficient data organization and retrieval. In research, these structures underpin a wide variety of algorithms and theories, especially in fields like Artificial Intelligence (AI), where they are crucial for understanding and optimizing learning processes. In real-world applications, trees and graphs have profound impacts. For instance, trees are at the heart of decision-making processes, from simple decision trees in machine learning to complex game trees in AI strategies for games like chess. Graphs, on the other hand, are vital in networking, whether in social networks, neural networks, or logistical networks, helping to map and optimize connections and flows. The versatility of these structures in modelling complex systems makes them indispensable in both theoretical research and practical applications, impacting industries from technology to transportation and beyond. This dual significance not only underscores the theoretical importance of our study but also enhances its applicability in solving real-world problems.

Problem: The main issue is that regression for trees and graphs is still not well explored in the literature. Many real-world problems for trees and graphs involve regressions, like predicting drug efficacy for molecular drugs or evolutionary outcomes for evolutionary biology trees.

Goal: In this thesis, we aim to enhance the regression analysis by proposing and utilising AI models on trees and graphs.

Solution Approaches: To that aim, we analysed the behaviour of different Tree-Based Neural Networks (TBNNs). Thus, Graph Neural Networks (GNNs), Tree-Convolutional Neural Networks (TreeCNN), path-based attention models, and transformer-based models are used. Then, we enhanced the behaviour of the transformer-based model by proposing our dual transformer based on cross attention as a model-centric AI approach to have a better representation. Then, we enhanced the regression analysis by focusing on data instead of the model. Thus, data-centric AI is used to augment the tree by adding more edges to represent more information. In this way, the augmented tree is converted into graphs, and then the same GNN models used in the previous analytical framework have better regression prediction by having a richer representation. Then, through data-centric AI, we improve the data by acquiring better labelling through interactive learning. Thus, we defined a unified active learning framework for labelling graphs for regression tasks. Through this framework, we select informative, representative, and diverse batches of samples for labelling.

Results: The results show that the effective TBNN models for classification tasks fail to generalise for regression tasks. Thus, our proposed model outperforms all other TBNN models as well as GNN models through different settings and experiments.

Moreover, The same GNN models used in the tree setting achieve higher Pearson correlation scores when we augment the tree and convert it into a graph, which shows that adding more information improves the prediction. Our results also show that the active learning framework can provide efficient query strategies for labelling the regression value on the entire graph level.

Keywords: Graph Neural Networks (GNNs), Active Learning, Tree-Based Neural Networks (TBNNs), Tree-Convolutional Neural Networks (TreeCNN), Transformers, Model-Centric AI, Data-Centric AI, Neural Tangent Kernel (NTK)

Acknowledgments

First and foremost, I extend my deepest gratitude to my supervisors, Morteza Haghiri Chehreghani and Philipp Leitner. **Morteza**, words cannot fully capture the profound impact you have had on my journey. Without your unwavering guidance and support, I would not be where I am today. You have been more than a supervisor; you have been a true mentor, a leader, and a friend. Your relentless drive to push me to my fullest potential has elevated my standards and reshaped my approach to research and life. I am deeply grateful for your time, attention, and genuine care. Your belief in me charged me every day with hope and passion, and every piece of advice you offered has been a cornerstone not only in the completion of this PhD but in the broader journey of my life. Your brilliance as a supervisor is beyond words, but it is your mentorship, support, and invaluable insights that have profoundly shaped the research presented in this thesis. I will forever cherish the wisdom you have imparted and the inspiration you have provided.

Philipp, thank you for your invaluable support as my line manager and for your guidance as a supervisor, especially during the initial stages of my PhD. Your administrative assistance and scholarly input were crucial in setting a strong foundation for my research journey.

I also extend my sincere thanks to my examiner, Prof. Mirosław Staron, for the constructive feedback that helped refine my work. I am grateful to Prof. Giovanna Guerrini for accepting the invitation to lead the discussion on my research and to all the committee members for their willingness to be part of my PhD defence. A special thank you to Ashkan Panahi for being on the committee and for all the friendly chats that provided much-needed encouragement.

I am deeply appreciative of the supportive and inspiring environment fostered by the Data Science and AI division. A heartfelt thank you to my office mates, Firooz, Mehrdad, Milad, and my friend Arman, for creating a friendly and vibrant working atmosphere that made every day enjoyable.

To my soul friend, Walaa, I thank you from the bottom of my heart for your endless support, care, and unwavering presence, regardless of the circumstances.

Khalid & Jounama, Mazen & Reem, Rudi & Berwin, thank you for welcoming me with open arms from the very first day of my arrival in Sweden. You made me feel at home, providing me with a loving family away from home.

My acknowledgement and gratitude also go to my bro, Benjamin Johnson, for all the funny and crazy special moments we shared in life and at the gym.

My dear friend Adham Tala, thank you for being a true brother and for all the love and support you showed me during my time in Sweden.

My friend of the beautiful days, Murad Ahmad, thank you for always being by my side.

A special thanks to the closest friend to my heart and the best scriptwriter ever, Bouthina Awad, for being a constant and steadfast part of my life.

I am grateful for the collaboration with Antonio Longa, Linus Aronsson, and Firas Bayram, and I look forward to continuing our work together in the future.

To Linda Erlinhov, thank you for all your help and support during my stay in Lindholmen.

Everything began in Italy at the University of Genoa, and I am deeply grateful to the incredibly supportive people there, including Prof. Barbara Catania, Prof. Alessandro Veirri, Prof. Giovanna Guerrini, and Prof. Giorgio Delzanno. Your guidance and encouragement were instrumental in setting the foundation for my academic journey.

This work received financial support from the Swedish Research Council VR under grant number 2018-04127 (Developer-Targeted Performance Engineering for Immersed Release and Software Engineering).

Peter Samoaa
Göteborg, September 2024

List of Publications

This thesis is based on the following appended papers:

Paper 1. Peter Samoaa, Firas Bayram, Pasquale Salza, and Philipp Leitner. *A systematic mapping study of source code representation for deep learning in software engineering*. IET Software Journal, 2022, 351-385.

Paper 2. Peter Samoaa, Mehrdad Farahani, Antonio Longa, Philipp Leitner, Morteza Haghiri Chehreghani. *Analyzing the Behaviour of Tree-Based Neural Networks in Regression Tasks*. IEEE Transactions on Neural Networks and Learning Systems - 2024.

Paper 3. Peter Samoaa, Antonio Longa, Mazen Mohamad, Morteza Haghiri Chehreghani, and Philipp Leitner. *TEP-GNN: Accurate Execution Time Prediction of Functional Tests using Graph Neural Networks*. PROFES'22, the International Conference on Product-Focused Software Process Improvement (November 2022).

Paper 4. Peter Samoaa, Linus Aronsson, Antonio Longa, Philipp Leitner, Morteza Haghiri Chehreghani. *A Unified Active Learning Framework for Annotating Graph Data For Regression Task*. Journal of Engineering Applications of Artificial Intelligence - 2024.

Paper 5. Peter Samoaa, Linus Aronsson, Philipp Leitner, Morteza Haghiri Chehreghani. *Batch Mode Deep Active Learning for Regression on Graph Data*. International Conference on Big Data (BigData) - 2023.

The following publications were published during my PhD studies or are currently in submission/under revision. However, they are not appended to this thesis due to their contents overlapping those of appended publications or their content being not related to the thesis.:

Peter Samoaa, Marcus Vukojevic, Morteza Haghiri Chehreghani, Antonio Longa. *Broadening the Scope of Graph Regression: Introducing A Novel Dataset with Multiple Representation Settings*. Submitted to LOG 2024: Learning on Graphs Conference.

Peter Samoaa Linus Aronsson, Morteza Haghiri Chehreghani. *Optimizing Meta Graph Learning for Regression with Active Learning Strategies*. Submitted to International Conference on Big Data (BigData) - 2024

Peter Samoaa and Philipp Leitner. *An Exploratory Study of the Impact of Parameterization on JMH Measurement Results in Open-Source Projects*. Proceedings of the ACM/SPEC International Conference on Performance Engineering ICPE'21 (April 2021), 213–224.

Peter Samoaa and Barbara Catania. *A Pipeline for Measuring Brand Loyalty Through Social Media Mining* SOFSEM 2021: Theory and Practice of Computer Science. (January 2022).

Research Contribution

I (Peter Samoaa) was the main driver and contributor of the all papers. A summary of the contributions is presented in Table 1.

Role	Paper I	Paper II	Paper III	Paper IV	Paper V
Conceptualization	X	X	X	X	X
Data curation	X	X		X	X
Problem Formulation	X	X	X	X	X
Investigation	X	X	X	X	X
Methodology	X	X	X	X	X
Implementation	NA	X	P	X	X
Validation	X	X	X	X	X
Visualization	X	X	X	P	X
Writing - original draft	X	X	X	X	X

Table 1: The Individual Contributions of this thesis' author to the appended papers (Allen et al. 2019).

Note: "P" in the table denotes Partial/shared contribution.

List of Acronyms

AI	–	Artificial Intelligence
GNN	–	Graph Neural Networks
TBNN	–	Tree Based Neural Networks
TreeCNN	–	Tree Based Convolutional Networks
NTK	–	Neural Network Kernel
NN	–	Neural Network
GP	–	Gaussian Process
GPR	–	Gaussian Process Regressor

Contents

Abstract	v
Acknowledgments	vii
List of Publications	ix
Research Contribution	xi
List of Acronyms	xiii
I Introductory chapters	1
1 Introduction	3
2 Background	9
2.1 Graphs and Trees	9
2.2 Model-Centric AI	9
2.3 Data-Centric AI	10
2.4 Graph Neural Networks (GNNs)	12
2.5 Active Learning	13
3 General Overview of The Papers	17
3.1 Exploration of Tree and Graph Representation	17
3.1.1 Analyzing Trees and Graphs as Intermediate Representations	18
3.1.2 Exploring the Integration of Multiple Representations	21
3.2 Tree Regression Analysis and Model-Centric AI for Trees	21
3.2.1 Behaviour of TBNN models in Regression Context	22
3.2.2 Model-Centric AI for Trees	23
3.2.3 Error and Correlation Analysis for TBNN models on Regression	24
3.3 Data-Centric AI for Graphs	24
3.3.1 From Tree to Graph over Data-Centric AI	25
3.3.2 Validating the Data-Centric AI Approach	26
3.4 Active Learning for Graphs	27
3.4.1 Informativeness and Representativeness	28
3.4.2 Diversity	29

3.5	Contributions	30
3.6	Limitations and Challenges	31
4	Concluding Remarks and Future Works	33
4.1	Conclusion	33
4.2	Future Work	33
II	Appended papers	43
	Paper 1: A systematic mapping study of source code representation for deep learning in software engineering	45
	Paper 2: Analysing the Behaviour of Tree-Based Neural Networks in Regression Tasks	83
	Paper 3: Tep-gnn: Accurate execution time prediction of functional tests using graph neural networks	101
	Paper 4: A Unified Active Learning Framework for Annotating Graph Data For Regression Task	119
	Paper 5: Batch Mode Deep Active Learning for Regression on Graph Data	159

Part I

Introductory chapters

Chapter 1

Introduction

Graphs and trees are fundamental data structures that play a critical role in various applications within artificial intelligence (AI). These structures enable efficient organization, modelling, and retrieval of complex data, forming the backbone of numerous algorithms and theoretical frameworks essential for advancing AI. Trees are pivotal in decision-making processes, including applications such as biological taxonomies (Cramer et al. 2020; Adams and Collyer 2019), genealogical trees (Suissa et al. 2023; He et al. 2021), genetic information (Whitehouse et al. 2024) and game trees used in AI strategies for games like chess and Go (Sironi 2019; Thangaramya et al. 2024). Conversely, graphs are indispensable in a multitude of networking contexts, encompassing social networks (Min et al. 2021; Jain et al. 2023), molecular structures in drug discovery (Ye et al. 2022; Bongini et al. 2021), and road networks for optimizing transportation logistics (Åkerblom et al. 2023; Ren et al. 2019).

However, regression analysis is still poorly explored for tree and graph data structures. Thus, we will enhance the regression analysis for trees and graphs i) from the model perspective by improving the capability of the model in predicting scalar values with a low margin of error ii) and from data perspectives by improving the quality of data representation as well as gaining more labelled data.

Through that aim, we first explore tree regression models by building an analytical framework for regression analysis using the existing tree-based models in the literature. These models are graph-based (Talak et al. 2021), convolutional-based (Roy et al. 2020), path-based attention (Peng et al. 2021) and sequential-based tree transformers (Sun et al. 2020). However, all these models are used for classification but not for regression. When we put these models in the context of regression, they tend to have poor efficiency despite their remarkable performance in the classification tasks. This indicates the models' weakness in exploring an unlimited number of prediction options, as in predicting scalar values of regression. To handle this gap, we follow the model-centric AI (Hamid 2022) by enhancing the behaviour of the tree transformer model by adding more components that manipulate the extra context of information alongside cross-attention (H. Lin et al. 2022), which leads to a way better efficiency and make our model performs remarkably better than all other models.

Enhancing data quality can improve regression analysis for trees, either by

improving the representation to have a richer representation that can help the model to detect more patterns from the data, and that can be done by following data-centric AI (P. Samoaa 2023) or by enhancing the quality of the labels through the usage of active learning (Settles 2009).

Throughout data-centric AI, we move from trees to graphs to enhance the regression analysis by augmenting the tree with more edges that describe more information. By augmenting the tree, the tree is then converted into a graph, which is a richer representation. The augmentation strategy can be different according to the domain and case study. Thus, we augment the tree for a specific case study in the thesis. The GNN models are then applied to the resulting graphs for a better regression prediction. We observe a remarkable improvement in the behaviour of GNN models in the augmented trees compared to the original trees before the augmentation.

Conversely, the data-hungry problem, characterized by insufficiency and low-quality data, poses obstacles for deep learning models (Bi et al. 2023). Thus, based on the generated graphs from the augmented trees, we aim to address the quality issue of the data, which is the lack of labelled data. For data-hungry models like GNNs, the more labelled data we have, the more the model can detect and learn from these patterns, ultimately enhancing the regression analysis. For that aim, we tackle the active learning problem for graphs (Hu et al. 2020). Active learning is an online learning process (Cacciarelli et al. 2024) that aims to define the most informative samples for labelling iteratively (Hsu and H.-T. Lin 2015). Thus, instead of asking the oracle to label all graphs, we can select only the most informative graphs for labelling where the used models are uncertain about the informative graph samples. To the best of our knowledge, active learning for graphs and regression is still not well explored in the literature since most of the attention goes toward the investigation of active learning for classification (Caramalau et al. 2021; Miller et al. 2022; Q. Wang et al. 2021). In active learning, informativeness is measured by the uncertainty quantification of the models. In classification, measuring the uncertainty is straightforward through the softmax layer of the model. However, such methods are not directly applicable when it comes to regression. A straightforward uncertainty quantification mechanism is absent in regression settings that yield scalar outputs. This gap is bridged by using Gaussian Process (GP), a Bayesian technique that computes uncertainties via kernel methods. Another issue with active learning for graphs is that the investment for active learning at the entire graph level is not explored in the literature. In many domain cases, the label is mapped to the entire graph instead of nodes or edges. Thus, several approaches have been proposed to address active learning for graphs on node-level tasks (Cai et al. 2017; Y. Wu et al. 2020) but not for the entire graph level. To handle all the previous issues, we design a unified active learning framework for graph-level learning on regression tasks. The following three criteria are generally considered for selecting batches (D. Wu 2019):

- *Informativeness*: The selection method should select samples where the model is mostly uncertain about the label.
- *Diversity*: The selection methods must ensure that the samples in the batch

must be diverse and different from each other.

- *Representative*: The training set selection should be concentrated on the region where the pool data distribution has high density.

The Matern kernel of the GP and Neural Tangent Kernel (NTK) with neural networks are used as base kernels for that aim, in addition to utilising supervised and unsupervised learning for the entire graph level. Our framework is task-agnostic, allowing for applying any regression method and active learning query strategy available in the literature. The obtained results are promising, meaning that our framework can be adapted to any graph domain, not only for our case study.

Through this thesis and to have more informative results, we decided to invest in predicting the scalar value of the execution time of the source code as a case study. The main reason for investing in this case study is that the source code can be represented as both tree and graphs simultaneously, meaning we have multiple intermediate representations for the same data sample. Moreover, real-life source code files are widely available on GitHub without any restrictions or constraints on access.

In this thesis and through the included papers, we are trying to address the following research questions:

RQ1 *What is the information that trees and graphs as intermediate representations convey?*

For this question, we will systematically investigate the tree and graph representation for the used case study. In addition to the semantic meaning of the structure of each representation for the models in addition to the information that trees and graphs convey for the case study.

RQ2 *Is it feasible to combine more than one representation?*

In this question, and through the systematic and mapping study, we investigate the methods, approaches, and consequences of using trees and graphs simultaneously for different learning tasks and the impact of that usage.

RQ3 *What is the behaviour of TBNN models in regression context?*

For this question, we focus more on the tree representation and the behaviour of the different deep-learning architectures used for tree classification. We investigate the literature, select the most successful TBNN models for classification, use them for regression, and detect their efficiency in that context. We build a framework that analyzes the behaviour of TBNN models for regression.

RQ4 *How to improve the behaviour of TBNN models for regression?*

In this research question and based on the results of the behaviour of the TBNN in **RQ3**, we will try to improve the behaviour of one of the TBNN models to have better behaviour for regression prediction. To address this question, we

will rely on model-centric AI to improve the model architecture and learning and utilise extra information that supports the learning process for the model on the tree. On that basis, we extend the analytical framework for TBNN by adding our model.

RQ5 *What is the impact of error analysis and other metrics?*

In this question, and through the framework built in **RQ3**, **RQ4**, we will use different error metrics and also Pearson correlation to compare the TBNN models from the perspective of different error and correlation score metrics since every metric can deliver different semantic to analyze the behaviour of the TBNN models.

RQ6 *How to enhance the regression analysis from a data perspective rather than the model?*

In this research question, we wanted to enhance the regression analysis using data-centric AI by enhancing the tree representation to have more information. Thus, we combine the tree and graph representation to have one representation of learning that holds the information conveyed by trees and graphs. The way to do that is by keeping the tree presentation and augmenting it by adding the graph edges that describe different semantics. The augmented tree is then converted into a graph accordingly as a result of the augmentation.

RQ7 *How well can a hybrid representation approach that combines tree and graph-based approaches perform for regression?*

For this question, we will validate the graph generated as a result of the **RQ6** by investigating different architectures of GNN that learn basically based on the edge types as initial information for learning. Then, we compare the behaviour of the GNN for the augmented tree by data-centric AI with their behaviour for the tree in **RQ3**.

RQ8 *What is the impact of batch mode active learning for graph level learning?*

In this research question, we will try to enhance the quality of our graph data generated in **RQ6** by acquiring labels to extend our datasets. In this question, we try to acquire labels for the most informative and representative samples using active learning query strategies.

RQ9 *What is the impact of using neural network and corresponding kernels of the quality of active learning for regression tasks?*

This question enhances the approach used to address the question **RQ8** by selecting the batch of samples based not only on the informativeness and representativeness used in **RQ8** but also the diversity. Thus, we will try to address the impact of incorporating the neural network and the corresponding kernels with query strategies to select the batch of samples based on representativeness, informativeness, and diversity.

RQ10 *How robust is the active learning framework when the graphs are expanded and updated?*

The graphs generated in **RQ7** are extended by adding more nodes and edges that express extra information related to our case study. Thus, through this question, we would check whether the unified framework we had provided for active learning for graphs on regression task cases is robust when the graphs are updated and expanded.

Chapter 2

Background

2.1 Graphs and Trees

A graph is a mathematical structure used to model relational data across various domains such as social networks (Lachi et al. 2023; Nguyen et al. 2022), biological networks (Huber et al. 2007), interaction networks (Longa, Cencetti, Lehmann, et al. 2024; Arregui-García et al. 2024; Longa, Cencetti, Lepri, et al. 2022), and mobility networks (Mauro et al. 2022; Cardia et al. 2022). It is represented as a pair (V, E) where V is the set of vertices or nodes and E is the set of edges between the nodes, $E \subseteq \{(u, v) \mid u, v \in V\}$. The graph can be undirected if it lacks self-loops and has a symmetric adjacency matrix, or directed otherwise. A *path* $P = \{v_1, \dots, v_k\}$ is an ordered sequence of connected nodes, with its length being the number of nodes it contains, and the shortest path between two nodes is the path with the minimal length connecting them. The *node neighborhood* of a node v in graph $G = (V, E)$ is the set of nodes adjacent to v , and the *degree* of a node is the number of its neighbors. The *density* of a directed graph is defined as $\text{Density} = \frac{|E|}{|V|(|V|-1)}$. A *triad* in a graph is a subset of three connected nodes, classified as *closed* if it forms a triangle with three edges, otherwise *open*.

Rooted in graph theory, a tree is a type of graph that is connected and acyclic, featuring nodes (or vertices) connected by edges (or links) with no cycles, making them a natural fit for representing data with inherent hierarchical relationships. A tree consists of nodes connected by edges, with one node designated as the root. Every node other than the root is connected by exactly one incoming edge from another node. Each node can have zero or more children nodes, leading to a hierarchical structure with levels that denote the depth of nodes from the root. Trees are particularly useful in applications where data is organized hierarchically or needs to be processed in a hierarchical manner.

2.2 Model-Centric AI

Model-centric AI represents a traditional approach that has dominated the AI research and application landscape. This paradigm focuses primarily on developing

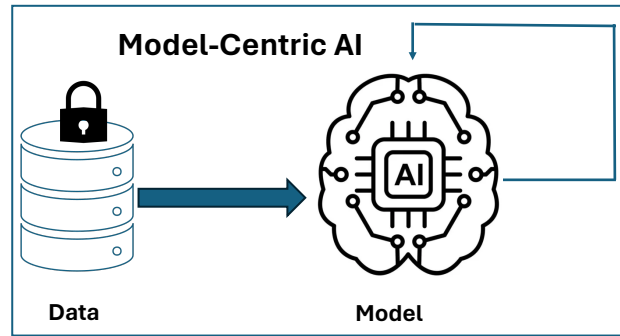


Figure 2.1: Model-Centric AI Paradigm (Figure adapted from (Jakubik et al. 2024))

and optimising machine learning models, using fixed datasets to benchmark and improve model performance. As in Figure 2.1, the dataset is always fixed, and the model is iteratively improved. Model-centric AI is defined by its emphasis on selecting and refining the appropriate machine learning models, architectures, and hyperparameters to solve specific problems (Jakubik et al. 2024). This approach aims to build AI systems that are both effective and efficient, leveraging a wide array of potential models to find the best fit for the given data and task.

Historically, the model-centric approach has propelled the advancement of AI through extensive research on model types and architectures, accompanied by rigorous hyperparameter tuning to enhance performance. This method has been prevalent in both academic settings and practical applications, where AI models are evaluated and compared using benchmark datasets. Such datasets facilitate scientific and statistically sound comparisons across different methods, significantly accelerating the evolution of AI capabilities.

However, the exclusive focus on model optimization has shown diminishing returns over time, especially as improvements plateau for many datasets. In real-world applications, merely advancing model complexity often does not translate into significant performance gains (Baesens et al. 2021). This is particularly evident when custom problems arise for which no public datasets or pre-trained models are readily available. Moreover, this model-centric focus overlooks the equally crucial role of the data that feeds these models. In conclusion, while model-centric AI has significantly advanced the field of artificial intelligence by refining computational models, the evolving landscape of AI development is increasingly acknowledging the critical role of data. This holistic view promises to enhance the robustness, applicability, and effectiveness of AI systems in diverse real-world settings.

2.3 Data-Centric AI

Data-centric AI is an emerging paradigm that emphasizes the importance of high-quality data in developing and deploying artificial intelligence systems. Unlike the traditional model-centric approach, which focuses on refining models while keeping the dataset fixed, data-centric AI aims to engineer and maintain datasets to improve

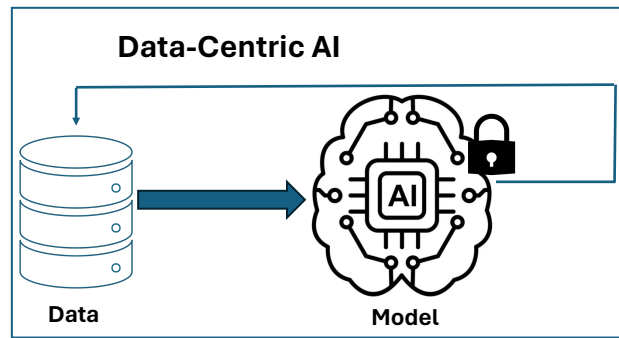


Figure 2.2: Data-Centric AI Paradigm (Figure adapted from (Jakubik et al. 2024))

AI performance systematically (Zha, Bhat, et al. 2023). Thus, as in Figure 2.2, the model is fixed, and we enhance the data quality to improve the model's performance. This approach ensures that the data used for training and inference is of the highest quality, which is critical for developing robust and reliable AI systems. The shift towards data-centric AI is driven by several factors:

1. **Model Transferability and Generalizability:** Traditional model-centric AI often struggles with transferring and generalizing models across different datasets. Enhancing data quality helps to address this issue, leading to more adaptable models.
2. **Reduction of Data Cascades:** Poor data quality can lead to a cascade of negative effects (Sambasivan et al. 2021), including biases and inaccuracies in AI systems. Focusing on data quality helps mitigate these risks.
3. **Efficiency in High-stakes Domains:** In domains such as healthcare and finance, the cost of errors is high. Ensuring data quality can significantly enhance model reliability and performance in these critical areas.

Data-centric AI organizes its pursuits around three principal goals: training data development, inference data development, and data maintenance. Each goal encompasses specific tasks tailored to enhance the data's role in improving AI outcomes. Training data development focuses on curating and refining data that teaches AI models to behave predictably and effectively. Inference data development involves crafting data sets that accurately test and challenge AI models, ensuring they are capable of performing under varied or unexpected conditions. Data maintenance is critical for sustaining the data's accuracy, consistency, and relevance, which is vital for the long-term reliability of AI systems.

In data-centric AI, the balance between automation and human involvement is critical. Automation in data-centric AI aims to streamline processes like data labelling and augmentation (Zha, Lai, et al. 2022; H. P. Samoaa, Longa, et al. 2022), reducing the need for intensive human labour while enhancing efficiency and consistency. Conversely, human expertise remains indispensable, particularly in tasks requiring nuanced judgment and decision-making, such as validating data quality or managing complex data integration scenarios. This interplay ensures that while

AI systems can operate independently, they also benefit from human oversight and intervention, leading to more trustworthy and aligned AI outcomes.

In essence, data-centric AI represents a transformative approach that redefines the priorities of AI development, advocating for a data-first strategy that promises to enhance the efficacy, fairness, and adaptability of AI systems across various sectors.

2.4 Graph Neural Networks (GNNs)

If the source code is to be represented as a graph, then Graph Neural Networks (GNNs) are the right model to handle this type of representation.

Graphs are complex structures, and verifying if two graphs are identical (also known as the isomorphism test) is an important and difficult task. It is unknown if the problem can be solved in polynomial time or if it is computationally intractable for large graphs. A fast heuristic to verify if two graphs are the same is the k -Weisfeiler-Leman test (Weisfeiler and Leman 1968). The algorithm produces a representation for each graph. Then, if the representations of two graphs are not equivalent, the graphs are not considered isomorphic. However, there is the possibility that two non-isomorphic graphs share a representation. Thereby, this test might not provide conclusive evidence that the two graphs are isomorphic. GNN network can be as powerful as the k -Weisfeiler-Leman test with k equal to 1, in which the representation propagates the information by nodes. With k greater than 1, the information is propagated among substructures of order k . A higher-order graph convolution layer (k -GNN) is also proposed, wherein messages are exchanged among nodes, edges and substructure with tree nodes (triads). Once messages are exchanged among substructures, each node has a latent representation. In order to predict the property of the graphs (i.e., the execution time of a graph representing Java code), node embeddings are globally aggregated (pooling step) with an invariant ordering function (i.e. sum, max, mean). In particular, k -GNN is defined as: given is an integer k the k -element subset $[V(G)]^k$ over $V(G)$. Let $s = \{s_1, s_2, \dots\}$ be k -set in $[V(G)]^k$, then the *neighborhood* of s is defined as:

$$N(s) = \{t \in [V(G)]^k \mid |s \cup t| = k + 1\} \quad (2.1)$$

In Equation 2.1, the neighbour of a k -set is defined as the set of k -set such that the intersection of their cardinality is equal to $k - 1$. The local neighbourhood is defined as:

$$N_L(s) = \{t \in N(s) \mid (u, w) \in E(G) \text{ with } u \in s/t \text{ and } w \in t/s\} \quad (2.2)$$

The local neighbourhood defined in Equation 2.2 is a subset of the neighbour (Equation. 2.1). Finally, the k -GNN is defined as:

$$f_{k,L}^{(l)}(s) = \sigma(f_{k,L}^{(l-1)}(s) \cdot W_1^{(t)} + \sum_{u \in N_L(s)} f_{k,L}^{(t-1)}(u) \cdot W_2^{(t)}) \quad (2.3)$$

The l -th layer of the k -GNN computes an embedding of s , using the non-linear activation function σ of the summation over the substructure itself in the previous layer (i.e., layer $l - 1$) and the summation over the previous layer embedding of each local neighbourhood of the substructure s .

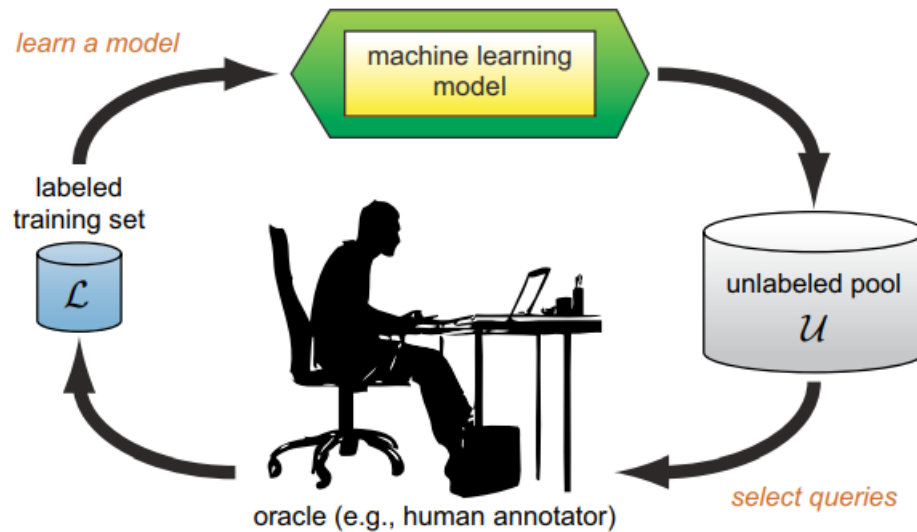


Figure 2.3: Pool Based Active Learning (Settles 2009)

2.5 Active Learning

Active Learning is a subset of machine learning where the algorithm selectively queries the most informative data points to label, optimizing the learning process using fewer training examples. This method is particularly useful in scenarios where labelled data are scarce, costly, or time-consuming to acquire.

Given set of N data points $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$. Assume that we have $m \ll N$ labeled data points $\mathcal{L} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)\}$. Then, let $\mathcal{U} = \{\mathbf{x}_{m+1}, \dots, \mathbf{x}_N\}$ be the remaining data points for which we have **no labels**. In the field of machine learning, we categorize techniques based on how they utilize data:

- **Supervised learning:** This paradigm learns from a dataset that includes input-output pairs, focusing on mapping function from inputs to outputs. It requires a fully labelled dataset \mathcal{L} .
- **Unsupervised learning:** Unsupervised techniques infer patterns from a dataset without referring to known or labelled outcomes. They are useful for discovering the underlying structure of data. They learn from all available data features \mathbf{X} .
- **Semi-supervised learning:** Semi-supervised learning falls between supervised and unsupervised learning. It uses both labelled and unlabeled data (i.e., \mathcal{L} and \mathcal{U} .) to improve learning accuracy with a minimal set of labelled instances.
- **Active Learning:** Particularly useful when labels are costly or difficult to obtain. They iteratively query an oracle (e.g., human annotator) for labels of the data points in \mathcal{U} in an efficient way. Learn from \mathcal{L} and additional labels queried so far, thus integrating human oversight directly into the learning process.

Figure 2.3 shows the iterative process of active learning explained in detail in Algorithm 1. We have an initially labelled dataset $\mathcal{L}_0 := \mathcal{L}$ and an initial unlabeled dataset $\mathcal{U}_0 := \mathcal{U}$. We also have a model f_θ , parameterized by θ . Given this, an iterative active learning procedure works based on training the model iteratively and detecting the most informative samples in each iteration, then extending the labelled dataset \mathcal{L} and training the model on the extended labelled dataset.

Algorithm 1 Train model using active learning

```

1:  $i \leftarrow 0$ 
2: repeat
3:   Train  $f_\theta$  on  $\mathcal{L}$ 
4:   Query the label  $y$  of the most informative data point  $\mathbf{x} \in \mathcal{U}$  based on the
     current model  $f_\theta$ 
5:    $\mathcal{L} \leftarrow \mathcal{L} \cup \{(\mathbf{x}, y)\}$ 
6:    $\mathcal{U} \leftarrow \mathcal{U} \setminus \{\mathbf{x}\}$ 
7:    $i \leftarrow i + 1$ 
8: until some stopping criterion is met

```

As in Algorithm 1, we query the label y of the most informative data point using different query strategies. A query strategy can consider one or both of the following notions of informativeness:

- **Uncertainty:** This corresponds to selecting the data point $\mathbf{x} \in \mathcal{U}$ for which the current model f_θ is the most uncertain.
- **Representative:** This corresponds to selecting a data point $\mathbf{x} \in \mathcal{U}$ that is representative of \mathcal{U} .

As for uncertainty, the strategies differ in that they are based on sampling for probabilistic models, query by committee, or expected model change. As for Uncertainty Sampling for Probabilistic Models, if the model f_θ is probabilistic, we assume that it estimates the posterior probability $P_\theta(\hat{y} | x)$.

- **Least confident:** Choose data points where the model's prediction is least certain. Thus, $x_{LC}^* = \underset{x}{\operatorname{argmax}}(1 - P_\theta(\hat{y} | x))$, where $\hat{y} = \operatorname{argmax}_y(P_\theta(y | x))$.
- **Margin sampling:** Select data points where the difference between the first and second most probable class labels is minimal. So $x_M^* = \underset{x}{\operatorname{argmin}}(P_\theta(\hat{y}_1 | x) - P_\theta(\hat{y}_2 | x))$, where \hat{y}_1 and \hat{y}_2 are the first and second most probable class labels under the model, respectively.
- **Entropy:** Focus on data points with the highest prediction entropy across possible outcomes. That described mathematically as:
 $x_H^* = \underset{x}{\operatorname{argmax}}(-\sum_i P_\theta(y_i | x) \log P_\theta(y_i | x))$, where y_i ranges over all possible labelings.

As for Query by Committee (QBC) approach, it involves maintaining a committee $\mathcal{C} = \{f_{\theta^{(1)}}, \dots, f_{\theta^{(C)}}\}$ of models which are all trained on the current labelled set \mathcal{L} , but represent competing hypotheses.

Given this, there are a number of ways to select a sample, and one possibility is the **vote entropy**:

$$x_{VE}^* = \operatorname{argmax}_x - \sum_i \frac{V(y_i)}{C} \log \frac{V(y_i)}{C}$$

where y_i again ranges over all possible labellings, and $V(y_i)$ is the number of "votes" that a label receives from among the committee members' predictions, and C is the committee size. This can be thought of as a QBC generalization of entropy-based uncertainty sampling.

Expected model change selects the data point that would impart the greatest change to the current model if we knew its label. An example query strategy in this framework is the "expected gradient length" (EGL). In theory, the EGL strategy can be applied to any learning problem where gradient-based training is used.

Let $\nabla \ell_{\theta}(\mathcal{L})$ be the gradient of the objective function ℓ with respect to the model parameters θ . Now let $\nabla \ell_{\theta}(\mathcal{L} \cup \langle x, y \rangle)$ be the new gradient that would be obtained by adding the training tuple $\langle x, y \rangle$ to \mathcal{L} . Since the query algorithm does not know the true label y in advance, we must instead calculate the length as an expectation over the possible labellings:

$$x_{EGL}^* = \operatorname{argmax}_x \sum_i P_{\theta}(y_i | x) \|\nabla \ell_{\theta}(\mathcal{L} \cup \langle x, y_i \rangle)\|,$$

where $\|\cdot\|$ is, in this case, the Euclidean norm of each resulting gradient vector.

As for representativeness, expected error reduction is another decision-theoretic approach that measures not how much the model is likely to change but how much its generalization error is likely to be reduced. The idea is to estimate the expected future error of a model trained using $\mathcal{L} \cup \langle x, y \rangle$ on the remaining unlabeled instances in \mathcal{U} (which is assumed to be representative of the test distribution, and used as a sort of validation set), and query the instance with minimal expected future error (sometimes called risk). One approach is to minimize the expected 0/1-loss:

$$x_{0/1}^* = \operatorname{argmin}_x \sum_i P_{\theta}(y_i | x) \left(\sum_{u=1}^U 1 - P_{\theta^+(x, y_i)}(\hat{y} | x^{(u)}) \right),$$

where $\theta^+(x, y_i)$ refers to the new model after it has been re-trained with the training tuple $\langle x, y_i \rangle$ added to \mathcal{L} .

Expected error reduction is often intractable. Variance reduction attempts to approximate it. The expected future error can be decomposed in the following way:

$$\begin{aligned} E_T [(\hat{y} - y)^2 | x] &= E [(y - E[y | x])^2] \\ &\quad + (E_{\mathcal{L}}[\hat{y}] - E[y | x])^2 \\ &\quad + E_{\mathcal{L}} [(\hat{y} - E_{\mathcal{L}}[\hat{y}])^2] \end{aligned}$$

The main idea is that informative instances should be not only uncertain but also "representative" of the underlying distribution (i.e., inhabit dense regions of the input space). Therefore, we wish to query instances as follows:

$$x_{ID}^* = \operatorname{argmax}_x \phi_A(x) \times \left(\frac{1}{U} \sum_{u=1}^U \operatorname{sim}(x, x^{(u)}) \right)^\beta .$$

Here, $\phi_A(x)$ represents the informativeness of x according to some "base" query strategy A , such as an uncertainty sampling or QBC approach. The second term weights the informativeness of x by its average similarity to all other instances in the input distribution (as approximated by \mathcal{U}), subject to a parameter β that controls the relative importance of the density term.

Active learning involves several practical considerations:

- **Batch-Mode Learning:** How to efficiently query multiple instances at once.
- **Noisy Oracles:** Dealing with inaccuracies in the labels provided by human annotators.
- **Variable Labeling Costs:** Managing the differing costs associated with labelling various data points.
- **Multi-Task and Multi-Class Scenarios:** Extending the active learning framework to handle multiple related tasks or classification problems.
- **Stopping Criteria:** Determining when the model has learned sufficiently to cease active querying.

By addressing these practical considerations and employing robust query strategies, active learning can significantly enhance learning efficiency, especially in scenarios where labelled data are scarce or expensive to obtain.

Chapter 3

General Overview of The Papers

This section will present the main components discussed in this thesis in order to address the research questions mentioned earlier. The ultimate aim is to enhance the regression analysis for both trees and graphs data structure. To that aim and as in Figure 3.1, we start by exploring the behaviour of the trees and graphs representations and the information they convey. Then, we analyse the regression for trees by analyzing the behaviour of TBNN models for the regression task. Then, based on this analysis, we enhance the behaviour of TBNN for regression by proposing a new model based on model-centric AI. Then, we move to graphs from the trees by augmenting the trees with extra edges based on data-centric AI to convert the tree into a graph. Then, GNN models enhance the regression for our generated graphs. Finally, we invest in interactive learning and active learning for graphs to enhance the data quality for regression analysis.

To achieve the mentioned goal, we use the source code analysis for scalar value prediction of execution time as a case study. The reason is that i) source code can be represented as trees and graphs simultaneously. ii) the source code files are available on public host platforms like GitHub for free iii) since the source code can be available for free, then we don't need to care so much about the privacy issue since we are not dealing with sensitive data like patients records or human biological trees or even the graph of molecular where each company has its own drug. Above all that, execution time prediction is an important task since it gives the developer an early indication of the complexity of the source code before it runs.

3.1 Exploration of Tree and Graph Representation

This section will answer the research questions **RQ1**, **RQ2**. To that aim, we conducted a systematic literature review and systematic mapping study in our first

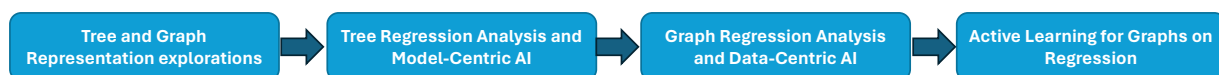


Figure 3.1: General Overview of the Thesis

paper titled: "A systematic mapping study of source code representation for deep learning in software engineering" (H. P. Samoaa, Bayram, et al. 2022).

RQ1 *What information do trees and graphs convey as intermediate representations?*

Answer: The source code can be represented as token-based, tree-based, and graph-based. As for tree representation, it represents the syntactical information. On the other hand, graph representation conveys the semantic information of the code.

RQ2 *Is it feasible to combine more than one representation?*

Answer: Notably, a substantial number of publications use a hybrid representation approach, combining multiple different representations.

3.1.1 Analyzing Trees and Graphs as Intermediate Representations

In this section, we will dig deeply into the answer to research question **RQ1**. Through that, we discuss the intermediate representation of the source code for the deep learning models. The first step in the deep learning process is proper data representation. We must represent the source code in a format suitable for the model and the task of interest. Thus, the literature mainly classifies code representation approaches into three categories: *token-based*, *Tree-based*, and *Graph-based*. Every form maps the source code's syntactical and semantic aspects to a specific data structure. These representations can then be embedded in a neural network so that they can use source code as input. Source code is originally a text encoding representing a program. This can be processed and further transformed into different representation forms. In this section, we describe three well-known representations, each one mapping certain aspects of the original source code. We use the C snippet depicted in Listing 3.1 as a running example. This example has originally been proposed by Yamaguchi et al. (Yamaguchi et al. 2014)

Listing 3.1: Example of C code (Yamaguchi et al. 2014).

```
void foo() {
    int x = source();
    if( x < MAX ) {
        int y = 2*x;
        sink(y);
    }
}
```


Token-Based Representation This representation treats code as free text. Thus, it converts the code into a list of tokens where each word (e.g., "void") is a token, but each special character (e.g., '(') is also a token (rather than considering it as part of a word). An example is given in Listing 3.2.

Listing 3.2: Token Representation for the code in Listing 3.1.

```
[ 'void ', 'foo ', '(', ')', '{', 'int ', 'x', '=',  
  'source ', '(', ')', ';', 'if ', '(', 'x', '<',  
  'MAX', ')', '{', 'int ', 'y', '=', '2*x', ';',  
  'sink ', '(', 'y', ')', ';', '}', '}' ]
```

Then, each token will be encoded into a numerical vector using different statistical language models, such as word embedding (Teller 2000) or n-grams (Niesler and Woodland 1996). In principle, word embedding is a learned representation for text where words that have the same meaning get a similar representation. Technically, word embeddings are a class of techniques where individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector, and the vector values are learned in a way that resembles a neural network. Hence the technique is often lumped into the field of deep learning. N-grams are several words appearing together. They are useful abstractions for modelling sequential data such as text, where there are dependencies among the terms in a sequence. However, a corpus of code can be regarded as a sequence of sequences, and corpus-based models such as n-grams learn conditional probability distributions from the order of terms in a corpus. Corpus-based models can be used for many different types of tasks, such as discriminating data instances or generating new data characteristics of a domain. Embeddings can be considered a way to represent words and help the DL model learn the source code's representation. An embedding can be trained to represent n-grams or just individual words.

Tree-Based Representation This representation captures the abstract syntactic structure of the source code. Abstract syntax trees (ASTs) are a kind of tree representation approach that is widely used by programming language tools.

Figure 3.2 shows an example of an AST representation. The nodes of the AST tree are related to constructs or symbols of the source code. In comparison to the token-based approach, AST representation is abstract and does not include all available details, such as punctuation and delimiters. Theoretically, ASTs can be used to illustrate the lexical information and the syntactic structure of source code, such as the function name and the flow of the instructions (for example, in an if or while construct).

Graph-Based Representation This approach represents source code as a graph at many different levels. Levels of representation define the type of the representation graph. Thus, a control flow graph (CFG, see 3.3 (a)) describes the sequence in which the instructions of a program will be executed. Thus, the graph is determined by

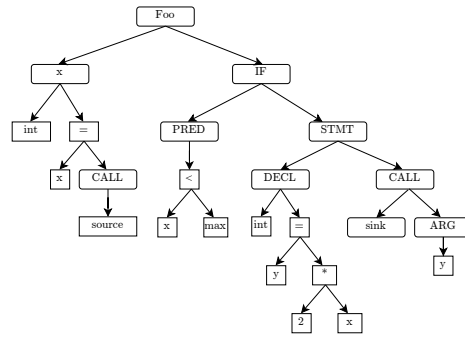


Figure 3.2: Abstract syntax tree (AST) for the code snippet in Listing 3.1 (Yamaguchi et al. 2014).

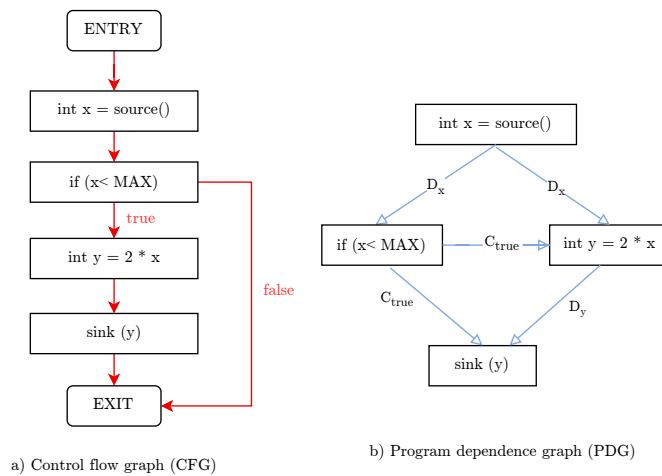


Figure 3.3: Graph-based representations for the code snippet in Listing 3.1 (Yamaguchi et al. 2014).

conditional statements, e.g., if, for, and switch statements. In CFGs, nodes denote statements and conditions, and directed edges connect them to indicate the transfer of control.

Alternatively, the representation might be a variable-oriented data flow. Thus, a data flow graph (DFG) is used to follow and track the usage of the variables through the CFG. A DFG edge represents the subsequent access or modification of the same variables. The call flow graph (CallFG) captures the relation between a statement which calls a function and the called function (Cummins et al. 2020). Finally, the entire program can be represented as a graph using a program dependence graph (PDG, see 3.3 (b)), where the nodes can characterize statements and predicate expressions. In this study, we differentiate between tree-based and graph-based approaches since each representation retrieves a different level of information from the source code. Thus, the tree-based approach, such as using the AST, extracts syntactical information from the source code, whereas graph-based approaches, such as CFG or DFG, extract semantic information.

3.1.2 Exploring the Integration of Multiple Representations

This section will deeply investigate the research question **RQ2**. As we observed in our first paper (H. P. Samoaa, Bayram, et al. 2022), some studies have utilized a hybrid approach for code representation to capture more information on the source code. This is often promising as tree-based approaches capture syntactical information, graph-based approaches better retain semantics, and token-based approaches preserve lexical information. Thus, studies like (J. Hua and H. Wang 2021; Z. Li et al. 2021; Fang et al. 2020) combined representations from all three groups. The most common hybrid approach combines token- and tree-based approaches. Combinations of the tree- and graph-based approaches are also fairly popular. The problem with all mentioned combined representation approaches is that they are separated into multiple representations, not combined in one representation. These separated representations constitute multiple inputs for either one deep learning model or multiple models (each with one input representation) to address one or more tasks. Thus, we do not have one rich representation approach that compresses all the information from the source code. More details about these results can be found in Section 8.2 in Paper I.

3.2 Tree Regression Analysis and Model-Centric AI for Trees

This section will be dedicated to answer the research questions **RQ3**, **RQ4**, and **RQ5**. For that purpose, in our second paper titled: "*Analysing the Behaviour of Tree-Based Neural Networks in Regression Tasks*" (P. Samoaa, Farahani, et al. 2024), we conduct a comparative and empirical study. We will first provide a brief answer for each research question and then discuss each question in depth.

RQ3 *What is the behaviour of TBNN models in the regression context?*

Answer: The TBNN models failed to deliver an efficient performance despite their remarkable efficiency in classification tasks.

RQ4 *How to improve the behaviour of TBNN models for regression?*

Answer: Model-centric AI is then used to improve the behaviour of the tree-based transformer model. The improvement is proposed as an alternative dual-transfer based on cross-attention. In the dual transformer, a token-based input is also utilised beside the tree, which improves the transformer's behaviour as TBNN in regression.

RQ5 *What is the impact of error analysis and Pearson correlation metrics?*

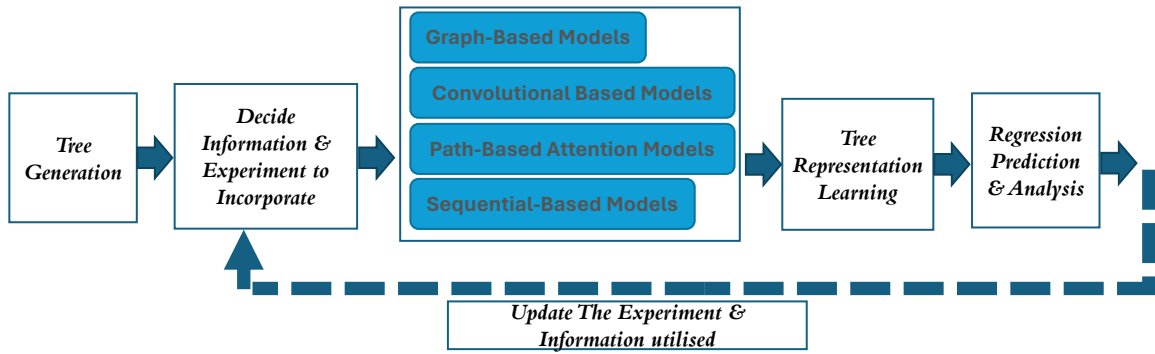


Figure 3.4: Tree Based Neural Networks Analytical Framework

Answer: The discrepancy between Pearson correlation and error metrics like MSE and MAE in different models like GNNs and other TBNNs can be attributed to the nature of the models and the type of data they process. GNNs often exhibit good Pearson correlation because they effectively capture the overall patterns and relationships in trees through their structural and node-level interactions, making them adept at predicting trends accurately. However, they might not minimize individual prediction errors effectively, leading to less efficient MSE and MAE scores. Conversely, some TBNN models might achieve lower Pearson correlation but better error metrics. This could occur if these models are very accurate on average (hence low MSE/MAE) but fail to effectively capture the underlying patterns or dependencies between the predicted and actual values, particularly in complex or noisy datasets where structural nuances significantly impact model performance. This suggests a trade-off between capturing overall trends and minimizing point-specific errors, highlighting the importance of choosing the right model based on the data's specific analytical requirements and characteristics.

3.2.1 Behaviour of TBNN models in Regression Context

This section will deeply investigate the research question **RQ3**. Extracting the tree representation for the source code is easier and straightforward since we need to parse the code without running it, so we start with the AST, the tree representation of the source code, for regression analysis. Recently, some approaches have combined neural networks and ASTs to constitute tree-based neural networks (TBNNs) (Zhang et al. 2019). Given a tree, TBNNs learn the vector representation by recursively computing node embeddings in a bottom-up way. Popular TBNN models are the Recursive Neural Network (RvNN) (White et al. 2016), Tree-based CNN (TBCNN) (Mou et al. 2016), and Tree-based Long Short-Term Memory (Tree-LSTM) (Wei and M. Li 2017). Based on Figure 3.4, many TBNN architectures are used for regression analysis like GNNs Convolutional models, Code2Vec (Alon et al. 2019), which is a Path-Based attention model, and sequential-based transformer (W. Hua and Liu 2022). All the

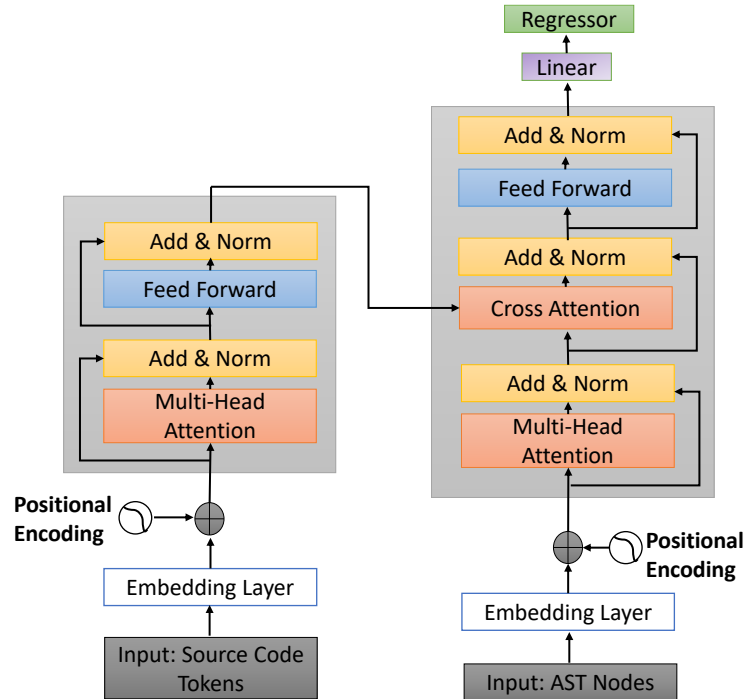


Figure 3.5: The architecture of the Dual-Transformer model. The framework features two transformer encoders: NLEncoder for source code tokens and AS-TEncoder for AST nodes, each with layers for embedding, multi-head attention, and feed-forward networks, complemented by add & norm layers for stabilization. Their outputs are merged via cross-attention and passed to a linear regressor for error prediction, leveraging both textual and syntactical insights.

details for the aforementioned models are reported in Section 4, Paper 2. We chose these models because they have achieved a remarkable performance for classification tasks. Thus, we want to try them for regression-related tasks.

Hence, to address **RQ3**, we designed an analytical framework 3.4 that examines the behaviour of different TBNN architectures in a regression context. These models do not deliver the same efficiency in regression tasks, specifically for transformer-based models. In this model, the tree nodes are flattened, and then the sequences of the nodes will be the input for the model, leading to a limitation in dealing with extended sequence lengths. Moreover, we realise that one of the issues in the transformer-based tree model is the inability to effectively capture the interplay between the different types of input data, such as textual and structural representation of the code.

3.2.2 Model-Centric AI for Trees

This section addresses the gap in the poor behaviour of different TBNN models in regression and thus answers the research question **RQ4** regarding the possibilities to enhance the behaviour and efficiency of TBNN in a regression context.

Thus, by model-centric AI, we enhance the transformer model by proposing our dual transformer model 3.5. In this model, we use the transformer block for the tree and another transformer block for the source code token. Thus, we also utilise the

token-based representation of the source code alongside the tree representation. Then, we modify the AST transformer block by adding cross attention to understand the impact of each code token on the tree node as in Figure 3.5. More details about our model are in Section 5, Paper 2. Our model shows efficiency across all architectures of TBNN models on all used metrics.

3.2.3 Error and Correlation Analysis for TBNN models on Regression

This section will address the research question **RQ5**. Throughout the comparative study of the TBNN used in regression, we will analyze the metrics used to evaluate the used models in our analytical framework. There is a general discrepancy between the error metrics used, such as MSE and MAE, and the Pearson correlation that finds the correlation between the predicted value and the true one over the test set. GNN models demonstrate good Pearson correlation but may not perform as well on error metrics, indicating that while they can capture the trend of the data well, they might still make large individual errors. On the other hand, certain TBNN models (code2vec, TreeCNN, and Tree-based Transformer) may show strong performance in minimizing error metrics but exhibit lower Pearson correlation compared to GNN, suggesting that these models, while accurate on average, may fail to capture underlying trends or dependencies in the data effectively. However, Our dual-transformer model shows the best scores in error and correlation metrics, showing its ability to detect patterns from the data and minimise the error.

This variation highlights the importance of selecting appropriate metrics based on the specific requirements and goals of the analysis. For instance, when precise error minimization is critical, focusing on MSE and MAE might be preferable. However, for applications where understanding the strength of relationships within the trees is more important, Pearson correlation becomes a crucial metric. Understanding these metrics in conjunction helps in refining model selection and tuning, ensuring that the chosen models are not only accurate but also align well with the specific analytical objectives of a study.

3.3 Data-Centric AI for Graphs

This section will answer the research questions **RQ6** and **RQ7**. For that purpose, in our third paper titled: "*TEP-GNN: Accurate Execution Time Prediction of Functional Tests Using Graph Neural Networks*" (H. P. Samoaa, Longa, et al. 2022), we conduct an empirical study. We will first provide a brief answer for each research question and then discuss each question in depth.

RQ6 *How to enhance the regression analysis from a data perspective rather than the model?*

Answer: We must improve the tree representation to have a richer representation so the model can learn and detect more patterns from the new representation. Thus, we follow the data-centric AI to enhance the quality of the tree representation by adding more edges that can add more information to the tree structure, which leads to better learning for the models, especially for regression tasks. Connecting this approach with our case study, we merged the tree and graph representations discussed in Section 3.1.1 in one solid and rich representation. By doing so, we address the gap of misuse of all representations together, as we had discussed in Section 3.1.2

RQ7 *How well can a hybrid representation approach that combines the tree and graph-based methods perform for regression?*

Answer: The outcome of the tree augmentation and having a hybrid representation of tree and graphs in **RQ6** will lead to having a graphs data structure. Thus, GNN models are used to examine the quality of the produced graphs in a regression context. Using the same datasets and metrics used to examine the GNNs in tree contexts, the GNNs tend to have a way better performance for error metrics and also Pearson correlation. This leads to the conclusion that improving the quality of data representation will certainly improve the model behaviour for regression tasks.

3.3.1 From Tree to Graph over Data-Centric AI

This section will deeply investigate the research question **RQ6**. As we saw in Section 3.2, we enhanced the regression analysis by improving the models based on Model-Centric AI. In this Section, we enhance the regression analysis by only focusing on the data. Thus, we will follow the data-centric AI to improve the tree representation to have a richer representation that can deliver more information and patterns to the model. From a case study point of view, we manipulated the token-based and tree-based representations in our dual transformer in 3.2.2, but we still do not use the graph-based representations. As we clarified in Section 3.1, the graph representations might have different topological structures depending on the semantic information the graph delivers. Thus, we have control flow graphs that explain the semantics of the execution and data flow graphs that describe how the data is updated through the logic of the code. However, one of the issues is that each statement is represented as a node as in Figure 3.3. That means that we have fewer edges compared to the tree, which leads to less information being exchanged throughout the structure of the graph, so there is less ability for the GNN models to learn from these graphs. Moreover, generating these graphs requires running each code in order to extract the corresponding flow graphs, which is more time, effort, and computational resources consuming. To extract the tree, we need to parse the code. Thus, we decided to stick with the tree representation and augment the tree

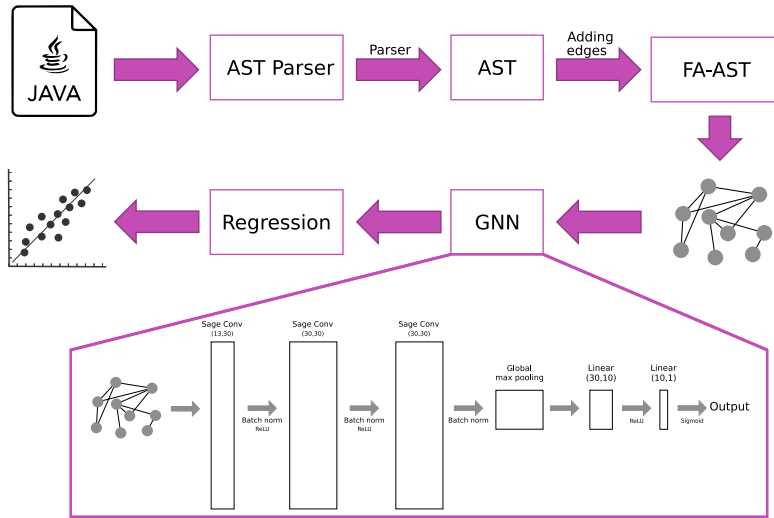


Figure 3.6: Schematic Overview of Data-Centric AI Approach to Enhance the Regression Analysis

representation only by adding edges that explain the control and data flow graphs. We called the new graphs generated from trees Flow-Augmented AST (FA-AST). We designed the augmentation strategies for each control statement in addition to designing strategies for tracking the data flow through the tree. Then, we tackled the nodes of the trees that describe the control and data statements and attached the new edges according to our strategies. Each edge is assigned a label that describes the functionality of these edges. Section 2.3 of Paper 3 provides a deeper understanding of how we generate the graphs of FA-AST with motivation examples, in addition to the augmentation of different control and data statements.

3.3.2 Validating the Data-Centric AI Approach

This section will deeply investigate the research question **RQ7**.

As Figure 3.6 illustrates, we build FA-AST graphs and store them as *PyTorch Geometric*¹ objects. Then, GNN models can be directly used to examine the quality of the transformed graphs in a regression context. Applying the GNN models on the same datasets and using the same evaluation metrics previously used in tree contexts makes it evident that GNNs exhibit significantly improved performance, both in error metrics and Pearson correlation. This stark improvement confirms that refining data representation quality directly contributes to more effective model performance in regression tasks, highlighting the potential of hybrid structures to optimize predictive accuracy and model reliability in complex data environments. Out of all used GNN architectures, Graph Conv (Spectral Graph Convolution) (Defferrard et al. 2016) achieved the highest efficiency both in error and Pearson correlation metrics. We refer to Section 2.4 in Paper 3 for further details about the model.

¹<https://pytorch-geometric.readthedocs.io/en/latest>

3.4 Active Learning for Graphs

This section will answer the research questions **RQ8**, **RQ9**, and **RQ10**. For that purpose, in our fourth paper titled: "*A Unified Active Learning Framework for Annotating Graph Data For Regression Task*" (P. Samoaa, Aronsson, Longa, et al. 2023), and in the fifth paper titled: "*Batch Mode Deep Active Learning for Regression on Graph Data*" (P. Samoaa, Aronsson, Chehreghani, et al. 2023), we conduct an empirical study to use active learning for acquiring labels for graphs and extend the labelled dataset for best investment of learning models on graphs. We will first provide a brief answer for each research question and then discuss each question in depth.

RQ8 *What is the impact of batch mode active learning for graph level learning?*

Answer: Through our unified active learning framework designed for graphs in the regression context, we find out that the active learning selection method efficiently selects batches containing informative and representative samples. However, the effectiveness of the selection methods depends on the embedding techniques used for graph representation learning.

RQ9 *What is the impact of using a neural network and corresponding kernels on the quality of active learning for regression tasks?*

Answer: On top of the graph models used for graph embeddings, a fully connected neural network is also used to transform the embedding from one feature space to another. Then, in the last neural network layer, the NTK kernel is used to measure the similarities between the vectors of the embedded data points. Then, the kernel is manipulated in the GP, which is involved in active learning query strategies to preserve the diversity of the selected samples in each batch. Through that aim, we iteratively add one new data point to the batch based on the current GP, then retrain the GP with the selected data point and learned kernel based on the parameters of the neural network to select a new data point that is different from the first one, leading to diversity. So, each selected data point in the batch takes the previous ones into account. When a neural network is used, the framework tends to be batch mode **deep** active learning on graphs for regression tasks.

RQ10 *How robust is the active learning framework when the graphs are expanded and updated?*

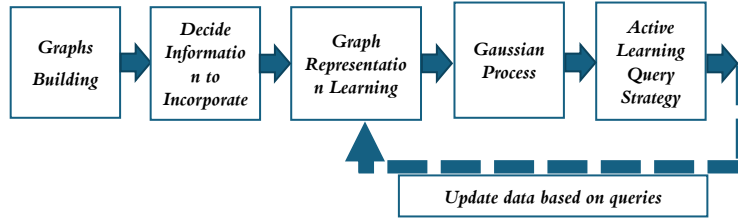


Figure 3.7: Active Learning Framework for Graphs in Regression settings

Answer: We have a case where the graphs can be expanded and updated, keeping the same regression value. Thus, we test the robustness of our framework for this scenario. Throughout our experiment, we find that the benefit of active learning increases for the expanded versions of the graphs (i.e., when the data is more complex). Arguably, this is when active learning is the most important.

3.4.1 Informativeness and Representativeness

In this section, we will dive deeply into the research questions **RQ8** and **RQ10**, where we investigate the behaviour of our active learning framework in terms of representativeness and informativeness for original graphs and the expanded version of the graphs.

Starting from the graphs created by the data-centric AI in Section 3.3, we tackle the problem of insufficient data, which is a lack of quality and availability of labels. Thus, we will use active learning as an interactive learning approach to acquire labels for our generated graphs. Through active learning, each iteration selects the most informative samples for labelling. Active learning for graphs is widely investigated in the literature, mostly for classification and at the node level. However, to our knowledge, active learning for graph-level and regression has still not been explored well in the literature. Since we have to map the entire graph to a regression value, it is important to invest the query strategies of active learning for the entire graph instead of one single node. To tackle this issue, we provided a unified active learning framework for graph annotation for the entire graph level and regression task. In active learning, we only access a small portion of labelled data. Then, a model is trained based on the labelled data and updated based on the acquired labelled data through active learning. On that basis, for active learning, we have three datasets: Labelled set \mathcal{L}_i , Unlabelled pool \mathcal{U}_i , and Test set \mathcal{T} . In principle, the information that can be used to perform the active learning are the labels and features of these datasets, i.e., $\mathbf{X}_{\mathcal{L}_i}$, $\mathbf{X}_{\mathcal{U}_i}$, $\mathbf{X}_{\mathcal{T}}$, $\mathbf{Y}_{\mathcal{L}_i}$, $\mathbf{Y}_{\mathcal{U}_i}$ and $\mathbf{Y}_{\mathcal{T}}$.

As in Figure 3.7, and based on graphs generated in Section 3.3, we manipulate different information for active learning, like training features $\mathbf{X}_{\mathcal{L}_i}$ and test set features $\mathbf{X}_{\mathcal{T}_i}$, besides the initially available labels $\mathbf{Y}_{\mathcal{T}_i}$. We get the features of the graphs by learning the representation of the graphs using unsupervised based on graph shallow embedding, supervised based on GNN, and manual embedding, which is based on the graph metrics without the need for any learning process (more

information about the used approach for graph learning used in our framework is available in Section 4.4 of Paper 4). Regression tasks inherently lack a natural measure of uncertainty, which is often straightforward in classification tasks through softmax layers. Computing uncertainties in regression, therefore, becomes less straightforward, necessitating the use of kernel methods. Therefore, we admit a Gaussian Process (GP) framework (Rasmussen and Williams 2005) in order to investigate and utilize different notions of uncertainty due to their probabilistic nature (Rasmussen and Williams 2005). Through the GP, we learn the Matern kernel for entropy measurement. This uncertainty model allows us to define natural acquisition functions that can be used in an active learning setting. It is also worth mentioning that GP is also used for regression prediction to obtain the Pearson correlation. Then, active learning query strategies are used to select samples for labelling. In our framework, we manipulated different query strategies:

- **Coreset** (Sener and Savarese 2018): It preserves the representativeness based on the distance in the feature space of the graph embeddings.
- **Variance**: It is based on uncertainty estimations provided by GP.
- **Query-by-committee (QBC)**: This corresponds to fitting n estimators to subsets of the labelled data. If the estimators disagree strongly about a data point, this indicates large uncertainty and, thus, informativeness.

More details about the query strategies are in paper 4, section 4.7.

Results have shown the efficiency of the query strategies employed in our framework for graph-level learning in the regression. It is worth mentioning that the embedding quality determines the effectiveness of each query strategy. Thus, for our dataset, the unsupervised embedding using the graph2vec approach delivers the best results for the original and extended graphs. As for the supervised embeddings through the GNN models, the efficiency of query strategies is delivered only for the original graphs, whereas, for expanded graphs, the embedding quality is drastically decreased, affecting the active learning acquisition functions behaviour. As for the behaviour of selection methods, they tend to be close to each other, especially in supervised embeddings. Details about the results can be found in paper 4, section 5.5.2 and appendix B.3 in the same paper.

3.4.2 Diversity

This section will intensely discuss the research question **RQ9**. Matern kernel is a model-based kernel commonly used for GP, but it might not be the best for a particular task. We use the Matern kernel with GP to approximate the uncertainty for the selection methods in active learning. Our framework so far is able to select the batch of samples based on the informativeness and representativeness of the samples elected from the unlabelled pool. Thus, we select the batch of top- $|\mathcal{B}|$ data points from \mathcal{U}_i according to:

$$\sigma(\mathbf{x}): \mathcal{B}^* = \arg \max_{\mathcal{B} \subseteq \mathcal{U}_i, |\mathcal{B}|=B} \sum_{\mathbf{x} \in \mathcal{B}} \sigma(\mathbf{x})$$

where B is the batch size and $\sigma(\mathbf{x})$ is the variance of the model in data point x . As a result of that, diversity is still not preserved in our framework. In Diversity, the selection methods select the samples that are diverse and different from each other. We extend our framework to be a **deep** mode active learning framework by using NN as a deep learning model, which requires the use of an NTK kernel such that we can utilize GP posterior uncertainties for active learning. The NN is used to learn and approximate the kernel in order to find similarities between the data points and preserve the diversity. We chose the NTK kernel as a corresponding base kernel compatible with the NN. Then, kernel transformations (i.e., scaling, sketching, or GP posterior) are employed to improve the kernel approximation. GP is then used based on the kernel learned through the NN to approximate the uncertainty in query strategies. Algorithm 4 in paper 5 clarifies that we preserve the diversity through the loop (in line 8) by iteratively selecting the data point and then updating the GP based on the selected data point and the kernel. As in Algorithm 3 in Paper 5, We get the kernel from the NN model f_θ as the gradient of the last layer $\phi_{ll}(x) := \nabla_{\tilde{W}^{(L)}} f_{\theta_T}(x)$ (line 2). So, instead of taking the gradient of the entire NN, we only take the gradient of the last layer of the NN, which is now cheaper. Then, the kernel is used to update the GP and make the selection using the updated GP and the previously selected data point (line 3). In this context, GP is only used through the active learning process, whereas the NN is used in the prediction by obtaining the Pearson correlation score. In this setting, and by using the NTK kernel alongside an NN, the selection methods tend to reach the best score quickly. We notice that in this setting, the active learning selection methods vary from each other in the supervised embedding (Paper 5, Figure 6-b), which was not the case in the previous setting. That is reasonable because of the diversity and also because the GNN model training is repeated through the active learning process based on the extended training set after acquiring the labels (Paper 5, Algorithm 2). In an unsupervised setting, and since we ignore the label for computing the latent feature space of embedding, the selection methods tend to be close to each other (Paper 5, Figure 6-a) since we compute the embedding once before the active learning loop (Paper 5, Algorithm 1). It is also worth mentioning that in this setting, the consistency of the dataset is also defining the behaviour of selection methods. Thus, the selection methods fluctuate and vary when we have datasets collected from diverse sources (Paper 5, Figure 7.). More results are in Paper 5, Section IV-C.

It is worth mentioning that if we use Algorithm 4 in our first framework, then we can preserve the diversity by using the Maten kernel with GP without the need for the use of fully connected NN and the NTK kernel since the NN is used to have a better prediction, and NTK manipulates the embedded data points to find similarities between the data points by dot products.

3.5 Contributions

This thesis has made several significant contributions, summarized as follows:

- **Dataset Collection:** We curated a comprehensive dataset encompassing both

trees and graphs, which we have made available to the research community to foster further exploration and advancements.

- **Analytical Framework for TBNNs:** We developed a robust framework to analyze the behaviour of Tree-Based Neural Networks (TBNNs) in regression tasks. This framework integrates various architectural models, offering researchers a versatile tool for extending their investigative studies.
- **Dual Transformer Model:** Our innovative model addresses limitations inherent in existing TBNNs by effectively utilizing source code tokens for regression analysis of tree structures, with potential applications extended to other domains characterized by tree-like data and associated textual descriptions.
- **TEP-GNN Model:** We introduced a Graph Neural Network model designed for handling regression analyses of large and complex graphs. This model serves both as a direct tool for researchers and as a benchmark for future developments in the field.
- **Open Source Active Learning Framework for Graphs:** We have provided an open-source implementation of an active learning framework tailored for graph data. This framework supports diverse configurations and is adaptable to a wide array of graph-based applications, significantly broadening its applicability.

3.6 Limitations and Challenges

In this section, we will present the limitations and challenges of the proposed solutions in this thesis.

In the development of the TBNN analytical framework, we selectively included various architectural models. This selective approach means that not all possible TBNN models were analyzed, which may limit the generalizability of our findings across all TBNNs.

Furthermore, our active learning framework demands significant computational resources. This is particularly evident during the iterative training cycles in supervised active learning, where the GNN model requires retraining with each iteration to incorporate newly acquired labelled data. This intensive computation can be a barrier in scaling the application to larger datasets or more frequent updates.

The frameworks and solutions proposed in this thesis can be utilised in any domain represented as graphs or trees. However, some of our proposed solutions are domain-specific. Thus, the data-centric AI for AST, by adding control and data flow edges, as well as the proposed dual transformer based on using source code tokens in addition to the AST, are domain-specific.

Chapter 4

Concluding Remarks and Future Works

4.1 Conclusion

In this thesis, we enhance the regression analysis of trees and graphs. By developing the dual transformer model, we address specific gaps identified in the behaviour of Tree-Based Neural Networks (TBNNs) within regression contexts. Furthermore, we enhance tree representations by adding semantically richer edges and converting these augmented trees into graphs. Our approach also includes refining graph quality through active learning for label acquisition.

Source code performance prediction is used as a case study. However, we claim that the approaches proposed can be applied to other domains that can be represented as trees or graphs since the provided models learn based on the topological structure of the trees and graphs. The obtained results might be different from one domain to another. It is worth mentioning that if we would like to enhance the results for a specific domain, heterogeneous graph learning can be manipulated to consider the semantics of the nodes and edges of the trees and graphs.

4.2 Future Work

Identifying which parts of the graphs most significantly influence the regression values is crucial. Multi-agent reinforcement learning could enhance graph explainability by delving into each subgraph's complexities and interdependencies.

In our active learning framework, while the model remains fixed across scenarios with varying selection methods, the impact of the model on these methods warrants further investigation. Enhancing the model could potentially improve selection method behaviours. Therefore, proposing a dynamic active learning process, possibly through meta-learning, could be beneficial. This approach would allow for adjustments or changes in the model's architecture in response to different query strategies. Additionally, exploring how model hyperparameterization affects the behaviour of active learning acquisition functions will be an important area of study.

Bibliography

- Adams, Dean C. and Michael L. Collyer (2019). “Phylogenetic Comparative Methods and the Evolution of Multivariate Phenotypes”. In: *Annual Review of Ecology, Evolution, and Systematics* 50. Volume 50, 2019, pp. 405–425. ISSN: 1545-2069. DOI: <https://doi.org/10.1146/annurev-ecolsys-110218-024555> (cit. on p. 3).
- Åkerblom, Niklas, Fazeleh Sadat Hoseini, and Morteza Haghiri Chehreghani (2023). “Online learning of network bottlenecks via minimax paths”. In: *Machine Learning* 112.1, pp. 131–150 (cit. on p. 3).
- Alon, Uri, Meital Zilberstein, Omer Levy, and Eran Yahav (Jan. 2019). “Code2vec: Learning Distributed Representations of Code”. In: *Proc. ACM Program. Lang.* 3.POPL. DOI: [10.1145/3290353](https://doi.org/10.1145/3290353) (cit. on p. 22).
- Arregui-García, Beatriz, Antonio Longa, Quintino Francesco Lotito, Sandro Meloni, and Giulia Cencetti (2024). “Patterns in temporal networks with higher-order egocentric structures”. In: *Entropy* 26.3, p. 256 (cit. on p. 9).
- Baesens, Bart, Sebastiaan Höppner, and Tim Verdonck (2021). “Data engineering for fraud detection”. In: *Decision Support Systems* 150. Interpretable Data Science For Decision Making, p. 113492. ISSN: 0167-9236. DOI: <https://doi.org/10.1016/j.dss.2021.113492> (cit. on p. 10).
- Bi, Wendong, Xueqi Cheng, Bingbing Xu, Xiaoqian Sun, Li Xu, and Huawei Shen (2023). “Bridged-GNN: Knowledge Bridge Learning for Effective Knowledge Transfer”. In: *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. CIKM '23. <conf-loc>, <city>Birmingham</city>, <country>United Kingdom</country>, </conf-loc>: Association for Computing Machinery, pp. 99–109. ISBN: 9798400701245. DOI: [10.1145/3583780.3614796](https://doi.org/10.1145/3583780.3614796) (cit. on p. 4).
- Bongini, Pietro, Monica Bianchini, and Franco Scarselli (2021). “Molecular generative Graph Neural Networks for Drug Discovery”. In: *Neurocomputing* 450, pp. 242–252. ISSN: 0925-2312. DOI: <https://doi.org/10.1016/j.neucom.2021.04.039> (cit. on p. 3).
- Cacciarelli, Davide, Murat Kulahci, and John Sølve Tyssedal (2024). “Robust online active learning”. In: *Quality and Reliability Engineering International* 40.1, pp. 277–296. DOI: <https://doi.org/10.1002/qre.3392>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/qre.3392> (cit. on p. 4).
- Cai, Hongyun, Vincent W. Zheng, and Kevin Chen-Chuan Chang (2017). *Active Learning for Graph Embedding*. arXiv: 1705.05085 [cs.LG] (cit. on p. 4).

- Caramalau, Razvan, Binod Bhattarai, and Tae-Kyun Kim (June 2021). “Sequential Graph Convolutional Network for Active Learning”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9583–9592 (cit. on p. 4).
- Cardia, Marco, Massimiliano Luca, and Luca Pappalardo (2022). “Enhancing crowd flow prediction in various spatial and temporal granularities”. In: *Companion Proceedings of the Web Conference 2022*, pp. 1251–1259 (cit. on p. 9).
- Cramer, Aurora Linh, Vincent Lostanlen, Andrew Farnsworth, Justin Salamon, and Juan Pablo Bello (2020). “Chirping up the Right Tree: Incorporating Biological Taxonomies into Deep Bioacoustic Classifiers”. In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 901–905. DOI: 10.1109/ICASSP40776.2020.9052908 (cit. on p. 3).
- Cummins, Chris, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefer, and Hugh Leather (2020). *ProGraML: Graph-based Deep Learning for Program Optimization and Analysis*. arXiv: 2003.10536 [cs.LG] (cit. on p. 20).
- Defferrard, Michaël, Xavier Bresson, and Pierre Vandergheynst (2016). “Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering”. In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett. Vol. 29 (cit. on p. 26).
- Fang, Chunrong, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi (2020). “Functional Code Clone Detection with Syntax and Semantics Fusion Learning”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, pp. 516–527. ISBN: 9781450380089. DOI: 10.1145/3395363.3397362 (cit. on p. 21).
- Hamid, Oussama H. (2022). “From Model-Centric to Data-Centric AI: A Paradigm Shift or Rather a Complementary Approach?”. In: *2022 8th International Conference on Information Technology Trends (ITT)*, pp. 196–199. DOI: 10.1109/ITT56123.2022.9863935 (cit. on p. 3).
- He, Kai, Lixia Yao, JiaWei Zhang, Yufei Li, and Chen Li (Aug. 2021). “Construction of Genealogical Knowledge Graphs From Obituaries: Multitask Neural Network Extraction System”. In: *J Med Internet Res* 23.8, e25670. ISSN: 1438-8871. DOI: 10.2196/25670 (cit. on p. 3).
- Hsu, Wei-Ning and Hsuan-Tien Lin (Feb. 2015). “Active Learning by Learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 29.1. DOI: 10.1609/aaai.v29i1.9597 (cit. on p. 4).
- Hu, Shengding, Zheng Xiong, Meng Qu, Xingdi Yuan, Marc-Alexandre Côté, Zhiyuan Liu, and Jian Tang (2020). “Graph Policy Network for Transferable Active Learning on Graphs”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., pp. 10174–10185 (cit. on p. 4).
- Hua, Jiayi and Haoyu Wang (2021). “On the Effectiveness of Deep Vulnerability Detectors to Simple Stupid Bug Detection”. In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 530–534. DOI: 10.1109/MSR52588.2021.00068 (cit. on p. 21).

- Hua, Wei and Guangzhong Liu (2022). “Transformer-based networks over tree structures for code classification”. In: *Applied Intelligence*, pp. 1–15 (cit. on p. 22).
- Huber, Wolfgang, Vincent J Carey, Li Long, Seth Falcon, and Robert Gentleman (2007). “Graphs in molecular biology”. In: *BMC bioinformatics* 8.6, pp. 1–14. DOI: 10.1186/1471-2105-8-S6-S8 (cit. on p. 9).
- Jain, Lokesh, Rahul Katarya, and Shelly Sachdeva (Apr. 2023). “Opinion Leaders for Information Diffusion Using Graph Neural Network in Online Social Networks”. In: *ACM Trans. Web* 17.2. ISSN: 1559-1131. DOI: 10.1145/3580516 (cit. on p. 3).
- Jakubik, Johannes, Michael Vössing, Niklas Kühn, Jannis Walk, and Gerhard Satzger (2024). “Data-centric artificial intelligence”. In: *Business & Information Systems Engineering*, pp. 1–9 (cit. on pp. 10, 11).
- Lachi, Veronica, Giovanna Maria Dimitri, Alessandro Di Stefano, Pietro Liò, Monica Bianchini, and Chiara Mocenni (2023). “Impact of the Covid 19 outbreaks on the italian twitter vaccination debat: a network based analysis”. In: *arXiv preprint arXiv:2306.02838* (cit. on p. 9).
- Li, Zhen, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen (2021). “SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities”. In: *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1. DOI: 10.1109/TDSC.2021.3051525 (cit. on p. 21).
- Lin, Hezheng, Xing Cheng, Xiangyu Wu, and Dong Shen (2022). “CAT: Cross Attention in Vision Transformer”. In: *2022 IEEE International Conference on Multimedia and Expo (ICME)*, pp. 1–6. DOI: 10.1109/ICME52920.2022.9859720 (cit. on p. 3).
- Longa, Antonio, Giulia Cencetti, Sune Lehmann, Andrea Passerini, and Bruno Lepri (2024). “Generating fine-grained surrogate temporal networks”. In: *Communications Physics* 7.1, p. 22 (cit. on p. 9).
- Longa, Antonio, Giulia Cencetti, Bruno Lepri, and Andrea Passerini (2022). “An efficient procedure for mining egocentric temporal motifs”. In: *Data Mining and Knowledge Discovery*, pp. 1–24 (cit. on p. 9).
- Mauro, Giovanni, Massimiliano Luca, Antonio Longa, Bruno Lepri, and Luca Pappalardo (2022). “Generating mobility networks with generative adversarial networks”. In: *EPJ data science* 11.1, p. 58 (cit. on p. 9).
- Miller, Kevin, Jack Mauro, Jason Setiadi, Xoaquin Baca, Zhan Shi, Jeff Calder, and Andrea L. Bertozzi (2022). “Graph-based active learning for semi-supervised classification of SAR data”. In: *Algorithms for Synthetic Aperture Radar Imagery XXIX*. Ed. by Edmund Zelnio and Frederick D. Garber. Vol. 12095. International Society for Optics and Photonics. SPIE, p. 120950C. DOI: 10.1117/12.2618847 (cit. on p. 4).
- Min, Shengjie, Zhan Gao, Jing Peng, Liang Wang, Ke Qin, and Bo Fang (2021). “STGSN — A Spatial–Temporal Graph Neural Network framework for time-evolving social networks”. In: *Knowledge-Based Systems* 214, p. 106746. ISSN: 0950-7051. DOI: <https://doi.org/10.1016/j.knosys.2021.106746> (cit. on p. 3).

- Mou, Lili, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin (2016). “Convolutional Neural Networks over Tree Structures for Programming Language Processing”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI’16. Phoenix, Arizona: AAAI Press, pp. 1287–1293 (cit. on p. 22).
- Nguyen, Anna, Antonio Longa, Massimiliano Luca, Joe Kaul, and Gabriel Lopez (2022). “Emotion Analysis Using Multilayered Networks for Graphical Representation of Tweets”. In: *IEEE Access* 10, pp. 99467–99478 (cit. on p. 9).
- Niesler, T.R. and P.C. Woodland (1996). “A variable-length category-based n-gram language model”. In: *1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*. Vol. 1, 164–167 vol. 1. DOI: 10.1109/ICASSP.1996.540316 (cit. on p. 19).
- Peng, Han, Ge Li, Wenhan Wang, YunFei Zhao, and Zhi Jin (2021). “Integrating Tree Path in Transformer for Code Representation”. In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan. Vol. 34. Curran Associates, Inc., pp. 9343–9354 (cit. on p. 3).
- Rasmussen, Carl Edward and Christopher K. I. Williams (2005). *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press. ISBN: 026218253X (cit. on p. 29).
- Ren, Mu, Ziwei Fan, Jianjun Wu, Li Zhou, and Zhiping Du (2019). “Design and Optimization of Underground Logistics Transportation Networks”. In: *IEEE Access* 7, pp. 83384–83395. DOI: 10.1109/ACCESS.2019.2924438 (cit. on p. 3).
- Roy, Deboleena, Priyadarshini Panda, and Kaushik Roy (2020). “Tree-CNN: A hierarchical Deep Convolutional Neural Network for incremental learning”. In: *Neural Networks* 121, pp. 148–160. ISSN: 0893-6080. DOI: <https://doi.org/10.1016/j.neunet.2019.09.010> (cit. on p. 3).
- Sambasivan, Nithya, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, and Lora M Aroyo (2021). ““Everyone wants to do the model work, not the data work”: Data Cascades in High-Stakes AI”. In: CHI ’21. <conf-loc>, <city>Yokohama</city>, <country>Japan</country>, </conf-loc>: Association for Computing Machinery. ISBN: 9781450380966. DOI: 10.1145/3411764.3445518 (cit. on p. 11).
- Samoa, Hazem Peter, Firas Bayram, Pasquale Salza, and Philipp Leitner (2022). “A systematic mapping study of source code representation for deep learning in software engineering”. In: *IET Software* 16.4, pp. 351–385. DOI: <https://doi.org/10.1049/sfw2.12064>. eprint: <https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/sfw2.12064> (cit. on pp. 18, 21).
- Samoa, Hazem Peter, Antonio Longa, Mazen Mohamad, Morteza Haghiri Chehrehgani, and Philipp Leitner (2022). “TEP-GNN: Accurate Execution Time Prediction of Functional Tests Using Graph Neural Networks”. In: *Product-Focused Software Process Improvement*. Ed. by Davide Taibi, Marco Kuhrmann, Tommi Mikkonen, Jil Klünder, and Pekka Abrahamsson. Cham: Springer International Publishing, pp. 464–479. ISBN: 978-3-031-21388-5 (cit. on pp. 11, 24).
- Samoa, Peter (2023 2023). “Data-Centric AI for Software Performance Engineering - Predicting Workload Dependent and Independent Performance of Software

- Systems Using Machine Learning Based Approaches”. English. PhD thesis, p. 57. ISBN: 9798377684701. URL: <http://proxy.lib.chalmers.se/login?url=https://www.proquest.com/dissertations-theses/data-centric-ai-software-performance-engineering/docview/2800163992/se-2> (cit. on p. 4).
- Samoa, Peter, Linus Aronsson, Morteza Haghiri Chehreghani, Philipp Leitner, and Morteza Haghiri Chehreghani (2023). “Batch Mode Deep Active Learning for Regression on Graph Data”. In: *2023 IEEE International Conference on Big Data (BigData)*, pp. 5904–5913. DOI: 10.1109/BigData59044.2023.10386685 (cit. on p. 27).
- Samoa, Peter, Linus Aronsson, Antonio Longa, Philipp Leitner, and Morteza Haghiri Chehreghani (2023). *A Unified Active Learning Framework for Annotating Graph Data with Application to Software Source Code Performance Prediction*. arXiv: 2304.13032 (cit. on p. 27).
- Samoa, Peter, Mehrdad Farahani, Antonio Longa, Philipp Leitner, and Morteza Haghiri Chehreghani (2024). *Analysing the Behaviour of Tree-Based Neural Networks in Regression Tasks*. arXiv: 2406.11437 (cit. on p. 21).
- Sener, Ozan and Silvio Savarese (2018). *Active Learning for Convolutional Neural Networks: A Core-Set Approach*. arXiv: 1708.00489 [stat.ML]. URL: <https://arxiv.org/abs/1708.00489> (cit. on p. 29).
- Settles, Burr (2009). “Active learning literature survey”. In: (cit. on pp. 4, 13).
- Sironi, Chiara Federica (Nov. 2019). “Monte-Carlo Tree Search for Artificial General Intelligence in Games”. English. PhD thesis. Maastricht University. ISBN: 9789463805537. DOI: 10.26481/dis.20191113cs (cit. on p. 3).
- Suissa, Omri, Maayan Zhitomirsky-Geffet, and Avshalom Elmalech (2023). “Question answering with deep neural networks for semi-structured heterogeneous genealogical knowledge graphs”. In: *Semantic Web 14.2*, pp. 209–237 (cit. on p. 3).
- Sun, Zeyu, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang (Apr. 2020). “TreeGen: A Tree-Based Transformer Architecture for Code Generation”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.05, pp. 8984–8991. DOI: 10.1609/aaai.v34i05.6430 (cit. on p. 3).
- Talak, Rajat, Siyi Hu, Lisa Peng, and Luca Carlone (2021). “Neural Trees for Learning on Graphs”. In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan. Vol. 34. Curran Associates, Inc., pp. 26395–26408 (cit. on p. 3).
- Teller, Virginia (Dec. 2000). “Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition”. In: *Computational Linguistics* 26.4, pp. 638–641. ISSN: 0891-2017. DOI: 10.1162/089120100750105975. eprint: <https://direct.mit.edu/coli/article-pdf/26/4/638/1797597/089120100750105975.pdf> (cit. on p. 19).
- Thangaramya, K., G. Logeswari, G. Sudhakaran, R. Aadharsh, S. Bhuvaneshwar, R. Dheepakraaj, and Parasu Sunny (2024). “Predicting Optimal Moves in Chess Board Using Artificial Intelligence”. In: *Cognitive Analytics and Reinforcement Learning*. John Wiley and Sons, Ltd. Chap. 4, pp. 73–101. ISBN: 9781394214068.

- DOI: <https://doi.org/10.1002/9781394214068.ch4>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781394214068.ch4> (cit. on p. 3).
- Wang, Qunbo, Wenjun Wu, Yongchi Zhao, and Yuzhang Zhuang (2021). “Graph active learning for GCN-based zero-shot classification”. In: *Neurocomputing* 435, pp. 15–25. ISSN: 0925-2312. DOI: <https://doi.org/10.1016/j.neucom.2020.12.127> (cit. on p. 4).
- Wei, Hui-Hui and Ming Li (2017). “Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code”. In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. IJCAI’17. Melbourne, Australia: AAAI Press, pp. 3034–3040. ISBN: 9780999241103 (cit. on p. 22).
- Weisfeiler, Boris and Andrei Leman (1968). “The reduction of a graph to canonical form and the algebra which appears therein”. In: *NTI, Series 2.9*, pp. 12–16 (cit. on p. 12).
- White, M., M. Tufano, C. Vendome, and D. Poshyvanyk (2016). “Deep learning code fragments for code clone detection”. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (cit. on p. 22).
- Whitehouse, Logan S., Dylan Ray, and Daniel R. Schrider (2024). “Tree sequences as a general-purpose tool for population genetic inference”. In: *bioRxiv*. DOI: 10.1101/2024.02.20.581288. eprint: <https://www.biorxiv.org/content/early/2024/02/21/2024.02.20.581288.full.pdf> (cit. on p. 3).
- Wu, Dongrui (2019). “Pool-Based Sequential Active Learning for Regression”. In: *IEEE Transactions on Neural Networks and Learning Systems* 30.5, pp. 1348–1359. DOI: 10.1109/TNNLS.2018.2868649 (cit. on p. 4).
- Wu, Yuexin, Yichong Xu, Aarti Singh, Artur Dubrawski, and Yiming Yang (2020). *Active Learning Graph Neural Networks via Node Feature Propagation* (cit. on p. 4).
- Yamaguchi, Fabian, Nico Golde, Daniel Arp, and Konrad Rieck (2014). “Modeling and Discovering Vulnerabilities with Code Property Graphs”. In: *2014 IEEE Symposium on Security and Privacy*, pp. 590–604. DOI: 10.1109/SP.2014.44 (cit. on pp. 18, 20).
- Ye, Xian-bin, Quanlong Guan, Weiqi Luo, Liangda Fang, Zhao-Rong Lai, and Jun Wang (2022). “Molecular substructure graph attention network for molecular property identification in drug discovery”. In: *Pattern Recognition* 128, p. 108659. ISSN: 0031-3203. DOI: <https://doi.org/10.1016/j.patcog.2022.108659> (cit. on p. 3).
- Zha, Daochen, Zaid Pervaiz Bhat, Kwei-Herng Lai, Fan Yang, Zhimeng Jiang, Shaochen Zhong, and Xia Hu (2023). *Data-centric Artificial Intelligence: A Survey*. arXiv: 2303.10158 (cit. on p. 11).
- Zha, Daochen, Kwei-Herng Lai, Qiaoyu Tan, Sirui Ding, Na Zou, and Xia Ben Hu (2022). “Towards Automated Imbalanced Learning with Deep Hierarchical Reinforcement Learning”. In: *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. CIKM ’22. Atlanta, GA, USA:

- Association for Computing Machinery, pp. 2476–2485. ISBN: 9781450392365. DOI: 10.1145/3511808.3557474 (cit. on p. 11).
- Zhang, J., X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu (2019). “A Novel Neural Source Code Representation Based on Abstract Syntax Tree”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794. DOI: 10.1109/ICSE.2019.00086 (cit. on p. 22).

Part II

Appended papers

Paper 1

**A systematic mapping study of source code representation
for deep learning in software engineering**

Peter Samoaa, Firas Bayram, Pasquale Salza, Philipp Leitner

Journal of IET Software, 2022

IET Software

Special issue Call for Papers

**Be Seen. Be Cited.
Submit your work to a new
IET special issue**

**"Smart Blockchain for
Software Systems: Trust,
Security and Privacy"**

**Guest Editors: Gaurav
Dhiman, Atulya K. Nagar,
Wattana Viriyasitavat and
Seifedine Kadry**


Read more



The Institution of
Engineering and Technology

REVIEW

A systematic mapping study of source code representation for deep learning in software engineering

Hazem Peter Samoaa¹  | Firas Bayram² | Pasquale Salza³ | Philipp Leitner¹

¹Software Engineering and Interaction Design Division, Chalmers | University of Gothenburg, Gothenburg, Sweden

²Department of Mathematics and Computer Science, Karlstad University, Karlstad, Sweden

³Software Evolution & Architecture Lab, University of Zurich, Zurich, Switzerland

Correspondence

Hazem Peter Samoaa, Software Engineering and Interaction Design Division, Chalmers | University of Gothenburg, Lindholmsplatsen 1, Kuggen building, Room 22, Floor 3, 417 56, Gothenburg, Sweden.
Email: samoaa@chalmers.se

Funding information

Melise – Machine Learning Assisted Software Development, Grant/Award Number: Swiss National Science Foundation/SNSF 20; AIDA – A Holistic AI-driven Networking and Processing Framework for Industrial IoT, Grant/Award Number: Knowledge Foundation of Sweden/Rek:2020006; Developer-Targeted Performance Engineering for Immersed Release and Software Engineers, Grant/Award Number: Swedish Research Council VR/2018-04127

Abstract

The usage of deep learning (DL) approaches for software engineering has attracted much attention, particularly in source code modelling and analysis. However, in order to use DL, source code needs to be formatted to fit the expected input form of DL models. This problem is known as source code representation. Source code can be represented via different approaches, most importantly, the tree-based, token-based, and graph-based approaches. We use a systematic mapping study to investigate in detail the representation approaches adopted in 103 studies that use DL in the context of software engineering. Thus, studies are collected from 2014 to 2021 from 14 different journals and 27 conferences. We show that each way of representing source code can provide a different, yet orthogonal view of the same source code. Thus, different software engineering tasks might require different (combinations of) code representation approaches, depending on the nature and complexity of the task. Particularly, we show that it is crucial to define whether the DL approach requires lexical, syntactical, or semantic code information. Our analysis shows that a wide range of different representations and combinations of representations (hybrid representations) are used to solve a wide range of common software engineering problems. However, we also observe that current research does not generally attempt to transfer existing representations or models to other studies even though there are other contexts in which these representations and models may also be useful. We believe that there is potential for more reuse and the application of transfer learning when applying DL to software engineering tasks.

1 | INTRODUCTION

Machine learning (ML), and nowadays deep learning (DL), is increasingly used by software engineering (SE) researchers and practitioners for a wide range of tasks. Examples include source code classification [1–3], code clone detection [4–6], bug detection [7–9], or code summarisation [10–12]. The current interest in DL is enabled by the wide availability of large-scale data (e.g., through open-source systems hosted on platforms, such as GitHub). Particularly, DL is interesting to researchers as it promises good results (e.g., highly accurate code clone detection) without the need for cumbersome (and often limiting) explicit feature extraction process from the raw data as it is required by traditional machine learning models [13].

In classical machine learning approaches, a considerable amount of effort goes to the design of proper ways to capture the structure of the data, that is, feature engineering, which is a “human” effort in most of the cases. This is the reason why, in the last decade, attention in machine learning is moving to ‘representation learning’, which consists of automatically extracting or learning features without the need of human feature engineering. In representation learning, feature engineering and selection phases are taken away and replaced with deep learning neural networks. DL models are composed of multiple layers to learn data representations with multiple higher levels of abstraction [14]. The networks are supposed to learn the data representation automatically, simulating the human brain for learning and analysis. Moreover, neural networks

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs License, which permits use and distribution in any medium, provided the original work is properly cited, the use is non-commercial and no modifications or adaptations are made.

© 2022 The Authors. *IET Software* published by John Wiley & Sons Ltd on behalf of The Institution of Engineering and Technology.

can be used to learn a representation of input data, such as program source code.

However, using a DL model does not entirely free researchers from all preparatory work. In order to use these techniques, appropriate features first need first to be extracted from the program source code and represented in a way the DL model can understand. This process is known as 'code representation'. Code representation is the process of transforming the textual program source code into a generic input format acceptable to the DL model [15]. Researchers can make use of different representation approaches, depending on the kind of information that needs to be extracted. Examples include token-based representation for lexical information, tree-based for extract syntactical information, and graph-based for semantic information.

No single DL and code representation approach is a silver bullet that works ideally on every case. Furthermore, in practice, choosing a suitable code representation approach is not trivial as the choice is heavily impacted not only by which DL models should be employed, but also by the requirements of the software engineering task that should be addressed. Some problems might require to focus on the semantics of the code rather than the syntax. For example, a research contribution in code summarization will require different types of information to be extracted than a clone detection approach. Currently, there is no study that has investigated which representation approaches are predominantly used for which types of problems, nor is there collective evidence regarding which approaches work better for which use cases.

In this paper, we address this gap through a systematic mapping study. We systematically collected a dataset of 103 studies published between 2014 and 2021 in 20 different conferences and journals. Our primary goal was to investigate academic studies that propose or evaluate the usage of DL and code representation to address practical software engineering tasks, such as source code classification [1] or code clone detection [5]. Our main acceptance criterion was that studies needed to (a) employ DL to address a practical software engineering task (excluding studies that use DL as a tool to conduct software engineering research, such as identifying automated code contributions [16]), and (b) explicitly discuss their code representation approach. The goal of this study is to provide an exhaustive analysis and overview on the progress achieved in using DL models in different software engineering tasks. We further discuss current best practices and elaborate on gaps in the current state of research.

We show that each way of code representation can provide a different, yet orthogonal view of the same source code. Thus, different SE tasks might require different (combinations of) code representation approaches, depending on the nature and complexity of the task. Particularly, we show that it is crucial to define whether the DL approaches require lexical, syntactical, or semantic code information. Our analysis shows that a wide range of different representations are used to tackle a wide range of common SE problems. We find that all three major types of code representation (token-, tree-, and graph-based)

are employed, but tree-based (typically based on Abstract Syntax Trees, ASTs) approaches are currently the most used. Graph-based representations are not yet common, but a growing area of research. Hybrid representations, which combine different representations approaches in a single approach, are also seeing increasing use.

Nevertheless, our results also show a lack of generalizability of the presented approaches to other tasks as well as a lack of validation based on industrial datasets. Most studies construct models for a single limited-scope task based on open-source data and rarely validate the constructed model outside of the open-source domain. Evidently, industrial datasets are not inherently superior to open-source ones. However, during our review, it became clear that virtually all analysed studies are based on open-source data, published data sets (which are often also constructed based on open-source data), or in some cases, artificial data. We argue that this limits the generalizability of the investigated studies to closed-source industrial applications and denotes a gap in the current research.

The rest of this paper is structured as follows. We present necessary background about code representation in Section 2. In Section 3, we detail the applied mapping study methodology and research questions and also provide an overview of the 103 papers that form the basis of our discussion. Afterwards, in Sections 4–8, we elaborate on the findings of the mapping study per research question. This is followed by presenting the research gaps and challenges in Section 9 and potential future directions in Section 10. Finally, we conclude the paper in Section 11.

2 | PRELIMINARIES

To contextualise the rest of this study, we now present some background about code representation. In particular, we introduce three possible forms about how source code can be represented in DL. In the literature, the code representation approaches are classified into four categories: Token-based, tree-based, graph-based, and others [17]. Every form maps different syntactical and semantic aspects of the source code to a specific data structure. These representations can then be embedded in a neural network so that they can use source code as input.

Source code is originally a text encoding representing a programme. This can be processed and further transformed into different representations forms. In this section, we describe three well-known representations, each one mapping certain aspects of the original source code. We use the C snippet depicted in Listing 1 as a running example.

Listing 1 Example of C code

```

1 void foo() {
2     int x = source();
3     if (x < MAX) {
4         int y = 2*x;

```



```

5         sink(y);
6     }
7 }

```

2.1 | Token-based representation

This representation treats code as free text. Thus, it converts the code into a list of tokens where each word (e.g., “void”) is a token, but each special character (e.g., ‘(’) is also a token (rather than considering it as part of a word). An example is given in Listing 2.

Listing 2 Token representation for the code in Listing 1

```

1  ['void', 'foo', '(', ')', '{', 'int',
  'x',
2  '=', 'source', '(', ')', ';', 'if',
  '(', 'x', '<', 'MAX', ')', '{', 'int', 'y',
  '=',
3  '2*x', ';', 'sink', '(', 'y', ')', ';',
  '}', '}']

```

Then, each token will be encoded into a vector of numbers using different statistical language models, such as word embedding [18] or n-grams [19]. In principle, word embedding is a learned representation for text where words that have the same meaning get a similar representation. Technically, word embeddings are a class of techniques where individual words are represented as real-valued vectors in a predefined vector space [20]. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network and hence, the technique is often lumped into the field of DL. As for n-grams, they are useful abstractions for modelling sequential data, such as text, where there are dependencies among the terms in a sequence. However, a corpus of code can be regarded as a sequence of sequences, and corpus-based models, such as n-grams, learn conditional probability distributions from the order of terms in a corpus. Corpus-based models can be used for many different types of tasks, such as discriminating instances of data or generating new data that are characteristic of a domain. Embeddings can be considered as a way to represent words and help the DL model to learn the representation of the source code. N-grams are several words appearing together. An embedding can be trained to represent n-grams or just individual words.

2.2 | Tree-based representation

This representation treats the abstract syntactic structure of the source code. ASTs are a kind of tree representation approach that is widely used by a programming language and SE tools.

Figure 1 shows an example of an AST representation. The nodes of the AST tree are related to constructs or symbols of

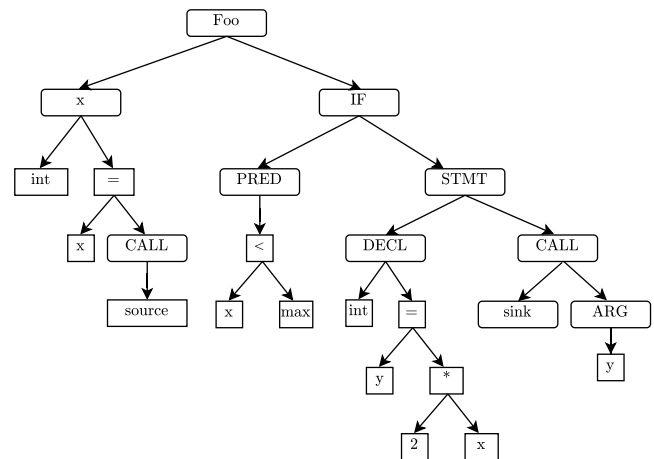


FIGURE 1 Abstract syntax tree for the code snippet in Listing 1

the source code. In comparison to the token-based approach, AST representation is abstract and does not include all available details, such as punctuation and delimiters. Theoretically, ASTs can be used to illustrate the lexical information and the syntactic structure of source code, such as the function name, and the flow of the instructions (e.g., in an if or while construct). Recently, some approaches combined neural networks and ASTs to constitute tree-based neural networks (TNNs) [21]. Given a tree, TNNs learn the vector representation by recursively computing node embeddings in a bottom-up way. Popular TNN models are the Recursive Neural Network (RvNN) [22], Tree-based Convolutional Neural Network (TBCNN) [3], and Tree-based Long Short-Term Memory (Tree-LSTM) [23].

2.3 | Graph-based representation

In this approach, source code is represented as a graph on many different levels. Levels of representation will define the type of the representation graph. Thus, a control flow graph (CFG, see Figure 2a) describes the sequence in which the instructions of a programme will be executed. Thus, the graph is determined by conditional statements, for example, if, for, and switch statements. In CFGs, nodes denote statements and conditions, and they are connected by directed edges to indicate the transfer of control.

Alternatively, the representation might be a data flow that is variable-oriented. Thus, a data flow graph (DFG) is used to follow and track the usage of the variables through the CFG. A DFG edge represents the subsequent access or modification onto the same variables. Call flow graph (CallFG) captures the relation between a statement which calls a function and the called function [24]. Finally, the entire programme can be represented as a graph using a programme dependency graph (PDG, see Figure 2b), where statements and predicate expressions can be characterised by the nodes. In this study, we differentiate between the tree- and graph-based approaches since each representation approach is used to retrieve a

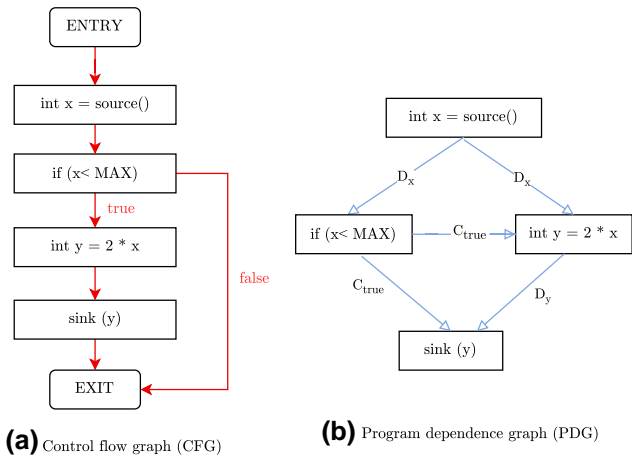


FIGURE 2 Graph-based representations for the code snippet in Listing 1

different level of information from the source code. Thus, the tree-based approach, such as using the AST, is used to extract the syntactical information from the source code [21], whereas graph-based approaches, such CFG or DFG, extract semantic information [25].

3 | RESEARCH METHODOLOGY

Our goal is to study what code representation approaches are used in combination with DL within the field of software engineering, and which code representation approaches are suitable for which tasks. Our primary method is a systematic mapping study. Systematic Mapping studies are a commonly used research method to systematically analyse a mature body of research and to derive recommendations from a disparate, large body of published works.

3.1 | Research questions

To effectively conduct a systematic mapping study, it is crucial to have well-defined research questions. The research questions analyse the main attributes of the study, which are code representation, DL, and tackled software engineering tasks, on multiple levels. In the following, we present the headlines of our research questions along with the corresponding detailed questions.

RQ 1 Main Attributes Analysis

In **RQ1**, we are primarily interested in which software engineering tasks, DL models, and code representation approaches are currently prominently investigated in the field of study.

RQ 1.1 Software Engineering Tasks: *For which software engineering tasks are DL and code representation being used?*

This research question explores the software engineering problems that are commonly tackled with DL using the code representation. This is crucial to contextualise and further analyse our subsequent findings.

RQ 1.2 DL Models: *Which DL models are being used in conjunction with code representation in software engineering research?*

While other review studies DL and SE tasks in more details, our goal is also to investigate what DL models are specifically used with a strong emphasis on code representation.

RQ 1.3 Code Representation Approaches: *Which code representation approaches are being used?*

Finally, it is evidently important to our study goal to identify the basic code representation approaches that literature currently has to offer to software engineering researchers.

RQ 2 Detailed Analysis Based on SE Tasks

Within **RQ2**, we conduct a deeper analysis of our dataset to identify which code representation approaches, on one side, and DL, on the other side, are commonly used to tackle which kinds of problems. Particularly, we are interested in identifying characteristics and commonalities of tasks that make them particularly amenable to a specific type of representation or model.

RQ 2.1 Tasks and Models: *Which DL models are being used to tackle which software engineering tasks?*

Firstly, we correlate software engineering tasks with used DL models with the goal of identifying which models are particularly suitable to solve which tasks.

RQ 2.2 Tasks and Representations: *Which software engineering tasks and representation approaches are being used?*

Secondly, we further correlate software engineering tasks with used code representation approaches.

RQ 3 Main Attributes- Cross Analysis—*Which code representation and DL models are commonly used approaches to solve a specific software engineering task?*

In **RQ3**, we perform the analysis on all three main attributes (task, representation, and model) together to map the code representation and DL models with different software tasks.

RQ 4 Hybrid Approach Analysis

In **RQ4**, we analyse the studies that combine different approaches in one framework. In the rest of this paper, we will

be referring to the overall solution presented in the retrieved studies as the 'framework'. These approaches are considered to provide valuable characteristics since they have either a wider scope to solve multiple tasks simultaneously, or more powerful capabilities in fulfilling many requirements by integrating multiple representation approaches. However, this integration between multiple approaches would increase the cost of implementing these (fairly complex) frameworks.

RQ 4.1 Hybrid Software Tasks: *What are the characteristics of frameworks that handle multiple software tasks? How are the different software tasks processed?*

We first scrutinise the studies that are set to solve different software tasks at once. The overarching aim is to elicit insights into the strategies followed to tackle multiple different tasks.

RQ 4.2 Hybrid Representation Approaches: *Which frameworks utilise multiple representation approaches? How are the different representations integrated?*

In this research question, we study research that exploits multiple representation approaches at the same time. We also examine how this integration is carried out to expand the efficiency of the framework.

RQ 5 Gaps in the Literature: *What are current research gaps and challenges in the software engineering field?*

Finally, our study raises the question which promising areas are currently underexplored, and warrant future research in the software engineering field.

3.2 | Literature search and selection

To conduct our study, we followed the process outlined in Figure 3. We used a two-step method for literature search. Firstly, we collected an initial set of candidate papers through a database search. Secondly, we used iterative backward and forward snowballing to extend this initial candidate set (the seed).

For constructing the initial candidate set, we have relied on a single primary search database (Google Scholar) rather than aggregating results from different digital libraries, such as the ACM Digital Library or IEEEXplorer. The reason for this was two-fold: (1) Google Scholar has a highly complete index, and it is unlikely that searching in other libraries would lead to additional search results, and (2) since we heavily made use of snowballing, completeness of the initial candidate set was deemed less crucial (as important missing work would appear during the snowballing process).

The initial candidate list was generated by executing the following search term on Google Scholar:

code representation for deep learning

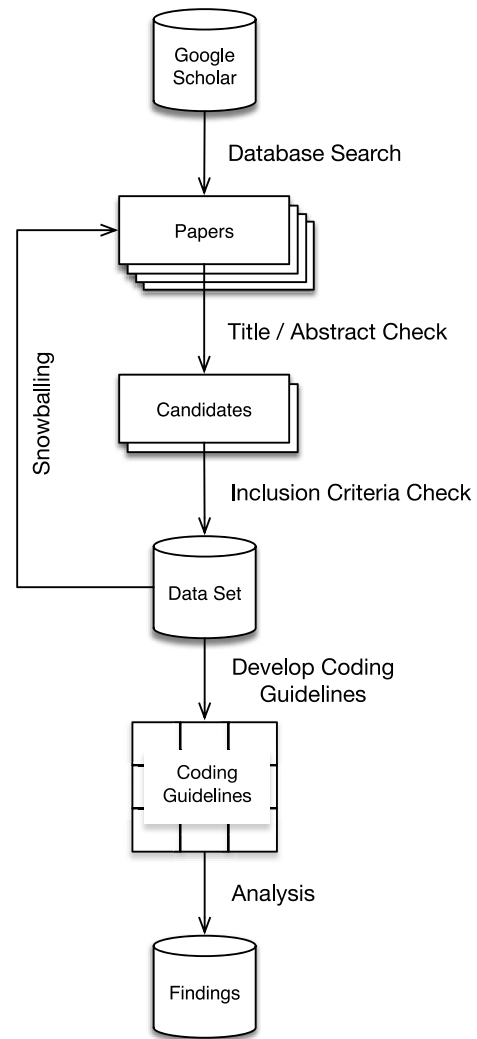


FIGURE 3 Overview of systematic mapping study process

We screened the first five pages of search results based on paper title and abstract. These potentially relevant papers were then evaluated with regard to our inclusion criteria (see Section 3.3). If a paper matched the criteria, it was added to the study dataset. After five pages (and initial snowballing), we have observed saturation, that is, investigation of the next two pages of search results did not lead to further papers matching the inclusion criteria. Hence, we stopped the search at this point.

We used explicit backward and forward snowballing to extend our initial set of candidate papers: for each selected paper, we further screened the reference list for additional relevant papers and also used Google Scholar's "cited by" functionality to discover later papers that have referenced papers in our initial set. We applied the same basic strategy to these additional candidate papers (screening based on title and abstract, followed by an explicit evaluation of inclusion criteria). This process has been repeated iteratively until no new papers could be found.

3.3 | Inclusion criteria

To clearly delineate papers that are within the scope of our study, we defined the following inclusion criteria:

- **I1:** Published in 2014 or later. We chose 2014 as a cutoff point because this was the year the TensorFlow system was initially released.
- **I2:** Making use of DL as a core contribution of the paper and explicitly reporting on the used code representation approach. To illustrate this criterion, we discuss the following study as a counterexample [26]. In this study, Laaber et al. tackled an SE task (predictability of system performance) and the authors used an artificial neural network (ANN) as a DL model for that task. However, the authors do not report on a specific code representation approach as they relied on the static features of the source code (e.g., lines of code or the number of loops). Hence, this study does not match the inclusion criterion I2.
- **I3:** Reporting on research in the wider field of software engineering. Particularly, we did not include pure DL research with no clear connection to software engineering.
- **I4:** Explicitly reporting (a) what software engineering task is being addressed, (b) what DL model is being used, (c) what code representation approach is being used, (d) what programming language(s) are being used, and (e) on what level (lines of code or functions/methods) DL is applied.

I1–I3 define the topical relevance to our study goals. I4 was important to ensure that all the data required for our study are actually reported by the papers in our dataset. We did not focus on publications in a specific venue and also accepted unpublished academic preprints if no published version of the paper exists. To be selected into the dataset, a paper had to fulfill all the four inclusion criteria.

3.4 | Resulting study dataset

Applying this literature search and selection procedure resulted in a dataset of 103 relevant studies, which are listed in Appendix A.

Figure 4 indicates the distribution of the papers in our dataset over time between 2014 and 2021. It can be observed that the number of relevant studies has increased over the years. With only two relevant publications in 2014 to reach 30 publications in 2019, then we observe a slight decline in 2020 (the last complete year in our study) with 19 publications. It is also interesting to notice the steadily increasing fraction of publications in academic journals rather than conferences or workshops.

In Figure 5, we have summarised the conference venues, which are common targets in this field of study. Conference venues with only one publication are not depicted in the figure. Unsurprisingly, ICLR, which is dedicated to presenting the advancement in representation learning, is the biggest contributor to our dataset with nine studies. It is followed by

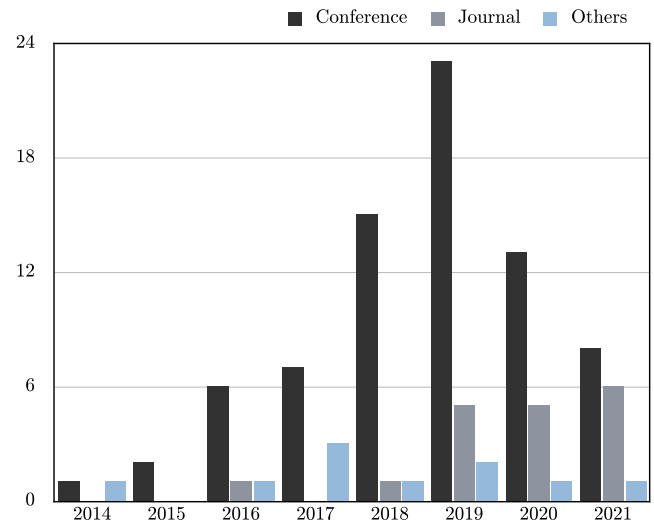


FIGURE 4 Number of publications per year. “Others” includes academic workshops and pre-prints for which no published versions exist

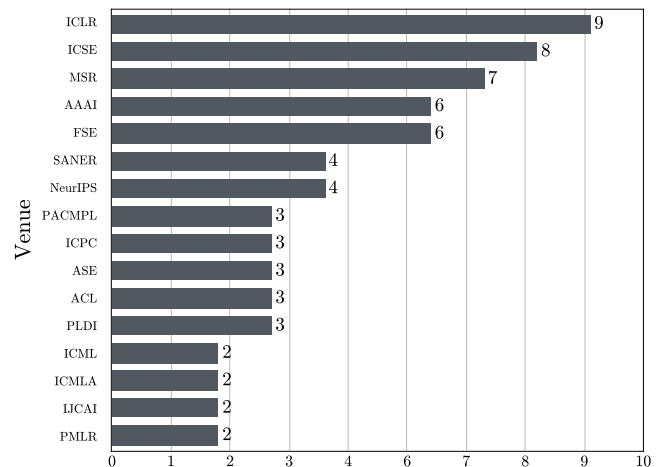


FIGURE 5 Number of publications per distinct conference venue

ICSE, which is widely seen as one of the highest ranked software engineering conferences, with eight studies, and MSR with seven studies. A smaller subset of our dataset has been published in ML venues, such as AAAI, NeurIPS, or ICML, or in programming languages venue, such as PLDI. The abbreviations of the venues presented in Figure 5 are listed in Appendix B.

3.5 | Data extraction, coding, and analysis

To analyse this dataset that answers the research questions of this study, a coding taxonomy was developed. The taxonomy is presented in Table 1. We consider the three categories (code representation approach, DL model, and Software Tasks) as primary attributes whereas we mentioned the code-level and programming languages in RQ2.3, in part related to code representation. Following our research questions, we iteratively

TABLE 1 Overview of systematic mapping study coding attributes

Code representation approach						Deep learning models		Software task	
Tree based	Graph based	Token based	Others	Programming language	Code-level granularity	Main models	Others	Task	Others
AST	CFG	Embedding	ByteCode	C	Method level	ANN	DBN	Code clone detection	Error handling
	DFG	n-grams	ASCII	C++	Statement level	RNN	NMT	Code similarity detection	Fixing format
	PDG		Code gadget	C#		LSTM	RL	Programme repair	Traceability
	CallFG		LSI	Java		CNN		Code completion	Compiler analysis
			Binary visualisation	JavaScript		GNN		Programme generation	Programme synthesise
				Python		Auto-Encoder		Vulnerability detection	Malicious behaviour detection
						Attention mechanism		Source code classification	Performance prediction
								Bug detection	Code smell detection
								Code summarisation	Type signature prediction
								Identifier generation	
								Code search	

Abbreviations: ANN, artificial neural network; AST, abstract syntax tree; CallFG, call flow graph; CFG, control flow graph; CNN, convolutional neural network; DBN, deep belief network; DFG, data flow graph; GNN, graph neural network; LSI, latent semantic indexing; LSTM, long short-term memory; NMT, neural machine translation; PDG, programme dependency graph; RL, reinforcement learning; RNN, recurrent neural network.

developed a coding guide with the following top-level codes: (1) programming language, (2) code-level granularity, (3) used code representation approach, (4) used DL model, and (5) the software task. Each publication in the dataset was coded by the first and second authors according to the taxonomy (in addition to collecting basic bibliographical information, such as the publication date and venue) with the other authors serving as sounding board and helping to resolve possible ambiguities. The resulting data were then analysed and plotted using Python scripts. We make the final coding sheet as well as the analysis script available in a replication package [27].

The detailed coding taxonomy is sketched in Table 1 and discussed in the following.

Programming Language: while DL is in principle not dependent on a specific programming language, concrete feature extraction techniques for code representation need to be built custom for individual programming languages. In our study, C, C++, C#, Java, JavaScript, and Python have emerged as target programming languages.

Code-Level Granularity: programme code can fundamentally be represented on different levels in a code representation approach. In our study, we distinguish between approaches that consider methods, functions, or similar as atomic unit [5, 28], from those that attempt to represent the programme code on a statement level [29, 30].

Code Representation: as the main target of this research, different code representation approaches were distinguished

on a fine-grained level. We distinguish between token-based, tree-based, graph-based, and other approaches. For token-based approaches, word embedding and n-grams [31] have emerged as clearly distinct groups. The only tree-based approaches [32] in our dataset are based on abstract syntax tree (AST). For graph-based approaches [33], we distinguish between CFG-, DFG-, PDG, and CallFG-based approaches, which capture the relation between a statement that calls a function and the called function [24]. Other code representation approaches that do not fall clearly into these groups are bytecode, ASCII, code gadget, latent semantic indexing, and binary visualisation since each approach has appeared only once in the retrieved list of papers. More examples for each approach will be mentioned as part of the discussion of results in Section 8.2.

DL Models: the main DL models that emerged in our coding as common methods in software engineering research are ANN [34], Convolutional Neural Network (CNN) [35], Recurrent Neural Network (RNN) [36], Graph Neural Network (GNN) [37], Long-Short Term Memory (LSTM) [38], and autoencoder and attention mechanism [39]. Additionally, three further models [deep belief network (DBN), neural machine translation (NMT), and deep reinforcement learning (RL)] emerged in two, four, and one publications, respectively, and we combine those in the group 'Others'. It is worth mentioning that we distinguished LSTM from RNN and was listed as a separate type (and not counted when referring to

RNN) since there are frameworks that combine AST with LSTM, which is referred as tree LSTM [23], and other frameworks that combine AST with RNN, which is referred as RvNN [22].

Software Engineering Tasks: to identify for which projects' code representation gets used, we also extracted the one or multiple software engineering tasks from the papers in the dataset. We observed that many common fields of study within software engineering were present. Particularly, we observed works related to code clone detection, code similarity detection [4], programme repair, programme generation [40], vulnerability detection [41], source code classification [1], bug detection [42], code summarisation [43], identifier generation [44], and code search [45]. Other tasks that emerged, but were investigated less frequently, were related to fixing formatting [46], traceability [47], compiler analysis [24], programme synthesis [10], malicious behaviour detection [48], performance prediction [49], code smell detection [50], and error handling [51].

3.6 | Data validation

To conduct a preliminary validation of the completeness of our data set, we selected five recent studies from high-profile software engineering venues that applied machine learning to one of the tasks in our study (see Table 1). We checked each reference cited by these recent studies against our inclusion criteria and validated for each study matching our criteria, whether they were indeed contained in our study set. No publications have been found to be missing.

3.7 | Threats to validity

Despite following a well-defined methodology, a review study such as ours is always subject to limitations and threats to validity. We use the classification proposed by Ampatzoglou et al. [52] to contextualise these threats.

- **Construction of the Search Process and Generalisability:** We chose to construct our dataset based on an initial search on Google Scholar followed by extensive snowballing, rather than a more conventional search strategy using major digital libraries, such as Scopus, IEEE Xplore, ScienceDirect, or the ACM Digital Library. We argue that relying on snowballing leads to a more complete and comprehensive dataset than traditional search, which suffers from limitations due to inconsistent naming and

terminology. However, one challenge is that it is hard to conduct an identical replication of our study since Google Scholar personalises search results. To mitigate this threat, we provide a replication package that includes all studied manuscripts as well as our resulting coding sheet.

- **Study Inclusion/Exclusion Bias:** DL is a rapidly growing area of research within software engineering. Hence, we needed to make decision when to stop accepting newly appearing papers into our dataset. While we do not believe that the overall findings would have been impacted if we had collected studies for a longer period of time, readers should still take our data collection period in mind when interpreting our results.
- **Validity of Primary Studies:** Four studies in our dataset are pre-prints retrieved from arXiv. While those are not peer-reviewed, the included studies are highly cited and highly influential in our field. Hence, we consider it important to include them in the analysis despite the threat that is introduced by the lack of peer review.
- **Data Extraction Bias:** While many of our coding dimensions lend themselves to objective categorisation, judgement calls still needed to be made in some cases. In these cases, we discussed among the author group to reach a consensus decision.

4 | AN OVERVIEW OF USAGE OF DEEP LEARNING IN SE TASKS

This section allows us to establish a general “process” overview of the steps required to make DL work in software engineering. While it is not expected that this general framework will differ drastically from DL in other domains, it will allow us to put the rest of the survey in context, identify the place of code representation in this general process, and serve as a guiding rail for novices to the domain. Thus, we provide a general framework of code representation and DL models' usage for tasks in software engineering based on the reviewed studies. This model has emerged from qualitatively investigating the DL models of the studies in our dataset.

4.1 | High-level process

The resulting model is depicted in Figure 6. Unsurprisingly, the high-level architecture is comparable to the usage of DL in other domains and consists of the well-known phases of data collection, data preparation and preprocessing, as well as learning and validation.

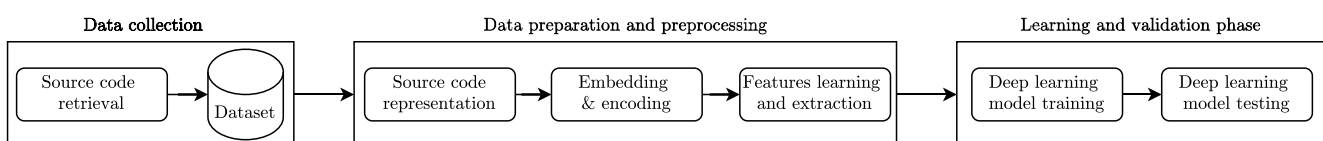


FIGURE 6 Abstracted general code representation and DL models in software engineering

Data Collection: The process starts with data collection, which in the domain of software engineering typically entails collecting the source code files for a specific programming language (e.g., through repository mining). Subsequently, the dataset needs to be annotated to serve as a training set. The annotation process is custom to the specific software engineering task that is intended to be tackled, for example, bug prediction evidently requires different annotations than, for example, code clone detection. The dataset is either ready and pre-annotated by domain experts or the researchers that conduct the study annotate the source code themselves. Annotations are task-specific and may for example, include information about the presence of bugs, or if the two code files are to be considered code clones [53].

Data Preparation and Preprocessing: Afterwards, in the data preparation and preprocessing phase, the collected code must be represented in a form that is compatible with DL. This is where code representation, the main subject of our study, comes into play. For example, in an AST representation, the collected code is converted into a tree form; then the tree paths need to be encoded or embedded as numeric values (vectorisation) using approaches such as one-hot encoding or word embedding. On the contrary, in a graph-based representation, a variety of graph embedding techniques are used, such as Graph2vec [54], HOPE [55], SDNE [56], or Node2vec [57]. Features can now be extracted from those vectors through different approaches, such as convolutional or sequential neural networks.

Learning and Validation Phase: Finally, the DL model will be trained and validated based on the tackled software engineering task.

4.2 | Examples

To concretise this process, we now present two examples of publications that follow the framework shown in Figure 6.

Example 1 (*Zhang et al. Retrieval-based neural source code summarisation, ICSE'20*):

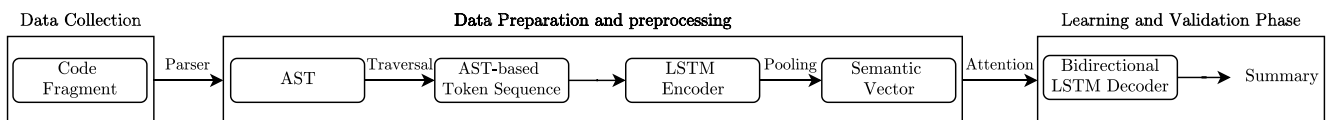


FIGURE 7 An example framework for code summarisation, based on Zhang et al. [58]. AST, abstract syntax tree; LSTM, long short-term memory

The first example [58] proposes a framework for (information retrieval) based neural source code summarisation. The solution specifically makes use of an attention encoder-decoder model. Figure 7 depicts the approach using the model introduced previously.

After collecting training data as a first step, source code is represented as ASTs, which are then turned into syntactic token sequences by tree traversal. Then, a trained encoder based on LSTM units is used to embed the code into a semantic vector using pooling, which is used to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network and preserve the most important features. Afterwards, a bidirectional LSTM decoder is used to capture the semantic context to generate natural-language summaries. The motivation behind this solution is that recent studies that use models of neural networks prefer high-frequency words in the corpus while struggling with low-frequency ones. The proposed method takes advantage of both neural and retrieval-based techniques to alleviate this problem.

Projecting Figure 7 on the main representative Figure 6, the code fragment part maps the data collection from AST to semantic vector is mapped to data preparation and preprocessing. The attention part, along with the bidirectional-LSTM decoder, presents the learning and validation phase.

Example 2 (*Wang et al. Detecting code clones with graph neural network and flow-augmented abstract syntax tree, SANER'20*):

A second example [59] uses code representation and DL for code clone detection. In this work, and as shown in Figure 8, the authors treat the AST as a graph by following a flow-augmented abstract syntax tree (FA-AST) to build a graph representation for code fragments. This is done by adding edges representing control and data flow to the AST. Graph representation is applied here as AST-based approaches cannot fully leverage the structural information of code fragments, especially semantic information, such as the control and data flow. After representing the AST as a graph, the vectors of

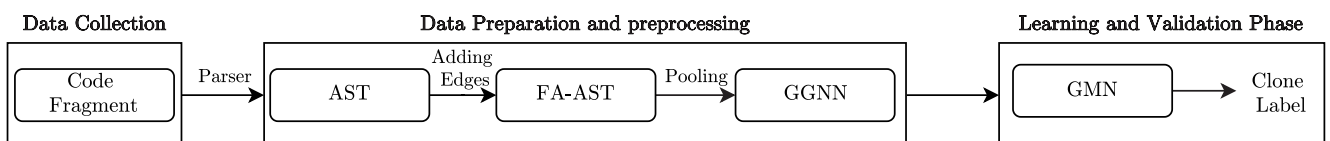


FIGURE 8 An example framework for code clone detection based on Wang et al. [59]. AST, abstract syntax tree; FA-AST, flow-augmented abstract syntax tree; GGNN, gated graph neural network; GMN, graph matching network

nodes are pooled into a graph-level vector representation. Hence, two different types of graph neural networks (GNN) are used: a gated graph neural network (GGNN) for graph embedding and a graph matching network (GMN), which can jointly learn embedding for a pair of graphs.

When mapping the approach explained in Figure 8 to the common architecture in Figure 6, the code fragment is part of data collecting, while going from AST to GGNN represents data preparation and preprocessing. Finally, the GMN is part of the learning and validation phase.

5 | MAIN ATTRIBUTES ANALYSIS

In this section, we will answer RQ1 by exploring this study's three main attributes in isolation. We answer the question about which software engineering tasks are tackled by the studies in our dataset, and what code representation and DL approaches are being used to do so.

5.1 | Software engineering tasks

DL is used for a large variety of different tasks in software engineering. Hence, to answer RQ1.1, we cluster the tasks into four broad groups inspired by work from Microsoft¹ based on high-level techniques and goals. Groups and concrete tasks, as well as their absolute and relative frequencies in our dataset, are shown in Table 2. It should be noted that the sum of percentages does not add up to 100% as some publications tackle multiple problems simultaneously.

Code-Code: the model's input is the code, and the model's output is also a source code (e.g., complete programs or code snippets). Example of tasks clustered under code-code are clone and similarity detection, code completion, programme generation and repair. Less-frequent code-code task in our dataset (grouped as “other”) is fixing formatting, traceability, and compiler analysis. Code-code tasks are a natural fit for DL and hence a frequent target in our dataset, representing 46% of all studies. Code clone detection is the most frequent individual task, followed by the (very related) task of code similarity detection and programme repair.

Code-Text: the input of the learning model is code, whereas the output is (often natural-language) text. A canonical example of this type of task is code summarisation, where the goal is to produce natural language summaries of source code constructs. The only other code-text task we found is identifier generation, which includes suggestions of method or variable names based on code information. As a group, code-text approaches represent about 20% of the studies in our dataset. However, this is primarily due to code summarisation individually being a common area of interest in DL for software engineering (representing 15% of the studies). Identifier

TABLE 2 Number and percentage of publications classified per addressed software engineering task

Code-code	46 (44.7%)
Code clone detection	16 (15.5%)
Code similarity detection	9 (8.7%)
Programme repair	9 (8.7%)
Code completion	7 (6.8%)
Programme generation	6 (5.8%)
Other	3 (2.9%)
Code-prediction	39 (37.8%)
Bug detection	14 (13.6%)
Vulnerability detection	11 (10.7%)
Source code classification	6 (5.8%)
Performance prediction	2 (1.9%)
Type signature prediction	2 (1.9%)
Malicious behaviour detection	2 (1.9%)
Others	2 (1.9%)
Code-text	21 (20%)
Code summarisation	15 (14.6%)
Identifier generation	7 (6.8%)
Text-code	6 (5.8%)
Code search	5 (4.9%)
Programme synthesis	1 (1%)

Generation appears in 7 studies. The total count of the papers that tackle code-text is 21, as one study [60] is about both, summarisation and identifier generation.

Text-Code: this group is the opposite of the previous group, where the input is the natural language text with code output. The only two tasks in our dataset of this type are code search and programme synthesis. As for code search, it uses the query text to find the corresponding source code. This task represents about 5% of the dataset. Programme synthesis, on the other hand, takes free text descriptions of programme functions as an input and returns source code as an output. There is only one study in our dataset that tackles this task.

Code-Prediction: finally, DL can be used to predict qualities based on code, such as detecting vulnerabilities, bugs, or malicious behaviour. We also group source code classification in this category. Two studies are grouped as “other” in this group: error handling and code smell detection. As a group, code-prediction is quite prevalent, accounting for 37.8% of the studies in our dataset. Within this group, different tasks are well distributed with the most common one being bug detection (14%) followed by vulnerability detection (11%).

We present the complete mapping of papers to our taxonomy of SE tasks in Appendix C.

¹<https://www.microsoft.com/en-us/research/blog/codexglue-a-benchmark-dataset-and-open-challenge-for-code-intelligence/>

RQ 1.1 Summary *We categorise the studies in our dataset in four main groups, depending on the inputs and outputs of DL. Code-code and code prediction tasks are most prevalent in our data. Code-text and text-code studies are more limited; however, these are also 'smaller' groups with a lower number of concrete subtasks.*

5.2 | Deep learning models

We now present the DL models used in the retrieved studies, as per RQ1.2. Various DL models have been identified in software engineering research. A graphical overview is given in Figure 9. LSTM [38], which is a type of RNN, is the most used DL approach and found in 49 (48%) studies. LSTM copes with the problem of RNNs known as “vanishing gradients” by adding the mechanism of “cell states” to selectively remember, or forget, part of the information that is needed during training [61]. Attention mechanism [39] and CNN [62] are the second and third most used DL models with 35 and 28 publications, respectively. CNNs are particularly efficient since they can work in parallel on sequences and have a structure for which the output and input have a logarithmic distance in terms of layers, which is linear for RNNs and LSTMs. The use of CNNs together with an attention mechanism (specifically “self-attention”) defines the architecture of “Transformers”. Autoencoders [63] and RNNs are almost equally present. The least applied DL models in our dataset are ANN and GNN. The category ‘Other’ includes deep belief networks, neural machine translation, and reinforcement learning.

It should be noted that counts in Figure 9 add up to substantially more than the total number of studies in our dataset (103) as many papers in practice combine multiple DL models. Particularly, we observe that there are specific DL models that are commonly used together for solving specific downstream tasks, such as studies that use attention mechanisms. The attention mechanism emerged as an improvement over the encoder-decoder neural machine translation system based on encoder-decoder RNNs/LSTMs. Both encoder and decoder are stacks of LSTM/RNN units. Further,

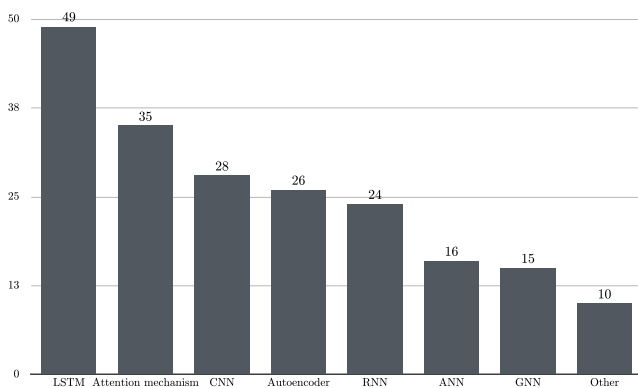


FIGURE 9 Summary of DL models used in conjunction with code representation in software engineering research

hybrid DL models are commonly used for tasks in the code-text or text-code groups as these require different models for different input and output. These issues will be discussed in more detail in Section 6.1.

RQ 1.2 Summary *Software engineering research uses a wide variety of DL models with LSTM and attention mechanisms currently receiving most attention.*

5.3 | Source code representation

We now turn towards what representations are being used in conjunction with these DL models to answer RQ1.3. We analysed the source code representation approaches that are utilised to encode source code into a form that is meaningful and can be fed into ML models. Three primary (groups of) techniques have emerged from our analysis: token-based representation, tree-based representation, and graph-based representation. Five concrete representation approaches emerged that do not clearly belong into any of these groups and have hence been categorised as ‘Other’. These are code gadget (the number of lines of code that are semantically related to each other [64]), binary visualisation (the raw representation of any type of file stored in the file system, which exhibits similar behaviours of the code while being syntactically different [65]), ASCII which used by Wang et al. [66] to convert each letter of JavaScript code into eight bit binary, latent semantic indexing (LSI, a method of analysing a set of documents in order to discover statistical co-occurrences of words that appear together which then give insights into the topics of those words and documents [47]), and bytecode (in this representation, a code fragment is expressed as a stream of bytecode mnemonic opcodes forming the compiled code [67]). An overview over the prevalence of the four groups is given in Figure 10.

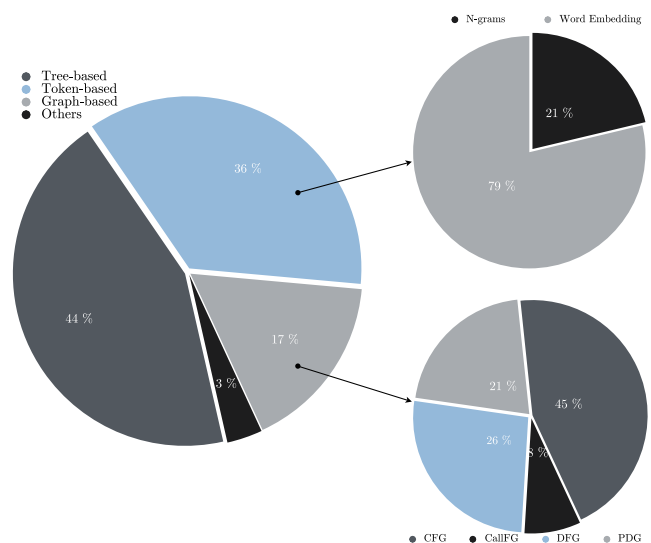


FIGURE 10 Summary of code representation approaches

All three groups see frequent use in software engineering. Tree-based and token-based representations are most common and are both utilised in over half of the studies in our dataset (66% or 64% and 54% or 52%, respectively). As before, some studies employ multiple representation approaches simultaneously. Graph-based approaches are less common and only used in 25 (24%) of studies, but the usage is increasing. The remaining techniques are only used in five individual publications.

For tree-based representation, the only specific technique that emerged from our study is AST. However, both token-based and graph-based representations can be split up into further subcategories. For token-based approaches, these are word embedding and n-grams, with word embedding being the dominant technique (used in 37% or 79% of the studies using a token-based approach, see also Figure 10).

There are a larger number of choices of graph-based representations, which are depicted in Figure 10. The most common ones are CFG (17% or 45%). Other options include PDG, DFG, and CallFG.

5.3.1 | Alternative representation approaches

In contrast, some studies have made use of code representation approaches without direct adoption of any of the methods that are categorised in Table 1. To take token-based approaches as an example, some works have tokenised the text without using word embedding or n-grams techniques. In a study by Fernandes et al. [68], the proposed framework breaks up all identifier tokens (i.e., variables, methods, classes, etc.) of the source code into sub-tokens by splitting them according to specific heuristics (*camelCase* and *pascal_case*). Gupta et al. [69] use an encoding map for each programme to map every token, based on its type (such as function, literal, variable, etc.), to a unique name in a pool of names. Similarly, there is a subset of graph-based solutions that have not used any graph-based methods that are classified in Figure 10. Yasunaga and Liang [70] have proposed a programme-feedback graph to model the reasoning process and capture the semantic correspondence involved in programme repair. Similarly, Fernandes et al. [71] extend sequence encoders with a graph neural network that can reason about long-distance relationships. Finally, Brockschmidt et al. [72] decode the code in a graph representation using GNN for partial programs to incorporate rich semantic information that is useful in programme repair tasks.

5.3.2 | Code representation depending on code-level granularity

Another question our review can answer is whether different code representation approaches are more commonly used to handle code on the statement or method levels. The results of this analysis are shown in Table 3.

TABLE 3 Main code representation approaches and code-level granularity

	Token (%)	Tree (%)	Graph (%)
Statement level	73	71	64
Method level	27	29	36

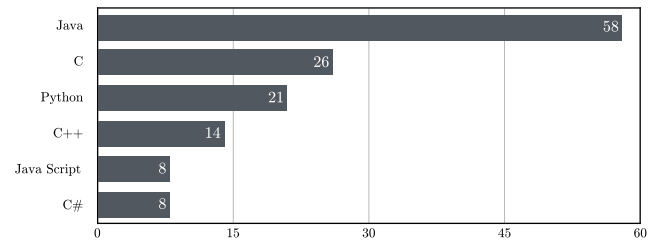


FIGURE 11 Programming languages considered in the dataset

As we can see, there is no clear-cut difference in the usage of representation approaches depending on the code level. However, token-based approaches are slightly more commonly used in studies that work with code at a statement level. This intuitively makes sense as such studies are less concerned with preserving the syntactical or semantic context of a software project.

5.3.3 | Code representation for different programming languages

As a final exploration of code representation approaches, we map which programming languages the studies in our corpus use. This is shown in Figure 11.

Unsurprisingly, Java is by far the most commonly considered programming language and is considered in over half the studies in the corpus (58 studies, or 56%). This can be explained by the wide availability of parsing tools that parse Java code into AST, which is compatible with the findings in Figure 10 that show AST to be the most common representation approach. Examples of common Java parsers are the Eclipse Java development tools (JDT) used by Büch and Andrzejak [73], SrcML [74] used by Bui et al. [4], or JavaParser used by Alon et al. [75]. However, SrcNL is a universal AST system that uses the same AST representations for multiple languages (Java, C#, C++, and C)

For graph-based representation approaches, different tooling is required. For example, Ben-Nun et al. [48] convert Java code to statements in an Intermediate Representation (IR) using the LLVM Compiler Infrastructure [76], which is then processed to contextual flow graphs. Mehrotra et al. [6] use the Soot optimization framework [77] to build program dependence graphs for Java code, followed by the Cytron's method [78] to compute control dependence. Reaching definition [79] and upward exposed analysis [80] are both used for computing data dependence graphs.

RQ 1.3 Summary *All three main groups of code representation introduced in Section 2 are used in literature with tree-based and token-based code representations being most prevalent. It is also notable that a substantial number of publications use a hybrid representation approach, combining multiple different representations.*

6 | DETAILED ANALYSIS BASED ON SOFTWARE ENGINEERING TASKS

So far, our analysis discussed the three main dimensions of the study (tasks, DL models, and code representation approaches) in isolation. Now, we turn to investigating the interplay between these dimensions as part of RQ2. Particularly, we investigate how DL models and chosen representation depend on tasks (Sections 6.1 and 6.2, respectively).

6.1 | Software tasks and DL models

In this section, we will discuss the results that explain RQ2.1, where we map the chosen DL models to tackle software engineering tasks. Figure 12 depicts a mapping of specific DL models identified in the study to the four high-level categories of tasks as a bubble plot.

We observe that a wide variety of models have been applied to the tasks in the code-code group, whereas there appears to be more dominant methods for code-text (LSTM with autoencoders and attention mechanisms) as well as code-prediction (CNN and LSTM). The data for the text-code group are too sparse to come to a clear conclusion, but initial evidence suggests that researchers also use a variety of models for this task. Further, LSTM is commonly used and proportionally distributed for all types of tasks. However, CNN is most frequently used for tasks in the group code-prediction. Both, autoencoders and attention mechanisms are used frequently for code-code and code-text tasks, but rarely for other tasks.

Figure 13 drills deeper into this and depicts the usage of different DL models for specific tasks in the code-code group. We observe that a variety of models are used for all specific tasks.

In programme repair, some approaches use sequence to sequence networks with encoder-decoder models attached with attention mechanisms. Bi-directional LSTM is mainly used in both encoder and decoder. However, attention might be used in the decoder part [40] or in encoder [70]. However, other approaches for handling programme repair do not rely on the encoder-decoder model. For example, Vasic et al. [81] use LSTM and attention mechanism to locate and handle the misuse of the variable defined in the programme. Other studies rely on sequential models for handling programme repair without using the encoder-decoder attention model [69, 82], whereas Dinella et al. [83] rely on graph neural networks for learning graph transformation to repair the bugs in the JavaScript programs.

Figure 14 presents a similar analysis for specific code-text tasks. It becomes evident that autoencoders are an important facet of contemporary code summarisation research. These approaches are based on the sequence-to-sequence paradigm over the words of some text with a sequence encoder (typically a RNN, but sometimes using self-attention [12]) processing the input and a sequence decoder generating the output. Recent successful implementations of this paradigm have substantially improved performance by focussing on the decoder, extending it with an attention mechanism over the input sequence and copying facilities [68]. However, while standard encoders (e.g., LSTMs) can in theory handle arbitrary long-distance relationships, in practice, they often fail to handle long texts (summarisation output) correctly [84].

RQ 2.1 Summary *Most of the software tasks studied are mainly tackled using the LSTM model. However, autoencoders and attention mechanisms are also widely adopted, particularly in code-code and code-text tasks. A high number of code-prediction publications utilise CNNs.*

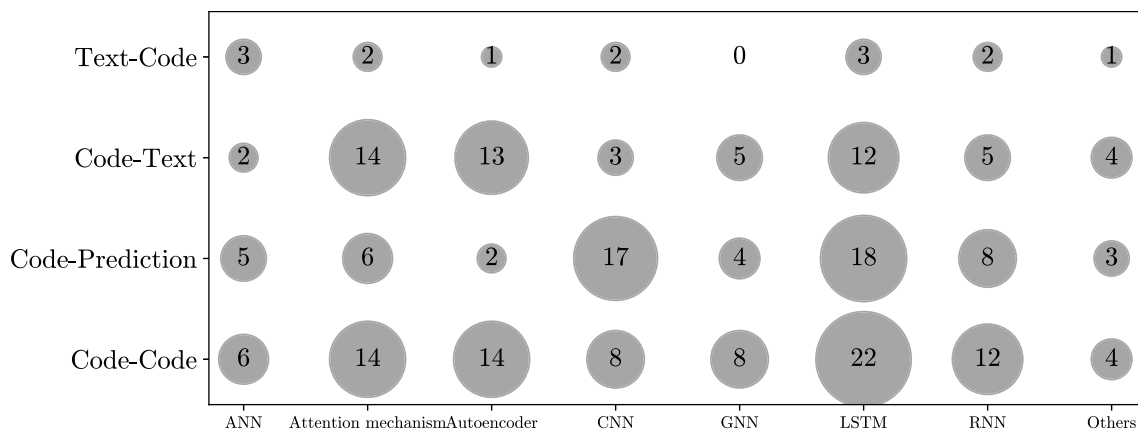


FIGURE 12 Software engineering tasks and applied DL approaches. ANN, artificial neural network; CNN, convolutional neural network; DL, deep learning; GNN, graph neural network; LSTM, long short-term memory; RNN, recurrent neural network

6.2 | Software tasks and code representation

We now turn towards RQ2.2 and explore how the choice of code representation approach is impacted by the chosen software engineering task. An overview for the four groups of tasks is provided in Figure 15.

We observe that the various code representation approaches are used across software engineering tasks. Text-code tasks are commonly addressed using token-based approaches. Only one study uses a tree-based approach for this type of task [10], and none uses a graph-based approach. However, this study handles multiple tasks within the same study. More

specifically, the authors have built multiple representations to handle tasks separately. The tree-based approach addresses code summarisation (a code-text task), whereas a token-based approach is used for code retrieval (text-code). Hence, we conclude that for text-code tasks, for example, code search, a token-based representation is the only method that is seeing current use. This can be explained as the freeform text of, for example, a query is better treated using natural language processing (NLP) techniques than the more code-specific tree- and graph-based representations.

Graph-based approaches are most commonly used in code-code tasks. However, also 38% of graph-based

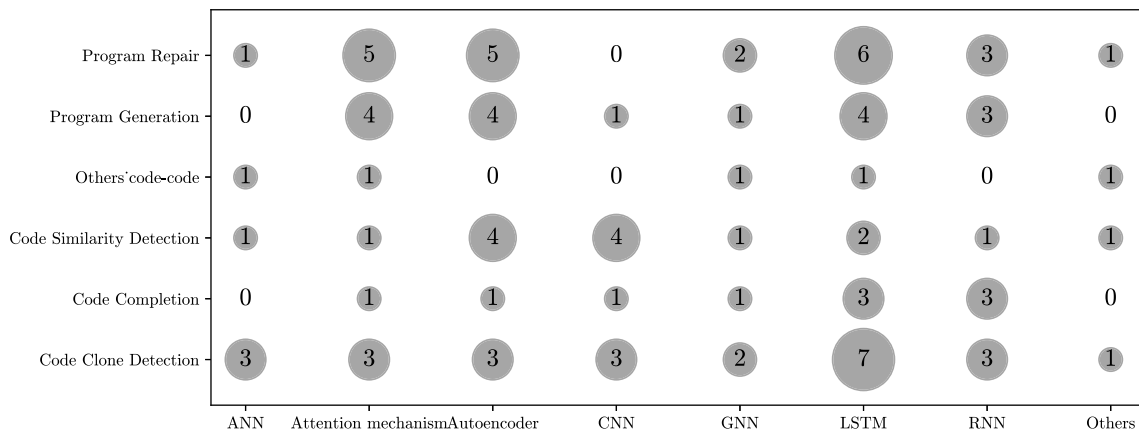


FIGURE 13 Applied DL approaches for specific code-code tasks. ANN, artificial neural network; CNN, convolutional neural network; DL, deep learning; GNN, graph neural network; LSTM, long short-term memory; RNN, recurrent neural network

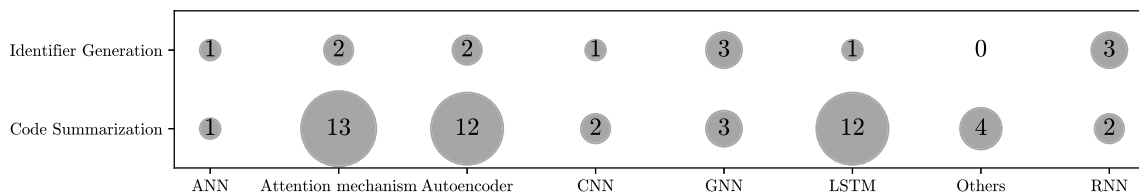


FIGURE 14 Applied DL approaches for specific code-text tasks. ANN, artificial neural network; CNN, convolutional neural network; DL, deep learning; GNN, graph neural network; LSTM, long short-term memory; RNN, recurrent neural network

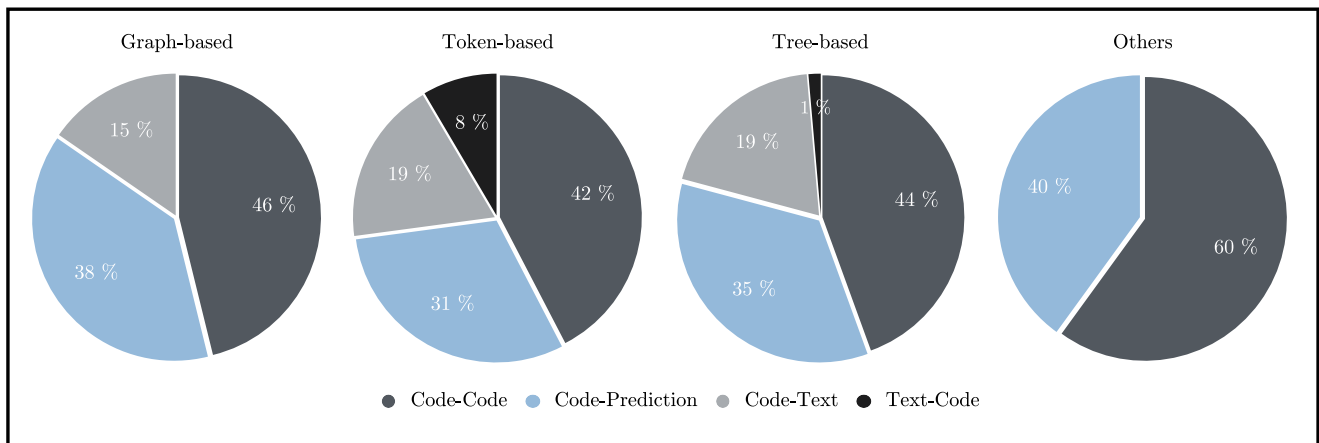


FIGURE 15 Code representation approaches per group of software engineering tasks

approaches are used for code-prediction tasks. To better understand this observation, we have again detailed further into specific tasks. In Figure 16, we present how often specific tasks in the code-code groups use a graph-based approach to represent the source code.

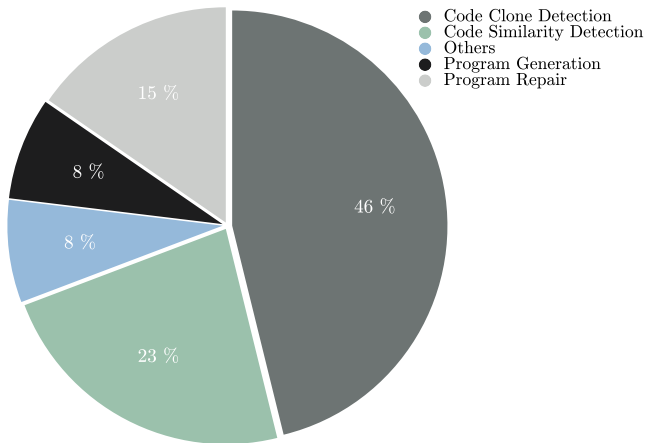


FIGURE 16 Usage of graph-based representation for specific code-code tasks

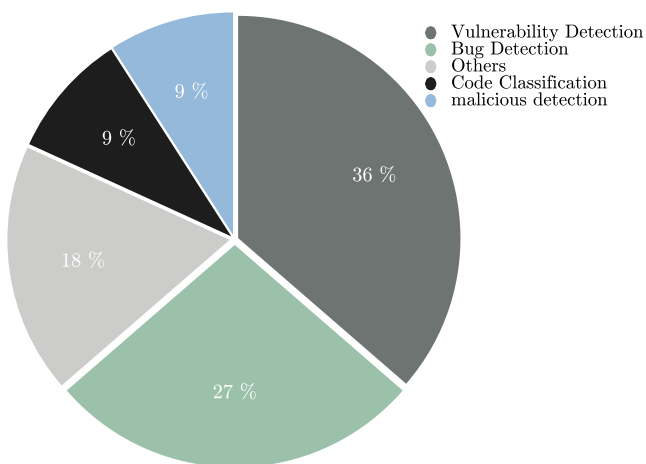


FIGURE 17 Usage of graph-based representation for specific code-prediction tasks

Both code clone and code similarity detection are proportionally overrepresented here. This is interesting, especially since these tasks have many similarities. It can be argued that a graph representation is highly appropriate for solving the problem of identifying similar code elements. By representing code snippets as a graph, those graphs are embedded into vectors (one vector for each graph). To measure the similarity, one can then simply compute the distance between those graphs. This approach is arguably more simple and effective than breaking each piece of code into tokens and then embedding each token into a vector.

We now conduct a similar analysis for the usage of graph-based representations in code-prediction tasks (Figure 17). We observe that graph-based representation approaches are commonly utilised in vulnerability and bug detection, together amounting to about two thirds of all usage of graph-based representation in code-prediction tasks. For these approaches, researchers commonly need to preserve semantic information for which graph representations are most suitable.

RQ 2.2 Summary *We were not able to identify a clear pattern that specific representations are more common for specific types of tasks with one exception: text-code tasks frequently call for token-based methods. Aside from this, software engineering researchers have tested different combinations of representations and tasks, and no clear consensus what the ideal way to address any specific task (except text-code tasks) has emerged yet.*

7 | MAIN ATTRIBUTES – CROSS ANALYSIS

We now discuss the interplay of all three dimensions of this study—tasks, DL approach, and code representation approach, answering RQ3. In the previous sections, we have separately analysed the three dimension task, DL model, and code representation approach. To answer RQ3 and get deeper insights into the current trends in the field, we now investigate all three dimensions together. Results of this analysis are summarised in Table 4.

LSTM is the most commonly used model for code-code tasks, using both tree- and token-based representations as

TABLE 4 Analysis of the main attributes

■ Code-Code ■ Code-Text ■ Text-Code ■ Code Prediction

	Tree Based				Graph Based				Token Based				Others			
ANN	4	1	0	3	3	0	0	2	4	1	3	3	1	0	0	0
CNN	5	1	1	14	1	0	0	5	6	3	2	8	1	0	0	0
RNN	8	4	0	5	3	1	0	4	9	2	2	6	1	0	0	0
LSTM	15	8	1	10	5	2	0	5	12	7	3	12	0	0	0	1
Attention Mechanism	9	8	0	4	6	3	0	1	4	8	2	3	0	0	0	0
Auto-encoder	10	9	1	1	4	3	0	0	8	7	1	0	1	0	0	1
GNN	5	4	0	3	7	4	0	3	1	1	0	1	0	0	0	0
Others	4	4	1	2	1	0	0	1	1	1	1	2	0	0	0	0

well as, to a certain extent, autoencoders. In contrast, LSTM and autoencoders are almost equally frequently used for code-text tasks. LSTM and autoencoders go hand in hand in solving sequential problems by treating the code as a sequence of tokens (using a token-based representation) or sequence of nodes (in the tree-based representation). Hence, sequential models, such as LSTM, are the most appropriate approach for such a problem. The sequential model needs to be encapsulated into an encoder-decoder model because for a code-text task, it is necessary to encode the code consistently through one model in order to generate natural language sequences from the corresponding source code. Attention mechanisms are used to dynamically select the distribution over the combined representations while decoding or encoding is selecting the relevant path in the AST [85].

Unsurprisingly, GNN is the most commonly used architecture in conjunction with a graph representation in the majority of SE tasks. In contrast, no common DL models can be identified for text-code tasks across all the representations. Instead, various different models are used across the studies in our dataset. This is because the text that represents the input in a text-code task can be treated using natural language processing (NLP) techniques, which according to literature, all DL models work properly on.

As for code-prediction tasks, CNN is the most dominant model in conjunction with a tree-based representation, while LSTM is most commonly used for a token-based representation. This difference is rooted in the different goals underlying tasks in the code-prediction group—in these tasks, the goal is not to generate code as in code-code and text-code tasks, or generating text as in code-text tasks. Much more, code-prediction tasks tend to deal with classical DL prediction problems, that is, classification and regression. For instance, bug or vulnerability detection is a binary classification problem to decide whether or not the code includes a bug or vulnerability. The same is true in performance prediction, where a specific performance value is predicted as a regression problem.

RQ 3 Summary *We analysed the retrieved frameworks from the viewpoint of the main three dimensions of our study, software task, code representation approach, and deep learning model applied. LSTM and autoencoders are the most used deep learning for code-code and code-text tasks using tree-based and token-based representations. While GNN is the most used model with graph representation with most of the SE tasks. For code-prediction, CNN with tree-based representation and LSTM with token-based representation are the most common techniques used in the studies.*

8 | ANALYSIS OF HYBRID APPROACHES

In this section, we will answer RQ4 by exploring frameworks that address either multiple SE tasks or which use multiple representations. We refer to such studies as using a hybrid approach.

8.1 | Hybrid software tasks within one framework

In this section, we address RQ4.1 and identify characteristics and main properties of frameworks that solve multiple SE tasks simultaneously. No study in our dataset is general in the sense that it is able to address all SE tasks.

8.1.1 | Solving many tasks with one framework

Bui et al. [10] propose an approach that integrates three different tasks—it tackles code similarity detection as a code-code task, code search as text-code tasks, and code summarisation as a code-text task. This study is singular in that it combines text-code tasks with other SE tasks. The proposed method is a self-supervised learning framework for source code modelling designed to mitigate the need for labelled data for different SE tasks. The key innovation here is that the source code model is trained to detect the similarity and dissimilarity across code snippets. This study also makes use of a hybrid representation approach, by merging an AST-based strategy with a token-based approach. The representation approaches are used in the encoder component of the discussed system. Hence, well-know AST-based code modelling techniques, such as Code2vec [44], TBCNN [3] are used besides token-based approaches by handling the source code as a sequence of tokens using a neural machine translation (NMT) baseline. Those techniques utilise node type and token information to initialise AST nodes. The hybrid representation approach will be discussed in more detail in Section 8.2. Throughout this approach, various encoders are used, and the choice of encoder depends on the task.

8.1.2 | Frameworks that solve two tasks

Besides the aforementioned study, we find that three other approaches tackle combinations of code-code and code-text tasks. Cvitkovic et al. [60] design a framework that solves code completion as a code-code task and identifier generation as a code-text task. They use ASTs to represent the source code. This tree is augmented with semantic information, such as data- and control-flow to eventually obtain an augmented AST as a directed multigraph. The augmented AST is then further augmented by adding a Graph Structured Cache. They add a node to the augmented AST for each token in the input instance. Then, all the nodes are vectorised to be processed than with the graph neural network.

Kang et al. [86] evaluate the generalisability of the Code2vec modelling technique by applying it along with a sequential model to address code clone detection as a code-code task as well as code summarisation as a code-text task. Then, they compare the results obtained from these techniques with a task-specific baseline. In this study, the authors do not focus on the overall effectiveness of the methods. Instead, they evaluate if the use of Code2vec can improve the performance

of the baselines. Based on their results, the authors claim that no improvements had been achieved by applying Code2vec.

Code summarisation is also investigated through one framework proposed by Wei et al. [87] that is generalised to solve programme synthesis as a text-code task. They use a token-based approach for code representation. The proposed framework consists of three main parts: a code summarisation model, a programme synthesis model, and dual constraints. The code summarisation and programme synthesis models both rely on a sequence-to-sequence neural network and an encoder-decoder attached with attention mechanism between encoding and decoder. To leverage the contextual information within the word embedding, a token-based, bi-directional LSTM is used as a unit in the encoder. Another LSTM is also used in the decoder. Dual constraints are used by adding regularisation terms in the loss function to constrain the duality between two models, which are enlightened by the probabilistic correlation and the symmetry of attention weights between code summarisation and programme synthesis models.

Finally, four studies design solutions that are transferable across code-code and code prediction tasks [21, 81–83]. Three of those proposed frameworks that tackle programme repair as code-code tasks and bug detection as code prediction tasks. These tasks are related in the sense that a bug is first detected in the code, which is subsequently fixed through programme repair. Hence, it makes sense to have one solution that addresses these tasks simultaneously. In the same context, one of those studies [83] uses a hybrid code representation approach by combining tree- and graph-based approaches. Thus, code is parsed into an AST to capture the programme's syntactic structure; then, the leaf nodes are connected with *SuccToken* edges. Additionally, the value of nodes that store the content of the leaf nodes is added with special semantic *ValueLink* edges connecting them together. Based on the study, the ultimate aim of introducing this additional set of nodes is to provide a name-independent strategy for code representation and modification. After representing the programme as a graph, a GNN is used to map the graph into a fixed dimension vector space. An LSTM is then trained to locate the bug through a sequence of graph transformations. That means that, given a buggy programme modelled by a graph structure, the proposed framework makes a sequence of predictions, including the position of bug nodes and corresponding graph edits to produce a fix.

The other related approaches [81, 82] use only a token-based approach combined with LSTM to locate and repair the bug in the programme. Moreover, the fourth study in this group [21] defines an AST-based neural network for source code representation in order to solve code-clone detection as a code-code task and code classification as a code-prediction task. This study discusses the problem of the long depth of the AST, which causes a long dependency between the sequence of nodes, leading to vanishing problems when injected into the sequential model. Thus, the tree is divided into a sequence of small statement trees. Those trees are encoded to be used with a bidirectional RNN model to leverage the naturalness of statements to achieve the tasks.

Statement trees are constructed using the preorder traversal algorithm.

It is interesting to observe that no study in our dataset proposes a framework that addresses a combination of code-text and code-prediction tasks, nor combinations of code-prediction and text-code tasks.

RQ 4.1 Summary *The integration between multiple tasks within one framework relies on the relatedness between these tasks. However, there currently appears to be no truly general framework for DL in software engineering, which could be applied independently of the tackled software tasks.*

8.2 | Hybrid representation approaches

Some studies have utilised a hybrid approach for code representation to capture more information on the source code. This is often promising as tree-based approaches capture syntactical information, graph-based approaches are better at retaining semantics, and token-based approaches preserve lexical information.

Table 5 summarises how often different types of code representation approaches are used alone or in conjunction. The diagonal elements represent the frequency of the frameworks that have used a single representation approach, while the non-diagonal elements represent the frequency of the frameworks that have used hybrid representations. Seven studies [5, 7, 21, 42, 67, 88, 89] combined representations from all three groups. The most common hybrid approach is a combination of token- and tree-based approaches, used by 25 studies, or almost a fourth of our dataset, in total (note that 18 approaches combine only tree- and token-based representation, plus the seven studies that use all three). Combinations of tree- and graph-based approaches are also fairly popular, used by 16 studies in total.

Particularly interesting are the seven studies that have used all three representation approaches in conjunction. For example, Hua and Li et al. [7, 42] present work on bug detection. The two approaches start with constructing AST representations of the source code in order to locate sensitive point-like object construction, method invocation, expression statement, conditional statement, and loop statements. Sensitive points are the syntax characteristics where most 'simple' bugs manifest. Then, Word2Vec [20], a token-based representation approach, is employed by taking all of the AST nodes of a method as the input and generating a learned vector representation for each given AST node. This vector

TABLE 5 Frequency of combinations of (types of) representation approaches

All = 7	Token	Tree	Graph
Token	25	18	4
Tree		32	9
Graph			5

representation is later used as input to the DL model. However, the local context of the method representation from AST node representations is preserved by representing each path as an ordered set of node vectors. Since the bug can be involved in multiple methods, it is crucial to capture also the global context by modelling the relations between different methods through a dependency graph (a PDG). Thus, semantic information in the source code, such as data and control flow, is traced. Then, when the graphs are generated, different embedding techniques for graphs are used on nodes, edges or the entire graph. For example, Node2Vec [90] is used to vectorise the nodes.

Similarly, other studies that use all three representation approaches are tackling code clone detection [5, 21, 67]. These studies show that using a stream of identifiers to represent the code, DL can effectively replace manual and hand-crafted feature engineering. Moreover, these works show that representation of the code at different levels of abstraction (identifiers, AST, and CFG) can provide a different, yet orthogonal, view of the same code fragment, thus enabling more reliable detection of code similarities.

Sonnekalb and Li et al. [42, 89] investigate a combination of all three main representation approaches for the task of vulnerability detection. These studies claim that there is a need to represent programs in a way that can adequately accommodate the syntax and semantic information related to vulnerabilities. This enables multiple kinds of neural networks to detect various kinds of vulnerabilities.

RQ 4.2 Summary *62 (60%) frameworks of the retrieved studies have used only one type of representation approach, while 31 (30%) studies have combined representations from two groups. Seven (7%) studies utilised representations of all three main groups in conjunction.*

9 | GAPS IN THE LITERATURE

In this section, we will discuss perceived limitations, research gaps, and challenges that we derived from the retrieved studies, addressing RQ5.

- **Lack of Topic Coverage:** Even though we have found DL to be applied to a wide variety of SE tasks, some crucial tasks appear to be underrepresented. For example, we have identified only one or two studies each tackling performance prediction, code smell detection, or traceability. This is surprising, as these tasks could profit substantially from an investment in DL. Taking performance prediction as an example, performance is often seen as a crucial non-functional property of software systems, and traditional performance engineering is challenging [91] and error-prone [92]. A deeper investment in DL in the style of some code clone detection or programme repair studies seems promising in these domains.
- **Lack of Generalisability:** According to Figure 6, DL models can be used in two phases—in the data preparation and

preprocessing phase for learning the representation of code (representation learning), and then again in the learning and validation phase to achieve the SE task. In principle, representation learning is independent of the tackled SE task. Transfer learning [93] could be used to generalise and reuse pre-trained models for representation learning to different tasks. In other application domains of DL (such as computer vision or NLP), transfer learning has led to generally useful models such as DenseNET [94] or BERT [95]. We observe a lack of such models in software engineering. However, we made the observation in this study that most of the proposed approaches are highly domain and problem-dependent. Thus, very few retrieved studies are applied to different SE tasks. Very few solutions are transferable or easily adapted to other SE problems. There are some approaches that explicitly present generalised SE representations [44, 85]. However, these approaches are for fixed code units, such as tokens, statements, or functions. They are not sufficiently flexible to generate encoding and embeddings for different units. Thus, the learned code representation may not be effective for a multitude of tasks. Two studies in our dataset already attempt to provide such a generalised representation model [4, 10]. We argue that this is an important area of future research that should be a focal point for future investigations.

- **Lack of Industrial Data:** Unsurprisingly, the vast majority of approaches in our dataset are trained and tested on open-source projects extracted from platforms, such as GitHub. However, validation of the resulting models on industrial data is rare. This is understandable especially in supervised learning model, which requires annotated datasets of considerable size. Annotations often need to be manually labelled by humans according to a specific downstream task. To address this challenge, and connecting to the previous point, recent research uses self-supervised learning [4, 10] to leverage unlabelled data to pre-train code representations which are reusable for building general models that are suitable for various downstream tasks. While the type of data that led to this challenge was not an explicit dimension that we coded for this study, it became abundantly clear during the review that virtually all analysed studies are based on open source data, published data sets (which are often also constructed based on open-source data), or in some cases artificial data.

RQ 5 Summary *We conclude that the core research gaps currently prevalent in the literature relate to a lack of coverage for some relevant SE tasks, a lack of the application of transfer learning, and a lack of validation based on industrial data.*

10 | DISCUSSION

- **Towards AST-Based Neural Networks:** As our work shows, token-based approaches are common in software engineering literature. These approaches tend to either

treat the code as a token sequence or bags of tokens, or they rely on latent semantic indexing (LSI) and latent dirichlet allocation (LDA) to represent the code. The problem of those token-based approaches is that they treat the source code as a natural language. To improve these approaches, code syntax and semantics need to be taken into account [96]. Some existing work [3, 22, 23] provide strong evidence that syntactical knowledge contributes positively and leads to better representations than traditional token-based methods. We speculate that this is the reason why ASTs are used in so many different approaches. Through the AST, researchers can easily capture lexical as well as syntactical information. Hence, many research works try to combine ASTs with deep learning, which is referred to as AST-based neural networks. These approaches combine ASTs with Recursive Neural networks (RvNN) [22], tree-based CNNs [3], or tree LSTMs [23].

- **The Limitations of Tree-based Approaches:** Despite the effectiveness of such tree-based neural network approaches in extracting both lexical and syntactical information, there are limitations. Similar to long texts in NLP, tree-based neural models are vulnerable to the gradient vanishing problem, where the gradient becomes vanishingly small during the training (especially when the tree is very large and deep, which it often is for real-life source code). Hence, traversing and encoding the entire AST tree in a bottom up way [22, 23] or using a sliding window technique [3] may lose long-term context information [21]. Another limitation of AST-based neural networks is that those approaches transform the AST or present it as full binary trees to improve simplicity and efficiency. However, this in turn destroys the original syntactic structure of the source code and makes the AST even deeper. Moreover, the transformed and deeper AST reduces the capability of neural network models to capture more real and complex semantics [21]. Finally, some SE tasks require not only syntactical, but also semantic information.
- **Towards Graph-based Code Representation:** Due to the problems of leveraging semantic information with AST-based approaches, more and more newer DL papers adopt graph-based representations, such as long-term CFG and Data Dependencies Graph (DDG). These representation approaches can overcome some of the limitations of AST-based neural networks. Examples of such works are Zhao et al. [25], who extract semantic features from the CFG of represented code, Allamanis et al. [33], who consider the long-range dependencies induced by the same variable or function in distant locations, or Tufano et al. [67], who directly construct CFGs of code fragments.
- **The Limitations of Graph-Based Approaches:** However, graph-based representation is not without challenges either. The drawback of CFGs is that they lack data flow information. Furthermore, most CFGs only contain control flows between code blocks and exclude the low-level syntactic structure within code blocks [59]. Another drawback of CFGs is that in some programming languages, CFGs are much harder to obtain than ASTs. Nevertheless, Henkel

et al. [97] show that embeddings learned from (mainly) semantic abstractions provide nearly triple the accuracy of those learned from (mainly) syntactic abstractions. Ultimately, many solution approaches choose to use a syntactic representation [75], because it was shown to be useful as a representation for modelling programming languages in machine learning models. It was also shown that they are more expressive than n-grams and manually designed features [44]. Other solutions use approaches based on semantic context [98] in which programme elements are graph nodes and semantic relations are edges in the graph. Due to the gap between syntax (e.g., tokens or ASTs) and the semantics of a procedure in a programme, the abstractions of traces obtained from symbolic execution of a programme are also used as a representation for learning word embeddings [97].

Based on the aforementioned discussion and the ongoing developments and current promising research directions, we expect a move towards more graph-based code representation as these representation models make it easier to learn semantic information. However, graph-based approaches are not without challenges, and more research in this direction will be needed.

11 | CONCLUSION

This study has presented a systematic mapping study on 103 primary studies that use code representation in the context of DL for software engineering. Our mapping study has classified the software task into four main categories depending on the input and output of the DL model (code-code, code-prediction, code-text, and text-code). Our study showed that code-code and code-prediction are the most addressed software tasks. We have also observed that tree-based and token-based approaches are the most common representation approaches applied in the investigated studies. However, we have also observed that there is a trend towards hybrid representations (which combine multiple different representation approaches) as well as the preferred usage of graph-based representations in newer studies. We identify two primary challenges in current literature: (1) there is a lack of generalisability of the presented approaches to other tasks (i.e., there are few attempts at transfer learning between tasks) and (2) very few studies validate the proposed framework on industrial datasets. We argue that these two problems constitute severe threats to the practical usefulness of current code representation research in the field of software engineering.

ACKNOWLEDGEMENTS

This work has been partially funded by the Swedish Research Council VR under grant number 2018-04127 (Developer-Targeted Performance Engineering for Immersed Release and Software Engineers), by the Knowledge Foundation of Sweden (KKS) through the Synergy Project AIDA—A Holistic AI-driven Networking and Processing Framework for Industrial

IoT (Rek:20200067), and by the Swiss National Science Foundation (SNSF) project “Melise—Machine Learning Assisted Software Development” (SNSF 204632).

CONFLICT OF INTEREST

The authors declared that they have no conflicts of interest to this work.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Zenodo at <https://doi.org/10.5281/zenodo.6466506>, reference number 6466506.

ORCID

Hazem Peter Samoaa  <https://orcid.org/0000-0001-5293-3388>

REFERENCES

- Bui, N., Jiang, L., Yu, Y.: Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In: The Workshops of the Thirty-Second AAAI Conference on Artificial Intelligence (2018)
- Kanade, A., et al.: Pre-trained contextual embedding of source code. In: ICLR 2020 Conference Program Chairs (2020)
- Mou, L., et al.: Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI '16, pp. 1287–1293. AAAI Press (2016)
- Bui, N.D.Q., Yu, Y., Jiang, L.: Infercode: self-supervised learning of code representations by predicting subtrees. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1186–1197 (2021)
- Fang, C., et al.: Functional code clone detection with syntax and semantics fusion learning. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, pp. 516–527. Association for Computing Machinery, New York, NY, USA (2020)
- Mehrotra, N., et al.: Modeling functional similarity in source code with graph-based siamese networks. *IEEE Trans. Softw. Eng.*(01), 1 (2020). <https://doi.org/10.1109/tse.2021.3105556>
- Hua, J., Wang, H.: On the effectiveness of deep vulnerability detectors to simple stupid bug detection. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 530–534 (2021)
- Li, Y., Wang, S., Nguyen, T.: Fault localization with code coverage representation learning. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 661–673 (2021)
- Shi, K., et al.: Mpt-embedding: an unsupervised representation learning of code for software defect prediction. *J. Softw.: Evol. Process.* 33(4), e2330 (2021). e2330 smr.2330. <https://doi.org/10.1002/smr.2330>
- Bui, N.D.Q., Yu, Y., Jiang, L.: Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In: SIGIR '21. Association for Computing Machinery, New York, NY, USA (2021)
- Liu, S., et al.: Retrieval-augmented generation for code summarization via hybrid GNN. In: International Conference on Learning Representations (2021)
- Zhang, J., et al.: Retrieval-based neural source code summarization. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 1385–1397 (2020)
- Jebnoun, H., et al.: Clones in deep learning code: what, where, and why? (2021). <https://doi.org/10.48550/arXiv.2107.13614>
- Bengio, Y.: Learning deep architectures for AI. Now Publishers Inc., Delft (2009)
- Keller, P., et al.: What you see is what it means! semantic representation learning of code based on visualization and transfer learning. *ACM Trans. Softw. Eng. Methodol.* 31(2), 1–34 (2021). <https://doi.org/10.1145/3485135>
- Dey, T., et al.: Detecting and characterizing bots that commit code. In: Kim, S., et al. (eds.) MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29–30 June, 2020, pp. 209–219. ACM (2020)
- Zhang, C., et al.: A survey of automatic source code summarization. *Symmetry.* 14(3), 471 (2022). <https://doi.org/10.3390/sym14030471>
- Teller, V.: Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition. *Comput. Ling.* 26(4), 638–641 (2000)
- Niesler, T., Woodland, P.: A variable-length category-based n-gram language model. In: 1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings, vol. 1, pp. 164–167 (1996)
- Mikolov, T., et al.: Efficient estimation of word representations in vector space. In: ICLR (Workshop Poster) (2013)
- Zhang, J., et al.: A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 783–794 (2019)
- White, M., et al.: Deep learning code fragments for code clone detection. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 87–98 (2016)
- Wei, H.-H., Li, M.: Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI'17, pp. 3034–3040. AAAI Press (2017)
- Cummins, C., et al.: Programl: graph-based deep learning for program optimization and analysis. In: arXiv preprint, arXiv:2003.10536 (2020)
- Zhao, G., Huang, J.: Deepsim: deep learning code functional similarity. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 141–151 (2018)
- Laaber, C., Basmaci, M., Salza, P.: Predicting unstable software benchmarks using static source code features. *Empir. Softw. Eng.* 26(5), 114 (2021). <https://doi.org/10.1007/s10664-021-09996-y>
- Samoa, H.P., et al.: A Structured Literature Study of Source Code Representation for Deep Learning in Software Engineering [Replication Package]. Zenodo (2021). <https://doi.org/10.5281/zenodo.6466506>
- Devlin, J., et al.: Semantic code repair using neuro-symbolic transformation networks. In: arXiv preprint, arXiv:1710.11054(2017)
- Malik, R.S., Patra, J., Pradel, M.: Nl2type: inferring javascript function types from natural language information. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 304–315 (2019)
- Pradel, M., et al.: Typewriter: neural type prediction with search-based validation. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, pp. 209–220. Association for Computing Machinery, New York, NY, USA (2020)
- Cambronero, J., et al.: When deep learning met code search. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, pp. 964–974. Association for Computing Machinery, New York, NY, USA (2019)
- Maurel, H., Vidal, S., Rezk, T.: Statically identifying XSS using deep learning. In: SECURE 2021 – 18th International Conference on Security and Cryptography, Virtual, France, July 2021 (2021)
- Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: arXiv preprint, arXiv:1711.00740 (2017)
- Liu, W., et al.: A survey of deep neural network architectures and their applications. *Neurocomputing.* 234, 11–26 (2017). <https://doi.org/10.1016/j.neucom.2016.12.038>
- Arel, I., Rose, D.C., Karnowski, T.P.: Deep machine learning—a new frontier in artificial intelligence research [research frontier]. *IEEE*

- Comput. Intell. Mag. 5(4), 13–18 (2010). <https://doi.org/10.1109/mci.2010.938364>
36. Sherstinsky, A.: Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Phys. D: Nonlinear Phenom.* 404, 132306 (2020). <https://doi.org/10.1016/j.physd.2019.132306>
 37. Scarselli, F., et al.: The graph neural network model. *IEEE Trans. Neural Network.* 20(1), 61–80 (2008). <https://doi.org/10.1109/tnn.2008.2005605>
 38. Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural Comput.* 9(8), 1735–1780 (1997). <https://doi.org/10.1162/neco.1997.9.8.1735>
 39. Vaswani, A., et al.: Attention is all you need. In: *Advances in Neural Information Processing Systems*, pp. 5998–6008 (2017)
 40. Chakraborty, S., et al.: Codit: code editing with tree-based neural models. *IEEE Trans. Softw. Eng.*, 48(4), 1–1399 (2020). <https://doi.org/10.1109/tse.2020.3020502>
 41. Cao, D., et al.: Ftclnet: convolutional lstm with Fourier transform for vulnerability detection. In: *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 539–546 (2020)
 42. Li, Y., et al.: Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.* 3(OOPSLA), 1–30 (2019). <https://doi.org/10.1145/3360588>
 43. Ahmad, W.U., et al.: A transformer-based approach for source code summarization. In: *arXiv preprint, arXiv:2005.00653* (2020)
 44. Alon, U., et al.: Code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3(POPL), 1–29 (2019). <https://doi.org/10.1145/3290353>
 45. Shuai, J., et al.: Improving code search with co-attentive representation learning. In: *Proceedings of the 28th International Conference on Program Comprehension, ICPC '20*, pp. 196–207. Association for Computing Machinery, New York, NY, USA (2020)
 46. Markovtsev, V., et al.: Style-analyzer: fixing code style inconsistencies with interpretable unsupervised algorithms. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 468–478 (2019)
 47. Csuvik, V., Kicsi, A., Vidács, L.: Source code level word embeddings in aiding semantic test-to-code traceability. In: *2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST)*, pp. 29–36 (2019)
 48. Ben-Nun, T., Jakobovits, A.S., Hoefler, T.: Neural code comprehension: a learnable representation of code semantics. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18*, pp. 3589–3601. Curran Associates Inc, Red Hook, NY, USA (2018)
 49. Ramadan, T., et al.: Comparative code structure analysis using deep learning for performance prediction. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 151–161 (2021)
 50. Hadj-Kacem, M., Bouassida, N.: Deep representation learning for code smells detection using variational auto-encoder. In: *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8 (2019)
 51. DeFrez, D., Thakur, A.V., Rubio-González, C.: Path-based function embedding and its application to error-handling specification mining. In: *ESEC/FSE 2018*, pp. 423–433. Association for Computing Machinery, New York, NY, USA (2018)
 52. Ampatzoglou, A., et al.: Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. *Inf. Softw. Technol.* 106, 201–230 (2019). <https://doi.org/10.1016/j.infsof.2018.10.006>
 53. Cheng, D., et al.: Manifesting bugs in machine learning code: an explorative study with mutation testing. In: *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 313–324. IEEE (2018)
 54. Narayanan, A., et al.: graph2vec: Learning distributed representations of graphs. In: *arXiv Preprints, arXiv:1707.05005v1* (2017)
 55. Ou, M., et al.: Asymmetric transitivity preserving graph embedding. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16*, pp. 1105–1114. Association for Computing Machinery, New York, NY, USA (2016)
 56. Goyal, P., Ferrara, E.: Graph embedding techniques, applications, and performance: a survey. *Knowl. Base Syst.* 151, 78–94 (2018). <https://doi.org/10.1016/j.knosys.2018.03.022>
 57. Grover, A., Leskovec, J.: Node2vec: scalable feature learning for networks. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16*, pp. 855–864. Association for Computing Machinery, New York, NY, USA (2016)
 58. Zhang, J., et al.: Retrieval-based neural source code summarization. In: *Rothermel, G., Bae, D. (eds.) ICSE '20: 42nd International Conference on Software Engineering*, Seoul, South Korea, 27 June – 19 July, 2020, pp. 1385–1397. ACM (2020)
 59. Wang, W., et al.: Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: *Kontogiannis, K., et al. (eds.) 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020*, London, ON, Canada, February 18–21, 2020, pp. 261–271. IEEE (2020)
 60. Cvitkovic, M., Singh, B., Anandkumar, A.: Open vocabulary learning on source code with a graph-structured cache. In: *International Conference on Machine Learning*, pp. 1475–1485. PMLR (2019)
 61. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: continual prediction with lstm. *Neural Comput.* 12(10), 2451–2471 (2000). <https://doi.org/10.1162/089976600300015015>
 62. LeCun, Y., et al.: Gradient-based learning applied to document recognition. *Proc. IEEE.* 86(11), 2278–2324 (1998). <https://doi.org/10.1109/5.726791>
 63. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: *Learning Internal Representations by Error Propagation* (Technical report). California Univ San Diego La Jolla Inst for Cognitive Science (1985)
 64. Li, Z., et al.: Vuldeepecker: a deep learning-based system for vulnerability detection. In: *Proceedings 2018 Network and Distributed System Security Symposium* (2018)
 65. Marastoni, N., Giacobazzi, R., Dalla Preda, M.: A deep learning approach to program similarity. In: *MASES 2018*, pp. 26–35. Association for Computing Machinery, New York, NY, USA (2018)
 66. Wang, Y., Cai, W.-d., Wei, P.-c.: A deep learning approach for detecting malicious javascript code. *Secur. Commun. Network.* 9(11), 1520–1534 (2016). <https://doi.org/10.1002/sec.1441>
 67. Tufano, M., et al.: Deep learning similarities from different representations of source code. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 542–553 (2018)
 68. Fernandes, P., Allamanis, M., Brockschmidt, M.: Structured neural summarization. In: *Conference paper at ICLR* (2019)
 69. Gupta, R., et al.: Fixing common c language errors by deep learning. In: *Thirty-First AAAI Conference on Artificial Intelligence* (2017)
 70. Yasunaga, M., Liang, P.: Graph-based, self-supervised program repair from diagnostic feedback. In: *International Conference on Machine Learning*, pp. 10799–10808. PMLR (2020)
 71. Fernandes, P., Allamanis, M., Brockschmidt, M.: Structured neural summarization. In: *International Conference on Learning Representations* (2019)
 72. Brockschmidt, M., et al.: Generative code modeling with graphs. In: *arXiv Preprint, arXiv:1805.08490* (2018).
 73. Büch, L., Andrzejak, A.: Learning-based recursive aggregation of abstract syntax trees for code clone detection. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 95–104 (2019)
 74. Collard, M.L., Decker, M.J., Maletic, J.I.: srcml: An infrastructure for the exploration, analysis, and manipulation of source code: a tool demonstration. In: *2013 IEEE International Conference on Software Maintenance*, pp. 516–519 (2013)
 75. Alon, U., et al.: A general path-based representation for predicting program properties. *SIGPLAN Not.* 53(4), 404–419 (2018). <https://doi.org/10.1145/3296979.3192412>

76. Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004, pp. 75–86 (2004)
77. Lam, P., et al.: The soot framework for java program analysis: a retrospective. In: Conference: Cetus Users and Compiler Infrastructure Workshop (CETUS 2011) (2011)
78. Cytron, R., et al.: Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program Lang. Syst.* 13(4), 451–490 (1991). <https://doi.org/10.1145/115372.115320>
79. Grunwald, D., Srinivasan, H.: Data flow equations for explicitly parallel programs. *SIGPLAN Not.* 28(7), 159–168 (1993). <https://doi.org/10.1145/173284.155349>
80. Allen, F.E., Cocke, J.: A program data flow analysis procedure. *Commun. ACM.* 19(3), 137 (1976). <https://doi.org/10.1145/360018.360025>
81. Vasic, M., et al.: Neural program repair by jointly learning to localize and repair. In: International Conference on Learning Representations (2019)
82. Santos, E.A., et al.: Syntax and sensibility: using language models to detect and correct syntax errors. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 311–322 (2018)
83. Dinella, E., et al.: Hoppity: learning graph transformations to detect and fix bugs in programs. In: International Conference on Learning Representations (2020)
84. Jia, R., Liang, P.: Adversarial examples for evaluating reading comprehension systems. In: arXiv preprint, arXiv:1707.07328 (2017)
85. Alon, U., et al.: code2seq: Generating sequences from structured representations of code. In: International Conference on Learning Representations (2019)
86. Kang, H.J., Bissyandé, T.F., Lo, D.: Assessing the generalizability of code2vec token embeddings. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1–12 (2019)
87. Wei, B., et al.: Code generation as a dual task of code summarization. In: 33rd Conference on Neural Information Processing Systems (NeurIPS) (2019)
88. Li, Z., et al.: Sysevr: a framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secure Comput.*, 1 (2021). <https://doi.org/10.1109/tdsc.2021.3051525>
89. Sonnekalb, T.: Machine-learning supported vulnerability detection in source code. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, pp. 1180–1183. Association for Computing Machinery, New York, NY, USA (2019)
90. Grover, A., Leskovec, J.: node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 855–864 (2016)
91. Costa, D., et al.: What's wrong with my benchmark results? Studying bad practices in jmh benchmarks. *IEEE Trans. Softw. Eng.* 47(7), 1452–1467 (2021). <https://doi.org/10.1109/tse.2019.2925345>
92. Laaber, C., Scheuner, J., Leitner, P.: Software microbenchmarking in the cloud. How bad is it really? *Empir. Softw. Eng.* 24(4), 2469–2508 (2019). <https://doi.org/10.1007/s10664-019-09681-1>
93. Tan, C., et al.: A survey on deep transfer learning. In: Kůrková, V., et al. (eds.) *Artificial Neural Networks and Machine Learning – ICANN 2018*, pp. 270–279. Springer International Publishing, Cham (2018)
94. Huang, G., et al.: Densely connected convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2017)
95. Devlin, J., et al.: Bert: pre-training of deep bidirectional transformers for language understanding. In: arXiv preprint, arXiv:1810.04805 (2018)
96. Panichella, A., et al.: How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 522–531 (2013)
97. Henkel, J., et al.: Code vectors: understanding programs through embedded abstracted symbolic traces. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pp. 163–174. Association for Computing Machinery, New York, NY, USA (2018)
98. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: International Conference on Learning Representations (2018)
99. Chen, Z., et al.: Sequencer: sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. Softw. Eng.* 47(9), 1943–1959 (2021). <https://doi.org/10.1109/tse.2019.2940179>
100. Cheng, X., et al.: Deepwukong: statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol.* 30(3), 1–33 (2021). <https://doi.org/10.1145/3436877>
101. Li, Z., et al.: Vuldelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Trans. Dependable Secure Comput.* (2021)
102. Amodio, M., Chaudhuri, S., Reps, T.W.: Neural attribute machines for program generation. In: arXiv e-prints, arXiv:1705.09231 (2017)
103. Haque, S., et al.: Improved automatic summarization of subroutines via attention to file context. In: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, pp. 300–310. Association for Computing Machinery, New York, NY, USA (2020)
104. Fujiwara, Y., et al.: Code-to-code search based on deep neural network and code mutation. In: 2019 IEEE 13th International Workshop on Software Clones (IWSC), pp. 1–7 (2019)
105. Gupta, R., Kanade, A., Shevade, S.: Neural attribution for semantic bug-localization in student programs. In: Wallach, H., et al. (eds.) *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., New York (2019)
106. Liu, S., et al.: Deepbalance: deep-learning and fuzzy oversampling for vulnerability detection. *IEEE Trans. Fuzzy Syst.* 28(7), 1329–1343 (2020). <https://doi.org/10.1109/tfuzz.2019.2958558>
107. Nair, A., Roy, A., Funcggn, K.M.: A graph neural network approach to program similarity. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '20. Association for Computing Machinery, New York, NY, USA (2020)
108. Sheneamer, A., Kalita, J.: Semantic clone detection using machine learning. In: 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 1024–1028 (2016)
109. Svyatkovskiy, A., et al.: Fast and memory-efficient neural code completion. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 329–340 (2021)
110. Liu, K., et al.: Learning to spot and refactor inconsistent method names. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 1–12 (2019)
111. Shi, K., et al.: Pathpair2vec: an ast path pair-based code representation method for defect prediction. *J. Comput. Lang.* 59, 100979 (2020). <https://doi.org/10.1016/j.cola.2020.100979>
112. Li, J., et al.: Software defect prediction via convolutional neural network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 318–328 (2017)
113. Perez, D., Chiba, S.: Cross-language clone detection by learning over abstract syntax trees. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 518–528 (2019)
114. Li, L., et al.: A deep learning-based clone detection approach. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 249–260 (2017)
115. Gu, X., Zhang, H., Kim, S.: Deep code search. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 933–944 (2018)
116. Hu, X., et al.: Deep code comment generation. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), pp. 200–20010 (2018)
117. Sun, Z., et al.: A grammar-based structural cnn decoder for code generation. *Proc. AAAI Conf. Artif. Intell.* 33, 7055–7062 (2019). <https://doi.org/10.1609/aaai.v33i01.33017055>

118. Wang, R., et al.: Fret: functional reinforced transformer with bert for code summarization. *IEEE Access*. 8, 135591–135604 (2020). <https://doi.org/10.1109/access.2020.3011744>
119. Murali, V., et al.: Neural sketch learning for conditional program generation. In: *International Conference on Learning Representations* (2018)
120. Svyatkovskiy, A., et al.: Pythia: AI-assisted code completion system. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining, KDD '19*, pp. 2727–2735. Association for Computing Machinery, New York, NY, USA (2019)
121. Wang, W., et al.: Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 261–271 (2020)
122. Wu, H., Zhao, H., Zhang, H.: Code summarization with structure-induced transformer. In: *Association for Computational Linguistics: ACL-IJCNLP 2021* (2021)
123. Li, J., et al.: Code completion with neural attention and pointer networks. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence* (2018)
124. Pradel, M., Sen, K.: Deep learning to find bugs. *TU Darmstadt, Department of Computer Science*. 4, 1 (2017)
125. Lin, G., et al.: Deep learning-based vulnerable function detection: a benchmark. In: Zhou, J., et al. (eds.) *Information and Communications Security*, pp. 219–232. Springer International Publishing, Cham (2020)
126. Iyer, S., et al.: Summarizing source code using a neural attention model. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2073–2083 (2016)
127. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. *SIGPLAN Not.* 49(6), 419–428 (2014). <https://doi.org/10.1145/2666356.2594321>
128. Xie, C., et al.: A source code similarity based on siamese neural network. *Appl. Sci.* 10(21), 7519 (2020). <https://doi.org/10.3390/app10217519>
129. Wang, S., Liu, T., Tan, L.: Automatically learning semantic features for defect prediction. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 297–308 (2016)
130. Tufano, M., et al.: An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.* 28(4), 1–29 (2019). <https://doi.org/10.1145/3340544>
131. Dam, H.K., Tran, T., Pham, T.: A deep language model for software code. In: *arXiv preprint, arXiv:1608.02715* (2016)
132. Pradel, M., Sen, K.: Deepbugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.* 2(OOPSLA), 1–25 (2018). <https://doi.org/10.1145/3276517>
133. Russell, R., et al.: Automated vulnerability detection in source code using deep representation learning. In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 757–762 (2018)
134. Wang, K., Su, Z.: Learning blended, precise semantic program embeddings. *Proc. ACM Program. Lang.* 1, 1–25 (2019)
135. White, M., et al.: Sorting and transforming program repair ingredients via deep learning code similarities. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 479–490 (2019)
136. Yu, H., et al.: Neural detection of semantic code clones via tree-based convolution. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 70–80 (2019)
137. Yin, P., Neubig, G.: A syntactic neural model for general-purpose code generation. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics* (2017)
138. Zeng, J., et al.: Fast code clone detection based on weighted recursive autoencoders. *IEEE Access*. 7, 125062–125078 (2019). <https://doi.org/10.1109/access.2019.2938825>
139. White, M., et al.: Toward deep learning software repositories. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 334–345 (2015)
140. Mou, L., et al.: Tbcnn: a tree-based convolutional neural network for programming language processing. In: *arXiv preprint, arXiv:1409.5718* (2014)
141. Zhou, M., et al.: Deeptle: learning code-level features to predict code performance before it runs. In: *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 252–259 (2019)
142. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: Balcan, M.F., Weinberger, K. Q. (eds.) *Proceedings of the 33rd International Conference on Machine Learning, Volume 48 of Proceedings of Machine Learning Research*, 20–22 Jun 2016, pp. 2091–2100. PMLR, New York, New York, USA (2016)
143. Wan, Y., et al.: Improving automatic source code summarization via deep reinforcement learning. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pp. 397–407. Association for Computing Machinery, New York, NY, USA (2018)
144. Zhou, Y., et al.: Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: *NeurIPS* (2019)

How to cite this article: Samoaa, H.P., et al.: A systematic mapping study of source code representation for deep learning in software engineering. *IET Soft.* 16(4), 351–385 (2022). <https://doi.org/10.1049/sfw2.12064>

APPENDICES

Analysis of the main attributes

Authors	Title	Venue	Year	Cit. Key
U. Alon, M. Zilberstein, O. Levy, and A. Yahav	code2vec: learning distributed representations of code	POPL	2019	[44]
U. Alon, S. Brody, O. Levy, E. Yahav	code2seq: Generating Sequences from Structured Representations of Code	ICLR	2019	[85]
M. Brockschmidt, M. Allamanis, A.L. Gaunt, O. Polozov	Generative Code Modelling with Graphs	ICLR	2019	[72]
W. U. Ahmad, S. Chakraborty, B. Ray, K. Chang	A Transformer-based Approach for Source Code Summarization	ACL	2020	[43]
Nghi D. Q. Bui, Lingxiao Jiang, Yijun Yu	Cross-Language Learning for Program Classification using Bilateral Tree-Based Convolutional Neural Networks	AAAI	2017	[1]

(Continues)

APPENDIX (Continued)

Authors	Title	Venue	Year	Cit. Key
Nghi D. Q. Bui, Yijun Yu, Lingxiao Jiang	Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations	SIGIR	2021	[10]
J. Devlin, J. Uesato, R. Singh, P. Kohli	Semantic Code Repair using Neuro-Symbolic Transformation Networks	arXiv	2017	[28]
M. Allamanis, M. Brockschmidt, M. Khademi	Learning to Represent Programs with Graphs	ICLR	2018	[33]
L. Büch, A. Andrzejak	Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection	SANER	2019	[73]
R. Gupta, S. Pal, A. Kanade, S. Shevade	DeepFix: Fixing Common C Language Errors by Deep Learning	AAAI	2017	[69]
J. Cambronero, H. Li, S. Kim, K. Sen, S. Chandra	When Deep Learning Met Code Search	FSE	2019	[31]
S. Chakraborty, Y. Ding, M. Allamanis, B. Ray	CODIT: Code Editing with Tree-Based Neural Models	TSE	2019	[40]
D. Cao, J. Huang, X. Zhang, X. Liu	FTCLNet: Convolutional LSTM with Fourier Transform for Vulnerability Detection	TrustCom	2020	[41]
Nghi D. Q. Bui, Y. Yu, L. Jiang	InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees	ICSE	2021	[4]
Z. Chen, S. Komrmusch, M. Tufano, L. Pouchet, D. Poshvanyk, M. Monperrus	SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair	TSE	2021	[99]
X. Cheng, H. Wang, J. Hua, G. Xu, Y. Sui	DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network	TOSEM	2021	[100]
J. Hua, H. Wang	On the Effectiveness of Deep Vulnerability Detectors to Simple Stupid Bug Detection	MSR	2021	[7]
U. Alon, M. Zilberstein, O. Levy, E. Yahav	A general path-based representation for predicting program properties	SIGPLAN	2018	[75]
T. Ben-Nun, A. S. Jakobovits, T. Hoefler	Neural Code Comprehension: A Learnable Representation of Code Semantics	NeurIPS	2018	[48]
V. Csuvik, A. Kicsi, L. Vidács	Source Code Level Word Embeddings in Aiding Semantic Test-to-Code Traceability	ICSE	2019	[47]
M. Cvitkovic, B. Singh, A. Anandkumar	Open Vocabulary Learning on Source Code with a Graph-Structured Cache	ICML	2019	[60]
Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, H. Jin	VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector	TDSC	2021	[101]
Y. Li, S. Wang, T. N. Nguyen	Fault Localization with Code Coverage Representation Learning	ICSE	2021	[8]
S. Liu, Y. Chen, X. Xie, J. K. Siow, Y. Liu	Retrieval-Augmented Generation for Code Summarization via Hybrid GNN	ICLR	2021	[11]
H. Maurel, S. Vidal, T. Rezk	Statically Identifying XSS using Deep Learning	SECURITY	2021	[32]
C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, H. Leather	PROGRAML: Graph-based Deep Learning for Program Optimization and Analysis	PMLR	2021	[24]
P. Fernandes, M. Allamanis, M. Brockschmidt	Structured Neural Summarization	ICLR	2019	[71]
M. Amodio, S. Chaudhuri, T. Reps	Neural Attribute Machines for Program Generation	arXiv	2021	[102]
E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, K. Wang	Hoppity: Learning graph transformations to detect and fix bugs in programs	ICLR	2020	[83]
C. Fang, Z. Liu, Y. Shi, J. Huang, Q. Shi	Functional Code Clone Detection with Syntax and Semantics Fusion Learning	ISSSTA	2020	[5]
S. Haque, A. LeClair, L. Wu, C. McMillan	Improved Automatic Summarization of Subroutines via Attention to File Context	MSR	2020	[103]
Y. Fujiwara, N. Yoshida, E. Choi, K. Inoue	Code-to-Code Search Based on Deep Neural Network and Code Mutation	IWSC	2019	[104]

APPENDIX (Continued)

Authors	Title	Venue	Year	Cit. Key
T. Ramadan, T.Z. Islam, C. Phelps	Comparative Code Structure Analysis using Deep Learning for Performance Prediction	ISPASS	2021	[49]
A. Kanade, P. Maniatis, G. Balakrishnan, K. Shi	Pre-trained Contextual Embedding of Source Code	ICLR	2020	[2]
D. DeFreez, A.V. Thakur, C. Rubio-González	Path-based function embedding and its application to error-handling specification mining	FSE	2018	[51]
R. Gupta, A. Kanade, S. Shevade	Neural Attribution for Semantic Bug-Localization in Student Programs	NeurIPS	2019	[105]
M. Hadj-Kacem, N. Bouassida	Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder	IJCNN	2019	[50]
S. Liu, G. Lin, Q.L. Han, S. Wen, J. Zhang, Y. Xiang	DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection	Transactions on Fuzzy Systems	2020	[106]
N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, R. Purandare	Modelling Functional Similarity in Source Code with Graph-Based Siamese Networks	TSE	2020	[6]
K. Shi, Y. Lu, G. Liu, Z. Wei, J. Chang	MPT-embedding: An unsupervised representation learning of code for software defect prediction	SEP	2021	[9]
A. Nair, A. Roy, K. Meinke	funcGNN: A Graph Neural Network Approach to Program Similarity	ESEM	2020	[107]
A. Sheneamer, J. Kalita	Semantic Clone Detection Using Machine Learning	ICMLA	2016	[108]
H.J. Kang, T.F. Bissyandé, D. Lo	Assessing the Generalizability of code2vec Token Embeddings	ASE	2019	[86]
M. Pradel, G. Gousios, J. Liu, S. Chandra	TypeWriter: Neural-Type Prediction with Search-Based Validation	FSE	2020	[30]
Y. Li, S. Wang, T.N. Nguyen, S. Van Nguyen	Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks	OOPSLA	2019	[42]
A. Svyatkovskiy, S. Lee, A. Hadjitofi	Fast and Memory-Efficient Neural Code Completion	MSR	2021	[109]
K. Liu, D. Kim, T.F. Bissyandé, T. Kim	Learning to Spot and Refactor Inconsistent Method Names	ICSE	2019	[110]
R.S. Malik, J. Patra, M. Pradel	NL2Type: Inferring JavaScript Function Types from Natural Language Information	ICSE	2019	[29]
K. Shi, Y. Lu, J. Chang, Z. Wei	PathPair2Vec: An AST path pair-based code representation method for defect prediction	JCL	2020	[111]
V. Markovtsev, W. Long, H. Mougard	STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms	MSR	2019	[46]
J. Li, P. He, J. Zhu, M.R. Lyu	Software Defect Prediction via Convolutional Neural Network	QRS	2017	[112]
D. Perez, S. Chiba	Cross-language clone detection by learning over abstract syntax trees	MSR	2019	[113]
L. Li, H. Feng, W. Zhuang, N. Meng	CCLearner: A Deep Learning-Based Clone Detection Approach	ICSME	2017	[114]
T. Sonnekalb	Machine-Learning Supported Vulnerability Detection in Source Code	FSE	2019	[89]
X. Gu, H. Zhang, S. Kim	Deep Code Search	ICSE	2018	[115]
J. Henkel, S.K. Lahiri, B. Liblit, T. Reps	Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces	FSE	2018	[97]
X. Hu, G. Li, X. Xia, D. Lo, Z. Jin	Deep Code Comment Generation	ICPC	2018	[116]
Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li	A Grammar-Based Structural CNN Decoder for Code Generation	AAAI	2019	[117]
J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, Y. Lei	Improving Code Search with Co-Attentive Representation Learning	ICPC	2020	[45]

(Continues)

APPENDIX (Continued)

Authors	Title	Venue	Year	Cit. Key
R. Wang, H. Zhang, G. Lu, L. Lyu, C. Lyu	Fret: Functional Reinforced Transformer With BERT for Code Summarization	IEEE Access	2020	[118]
V. Murali, L. Qi, S. Chaudhuri, C. Jermaine	Neural Sketch Learning for Conditional Program Generation	ICLR	2017	[119]
A. Svyatkovskiy, Y. Zhao, S. Fu	Pythia: AI-assisted Code Completion System	SIGKDD	2019	[120]
W. Wang, G. Li, B. Ma, X. Xia, Z. Jin	Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree	SANER	2020	[121]
H. Wu, H. Zhao, M. Zhang	SIT3: Code Summarization with Structure-Induced Transformer	ACL	2021	[122]
Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng	VulDeePecker: A Deep Learning-Based System for Vulnerability Detection	NDSS	2018	[64]
J. Li, Y. Wang, M.R. Lyu, I. King	Code Completion with Neural Attention and Pointer Networks	JICAI	2018	[123]
M. Pradel, K. Sen	Deep Learning to Find Bugs	arXiv	2017	[124]
M. White, M. Tufano, C. Vendome	Deep Learning Code Fragments for Code Clone Detection	ASE	2016	[22]
L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin	Convolutional Neural Networks over Tree Structures for Programming Language Processing	AAAI	2016	[3]
G. Lin, W. Xiao, J. Zhang, Y. Xiang	Deep Learning-Based Vulnerable Function Detection	ICICS	2020	[125]
S. Iyer, I. Konostas, A. Cheung, L. Zettlemoyer	Summarizing Source Code using a Neural Attention Model	ACL	2016	[126]
H. Wei, M. Li	Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code	AAAI	2017	[23]
V. Raychev, M. Vechev, E. Yahav	Code completion with statistical language models	SIGPLAN	2014	[127]
N. Marastoni, R. Giacobazzi, M. Dalla Preda	A Deep Learning Approach to Program Similarity	MASES	2018	[65]
C. Xie, X. Wang, C. Qian, M. Wang	A Source Code Similarity Based on Siamese Neural Network	Applied Science	2020	[128]
Y. Wang, W. Cai, P. Wei	A deep learning approach for detecting malicious JavaScript code	SCN	2016	[66]
S. Wang, T. Liu, L. Tan	Automatically Learning Semantic Features for Defect Prediction	2016	ICSE	[129]
M. Yasunaga, P. Liang	Graph-based, Self-Supervised Program Repair from Diagnostic Feedback	ICML	2020	[70]
M. Tufano, C. Watson, G. Bavota, M.D. Penta	An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation	TOSEM	2019	[130]
H. Wei, M. Li	Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code	JICAI	2017	[23]
H.K. Dam, T. Tran, T. Pham	A deep language model for software code	arXiv	2016	[131]
M. Vasic, A. Kanade, P. Maniatis, D. Bieber	Neural program repair by jointly learning to localize and repair	ICLR	2018	[81]
M. Pradel, K. Sen	DeepBugs: A Learning Approach to Name-Based Bug Detection	OOPSLA	2018	[132]
R. Russell, L. Kim, L. Hamilton, T. Lazovich	Automated Vulnerability Detection in Source Code Using Deep Representation Learning	ICMLA	2018	[133]
K. Wang, Z. Su	Learning Blended, Precise Semantic Program Embeddings	PLDI	2020	[134]
E.A. Santos, J.C. Campbell, D. Patel	Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors	SANER	2018	[82]
B. Wei, G. Li, X. Xia, Z. Fu, Z. Jin	Code Generation as a Dual Task of Code Summarization	NeurIPS	2019	[87]
M. White, M. Tufano, M. Martinez	Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities	SANER	2019	[135]

APPENDIX (Continued)

Authors	Title	Venue	Year	Cit. Key
H. Yu, W. Lam, L. Chen, G. Li, T. Xie	Neural Detection of Semantic Code Clones Via Tree-Based Convolution	ICPC	2019	[136]
P. Yin, G. Neubig	A Syntactic Neural Model for General-Purpose Code Generation	ACL	2017	[137]
J. Zeng, K. Ben, X. Li, X. Zhang	Fast Code Clone Detection Based on Weighted Recursive Autoencoders	IEEE Access	2019	[138]
M. White, C. Vendome	Toward deep learning software repositories	MSR	2015	[139]
J. Zhang, X. Wang, H. Zhang, H. Sun	A Novel Neural Source Code Representation Based on Abstract Syntax Tree	ICSE	2019	[21]
M. Tufano, C. Watson, G. Bavota	Deep Learning Similarities from Different Representations of Source Code	MSR	2018	[67]
J. Zhang, X. Wang, H. Zhang, H. Sun	Retrieval-based Neural Source Code Summarization	ICSE	2020	[58]
L. Mou, G. Li, Z. Jin, L. Zhang, T. Wang	TBCNN: A tree-based convolutional neural network for programming language processing	arXiv	2014	[140]
L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin	Convolutional Neural Networks over Tree Structures for Programming Language Processing	AAAI	2016	[3]
M. Zhou, J. Chen, H. Hu, J. Yu, Z. Li	DeepTLE: Learning Code-Level Features to Predict Code Performance before It Runs	APSEC	2019	[141]
M. Allamanis, H. Peng, C. Sutton	A Convolutional Attention Network for Extreme Summarization of Source Code	ICML	2016	[142]
G. Zhao, J. Huang	DeepSim: deep learning code functional similarity	FSE	2018	[25]
Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu	Improving Automatic Source Code Summarization via Deep Reinforcement Learning	ASE	2018	[143]
Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu	Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks	NeurIPS	2019	[144]

B List of Venue Acronyms

Acronyms	Venue
AAAI	Association for the Advancement of Artificial Intelligence
ACL	Association for Computational Linguistics
ASE	International Conference on Automated Software Engineering
FSE	Fast Software Encryption
ICLR	International Conference on Learning Representations
ICML	International Conference on Machine Learning
ICMLA	International Conference on Machine Learning and Applications
ICPC	International Conference on Program Comprehension
ICSE	International Conference on Software Engineering
IJCAI	International Joint Conference on Artificial Intelligence
MSR	Mining Software Repositories
NeurIPS	Neural Information Processing Systems
PACMPL	Proceedings of the ACM on Programming Languages
PLDI	Programming Language Design and Implementation
SANER	International Conference on Software Analysis, Evolution and Reengineering

C SE Tasks and Related Papers

C.1 Main SE Tasks and Related Papers

Title	Code-code	Code-text	Text-code	Code-prediction
code2vec: Learning Distributed Representations of Code		✓		
code2seq: Generating Sequences from Structured Representations of Code		✓		
Generative Code Modelling with Graphs	✓			
A Transformer-based Approach for Source Code Summarization		✓		
Cross-Language Learning for Program Classification using Bilateral Tree-Based Convolutional Neural Networks				✓
Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations	✓	✓	✓	
Semantic Code Repair using Neuro-Symbolic Transformation Networks				✓
Learning to Represent Programs with Graphs		✓		
Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection	✓			
DeepFix: Fixing Common C Language Errors by Deep Learning	✓			
When Deep Learning Met Code Search			✓	
CODIT: Code Editing with Tree-Based Neural Models	✓			
FTCLNet: Convolutional LSTM with Fourier Transform for Vulnerability Detection				✓
InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees	✓			
SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair	✓			
DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network				✓
On the Effectiveness of Deep Vulnerability Detectors to Simple Stupid Bug Detection				✓
A general path-based representation for predicting program properties		✓		
Neural Code Comprehension: A Learnable Representation of Code Semantics				✓
Source Code Level Word Embeddings in Aiding Semantic Test-to-Code Traceability	✓			
Open Vocabulary Learning on Source Code with a Graph-Structured Cache	✓	✓		
VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector				✓
Fault Localization with Code Coverage Representation Learning				✓
SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities				✓
Retrieval-Augmented Generation for Code Summarization via Hybrid GNN		✓		
Statically Identifying XSS using Deep Learning				✓
PROGRAML: GRAPH-BASED DEEP LEARNING FOR PROGRAM OPTIMIZATION AND ANALYSIS	✓			
STRUCTURED NEURAL SUMMARIZATION		✓		
Neural Attribute Machines for Program Generation	✓			
Hoppity: Learning graph transformations to detect and fix bugs in programs.	✓			✓
Functional Code Clone Detection with Syntax and Semantics Fusion Learning	✓			
Improved Automatic Summarization of Subroutines via Attention to File Context		✓		
Code-to-Code Search Based on Deep Neural Network and Code Mutation			✓	
Comparative Code Structure Analysis using Deep Learning for Performance Prediction				✓
PRE-TRAINED CONTEXTUAL EMBEDDING OF SOURCE CODE				✓
Path-based function embedding and its application to error-handling specification mining				✓
Neural Attribution for Semantic Bug-Localization in Student Programs				✓

APPENDIX (Continued)

Title	Code-code	Code-text	Text-code	Code-prediction
Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder				✓
DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection				✓
Modelling Functional Similarity in Source Code with Graph-Based Siamese Networks	✓			
MPT-embedding: An unsupervised representation learning of code for software defect prediction				✓
funcGNN: A Graph Neural Network Approach to Program Similarity	✓			
Semantic Clone Detection Using Machine Learning	✓			
Assessing the Generalizability of code2vec Token Embeddings	✓	✓		
TypeWriter: Neural Type Prediction with Search-Based Validation				✓
Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks				✓
Fast and Memory-Efficient Neural Code Completion	✓			
Learning to Spot and Refactor Inconsistent Method Names		✓		
NL2Type: Inferring JavaScript Function Types from Natural Language Information				✓
PathPair2Vec: An AST path pair-based code representation method for defect prediction				✓
STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms	✓			
Software Defect Prediction via Convolutional Neural Network				✓
Cross-language clone detection by learning over abstract syntax trees	✓			
CCLearner: A Deep Learning-Based Clone Detection Approach	✓			
Machine-Learning Supported Vulnerability Detection in Source Code				✓
Deep Code Search			✓	
Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces	✓			
Deep Code Comment Generation		✓		
A Grammar-Based Structural CNN Decoder for Code Generation	✓			
Improving Code Search with Co-Attentive Representation Learning			✓	
Fret: Functional Reinforced Transformer With BERT for Code Summarization		✓		
Neural Sketch Learning for Conditional Program Generation	✓			
Pythia: AI-assisted Code Completion System	✓			
Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree	✓			
SIT3: Code Summarization with Structure-Induced Transformer		✓		
VulDeePecker: A Deep Learning-Based System for Vulnerability Detection				✓
Code Completion with Neural Attention and Pointer Networks	✓			
Deep Learning to Find Bugs (With focus on name-based bug detectors)				✓
Deep Learning Code Fragments for Code Clone Detection	✓			
Convolutional Neural Networks over Tree Structures for Programming Language Processing				✓
Deep Learning-Based Vulnerable Function Detection: A Benchmark				✓
Summarizing Source Code using a Neural Attention Model		✓		
Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code	✓			
Code completion with statistical language models	✓			

(Continues)

APPENDIX (Continued)

Title	Code-code	Code-text	Text-code	Code-prediction
A Deep Learning Approach to Program Similarity	✓			
A Source Code Similarity Based on Siamese Neural Network	✓			
A deep learning approach for detecting malicious JavaScript code				✓
Automatically Learning Semantic Features for Defect Prediction				✓
Graph-based, Self-Supervised Program Repair from Diagnostic Feedback	✓			
An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation	✓			
Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code	✓			
A deep language model for software code	✓			
Neural program repair by jointly learning to localize and repair	✓			✓
DeepBugs: A Learning Approach to Name-Based Bug Detection				✓
Automated Vulnerability Detection in Source Code Using Deep Representation Learning				✓
Blended, precise semantic program embeddings		✓		
Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors	✓			✓
Code Generation as a Dual Task of Code Summarization		✓	✓	
Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities	✓			
Neural Detection of Semantic Code Clones Via Tree-Based Convolution	✓			
A Syntactic Neural Model for General-Purpose Code Generation	✓			
Fast Code Clone Detection Based on Weighted Recursive Autoencoders	✓			
Toward deep learning software repositories	✓			
A Novel Neural Source Code Representation Based on Abstract Syntax Tree	✓			✓
Deep Learning Similarities from Different Representations of Source Code	✓			
Retrieval-based Neural Source Code Summarization		✓		
TBCNN: A tree-based convolutional neural network for programming language processing				✓
Convolutional Neural Networks over Tree Structures for Programming Language Processing				✓
DeepTLE: Learning Code-Level Features to Predict Code Performance before It Runs				
A Convolutional Attention Network for Extreme Summarization of Source Code		✓		
DeepSim: deep learning code functional similarity	✓			
Improving Automatic Source Code Summarization via Deep Reinforcement Learning		✓		
Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks				✓

C.2 Code–Code Tasks and Related Papers

Title	Code clone detection	Traceability	Code similarity detection	Program repair	Fixing format	Code completion	Compiler analysis	Program generation
Generative Code Modelling with Graphs								✓
Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations			✓					

APPENDIX (Continued)

Title	Code clone detection	Traceability	Code similarity detection	Program repair	Fixing format	Code completion	Compiler analysis	Program generation
Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection	✓							
DeepFix: Fixing Common C Language Errors by Deep Learning				✓				
CODIT: Code Editing with Tree-Based Neural Models				✓				✓
InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees	✓		✓					
SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair				✓				
Source Code Level Word Embeddings in Aiding Semantic Test-to-Code Traceability		✓						
Open Vocabulary Learning on Source Code with a Graph-Structured Cache						✓		
PROGRAML: GRAPH-BASED DEEP LEARNING FOR PROGRAM OPTIMIZATION AND ANALYSIS							✓	
Neural Attribute Machines for Program Generation								✓
Hoppity: Learning graph transformations to detect and fix bugs in programs.				✓				
Functional Code Clone Detection with Syntax and Semantics Fusion Learning	✓							
Modelling Functional Similarity in Source Code with Graph-Based Siamese Networks	✓							
funcGNN: A Graph Neural Network Approach to Program Similarity			✓					
Semantic Clone Detection Using Machine Learning	✓							
Assessing the Generalizability of code2vec Token Embeddings	✓							
Fast and Memory-Efficient Neural Code Completion						✓		
STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms					✓			
Cross-language clone detection by learning over abstract syntax trees	✓							
CCLearner: A Deep Learning-Based Clone Detection Approach	✓							
Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces			✓					
A Grammar-Based Structural CNN Decoder for Code Generation								✓
Neural Sketch Learning for Conditional Program Generation						✓		
Pythia: AI-assisted Code Completion System						✓		
Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree	✓							

(Continues)

APPENDIX (Continued)

Title	Code clone detection	Traceability	Code similarity detection	Program repair	Fixing format	Code completion	Compiler analysis	Program generation
Code Completion with Neural Attention and Pointer Networks						✓		
Deep Learning Code Fragments for Code Clone Detection	✓							
Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code	✓							
Code completion with statistical language models						✓		
A Deep Learning Approach to Program Similarity			✓					
A Source Code Similarity Based on Siamese Neural Network			✓					
Graph-based, Self-Supervised Program Repair from Diagnostic Feedback				✓				
An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation				✓				
Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code	✓							
A deep language model for software code						✓		
Neural program repair by jointly learning to localize and repair				✓				
Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors				✓				
Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities				✓				
Neural Detection of Semantic Code Clones Via Tree-Based Convolution	✓							
A Syntactic Neural Model for General-Purpose Code Generation								✓
Fast Code Clone Detection Based on Weighted Recursive Autoencoders	✓		✓					
Toward deep learning software repositories								✓
A Novel Neural Source Code Representation Based on Abstract Syntax Tree	✓							
Deep Learning Similarities from Different Representations of Source Code	✓		✓					
DeepSim: deep learning code functional similarity			✓					

C.3 Code Prediction Tasks and Related Papers

Title	Source code classification	Code smell detection	Error handling	Bug detection	Malicious behaviour detection	Vulnerability detection	Performance prediction	Type signature prediction
Cross-Language Learning for Program Classification using Bilateral Tree-Based Convolutional Neural Networks	✓							

APPENDIX (Continued)

Title	Source code classification	Code smell detection	Error handling	Bug detection	Malicious behaviour detection	Vulnerability detection	Performance prediction	Type signature prediction
Semantic Code Repair using Neuro-Symbolic Transformation Networks				✓				
FTCLNet: Convolutional LSTM with Fourier Transform for Vulnerability Detection						✓		
DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network						✓		
On the Effectiveness of Deep Vulnerability Detectors to Simple Stupid Bug Detection				✓				
Neural Code Comprehension: A Learnable Representation of Code Semantics					✓			
VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector						✓		
Fault Localization with Code Coverage Representation Learning				✓				
SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities						✓		
Statically Identifying XSS using Deep Learning						✓		
Hoppity: Learning graph transformations to detect and fix bugs in programs				✓				
Comparative Code Structure Analysis using Deep Learning for Performance Prediction							✓	
PRE-TRAINED CONTEXTUAL EMBEDDING OF SOURCE CODE	✓							
Path-based function embedding and its application to error-handling specification mining			✓					
Neural Attribution for Semantic Bug-Localization in Student Programs				✓				
Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder		✓						
DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection						✓		
MPT-embedding: An unsupervised representation learning of code for software defect prediction				✓				
TypeWriter: Neural Type Prediction with Search-Based Validation								✓
Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks								
NL2Type: Inferring JavaScript Function Types from Natural Language Information								✓

(Continues)

APPENDIX (Continued)

Title	Source code classification	Code smell detection	Error handling	Bug detection	Malicious behaviour detection	Vulnerability detection	Performance prediction	Type signature prediction
PathPair2Vec: An AST path pair-based code representation method for defect prediction				✓				
Software Defect Prediction via Convolutional Neural Network				✓				
Machine-Learning Supported Vulnerability Detection in Source Code						✓		
VulDeePecker: A Deep Learning-Based System for Vulnerability Detection						✓		
Deep Learning to Find Bugs (With focus on name-based bug detectors)				✓				
Convolutional Neural Networks over Tree Structures for Programming Language Processing	✓							
Deep Learning-Based Vulnerable Function Detection: A Benchmark						✓		
A deep learning approach for detecting malicious JavaScript code					✓			
Automatically Learning Semantic Features for Defect Prediction				✓				
Neural program repair by jointly learning to localize and repair				✓				
DeepBugs: A Learning Approach to Name-Based Bug Detection				✓				
Automated Vulnerability Detection in Source Code Using Deep Representation Learning						✓		
Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors				✓				
A Novel Neural Source Code Representation Based on Abstract Syntax Tree	✓							
TBCNN: A tree-based convolutional neural network for programming language processing	✓							
Convolutional Neural Networks over Tree Structures for Programming Language Processing	✓							
DeepTLE: Learning Code-Level Features to Predict Code Performance before It Runs							✓	
Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks						✓		

C.4 Code-Text and Related Papers

Title	Identifier generation	Code summarisation
code2vec: Learning Distributed Representations of Code	✓	
code2seq: Generating Sequences from Structured Representations of Code		✓
A Transformer-based Approach for Source Code Summarization		✓
Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations		✓
Learning to Represent Programs with Graphs	✓	
A general path-based representation for predicting program properties	✓	
Open Vocabulary Learning on Source Code with a Graph-Structured Cache	✓	
Retrieval-Augmented Generation for Code Summarization via Hybrid GNN		✓
STRUCTURED NEURAL SUMMARIZATION	✓	✓
Improved Automatic Summarization of Subroutines via Attention to File Context		✓
Assessing the Generalizability of code2vec Token Embeddings		✓
Learning to Spot and Refactor Inconsistent Method Names	✓	
Deep Code Comment Generation		✓
Fret: Functional Reinforced Transformer With BERT for Code Summarization		✓
SIT3: Code Summarization with Structure-Induced Transformer		✓
Summarizing Source Code using a Neural Attention Model		✓
Blended, precise semantic program embeddings	✓	
Code Generation as a Dual Task of Code Summarization		✓
Retrieval-based Neural Source Code Summarization		✓
A Convolutional Attention Network for Extreme Summarization of Source Code		✓
Improving Automatic Source Code Summarization via Deep Reinforcement Learning		✓

C.5 Text-Code and Related Papers

Title	Program synthesis	Code search
Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations		✓
When Deep Learning Met Code Search		✓
Code-to-Code Search Based on Deep Neural Network and Code Mutation		✓
Deep Code Search		✓
Improving Code Search with Co-Attentive Representation Learning		✓
Code Generation as a Dual Task of Code Summarization	✓	

Paper 2

Analysing the Behaviour of Tree-Based Neural Networks in Regression Tasks

Peter Samoaa, Mehrdad Farahani, Antonio Longa, Philipp Leitner, Morteza Haghiri Chehreghani

Journal of IEEE Transactions on Neural Networks and Learning Systems, 2024

Analyzing the Behaviour of Tree-Based Neural Networks in Regression Tasks

Peter Samoaa*, Mehrdad Farahani*, Antonio Longa[‡], Philipp Leitner[†], Morteza Haghir Chehreghani*

*Data Science and AI Division

Chalmers University of Technology

Gothenburg, Sweden

Email: {samooa, mehrdad.farahani, morteza.chehreghani}@chalmers.se

[†]Interaction Design and Software Engineering Division

Chalmers University of Technology

Gothenburg, Sweden

Email: philipp.leitner@chalmers.se

[‡]Department of Information Engineering and Computer Science

University of Trento

Trento, Italy

Email: antonio.longa@unitn.it

Abstract—The landscape of deep learning has vastly expanded the frontiers of source code analysis, particularly through the utilization of structural representations such as Abstract Syntax Trees (ASTs). While these methodologies have demonstrated effectiveness in classification tasks, their efficacy in regression applications, such as execution time prediction from source code, remains underexplored. This paper endeavours to decode the behaviour of tree-based neural network models in the context of such regression challenges. We extend the application of established models—tree-based Convolutional Neural Networks (CNNs), Code2Vec, and Transformer-based methods—to predict the execution time of source code by parsing it to an AST. Our comparative analysis reveals that while these models are benchmarks in code representation, they exhibit limitations when tasked with regression. To address these deficiencies, we propose a novel dual-transformer approach that operates on both source code tokens and AST representations, employing cross-attention mechanisms to enhance interpretability between the two domains. Furthermore, we explore the adaptation of Graph Neural Networks (GNNs) to this tree-based problem, theorizing the inherent compatibility due to the graphical nature of ASTs. Empirical evaluations on real-world datasets showcase that our dual-transformer model outperforms all other tree-based neural networks and the GNN-based models. Moreover, our proposed dual transformer demonstrates remarkable adaptability and robust performance across diverse datasets.

Index Terms—Graph Neural Networks (GNNs), Tree-Based Neural Networks (TreeNN), Transformers.

I. INTRODUCTION

Deep learning models have widely used in the source code analysis for various tasks such as classification of source code [1]–[3], detection of code clones [4]–[6], identification of bugs [7]–[9], and generation of code summaries [10]–[12].

Source code can be represented as textual format, thereby encapsulating the lexical content of the code [13]. Through the textual representation of the source code we can extract the lexical information. For that aim, most traditional approaches to processing source code often adopt string-based

pattern matching, rule-based model transformation, and bag-of-words [14]. However, these methods treat code fragments as plain texts, which ignore the underlying semantic information in source code, resulting in poor performance.

Source code can be represented as a tree throughout the abstract syntax tree (AST) [13]. Thus, the code can be parsed to the tree directly without prior execution. AST representation is abstract and does not include all available details, such as punctuation and delimiters. Theoretically, ASTs can be used to illustrate the syntactic structure of source code, such as the function name and the flow of the instructions (for example, in an if or while construct).

Source code can also be represented as a graph which explains the semantic information from the source code [13]. The graph-structured representations can only be extracted via the intermediate representation or bytecode [15] (e.g. control flow graphs which describe the sequence in which the instructions of a program will be executed [16], data flow graphs which follow and tracks the usage of the variables through the program [16], call flow graphs which captures the relation between a statement which calls a function and the called function [17]), which means that the code fragments have to be compiled successfully. However, the graphs may contribute to enriching code representations. Unfortunately, arbitrary code fragments or incomplete code snippets usually lose the import libraries or dependency packages, making the compilation fail. Such a limitation may make a large number of labelled code snippets unavailable for training, hindering the application of graph representations in practice [18].

That is why, in our study, we will focus on trees as they are easier to extract since we just need to parse the source code.

Recently, some approaches combined neural networks and ASTs to constitute tree-based neural networks (TNNs) [19]. Given a tree, TNNs learn the vector representation by recursively computing node embeddings in a bottom-up

way. Popular TNN models are the Recursive Neural Network (RvNN) [20], Tree-based convolutional neural networks (TBCNN) [3], and Tree-based Long Short-Term Memory (Tree-LSTM) [21]. However, most TBNN approaches tackle a classification problem for the source code but not regression tasks. Regression tasks such as source code performance prediction (predicting the execution time for the application prior to running it) can give the developer an early indication of the runtime behaviour of their source code. Samoaa et al. [22] indicate that trying the TNNs approaches for regression tasks will lead to poor efficiency compared to classification tasks. That said, these solutions are not generic enough for any source code analysis tasks. Thus, to understand the behaviour of TBNN models in regression tasks, we design an analytical framework that uses the benchmark TBNN models for source code analysis to prove the claim that these models are efficient in classification tasks but in a regression context. The TBNN models that are used in our framework are code2vec [23], TBCNN [3], and Transformer-based networks over AST(TBAST) [18], taking into account that we have to make some changes in the architecture of these models to fit the regression tasks. Additionally, we explore various GNN architectures, focusing on neighbourhood information sampling and aggregation within the AST, to further enrich our analysis framework.

To address the lack of efficiency of these TBNN models in a regression context, we develop our model based on cross-attention dual transformers, which utilize sequences of source code tokens and AST nodes. By employing cross-attention mechanisms between the two transformers, our model aims to elucidate the influence of individual source code tokens on AST nodes, enhancing the understanding of code semantics.

Then, we analyse the behaviour of each type of architecture (convolution, sequence, and GNN) for different levels of information: node level (for every node in the AST) as in TBCNN, GNNs, and sequential transformers or path level (a path in AST, which is a sequence of nodes) as in code2vec. Since we have a regression value for each source code program, we have to map each AST to the regression value. Thus, the AST has to be represented as one vector. For that aim, we will aggregate the node and path representations through the models into one continuous vector. To increase the reliability, we use two different real-world datasets of performance measurements. The first dataset (*OSSBuild*) is real build data collected from the continuous integration systems of four open-source systems. The second (*HadoopTests*) is a larger dataset which we have collected ourselves by repeatedly executing unit tests from the Hadoop open-source system in a controlled environment.

The key findings of our experiments show that our dual-transformers model consistently outperformed traditional TBNN and GNN models across various metrics, including Mean Squared Error (MSE), Mean Absolute Error (MAE), and Pearson correlation. This superiority was observed in both dataset contexts (OssBuilds and Hadoop) and under different experimental setups, such as varying training sizes and cross-dataset transferability. In addition, The dual-transformers model demonstrated remarkable adaptability and robust performance across diverse datasets. This model effectively handled

the complexity and variability of datasets differing in size and composition, indicating their potential for general application not only in source code analysis but also in other tree data domains. The study also underscored how the characteristics of datasets, such as the diversity of the data and the structure of ASTs, significantly affect the performance of neural network models. This was evident from the varying performances of models on the OssBuilds dataset, which comprises data from multiple projects, compared to the Hadoop dataset, which is more homogeneous.

The aforementioned key findings highlight the potential of our study in understanding the behaviour of different types of neural network architectures for regression tasks. Our contributions are manifold and address several gaps in the current landscape of tree-based neural network methodologies for regressions:

- 1) **Novel Transformer-Based Model for Tree Learning:** We addressed the inefficiency of different used models for tree and regression by designing and implementing a model-centric AI for employing both code tokens and tree nodes in the transformer based on cross-attention.
- 2) **Development of Specialized Tree Datasets:** We propose new tree datasets designed to be directly usable by researchers, facilitating further exploration and validation of tree-based models.
- 3) **Novel Framework for Analyzing the Behaviour of Different Tree-Based Neural Networks Models :** We provide the research community with an open-source framework that merges all TBNN models with our dual-transformers model. So, researchers can directly use this framework for different tasks and research. The code files are available on GitHub https://github.com/petersamoaa/Tree_based_NN_Error_analysis, and the data files are available on Zenodo [24]

II. BACKGROUND

A. Abstract syntax trees

Abstract Syntax Trees (ASTs) offer a hierarchical representation of source code that abstracts away from the specific syntactic form, focusing instead on the underlying structure and logic of the code. This representation discards superficial elements like punctuation, concentrating on the nodes that signify the fundamental constructs of programming languages, such as variables, operators, and control structures.

An AST encapsulates the syntactic structure of code, where each node represents a construct occurring within the source code [25]. The tree's edges outline the relationship between these constructs, effectively mapping out the syntax rules of the language. The root of the tree often represents the entire program, with leaves corresponding to atomic elements like literals or variable names and internal nodes representing operator or control statements that dictate the flow of the program [13].

Transforming source code into an AST involves parsing, where the code is analyzed according to the grammar of the programming language, and its structure is broken down into a tree that reflects the hierarchical composition of the code's

elements. This process facilitates various code analysis tasks by providing a structured and navigable representation of the code, enabling more sophisticated and accurate analyses than linear source code examination.

ASTs are instrumental in various applications, from code compilation and optimization to more advanced analyses like static code analysis, refactoring, and understanding program behaviour. By providing a structured view of code, ASTs allow tools and developers to examine the abstract properties of the program without getting bogged down by syntactic details irrelevant to the analysis at hand.

In the context of programming language analysis, especially with the advent of machine learning techniques, ASTs serve as a crucial bridge between source code and its semantic understanding. They enable the application of advanced analytical models that can learn from the structural patterns of code to perform tasks such as bug detection, code summarization, and even automated code generation [13].

B. Motivation Example

To have a deeper understanding of the AST, this section explains by example how the source code can be represented as AST. Thus, we investigate Java source code files (see Listing 1).

Listing 1. A Simple JUnit 5 Test Case

```

package org.myorg.weather.tests;

import static
    org.junit.jupiter.api.Assertions.assertEquals;
import org.myorg.weather.WeatherAPI;
import org.myorg.weather.Flags;

public class WeatherAPITest {

    WeatherAPI api = new WeatherAPI();

    @Test
    public void testTemperatureOutput() {
        double currentTemp = api.currentTemp();
        Flags f = api.getFreezeFlag();
        if (currentTemp <= 3.0d)
            assertEquals(Flags.FREEZE, f);
        else
            assertEquals(Flags.THAW, f);
    }
}

```

In this example, a single test case `testTemperatureOutput()` is presented that tests a feature of an (imaginary) API. As common for test cases, the example is short and structurally relatively simple. Much of the body of the test case consists of invocations to the system-under-test and calls of JUnit standard methods, such as `assertEquals`.

A (slightly simplified) AST for this illustrative example is depicted in Figure 1. The produced AST does not contain purely syntactical elements, such as comments, brackets, or code location information. We make use of the pure Python Java parser `javalang`¹ to parse each test file and use the node types, values, and production rules in `javalang` to describe our ASTs.

¹<https://pypi.org/project/javalang/>

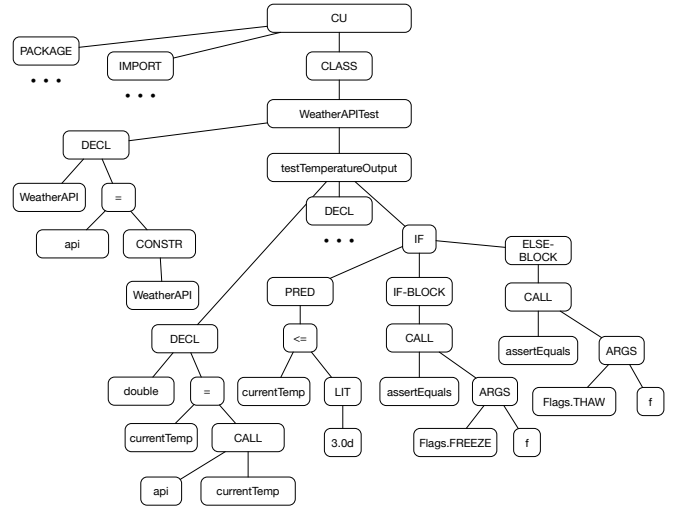


Fig. 1. Simplified abstract syntax tree (AST) representing the illustrative example presented in Listing 1. Package declarations, import statements, as well as the declaration in Line 15 are skipped for brevity.

III. RELATED WORK

The application of deep learning techniques to tree-structured representations of source code has garnered considerable attention within the research community. Mou et al. [3], [26] introduced a novel approach leveraging tree-based Convolutional Neural Networks (CNNs) to perform convolutional computations on Abstract Syntax Trees (ASTs) for code classification tasks. Similarly, Zhu et al. [27] employed tree-based Long Short-Term Memory (LSTM) networks to encode AST pairs into continuous vectors, facilitating code clone detection by measuring similarities.

Further exploring tree-based neural networks, Zhang et al. [19] utilized Recursive Neural Networks (RvNNs) to process ASTs at the path level, targeting classification objectives. Concurrently, While et al. [20] applied RvNNs to analyze ASTs at the node level for classification purposes, a method paralleled by Wei et al. [21] through the use of Labeled AST (LAST) structures.

Innovatively, Zhang et al. [28] introduced a transformer-based model that incorporates tree-based position embeddings to represent the nodes within ASTs, enhancing the classification of source code by learning from code tokens.

Beyond classification, the generation of source code has also been explored. A novel pre-trained model, TreeBERT [29], adapts the BERT architecture to understand programming languages through AST analysis, focusing on path-level node position embeddings for code summarization tasks. Yang et al. [30] further this exploration by proposing the use of multi-modal transformers, analyzing ASTs at the node level for code summarization.

In the realm of regression tasks, the work of Samoa et al. [22] stands out by applying Graph Neural Networks (GNNs) to augmented ASTs, representing a pioneering effort in leveraging tree-based neural network models for regression in source code analysis.

Despite the proliferation of deep learning methodologies for analyzing source code through AST representations, there remains a gap in the literature concerning the comparative analysis of different architectural approaches, particularly in the context of regression tasks. This study aims to bridge this gap by examining the behaviour and efficacy of various tree-based neural network models in regression scenarios.

IV. ANALYTIC FRAMEWORK

According to Samoaa [13], the majority of the deep-learning-based approaches for source code follow the same pipeline as in Figure 2. Thus, the approaches start with parsing the code into AST through the AST parser. Our study uses a Python Java parser `javalang`² as a parser that produces AST from the source code. AST represent the syntactic features. Then, deep learning models like recurrent neural networks (RNNs) or convolutional neural networks (CNNs) are used to encode the nodes of AST into vectors for downstream tasks like classification and regression.

These approaches have three major limitations: 1) RNN models inevitably suffer from the gradient vanishing problem, meaning that the gradients become vanishingly small values during model training, especially in the context of usage of AST which is very deep in most cases [31]; 2) CNN models cannot capture the long-distance dependency information from sequential nodes of tokens in AST due to the size limitation of the sliding windows, which scan only a few nodes/tokens at a time [31]; 3) apart from using the simplistic lexical features, the approaches for AST processing that recursively traverse entire trees from bottom to top may produce longer sequential inputs than the textual inputs, consume large amounts of computational resources and destroy the syntactical structures existing in AST [18]

Thus, based on the abovementioned limitations, we will use a transformer as a competitor for sequential models as well as the basis of our approach since also the transformer is the most mature of sequential models for the following reasons:

- **Handling Long Dependencies:** Transformers leverage self-attention mechanisms. This allows them to weigh the importance of different parts of the input sequence directly, regardless of the distance between elements [32], making them well-suited for the hierarchical and complex structures of ASTs.
- **Parallelization:** Transformers do not process data sequentially as RNNs do. Instead, they can process entire data sequences in parallel during training, significantly speeding up the learning process. This is particularly advantageous when dealing with the large and intricate structures of ASTs, where computational efficiency is paramount.
- **Flexibility in Capturing Structural Information:** The self-attention mechanism in transformers can easily adapt to the structured nature of ASTs, capturing both the local and global context within the tree. This flexibility allows for a more nuanced understanding of code semantics

compared to the fixed window size of CNNs or the sequential nature of RNNs.

- **Scalability:** Transformers are highly scalable and capable of handling large input sequences without significantly dropping performance. This makes them ideal for source code analysis, where ASTs can vary widely in size and complexity.

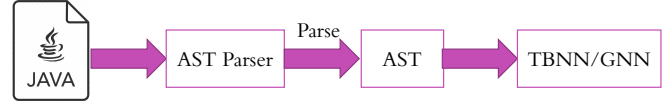


Fig. 2. Abstracted General Code Representation and DL Models in Software Engineering.

Despite its mentioned limitation, we will also use CNN in our framework to have diverse architecture types of neural networks.

V. DUAL TRANSFORMER MODEL

Most models use attention except the TBCNN. The main novelty of our work through the designing and developing of our approach, as well as the comparison with benchmark models, is the understanding that attention mechanism over multiple contexts is needed for embedding programs into a continuous space, and the use of this embedding to predict properties of a whole code snippet.

Our Dual-Transformer model is designed to integrate structural and lexical information within the source code to predict execution time. As illustrated in Figure 3, the architecture consists of two parallel transformer encoders: the NLEncoder for processing source code tokens and the ASTEncoder for processing AST node that the outputs of both encoders are integrated via a cross-attention mechanism, which allows the model to jointly consider textual and structural information.

A. NL-Encoder

The NL-Encoder serves to encode textual information from source code tokens. Input tokens x_{code} are transformed into embeddings E_{code} via a learned embedding matrix W_{code} , combined with positional encodings P_{code} to retain sequential information:

$$E_{code} = W_{code} \cdot x_{code} + P_{code} \quad (1)$$

These embeddings then pass through a series of transformer blocks, each comprising a multi-head self-attention mechanism and a position-wise feed-forward network. For the i^{th} block, the output O_i is computed as follows:

$$O'_i = \text{LayerNorm}(E_{code} + \text{MultiHead}(E_{code}, E_{code}, E_{code})) \quad (2)$$

$$O_i = \text{LayerNorm}(O'_i + \text{FFN}(O'_i)) \quad (3)$$

Where `LayerNorm` denotes layer normalization, `MultiHead` denotes the multi-head self-attention mechanism and `FFN` represents the feed-forward network. The embeddings are subsequently refined by transformer layers, with the output of the final layer being denoted as O_{code} .

²<https://pypi.org/project/javalang/>

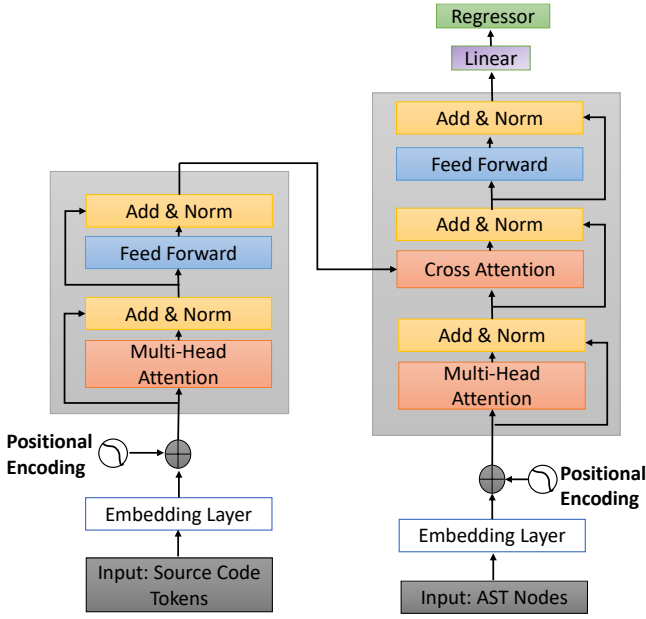


Fig. 3. The architecture of the Dual-Transformer model. The framework features two transformer encoders: NLEncoder for source code tokens and ASTEncoder for AST nodes, each with layers for embedding, multi-head attention, and feed-forward networks, complemented by add & norm layers for stabilization. Their outputs are merged via cross-attention and passed to a linear regressor for error prediction, leveraging both textual and syntactical insights.

B. AST-Encoder

The ASTEncoder parallels the NLEncoder in structure but operates on the AST's nodes. Similar to the NLEncoder, AST node inputs x_{ast} are embedded into vectors E_{ast} and supplemented with positional encoding:

$$E_{ast} = W_{ast} \cdot x_{ast} + P_{ast} \quad (4)$$

These embeddings are then processed through analogous transformer blocks, yielding a structured representation of the code's syntax as O_{ast} .

C. Attention Mechanisms

The crux of our model lies in the cross-attention mechanism that bridges the NLEncoder and ASTEncoder. For each pair of encoded sequences O_{code} and O_{ast} , cross-attention is computed as:

$$\begin{aligned} \text{CrossAttention}(O_{code}, O_{ast}) &= \text{Attention}(O_{code}W_{cross}^Q, O_{ast}W_{cross}^K, O_{ast}W_{cross}^V) \\ &= \text{softmax} \left(\frac{(O_{code}W_{cross}^Q)(O_{ast}W_{cross}^K)^T}{\sqrt{d_k}} \right) O_{ast}W_{cross}^V \end{aligned} \quad (5)$$

Where the learned weight matrices W_{cross}^Q , W_{cross}^K , and W_{cross}^V are central to the model's ability to integrate the outputs of the NLEncoder and ASTEncoder. These matrices transform the final layer outputs of the encoders into the

queries (Q), keys (K), and values (V) needed for the attention calculation. This allows each encoder to attend to the outputs of the other, integrating semantic and syntactic information into a unified representation.

D. Regression Head

At the top of the model, a regression head is applied to integrate the representation (z) of the output of ASTEncoder for error prediction:

$$\hat{y} = \text{Linear}(\text{ReLU}(\text{Linear}(z))) \quad (6)$$

Where z represents the result produced by the first token "[CLS]" from the ASTEncoder, which is designed to summarize the overall context of the input sequence.

VI. OTHER GNN AND TBNN MODELS

This section introduces the benchmark models against which our dual transformers model is evaluated. This includes discussing GNN-based models, a convolutional model leveraging tree structures, a sequential model transformer-based, and the path-attention mechanism employed by code2vec. Each approach offers a unique perspective on source code analysis through AST, setting the stage for a comprehensive comparative study.

A. Graph Learning Approach

Graph Neural Networks have demonstrated promise in various real-world applications [33]–[40]. Two primary models have played a pioneering role in the field, establishing the foundational frameworks for two key approaches to graph processing: the recurrent model proposed by Scarselli et al. [41] and the feedforward model introduced by Micheli [42]. Notably, the feedforward approach has evolved into the prevailing method [43]–[47].

In this section, we will explain the graph neural network architectures that we used in our experiment. The models accept the AST as an input and predict a scalar execution time value.

a) *GCN (Graph Convolutional Network)*: GCNs [43] leverage the concept of convolutional operations on graph-structured data. The model updates the representation of a node by aggregating the features of its neighbours.

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}) \quad (7)$$

Where $H^{(l)}$ is the matrix of node features at layer l , $\tilde{A} = A + I_N$ is the adjacency matrix A with added self-connections I_N , \tilde{D} is the degree matrix of \tilde{A} , $W^{(l)}$ is the weight matrix for layer l , and σ is a non-linear activation function.

b) *GAT (Graph Attention Network)*: GAT [44] introduces the attention mechanism to graph neural networks. It computes the hidden representations of each node by attending to its neighbours, following a self-attention strategy.

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(a^T [Wh_i || Wh_j]))}{\sum_{k \in \mathcal{N}(i)} \exp(\text{LeakyReLU}(a^T [Wh_i || Wh_k]))} \quad (8)$$

$$h'_i = \sigma \left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} W h_j \right) \quad (9)$$

Where h_i is the feature vector of node i , W is a shared linear transformation, a is the attention mechanism's learnable weight, \parallel denotes concatenation, and α_{ij} represents the attention coefficient between nodes i and j .

c) *GraphSAGE (Graph Sample and Aggregation)*: GraphSAGE [46] generates embeddings by sampling and aggregating features from a node's local neighbourhood.

$$h'_i = \sigma (W \cdot \text{MEAN}(\{h_i\} \cup \{h_j, \forall j \in \mathcal{N}(i)\})) \quad (10)$$

Where h_i is the feature vector of node i , $\mathcal{N}(i)$ is the set of its neighbours, and W is the weight matrix associated with the aggregator function.

d) *GIN (Graph Isomorphism Network)*: GIN [47] is designed to capture the power of the Weisfeiler-Lehman graph isomorphism test. It aggregates neighbour information to update node representations, aiming to distinguish graph structures.

$$h'_i = \text{MLP} \left((1 + \epsilon) \cdot h_i + \sum_{j \in \mathcal{N}(i)} h_j \right) \quad (11)$$

Where h_i represents the feature vector of node i , ϵ is a learnable parameter or a fixed scalar, and MLP denotes a multi-layer perceptron.

Since all baselines are used for classification, we changed the models to fit the regression tasks.

B. Tree-based CNN (TBCNN)

TBCNN models [3] are designed to process the structured data of an AST by leveraging convolutional layers tailored for tree structures. This approach involves several key components:

- **Representation Learning for AST Nodes:** Each AST node is represented as a distributed vector capturing the symbol features.

$$\vec{p} \approx \tanh \left(\sum_i l_i \mathbf{W}_{\text{code},i} \cdot \vec{c}_i + \mathbf{b}_{\text{code}} \right) \quad (1)$$

Where:

\vec{p} is the parent node's vector representation. l_i is a coefficient based on the subtree's leaf count. $\mathbf{W}_{\text{code},i}$ is learned weight matrices. \vec{c}_i and \vec{x}_i represent the children nodes' vectors.

- **Coding Layer:** This layer encodes the representation of a node by aggregating the features of its children through a learned transformation.

$$\vec{p} = \mathbf{W}_{\text{comb1}} \cdot \vec{p} + \mathbf{W}_{\text{comb2}} \cdot \tanh \left(\sum_i l_i \mathbf{W}_{\text{code},i} \cdot \vec{c}_i + \mathbf{b}_{\text{code}} \right) \quad (2)$$

where: $\mathbf{W}_{\text{comb1}}$ and $\mathbf{W}_{\text{comb2}}$ are learned weight matrices.

- **Tree-based Convolutional Layer:** A set of convolutional filters or kernels is applied over the AST to capture the hierarchical structure of the code.

$$\mathbf{y} = \tanh \left(\sum_i \mathbf{W}_{\text{conv},i} \cdot \vec{x}_i + \mathbf{b}_{\text{conv}} \right) \quad (3)$$

where $\mathbf{W}_{\text{conv},i}$ is learned weight matrices. \mathbf{b}_{code} and \mathbf{b}_{conv} are bias terms. \mathbf{y} is the output vector after applying the convolution operation.

- **Dynamic Pooling:** This layer aggregates the convolutional features from different parts of the AST to handle varying sizes and shapes.
- **The "Continuous Binary Tree" Model:** It addresses the challenge of AST nodes having varying numbers of children by considering each subtree as a binary tree during convolution.

This representation captures the essence of how TBCNNs operate on ASTs to learn meaningful representations of source code for various tasks such as program classification and pattern detection. However, we modified the model's architecture to fit regression tasks.

C. code2vec Path-Attention Model

The code2vec model operates on the principle of transforming code snippets into a distributed vector representation. It achieves this by embedding the paths and terminal nodes of AST and using an attention mechanism to identify and aggregate the most relevant features. The model can be decomposed into several components:

- **Embedding Vocabularies:** Two embedding matrices, $\text{value_vocab} \in \mathbb{R}^{|X| \times d}$ and $\text{path_vocab} \in \mathbb{R}^{|P| \times d}$, where $|X|$ is the number of unique AST terminal node values and $|P|$ is the number of unique AST paths observed during training. The embedding size d is a hyperparameter typically ranging between 100 and 500.
- **Context Vectors:** A path-context b_i is a triplet $\langle x_s, p_j, x_t \rangle$ representing the start and end tokens of a path in the AST and the path itself. Each component of b_i is mapped to its embedding and concatenated to form a context vector $c_i \in \mathbb{R}^{3d}$:

$$\begin{aligned} c_i &= \text{embedding} \langle x_s, p_j, x_t \rangle \\ &= [\text{value_vocab}[s], \text{path_vocab}[j], \text{value_vocab}[t]] \in \mathbb{R}^{3d} \end{aligned} \quad (12)$$

- **Fully Connected Layer:** Each context vector c_i is transformed by a fully connected layer with weights $W \in \mathbb{R}^{d \times 3d}$ and a tanh non-linearity to produce a combined context vector \tilde{c}_i :

$$\tilde{c}_i = \tanh(W \cdot c_i) \quad (13)$$

- **Attention Mechanism:** The attention mechanism computes a weighted average of the combined context vectors \tilde{c}_i , using an attention vector $a \in \mathbb{R}^d$ which is learned during training. The attention weight α_i for each \tilde{c}_i is computed using the softmax function:

$$\alpha_i = \frac{\exp(\tilde{c}_i^\top \cdot a)}{\sum_{j=1}^n \exp(\tilde{c}_j^\top \cdot a)} \quad (14)$$

- **Aggregated Code Vector:** The final code vector $v \in \mathbb{R}^d$ representing the entire code snippet is calculated as a weighted sum of the combined context vectors:

$$v = \sum_{i=1}^n \alpha_i \cdot \tilde{c}_i \quad (15)$$

The model learns to assign an appropriate amount of attention to each path context, effectively capturing the semantics of the code snippet. The final code vector can be used for various downstream tasks, such as method name prediction, with the attention weights offering insight into the model's decision process.

D. Transformer-based Networks for AST

This approach splits the deep ASTs into smaller subtrees that aim to exploit syntactical information in code statements. Then, the model gets the sequence of nodes of each subtree to eventually have a sequence of nodes of a sequence of subtrees. Thus, the transformer-based models are particularly adept at considering the sequential nature of code through the use of positional embeddings and self-attention mechanisms, drawing inspiration from their success in natural language processing tasks. This model utilizes multiple layers of self-attention and feed-forward networks to process data. The model can be mathematically described as follows:

- **Multi-Head Self-Attention:** The self-attention mechanism allows the model to weigh the importance of different tokens within the input sequence differently. This is done using queries (Q), keys (K), and values (V), which are derived from the input embedding matrix $X \in \mathbb{R}^{n \times d}$:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V, \quad (16)$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d \times d}$ are parameter matrices. The output of the attention function for a single head is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \quad (17)$$

where d_k is the dimension of the keys.

In the case of multi-head attention, the above computation is done in parallel for each head, and the outputs are concatenated and linearly transformed:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_n)W^O, \quad (18)$$

where each head is computed as $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$ and W^O is another parameter matrix.

- **Position-wise Feed-Forward Networks:** Each transformer block contains a position-wise feed-forward network, which applies two linear transformations with a ReLU activation in between:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2, \quad (19)$$

where W_1, W_2 and b_1, b_2 are parameters of the layers.

- **Layer Normalization and Residual Connections:** Each sub-layer in a transformer, including self-attention and feed-forward networks, has a residual connection around it followed by layer normalization:

$$\text{LayerNorm}(x + \text{Sublayer}(x)). \quad (20)$$

- **Output Layer:** The output of the transformer is typically taken from the first token's representation and passed through a final dense layer for classification tasks:

$$o = \text{softmax}(x_0W + b), \quad (21)$$

Where x_0 is the transformed embedding of the first token and W, b are parameters of the output layer.

VII. EXPERIMENT

A. Experiment Settings

In this experiment setup, all GNN-based models consist of two convolution layers with hidden dimensions of 40 and 30, followed by two linear layers. To facilitate graph prediction, node representation pooling was employed by concatenating mean and max global pooling techniques. Batch normalization and dropout techniques were applied for training regularization. All models were implemented using PyTorch-Geometric [48].

To standardize our experimental conditions across various utilised models, including TreeCNN, Code2Vec, Transformer-Based, and Dual-Transformer, we trained each for hundred epochs five times with different initialization seeds at a learning rate of 1×10^{-4} and a batch size of four. However, each model requires specific parameters to function optimally. For example, the TreeCNN model uses a node representation embedding size of 100 and a hidden layer size of 300. The Code2Vec model, employing a pre-trained version, was initially trained with 200 distinct contexts and had extensive vocabulary sizes for tokens and paths, set at 1,301,136 and 911,417, respectively, with an embedding size of 128.

We assess both scaled-down (small) and fully scaled versions (large) for the Transformer models. The scaled-down version includes a single transformer block with 768 hidden units, eight self-attention heads, and a maximum sequence length of 2,048 tokens, while the fully scaled version consists of 12 transformer blocks. The implementation of Transformer models utilized the Huggingface library.

By maintaining consistent training epochs, learning rates, and batch size across models and adjusting configurations to meet each model's architectural requirements, our experiment aims to deliver a balanced and comprehensive evaluation of models' performance across various metrics.

B. Dataset Collection

In our experiments, to increase reliability, we use two different real-world datasets of performance measurements. The first dataset (*OSSBuild*) is real build data collected from the continuous integration systems of four open-source systems. The second (*HadoopTests*) is a larger dataset we have collected ourselves by repeatedly executing the unit tests of the Hadoop open-source system in a controlled environment. A summary of both datasets is provided in Table I. In the following subsections, we provide some additional information about each of the two datasets that we used in the experimental studies.

1) *OSSBuild Dataset*: In this dataset (originally used in Samoa et al. [22]), information about test execution times in production build systems was collected for four open-source projects: systemDS, H2, Dubbo, and RDF4J. All four projects use public continuous integration servers containing (public) information about the project’s builds, which we harvested for test execution times as a proxy of performance in summer 2021. Basic statistics about the projects in this dataset are described in Table I (top). “Files” refers to the number of unit test files we collected execution times for, “Runs” is the (total) number of executions of files we extracted data for, whereas “Nodes” and “Vocab.” indicate the resulting trees. Prior to parsing the test files, we remove code comments to reduce the number of nodes in each tree (by construction, irrelevant). We notice that across 922 ASTs, we have almost 867,000 nodes with 36880 distinct labels as vocabs.

2) *HadoopTests Dataset*: To address limitations with the OSSBuilds dataset (primarily the limited number of files for each individual project in the dataset) [49], we additionally collected a second dataset for this study. We selected the Apache Hadoop framework since it entails a large number of test files (2895) of sufficient complexity. We then executed all unit tests in the project five times, recording the execution duration of each test file as reported by the JUnit framework (in millisecond granularity). As an execution environment for this data collection, we used a dedicated virtual machine running in a private cloud environment, with two virtualized CPUs and 8 GByte of RAM. Following performance engineering best practices, we deactivated all other non-essential services while running the tests. Statistics about the HadoopTests dataset are described in Table I (bottom). Since we have more files in HadoopTests, there are more nodes. Thus, ASTs for HadoopTests have almost 5 million nodes and almost 139,000 vocabs.

To better understand our dataset, Table II shows the average statistics of the input ASTs. In particular, we report the average number of nodes ($|V|$), the average number of edges ($|E|$), the diameter, and density. The data in Table II reveals key structural features of the ASTs in our datasets. The near-equal count of nodes ($|V|$) and edges ($|E|$) underscores the tree-like nature of ASTs, where most nodes are directly connected to only one parent. The substantial diameters indicate deep trees, suggesting complex nested code structures. Low-density points to sparse connectivity, emphasizing the depth over breadth in these ASTs, which could affect the performance of neural

models that process such hierarchical data.

VIII. RESULTS

In this section, we delineate the performance outcomes of our proposed model alongside those of competing frameworks, scrutinized from three distinct angles: initial efficacy across the OSSBuilds and Hadoop datasets on a standard data split, variations in model efficiency with increasing sizes of training data, and the adaptability of models through cross-dataset transferability assessments. The evaluation metrics include MSE, MAE, and Pearson correlation coefficient (Cor.), with lower MSE and MAE values indicating better performance and a higher correlation coefficient signifying a stronger relationship between predicted and actual execution times.

A. Models Performance Evaluation

This section presents a comprehensive analysis of our experimental results, comparing the performance of various models on the standard split of datasets. Thus, for five different seeds, we split the data into 80% used for training and the rest 20% for testing, and then we report the average results across the seeds.

a) *GNN models*: As presented in Table VIII-A, among the GNN architectures, GraphSage exhibits superior performance on the OSSBuilds dataset, with the lowest MSE of 0.06 and MAE of 0.20, alongside a commendable correlation of 0.68. However, on the Hadoop dataset, all GNN models demonstrate relatively similar MSE scores of 0.06 (except for GAT). As for MAE and Correlation scores, both GCN and GraphSage have similar scores, 0.21 and 0.52, respectively, with GraphSage maintaining a slight edge in standard deviation (STD). It is worth mentioning that the GIN model performs well for the Hadoop dataset with the best MAE score.

b) *TBNN Models*: Transitioning to the TBNN competitors, TreeCNN TreeCNN emerges as a strong contender in all metrics (particularly in Pearson correlation), showcasing the best scores among TBNN models for both datasets, with the lowest MSE of 0.03, an MAE of 0.13 and the highest correlation score of 0.57 on the OSSBuilds dataset and 0.02, 0.12, and 0.57 for MSE, MAE, and Pearson correlation On the Hadoop dataset. It is worth mentioning that Code2Vec is a good competitor with a very small margin of difference from TreeCNN in both datasets, especially for error metrics. As for transformer-based, it achieves the worst results, especially in the large setting on the OSSBuilding dataset, explaining that the model is too complex for this dataset. In contrast, the efficiency of the same model with the same setting is improving in the Hadoop dataset, which is the larger dataset with more complex trees.

c) *Our Dual-Transformer Model*: Our proposed Dual-Transformer model (in both large and small settings) significantly outperforms both the GNN architectures and TBNN competitors across all metrics on both datasets. It achieves the best MSE of 0.01 and 0.02 and an outstanding MAE of 0.08 and 0.09 on the OSSBuilds dataset, coupled with a remarkable correlation of 0.85 and 0.83 for small and large settings, respectively. Although the Hadoop dataset shares the best MSE

TABLE I
OVERVIEW OF THE OSSBUILDS AND HADOOPTESTS DATASETS.

Project	Description	Files	Runs	Nodes	Vocab.
sysDS	Apache ML for Data Science lifecycle	127	1321	114904	3205
H2	Java SQL DB	194	1391	432375	18326
Dubbo	Apache Remote Procedure Call framework	123	524	77142	4505
RDF4J	Scalable RDF	478	1055	242673	10844
(OSSBuilds) Tot.		922	4291	867094	36880
Hadoop	Apache framework for big data	2895	24348	5090798	138952

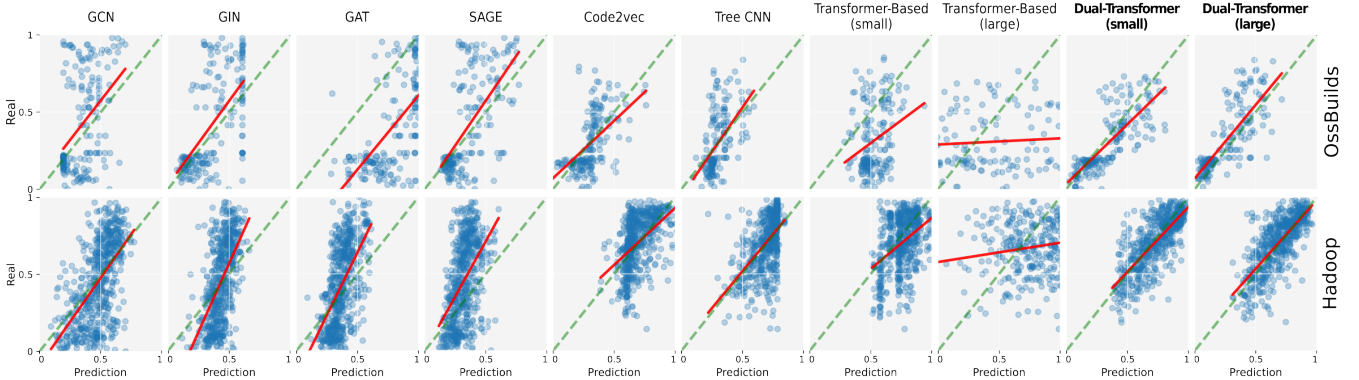


Fig. 4. **Real vs predicted values** Each panel reports the real (y-axes) and predicted (x-axes) values for each model. Each pair that is real-predicted is represented as a blue point, while the dashed red line shows a linear regression model fitted to the data.

TABLE II
AVERAGE STATISTICS OF THE INPUT TREES.

Dataset	$ V $	$ E $	Diameter	Density
OSSBuilds	875	874	17	0.015
HadoopTests	1490	1489	19	0.006

score of 0.02 with TreeCNN, its MAE of 0.12 and 0.11, as well as the correlation of 0.67 and 0.72, is unparalleled, underscoring its superior predictive capability and efficiency in capturing the underlying complexities of source code.

d) Models' Prediction Trending Analysis: In Figure 4, a correlation chart is presented for the OssBuilds dataset, each panel depicting a distinct model. The figure clearly indicates that the GNN-based models struggle to predict real values accurately. Notably, it appears that all GNN-based models exhibit a tendency to predict values that are either close to one or close to zero. The Transformer-based, TreeCNN, and Code2Vec models encounter challenges in predicting larger values. While our model also faces difficulty in predicting larger values, however, it outperforms the other approaches.

e) Conclusion: These results in Table VIII-A underscore the efficacy of our Dual-Transformer approach, particularly in its ability to harness the syntactic and semantic intricacies of ASTs for source code analysis. The significant improvement in correlation coefficients highlights the model's adeptness at understanding the nuanced relationships within the code, making it a promising tool for developers seeking early insights into the potential execution characteristics of their programs. When compared with transformer-based models, our dual transformer has displayed superior performance, which can

be attributed to its specialized architecture designed to handle dual input modalities. On the other hand, the Transformer-based model may not effectively capture the interplay between different types of input data, such as textual and structural representations in programming code. Furthermore, the transformer-based model may have limitations in dealing with extended sequence lengths compared to the dual transformer.

GNN models seem to be the second-best regarding Pearson scores for both datasets (except for TreeCNN in Hadoop). However, both error metrics for code2vec and TreeCNN models are better for both datasets compared with GNN models.

Regarding trending results for the models between OssBuilding and Hadoop, only the TreeCNN model has an upward trend. In contrast, the scores for the rest of the models have decreased in the Hadoop dataset compared to OssBuilding.

In conclusion, the empirical evidence strongly supports adopting our dual transformer model for the regression task. By effectively leveraging both the lexical and syntactic features of the source code, our approach establishes a new state-of-the-art performance, paving the way for future research in this domain.

B. Assessing Model Efficiency Across Incremental Training Data Sizes

Given the expense associated with data collection, this section delves into the effectiveness of our and other models when trained on a reduced dataset. Specifically, we employ random selection to allocate 20%, 40%, and 60% of the dataset for training the models, while a fixed 20% of the remaining portion is designated for testing. Through the outcomes shown in Table IV, we observed distinct patterns of performance

TABLE III
TEST MSE, MAE, AND PEARSON CORRELATION FOR BOTH DATASETS TREES

	OssBuilds			Hadoop		
	MSE ↓	MAE ↓	Cor. ↑	MSE ↓	MAE ↓	Cor. ↑
GCN	0.07±0.02	0.21±0.03	0.65±0.04	0.06±0.02	0.21±0.03	0.52±0.05
GAT	0.07±0.01	0.23±0.02	0.61±0.03	0.09±0.01	0.25±0.02	0.38±0.07
GIN	0.08±0.01	0.21±0.01	0.60±0.04	0.06±0.003	0.20±0.01	0.50±0.04
GraphSage	0.06±0.01	0.20±0.02	0.68±0.02	0.06±0.01	0.21±0.02	0.52±0.03
Code2Vec	0.03±0.003	0.14±0.01	0.44±0.11	0.03±0.002	0.14±0.01	0.28±0.05
TreeCNN	0.03±0.002	0.13±0.004	0.51±0.03	0.02 ± 0.001	0.12±0.01	0.57±0.03
Transformer-Based (small)	0.09±0.04	0.25±0.07	0.45±0.15	0.08±0.0.03	0.24±0.06	0.27±0.06
Transformer-Based (large)	0.46±0.25	0.56±0.20	0.30±0.10	0.05±0.01	0.17±0.03	0.40±0.07
Dual-Transformer (small)	0.01 ± 0.002	0.08 ± 0.006	0.85 ± 0.02	0.02 ± 0.01	0.12±0.03	0.67±0.04
Dual-Transformer (large)	0.02±0.006	0.09±0.02	0.83±0.03	0.02±0.004	0.11±0.01	0.72±0.03

adaptation as the models were exposed to increasing proportions of the dataset.

a) *GNN models*: For the OssBuilds dataset, GNN models generally demonstrated an expected trend: improvements in MSE (except for GIN), MAE (except for GIN and GraphSage), and Pearson correlation (except GCN, GIN) as the training data size expanded from 20% to 60%. Except for GIN, all other GNN models have stability in MSE score in the training data size of 40% to 60%. Moreover, the GAT model also has stability in MAE for both 20% and 40% of training data.

For the Hadoop dataset, GNN models tend to have better error metrics (except for GAT) across all sizes used in training. However, the opposite is true when it comes to Pearson’s score. Similar to OssBuilds, there is some stability in error metrics scores as in GIN in 20% and 40% and GraphSage in 40% and 60% sizes of training data. As for GCN, the MAE is stable across all training sizes. As for GAT, we got the same MSE score when we used 20% and 40% for training. Despite the stability in error metrics most of the time, the prediction correlation of Pearson correlation still changes through the different sizes, which justifies the importance of adding more data for training. Thus, notably, GNN-based models benefit from more extensive data to better capture the structural nuances of the ASTs. However, their performance plateaus, suggesting a limit to how much simply increasing data can benefit these models without corresponding adjustments in model complexity or architecture

b) *TBNN Models*: For the OssBuilds dataset, the TBNN models show varying degrees of sensitivity to the amount of training data. Code2Vec and TreeCNN exhibit stable performance in terms of MSE and MAE across different training sizes, but Pearson correlation shows a slight increase, indicating better alignment between predicted and actual values with more data. Transformer-based models, both small and large, display inconsistent performance, with significant fluctuations in MSE and MAE and only modest improvements in Pearson correlation, suggesting instability in their learning process with varied data sizes.

In the Hadoop dataset, TBNN models follow a similar trend. Code2Vec and TreeCNN maintain stable error metrics, while the Pearson correlation improves as more training data is provided. The Transformer-Based models show less stability, with large variations in MSE, MAE, and Pearson correlation across different training sizes. This instability indicates that

Transformer-Based models might require more fine-tuning or adjustments in their architecture to better handle varied data sizes.

c) *Our Dual-Transformer Model*: The Dual-Transformer models, our proposed approach, show robust performance improvements across all metrics as the training data size increases. For the OssBuilds dataset, both small and large variants of the Dual-Transformer model exhibit significant decreases in MSE and MAE and substantial increases in Pearson correlation, reaching 0.82 with 60% training data. This indicates the model’s ability to effectively utilize additional training data to enhance predictive accuracy and correlation alignment.

Similarly, in the Hadoop dataset, the Dual-Transformer models demonstrate consistent performance gains. The MSE and MAE decrease steadily, and the Pearson correlation shows marked improvement, reaching 0.73 with 60% training data. This consistent improvement across different data sizes underscores the effectiveness of the Dual-Transformer architecture in handling complex regression tasks on ASTs, leveraging the added data to refine its predictive capabilities.

d) *Conclusion*: The analysis across incremental training data sizes reveals that while GNN models benefit from increased data, their performance gains plateau. TBNN models show varying degrees of stability, with Code2Vec and TreeCNN being more resilient to changes in data size.

The overall error values for the Hadoop dataset were generally smaller than the OssBuilds dataset (except for GAT). On the other hand, Pearson Correlation scores are better for OssBuilds for all models except TreeCNN, suggesting that Hadoop’s complexity and tree structures might pose additional challenges in prediction. Despite this, the Dual-Transformer model consistently outperformed other models, reaffirming its adaptability and efficiency in handling complex trees. In addition, it demonstrates robust scalability and effectiveness in leveraging additional data to enhance predictive accuracy and correlation, making them well-suited for regression tasks for trees. It is worth mentioning that the Dual-Transformer model can be satisfied with 40% of training data to achieve the best performance when it comes to the Hadoop dataset. Still, it consistently needs more data to be utilised in training in OssBuilds, which contains samples from 4 different projects.

These findings underscore the critical importance of data volume in training neural network models for source code

TABLE IV
TEST MSE, MAE, AND PEARSON CORRELATION ACROSS DIFFERENT TRAINING DATA SIZES FOR BOTH DATASETS

Model	OssBuilds								
	20%			40%			60%		
	MSE	MAE	Cor.	MSE	MAE	Cor.	MSE	MAE	Cor.
GCN	0.09±0.02	0.26±0.03	0.51±0.07	0.08±0.01	0.23±0.03	0.63±0.03	0.08±0.01	0.22±0.02	0.62±0.05
GAT	0.08±0.005	0.23±0.02	0.52±0.04	0.07±0.005	0.23±0.02	0.57±0.02	0.07±0.004	0.22±0.02	0.59±0.04
GIN	0.09±0.01	0.24±0.02	0.49±0.06	0.07±0.01	0.21±0.009	0.57±0.05	0.09±0.02	0.24±0.04	0.56±0.08
GraphSage	0.08±0.007	0.24±0.02	0.58±0.005	0.07±0.01	0.22±0.03	0.64±0.03	0.07±0.01	0.23±0.03	0.65±0.04
Code2Vec	0.03±0.002	0.15±0.006	0.33±0.08	0.03±0.002	0.15±0.01	0.35±0.11	0.03±0.002	0.15±0.01	0.39±0.07
TreeCNN	0.03±0.001	0.15±0.005	0.32±0.06	0.03±0.001	0.15±0.003	0.40±0.05	0.03±0.003	0.14±0.01	0.46±0.05
Transformer-Based (small)	0.13±0.04	0.29±0.05	0.37±0.08	0.14±0.09	0.30±0.10	0.39±0.12	0.09±0.05	0.25±0.07	0.34±0.22
Transformer-Based (large)	0.56±0.47	0.61±0.17	0.18±0.11	0.60±0.26	0.62±0.14	0.13±0.12	0.34±0.06	0.45±0.03	0.26±0.05
Dual-Transformer (small)	0.02±0.004	0.10±0.01	0.79±0.04	0.01±0.003	0.09±0.007	0.84±0.03	0.02±0.01	0.10±0.03	0.82±0.05
Dual-Transformer (large)	0.02±0.004	0.11±0.01	0.71±0.04	0.02±0.002	0.09±0.005	0.79±0.05	0.01±0.002	0.09±0.01	0.82±0.02
Model	Hadoop								
	20%			40%			60%		
	MSE	MAE	Cor.	MSE	MAE	Cor.	MSE	MAE	Cor.
GCN	0.07±0.01	0.22±0.02	0.42±0.12	0.08±0.02	0.22±0.03	0.50±0.03	0.07±0.01	0.22±0.02	0.52±0.06
GAT	0.09±0.02	0.25±0.02	0.17±0.17	0.09±0.009	0.24±0.1	0.19±0.09	0.11±0.04	0.27±0.04	0.17±0.22
GIN	0.07±0.02	0.22±0.03	0.36±0.14	0.07±0.01	0.22±0.02	0.46±0.06	0.06±0.006	0.20±0.009	0.48±0.09
GraphSage	0.07±0.003	0.22±0.009	0.35±0.08	0.06±0.008	0.21±0.02	0.57±0.09	0.06±0.01	0.21±0.02	0.55±0.08
Code2Vec	0.04±0.003	0.13±0.006	0.38±0.04	0.03±0.003	0.14±0.01	0.24±0.04	0.03±0.001	0.13±0.004	0.38±0.06
TreeCNN	0.02±0.003	0.12±0.002	0.44±0.03	0.02±0.001	0.12±0.006	0.51±0.02	0.02±0.001	0.11±0.001	0.53±0.01
Transformer-Based (small)	0.09±0.05	0.23±0.08	0.26±0.16	0.10±0.04	0.26±0.07	0.34±0.11	0.07±0.04	0.21±0.07	0.38±0.17
Transformer-Based (large)	0.28±0.09	0.42±0.07	0.14±0.05	0.08±0.01	0.22±0.02	0.29±0.07	0.09±0.06	0.24±0.10	0.34±0.02
Dual-Transformer (small)	0.02±0.002	0.11±0.008	0.70±0.04	0.02±0.005	0.11±0.02	0.72±0.03	0.02±0.005	0.11±0.02	0.68±0.02
Dual-Transformer (large)	0.02±0.007	0.13±0.02	0.67±0.03	0.02±0.002	0.10±0.007	0.72±0.01	0.02±0.002	0.09±0.008	0.73±0.01

analysis. They also highlight the Dual-Transformer model’s superiority in adapting to varied training sizes while maintaining robust performance, marking it as a promising approach for efficient source code analysis.

C. Cross-Dataset Transferability

In this section, we explore the models’ performance in an inductive scenario, where they are trained on one dataset and tested on another dataset [50]. The results, shown in Table V, unfold distinct patterns of performance across the GNN-based models, TBNNs, and our Dual-Transformer model. Due to the differing scales of the machines used to collect the datasets (since the hardware used is one of the factors that affect the execution time), it was imperative to adapt the model to each specific context. To this end, we initially trained the model on one dataset and subsequently fine-tuned it using a small subset of the other dataset to optimize its parameters. This fine-tuning process involved using incremental portions of the test dataset—specifically 10%, 20%, and 30%—to refine the model’s ability to generalize across different operational conditions. The efficacy of the fine-tuning was then evaluated by testing the model on 20% of the test dataset, which is fixed across all portions, ensuring a consistent assessment framework across all experimental conditions. This methodological approach allowed us to rigorously assess the model’s adaptability and performance across datasets characterized by diverse computational environments.

When we **trained the models on Hadoop and fine-tuned and evaluated on Ossbuilds**, models exhibited relatively stable MSE and MAE across all fine-tuning portions (except for the small version of our dual transformer model), indicating a capacity to maintain consistent error rates when

transferring knowledge from a larger (Hadoop) to a smaller dataset (OssBuilds). Pearson correlation showed gradual improvement as the fine-tuning portion increased (except for a small model of transformed-based), highlighting a modest but positive adaptation. The Dual-Transformer models demonstrated the best performance, with significant improvements in correlation and error rates, making them the most adaptable across dataset sizes. Since the error metrics are slightly better for the large version of our dual transformer, the prediction correlation score for the small version, however, is largely better across the usage of all portions of fine-tuning. Conversely, the Transformer-Based models struggled the most, especially in larger configurations, showing limited correlation improvements and variable error rates, suggesting challenges in adapting to the smaller dataset’s nuances. It is worth mentioning that GNN models show more efficient predictions since the Pearson correlation score is better than all other TBNN models. However, this is also the case regarding the error metrics. Compared to other experiments, TBNN, we see a huge decrease in error metrics compared to the previous experiments. However, the error metrics of GNN are somehow stable across all our experiments. As for our model, only the MAE metric increased.

In the scenario where **OssBuilds was used for training and Hadoop for fine-tuning and evaluation**, all models generally showed better MSE and MAE scores compared to the previous setting, especially for TBNN models since both code2vec and TreeCNN competing our model, particularly with a small portion of test data used for fine-tuning. That said, when we train TBNN models on trees for different projects, the models can easily generalized to other trees. However, Pearson correlation is still an issue for these models. GNN models have

TABLE V
TRANSFERABILITY TEST MSE, MAE, AND PEARSON CORRELATION FOR BOTH DATASETS

Model	Train Set = Hadoop & Test Set = OssBuild								
	10%			20%			30%		
	MSE	MAE	Cor.	MSE	MAE	Cor.	MSE	MAE	Cor.
GCN	0.08±0.01	0.23±0.02	0.47±0.02	0.08±0.01	0.22±0.03	0.51±0.02	0.08±0.01	0.22±0.03	0.53±0.02
GAT	0.09±0.01	0.26±0.02	0.35±0.20	0.08±0.01	0.24±0.03	0.40±0.21	0.08±0.01	0.24±0.03	0.41±0.19
GIN	0.08±0.01	0.24±0.02	0.51±0.02	0.08±0.01	0.22±0.02	0.54±0.02	0.07±0.01	0.21±0.02	0.55±0.02
GraphSage	0.08±0.01	0.25±0.02	0.51±0.02	0.08±0.01	0.23±0.03	0.55±0.02	0.08±0.01	0.23±0.03	0.56±0.02
Code2Vec	0.20±0.01	0.37±0.02	0.21±0.09	0.20±0.01	0.37±0.02	0.30±0.10	0.19±0.02	0.36±0.02	0.33±0.09
TreeCNN	0.21±0.01	0.39±0.01	0.14±0.07	0.20±0.005	0.38±0.01	0.25±0.05	0.19±0.005	0.37±0.01	0.32±0.03
Transformer-Based (small)	0.24±0.07	0.39±0.05	0.40±0.10	0.34±0.13	0.44±0.10	0.40±0.17	0.24±0.03	0.40±0.04	0.35±0.15
Transformer-Based (large)	0.64±0.35	0.90±0.60	0.02±0.09	0.88±0.48	0.82±0.04	0.07±0.04	0.42±0.78	0.68±0.41	0.15±0.07
Dual-Transformer (small)	0.11±0.02	0.26±0.02	0.47±0.07	0.08±0.02	0.23±0.02	0.56±0.10	0.06±0.01	0.20±0.01	0.65±0.09
Dual-Transformer (large)	0.05±0.005	0.17±0.006	0.15±0.05	0.04±0.006	0.16±0.01	0.23±0.04	0.03±0.01	0.15±0.04	0.42±0.06
Model	Train Set = OssBuilds & Test Set = Hadoop								
	10%			20%			30%		
	MSE	MAE	Cor.	MSE	MAE	Cor.	MSE	MAE	Cor.
GCN	0.07±0.01	0.22±0.01	0.43±0.05	0.07±0.01	0.21±0.01	0.46±0.04	0.09±0.02	0.24±0.03	0.44±0.05
GAT	0.10±0.01	0.25±0.01	0.37±0.03	0.09±0.01	0.24±0.1	0.43±0.02	0.09±0.01	0.25±0.02	0.44±0.01
GIN	0.09±0.03	0.24±0.03	0.42±0.06	0.08±0.02	0.23±0.03	0.46±0.05	0.07±0.02	0.22±0.02	0.47±0.04
GraphSage	0.07±0.01	0.22±0.01	0.45±0.04	0.07±0.01	0.22±0.01	0.48±0.03	0.06±0.004	0.21±0.005	0.52±0.02
Code2Vec	0.005±0.0005	0.06±0.003	0.16±0.07	0.004±0.002	0.06±0.005	0.38±0.03	0.005±0.0002	0.06±0.004	0.38±0.05
TreeCNN	0.005±0.0004	0.06±0.003	0.30±0.04	0.005±0.001	0.06±0.004	0.38±0.03	0.004±0.0003	0.05±0.001	0.43±0.02
Transformer-Based (small)	0.04±0.01	0.16±0.03	0.04±0.09	0.08±0.05	0.23±0.09	0.11±0.12	0.04±0.03	0.16±0.07	0.14±0.07
Transformer-Based (large)	0.26±0.02	0.41±0.02	0.11±0.06	0.15±0.08	0.31±0.10	0.06±0.02	0.14±0.04	0.31±0.05	0.12±0.07
Dual-Transformer (small)	0.006±0.005	0.06±0.03	0.62±0.03	0.004±0.001	0.05±0.006	0.64±0.05	0.004±0.001	0.05±0.006	0.61±0.06
Dual-Transformer (large)	0.005±0.002	0.06±0.01	0.56±0.09	0.003±0.002	0.04±0.001	0.67±0.03	0.003±0.0004	0.05±0.004	0.67±0.03

slightly similar error metrics; however, the correlation score is decreasing (except for GAT). That is reasonable since GNN models mainly learn based on the tree’s structure throughout collecting information from the neighbour nodes. Although transformer-based is still the worst model in terms of Pearson score, the error metrics have improved hugely, especially for the small version of the model, which puts this version in a better position compared to GNN. The large version of transformer-based is generally still the worst in all metrics. Our model is still the best model, even in this scenario, with significantly better errors and correlation scores than Hadoop’s usage in the training. We also observe in this scenario that with increased fine-tuning, GraphSage and TreeCNN are stable across the portions. The improvement in the models in this scenario shows that when we train the models on diverse trees(since OssBuild contains four projects with four different trees), we can have a better generalization compared to training the models on a large but not diverse tree (as in Hadoop, where all trees come from one project).

IX. DISCUSSION

The analysis of our experimental results, particularly focusing on the Pearson correlation coefficients and error metrics (MSE and MAE), reveals insightful trends in the performance of various models applied to trees. Remarkably, our model demonstrates superior performance compared to other methods in all preceding experiments.

Based on Sections V, VI, we can differentiate the learning of the models into two categories: 1) GNNs and TreeCNN, which learn based on the structure of the tree without any interest in the tokens of the nodes. 2) The rest of the models use the sequence of tokens of the nodes for learning. The first category of models looks at the topological structure of

the tree. Whereas the second category focuses on the exact sequence of tokens of the tree node (e.g., class definition, control statement, method declaration, etc.) Thus, in this section, we will comment on the results we observed in the previous section.

A. Our transformer model vs the competitor

The Dual-Transformer’s design, which incorporates cross-attention between the token-level transformer and the tree node-level transformer, allows for a richer representation of the source code by highlighting the interaction between lexical and syntactic features. This nuanced representation is likely the reason for the observed improvement in performance. In contrast, the baseline Transformer-Based model, which operates solely on the tree, does not capture the lexical context to the same extent, thereby limiting its effectiveness for the regression task.

B. GNN part

GNN-based approaches consistently underperform compared to our model. The failure of GNNs in learning meaningful representations is attributed to the inherent topological structure of trees. Table II reveals that the average network diameter is 17 and 19 for OssBuilds and Hadoop, respectively. The elevated diameter poses a significant challenge for GNN-based methods, necessitating deeper networks. However, this exacerbates well-known issues such as over-smoothing [51] and over-squashing [52].

C. Attention mechanism

The application of attention mechanisms across different models unveils varied outcomes. Our Dual-Transformer

model, leveraging cross-attention, consistently outperforms other approaches, underscoring the efficacy of attention in distilling relevant features from both token sequences and AST node sequences. In contrast, the GAT model exhibits a mixed performance, ranking well on the OssBuilds dataset but falling short on the Hadoop dataset. This inconsistency highlights the potential sensitivity of attention-based GNNs to the underlying dataset characteristics. Interestingly, the TreeCNN model, which does not employ attention, demonstrates resilience across datasets, suggesting that attention mechanisms, while powerful, are not a panacea and may introduce complexity that does not always translate to improved performance.

D. Level of Information

The distinction between node-level and path-level information processing is another critical factor in our analysis. Except for code2vec, which operates at the path level, all other models process information at the node level. This distinction might contribute to the unique positioning of code2vec in the performance spectrum, indicating that the granularity of analysis (node vs. path) can significantly influence model outcomes.

E. Error Analysis and Pearson correlation score

It is noteworthy to highlight that the MAE, MSE, and Pearson correlation capture distinct aspects. To elaborate, MAE offers an assessment of error magnitude, disregarding their direction. Conversely, MSE accentuates larger errors through the squaring operation, rendering it sensitive to outliers. Lastly, Pearson correlation gauges the linear relationship between two variables, providing a measure of the strength and direction of the linear association between predicted and actual values.

- 1) **Mean Squared Error (MSE) and Mean Absolute Error (MAE)** primarily measure the accuracy of predictions in terms of error magnitude. Both metrics are direct measures of the average errors made by the model:
 - **MSE** gives a higher weight to larger errors due to the squaring of each term. This makes it more sensitive to outliers or large deviations from the true values.
 - **MAE** provides a straightforward arithmetic mean of absolute errors, thus not disproportionately penalizing larger errors compared to smaller ones.

When MSE and MAE remain stable across different training data sizes, it suggests that the overall magnitude of errors does not significantly change as more data is used for training. This could imply that adding more training data under these conditions does not necessarily improve the model’s ability to predict more accurately in terms of error reduction. It might suggest that the model has reached a plateau in learning from the additional data where the average error remains consistent.

- 2) **Pearson Correlation**, on the other hand, measures the strength and direction of a linear relationship between the predicted and actual values. An increase in the Pearson correlation as training data size increases could indicate several things:

- As more data is available for training, the model may be getting better at capturing underlying patterns that influence both the scale and trend of the predictions relative to actual outcomes. This doesn’t necessarily mean that the model is becoming more precise in a point-by-point prediction (as indicated by stable MSE and MAE), but rather that it is improving in aligning the direction and trends of its predictions with the actual values.
- A higher Pearson correlation means the model’s predictions are better aligned with the actual values’ variability, even if the absolute errors (magnitude of errors) aren’t improving. This could be critical in applications where understanding the direction of changes is more important than the exact errors.

Therefore, the observed pattern—stable MSE and MAE but increasing Pearson correlation—suggests that while the precision of the model in absolute terms does not improve with more data, the model’s ability to capture the relative movements or trends in the data improves. This distinction is crucial in scenarios where the relationship dynamics between predicted and actual values are more significant than the sheer accuracy of point predictions. It highlights the model’s growing capacity to reflect the true data structure in its predictions over increasing the size of the dataset.

F. Datasets properties

Dataset characteristics play a pivotal role in model performance. The OssBuilds dataset, with its diversity stemming from four distinct projects, ostensibly presents a more challenging environment for models due to the variability in code patterns and AST structures. However, models generally perform better on this dataset compared to Hadoop, which, despite its larger size, consists of samples from a single project. This counterintuitive result may be attributed to the complexity of Hadoop’s ASTs, particularly their depth and diameter, which could pose difficulties for models, especially GNNs, in effectively capturing and propagating information across the tree structure.

G. Transferability across different datasets

The cross-dataset transferability results highlight the challenges inherent in generalizing models trained on one source code dataset to another. Most models exhibited a decrease in performance when applied to an unfamiliar dataset, underscoring the specificity of learned patterns to the training data’s structure and semantics. However, the TreeCNN model and our Dual-Transformer showcased notable resilience, with the latter demonstrating a promising balance between error metrics and correlation, especially on the Hadoop dataset. This suggests that models with sophisticated attention mechanisms, like the Dual-Transformer, may possess an inherent advantage in capturing more generalizable features of source code, transcending dataset-specific idiosyncrasies.

H. models' efficiency across incremental training data sizes

The exploration into model efficiency with varying sizes of training data revealed an expected trend: model performance generally improved as the amount of training data increased. This trend underscores the importance of data volume in model training, particularly for complex models like GNNs and Transformers, which require substantial data to effectively learn and generalize from the intricate structures of ASTs. The consistent performance improvement of our Dual-Transformer model across incremental training sizes further emphasizes its robustness and the effectiveness of its architectural design in leveraging larger datasets for enhanced source code analysis.

I. Conclusion

In summary, our findings illuminate the multifaceted nature of source code analysis using machine learning models. The interplay between model architecture (especially the use of attention mechanisms), the level of information granularity, and dataset characteristics significantly influences performance. These insights not only contribute to our understanding of the strengths and limitations of various approaches but also pave the way for future research aimed at optimizing model design and data preprocessing techniques for enhanced source code analysis.

X. CONCLUSION

In this study, we provided an analytical framework to examine the performance of tree-based neural network models in regression tasks. At the heart of our investigation was the introduction of an innovative model predicated on a dual-transformer architecture. This model was meticulously evaluated against an array of models grounded in Graph Neural Network (GNN) and Tree-Based Neural Network (TBNN) paradigms. The rigour of our experimental methodology, applied to two distinct real-world datasets, firmly establishes the dual-transformer model as a superior contender, outshining its counterparts across various error metrics and Pearson correlation indices.

A recurrent theme observed across the evaluated models was the prevalent incorporation of attention mechanisms and a node-level analytical approach within tree structures. This observation accentuates the pivotal role of attention in effectively navigating the structural intricacies inherent in our case study tree.

The contributions of this paper are twofold. Firstly, it introduces a potent model that redefines the benchmark for regression tasks within the realm of source code analysis. Secondly, it facilitates a nuanced comparative analysis of tree-based neural network models, thereby bolstering the understanding of their efficacy and broadening their applicability in practical settings.

Furthermore, the research underscores the Dual-Transformer model's prowess in accurately forecasting source code execution times. By leveraging a dual encoder framework that intricately captures the nuances of source code tokens and AST nodes, our model demonstrates a marked improvement over conventional tree-based neural network approaches. This

finding signifies the untapped potential of advanced deep learning architectures in the field of source code analysis, setting a promising direction for future inquiries.

ACKNOWLEDGMENTS

Anonymized Acknowledgement

REFERENCES

- [1] N. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," 2018.
- [2] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Pre-trained contextual embedding of source code," 2020. [Online]. Available: <https://openreview.net/forum?id=rygoURNYvS>
- [3] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 1287–1293.
- [4] N. D. Q. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1186–1197.
- [5] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 516–527. [Online]. Available: <https://doi.org/10.1145/3395363.3397362>
- [6] 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*, no. 01, pp. 1–1, aug 2020.
- [7] J. Hua and H. Wang, "On the effectiveness of deep vulnerability detectors to simple stupid bug detection," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 530–534.
- [8] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 661–673.
- [9] K. Shi, Y. Lu, G. Liu, Z. Wei, and J. Chang, "Mpt-embedding: An unsupervised representation learning of code for software defect prediction," *Journal of Software: Evolution and Process*, vol. 33, no. 4, p. e2330, 2021, e2330 smr.2330. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2330>
- [10] N. D. Q. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," ser. SIGIR '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3404835.3462840>
- [11] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1385–1397.
- [12] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid GNN," in *International Conference on Learning Representations*, 2021. [Online]. Available: <https://openreview.net/forum?id=zv-typl1gPxA>
- [13] H. P. Samoaa, F. Bayram, P. Salza, and P. Leitner, "A systematic mapping study of source code representation for deep learning in software engineering," *IET Software*, vol. 16, no. 4, pp. 351–385, 2022. [Online]. Available: <https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/sfw2.12064>
- [14] K. W. Al-Sabbagh, M. Staron, and R. Hebig, "Improving test case selection by handling class and attribute noise," *Journal of Systems and Software*, vol. 183, p. 111093, 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S01641221001904>
- [15] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 141–151. [Online]. Available: <https://doi.org/10.1145/3236024.3236068>
- [16] P. Samoaa, "Data-centric ai for software performance engineering - predicting workload dependent and independent performance of software systems using machine learning based approaches," Ph.D. dissertation, 2023. [Online]. Available: <https://www.proquest.com/dissertations-theses/data-centric-ai-software-performance-engineering/docview/2800163992/se-2>

- [17] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," 2020.
- [18] W. Hua and G. Liu, "Transformer-based networks over tree structures for code classification," *Applied Intelligence*, pp. 1–15, 2022.
- [19] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 783–794.
- [20] M. White, M. Tufano, C. Vendome, and D. Poshvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 87–98.
- [21] H.-H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17. AAAI Press, 2017, p. 3034–3040.
- [22] H. P. Samoaa, A. Longa, M. Mohamad, M. H. Chehreghani, and P. Leitner, "Tep-gnn: Accurate execution time prediction of functional tests using graph neural networks," in *Product-Focused Software Process Improvement*, D. Taibi, M. Kuhrmann, T. Mikkonen, J. Klünder, and P. Abrahamsson, Eds. Cham: Springer International Publishing, 2022, pp. 464–479.
- [23] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: <https://doi.org/10.1145/3290353>
- [24] P. Samoaa, "Analyzing the Behaviour of Tree-Based Neural Networks in Regression Tasks," May 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.11383081>
- [25] P. Samoaa, L. Aronsson, P. Leitner, and M. H. Chehreghani, "Batch mode deep active learning for regression on graph data," in *2023 IEEE International Conference on Big Data (BigData)*, 2023, pp. 5904–5913.
- [26] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, Feb. 2016. [Online]. Available: <https://ojs.aaai.org/index.php/AAAI/article/view/10139>
- [27] X. Zhu, P. Sobihani, and H. Guo, "Long short-term memory over recursive structures," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 1604–1612. [Online]. Available: <https://proceedings.mlr.press/v37/zhub15.html>
- [28] A. Zhang, L. Fang, C. Ge, P. Li, and Z. Liu, "Efficient transformer with code token learner for code clone detection," *Journal of Systems and Software*, vol. 197, p. 111557, 2023. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121222002333>
- [29] X. Jiang, Z. Zheng, C. Lyu, L. Li, and L. Lyu, "Treebert: A tree-based pre-trained model for programming language," in *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, ser. Proceedings of Machine Learning Research, C. de Campos and M. H. Maathuis, Eds., vol. 161. PMLR, 27–30 Jul 2021, pp. 54–63. [Online]. Available: <https://proceedings.mlr.press/v161/jiang21a.html>
- [30] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 1–12.
- [31] L. Mou and Z. Jin, *Tree-based convolutional neural networks: principles and applications*. Springer, 2018.
- [32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
- [33] D. Buffelli and F. Vandin, "Graph representation learning for multi-task settings: a meta-learning approach," 2021. [Online]. Available: <https://openreview.net/forum?id=HmAHqnu3qu>
- [34] F. M. Bianchi and V. Lachi, "The expressive power of pooling in graph neural networks," *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [35] L. Pasa, N. Navarin, W. Erb, and A. Sperduti, "A unified framework for backpropagation-free soft and hard gated graph neural networks," *Knowledge and Information Systems*, pp. 1–24, 2023.
- [36] F. Ferrini, A. Longa, A. Passerini, and M. Jaeger, "Meta-path learning for multi-relational graph neural networks," in *The Second Learning on Graphs Conference*, 2023. [Online]. Available: <https://openreview.net/forum?id=gW9ZmT9hAe>
- [37] A. Nguyen, A. Longa, M. Luca, J. Kaul, and G. Lopez, "Emotion analysis using multilayered networks for graphical representation of tweets," *IEEE Access*, vol. 10, pp. 99 467–99 478, 2022.
- [38] L. Telyatnikov, M. S. Bucarelli, G. Bernardez, O. Zaghen, S. Scardapane, and P. Lio, "Hypergraph neural networks through the lens of message passing: a common perspective to homophily and architecture design," *arXiv preprint arXiv:2310.07684*, 2023.
- [39] J. Thomas, A. Moallem-Oureh, S. Beddar-Wiesing, and C. Holzhiiter, "Graph neural networks designed for different graph types: A survey," *Transactions on Machine Learning Research*, 2023. [Online]. Available: <https://openreview.net/forum?id=h4BYtZ79uy>
- [40] M. Tiezzi, G. Ciravegna, and M. Gori, "Graph neural networks for graph drawing," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.
- [41] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.
- [42] A. Micheli, "Neural network for graphs: A contextual constructive approach," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 498–511, 2009.
- [43] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017. [Online]. Available: <https://openreview.net/forum?id=SJU4ayYgl>
- [44] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: <https://openreview.net/forum?id=rJXMPikCZ>
- [45] D. Bacciu, F. Errica, A. Micheli, and M. Podda, "A gentle introduction to deep learning for graphs," *Neural Networks*, vol. 129, pp. 203–221, 2020.
- [46] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.
- [47] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2018.
- [48] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.
- [49] P. Samoaa, L. Aronsson, A. Longa, P. Leitner, and M. H. Chehreghani, "A unified active learning framework for annotating graph data with application to software source code performance prediction," 2023.
- [50] A. Longa, V. Lachi, G. Santin, M. Bianchini, B. Lepri, P. Lio, franco scarselli, and A. Passerini, "Graph neural networks for temporal graphs: State of the art, open challenges, and opportunities," *Transactions on Machine Learning Research*, 2023. [Online]. Available: <https://openreview.net/forum?id=pHCdMatOgl>
- [51] T. K. Rusch, M. M. Bronstein, and S. Mishra, "A survey on oversmoothing in graph neural networks," *arXiv preprint arXiv:2303.10993*, 2023.
- [52] U. Alon and E. Yahav, "On the bottleneck of graph neural networks and its practical implications," in *International Conference on Learning Representations*, 2021. [Online]. Available: <https://openreview.net/forum?id=i80OPhOCVH2>

Paper 3






Tep-gnn: Accurate execution time prediction of functional tests using graph neural networks

Peter Samoaa, Antonio Longa, Mazen Mohamad, Morteza Haghiri Chehreghani, Philipp Leitner

International Conference on Product-Focused Software Process Improvement (PROFES),
2022



TEP-GNN: Accurate Execution Time Prediction of Functional Tests Using Graph Neural Networks

Hazem Peter Samoaa¹ , Antonio Longa² , Mazen Mohamad¹ ,
Morteza Haghiri Chehrehghani¹ , and Philipp Leitner¹ 

¹ Chalmers—University of Gothenburg, Gothenburg, Sweden
{samoaa,mazenm,morteza.chehrehghani,philipp.leitner}@chalmers.se

² Fondazione Bruno Kessler and University of Trento, Trento, Italy
alonga@fbk.eu

Abstract. Predicting the performance of production code prior to actual execution is known to be highly challenging. In this paper, we propose a predictive model, dubbed TEP-GNN, which demonstrates that high-accuracy performance prediction is possible for the special case of predicting unit test execution times. TEP-GNN uses FA-ASTs, or flow-augmented ASTs, as a graph-based code representation approach, and predicts test execution times using a powerful graph neural network (GNN) deep learning model. We evaluate TEP-GNN using four real-life Java open source programs, based on 922 test files mined from the projects' public repositories. We find that our approach achieves a high Pearson correlation of 0.789, considerably outperforming a baseline deep learning model. Our work demonstrates that FA-ASTs and GNNs are a feasible approach for predicting absolute performance values, and serves as an important intermediary step towards being able to predict the performance of arbitrary code prior to execution.

Keywords: Performance · Software testing · Machine learning

1 Introduction

Performance is a critical quality property of many real-live software systems. Hence, performance modeling and analysis have gradually become an increasingly important part of the software development life-cycle. Unfortunately, predicting the performance of real-life production code is well-known to be a difficult problem – predicting the absolute execution time of applications based on code structure is challenging as it is a function of many factors, including the underlying architecture, the input parameters, and the application's interactions with the operating system [22]. Consequently, works that attempted to predict absolute performance counters (e.g., execution time) for arbitrary applications from source code generally report poor accuracy [19, 21].

However, recent research has shown that predicting performance characteristics is indeed possible in more specialized contexts, via the application of modern machine learning architectures. For example, Guo et al. successfully predict the execution time of a specific untested configuration of a configurable system [6, 7], Samoaa and Leitner have shown that the execution time of a benchmark with specific workload configuration can be predicted [24], and Laaber et al. have shown that a categorical classification of benchmarks into high- or low-variability is feasible [12].

In this work, we demonstrate that another context where performance prediction is possible is the prediction of execution times of functional tests. Test execution times are crucial in agile software development and continuous integration. While individual test cases might have short execution times, software products often have thousands of test cases, which makes the total execution time in the build process high. Researchers have been working on solutions to speed up the testing process by optimizing the code or prioritizing test cases [4, 11, 18, 28]. The goal of this study is to provide the developers with predictions of the execution times of their test cases, and consequently giving them an early indication of the time required to run the cases in the build process. We believe that this would support decisions regarding code optimization and test case selection in early stages of the software life-cycle.

Graphs are mathematical structures used to model pairwise relations between objects. A graph can be used to model a wide number of different domains, ranging from biology [9], face-to-face human interactions [17] and software. Indeed, we propose an approach dubbed TEP-GNN (Test Execution Time Prediction using Graph Neural Networks) that makes use of structural features of test cases (the abstract syntax tree, AST). We enrich the AST with various types of edges representing data and control flow. Following Wang and Jin, we refer to this resulting graph as flow-augmented abstract syntax trees (FA-AST) [30]. We use a graph neural network (GNN) model, GraphConv [20], on the resulting FA-ASTs. We train and test our model on a dataset collected from four well-known open source projects hosted on GitHub: *H2 database*¹, a relational database, *RDF4J*², a project for handling RDF data, *systemDS*³, an Apache project to manage the data science life cycle, and finally the Apache remote procedure call library *Dubbo*⁴. As labelled ground truth data, we collect 922 real test execution traces from these projects' publicly available build systems.

We conduct experiments with our TEP-GNN model to answer the following research questions:

- **RQ1:** How accurately can the absolute execution time of a test file consisting of one or multiple test cases be predicted using FA-ASTs and GNNs?

¹ <https://github.com/h2database/h2database>.

² <https://github.com/eclipse/rdf4j>.

³ <https://github.com/apache/systemds>.

⁴ <https://github.com/apache/dubbo>.

- **RQ2:** Does our usage of GraphConv improve execution time prediction compared to a baseline using Gated Graph Neural Networks (GGNN), as frequently used in previous software engineering research [1,5]?

Our results show that using TEP-GNN, test execution time can be predicted with a very high prediction accuracy (Pearson correlation of 0.789). Further, we show that our usage of GraphConv indeed improves the model significantly over GGNN. We conclude that test execution times can indeed be predicted using GNN models with high accuracy, even based on performance counters that have been collected “in the wild” by real projects (as opposed to performance measurements collected on a dedicated performance testing machine). The main novelty of our work lies in the application of a rarely used way of graph-encoding source code (FA-AST), combined with a powerful GNN model (GraphConv), to the problem of performance prediction. Even though test cases are shorter and structurally simpler than arbitrary programs, we see our results as an important stepping stone towards the prediction of the performance of arbitrary software systems prior to execution.

2 The TEP-GNN Approach

In this section, we introduce TEP-GNN. We first provide a general overview of the model and discuss the problem addressed in this paper, followed by a detailed discussion of the main components of TEP-GNN (FA-ASTs and the machine learning pipeline based on the GraphConv [20] higher order GNN).

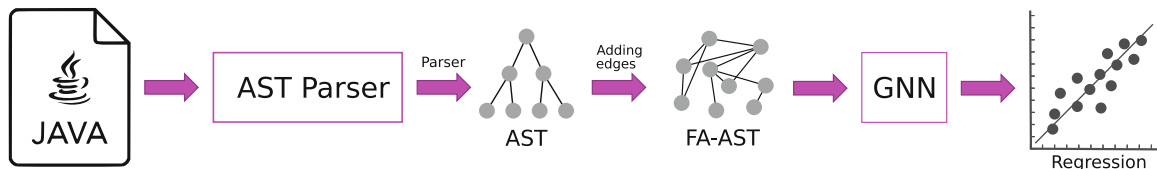


Fig. 1. Schematic overview of the main phases of TEP-GNN.

2.1 Approach Overview

Our goal in this paper is to predict the execution time of test cases based on static code information alone, i.e., without access to prior benchmarking of the test case or dynamic analysis data. The general procedure of our TEP-GNN approach is sketched in Fig. 1. To process a test file, we first parse it into its AST. Next, we build a graph representation (FA-AST) by adding edges representing control and data flow to the AST. We then initialize the embeddings of FA-AST nodes and edges before jointly feeding a vectorized FA-AST into a GNN.

2.2 Problem Definition

Given a test file (source code containing test cases) C_i and the corresponding run-time value X_i (execution time of all test cases in the file), for a set of test files with known execution times we can build a training set $D = (C_i, X_i)$. We aim to train a deep learning model for learning a function ϕ that maps a test file C_i to a feature vector v mapped to the corresponding value X_i .

2.3 Building Flow-Augmented Abstract Syntax Trees

Recent studies [25] emphasize the importance of the code representation when using deep learning in software engineering. Hence, and given the complexity of predicting performance, prediction based on the syntactical information extracted from ASTs alone is not sufficient to achieve high-quality predictions. In TEP-GNN, the basic structural information provided by the AST is enriched with semantic information representing data and control flow. Consequently, the tree structure of the AST is generalized to a (substantially richer) graph, encoding more information than code structure alone. This idea is based on the earlier work by Wang and Jin [30], who have also introduced the term FA-AST for this kind of source code representation.

```

1 package org.myorg.weather.tests;
2
3 import static
4     org.junit.jupiter.api.Assertions.assertEquals;
5 import org.myorg.weather.WeatherAPI;
6 import org.myorg.weather.Flags;
7
8 public class WeatherAPITest {
9
10     WeatherAPI api = new WeatherAPI();
11
12     @Test
13     public void testTemperatureOutput() {
14         double currentTemp = api.currentTemp();
15         Flags f = api.getFreezeFlag();
16         if (currentTemp <= 3.0d)
17             assertEquals(Flags.FREEZE, f);
18         else
19             assertEquals(Flags.THAW, f);
20     }
21 }

```

Listing 1.1. A Simple JUnit 5 Test Case

AST Parsing. We demonstrate our approach for constructing FA-ASTs for test files using the example of a Java JUnit 5 test case (see Listing 1.1). In this example, a single test case `testTemperatureOutput()` is presented that tests a feature of an (imaginary) API. As common for test cases, the example is short and structurally relatively simple. Much of the body of the test case consists of invocations to the system-under-test and calls of JUnit standard methods, such as `assertEquals`. We speculate that these properties make predicting test execution time a more tractable problem than predicting performance of general-purpose programs, which previous authors have argued to be extremely challenging [19, 21].

A (slightly simplified) AST for this illustrative example is depicted in Fig. 2. The produced AST does not contain purely syntactical elements, such as comments, brackets, or code location information. We make use of the pure Python Java parser `javalang`⁵ to parse each test file, and use the node types, values, and production rules in `javalang` to describe our ASTs.

⁵ <https://pypi.org/project/javalang/>.

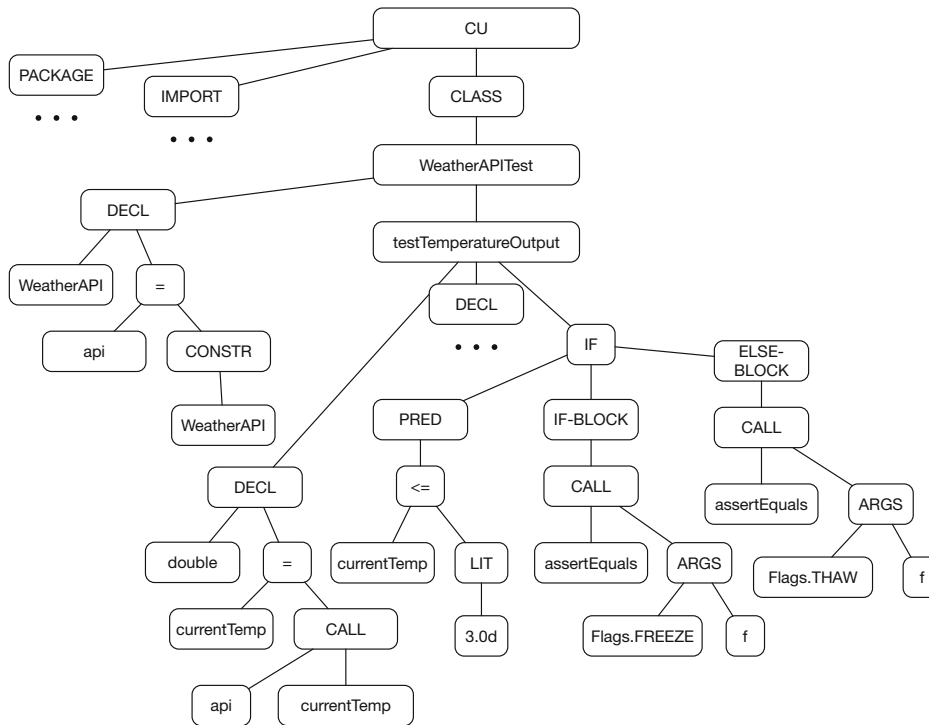


Fig. 2. Simplified abstract syntax tree (AST) representing the illustrative example presented in Listing 1.1. Package declarations, import statements, as well as the declaration in Line 15 are skipped for brevity.

Capturing Ordering and Data Flow. In the next step, we augment this AST with different types of additional edges representing data flow and node order in the AST. Specifically, we use the following additional flow augmentation edges, in addition to the **AST child** and **AST parent** edges that are produced readily by AST parsing:

FA Next Token (b):

This type of edge connects a terminal node (leaf) in the AST to the next terminal node. Terminal nodes are nodes without children. In Fig. 2, an FA Next Token edge would be added, for example, between `WeatherAPI` and `api`.

FA Next Sibling (c):

This connects each node (both terminal and non-terminal) to its next sibling, and allows us to model the order of instructions in an otherwise unordered graph. In Fig. 2, such an edge would be added, for example, connecting the first usage of `api` and with the `CONSTR` node (representing a Java constructor call).

FA Next Use (d):

This type of edge connects a node representing a variable to the place where this variable is next used. For example, the variable `api` is declared in Line 10 in Listing 1.1, and then used next in Line 14.

Figure 3 shows an example augmenting the AST in Fig. 2 (and, consequently, the example test case in Listing 1.1). Solid black lines indicate the AST parent and child relationships (for simplicity indicated through a single arrow, read from top to bottom). Red dashed arrows refer to the new edges added to represent the

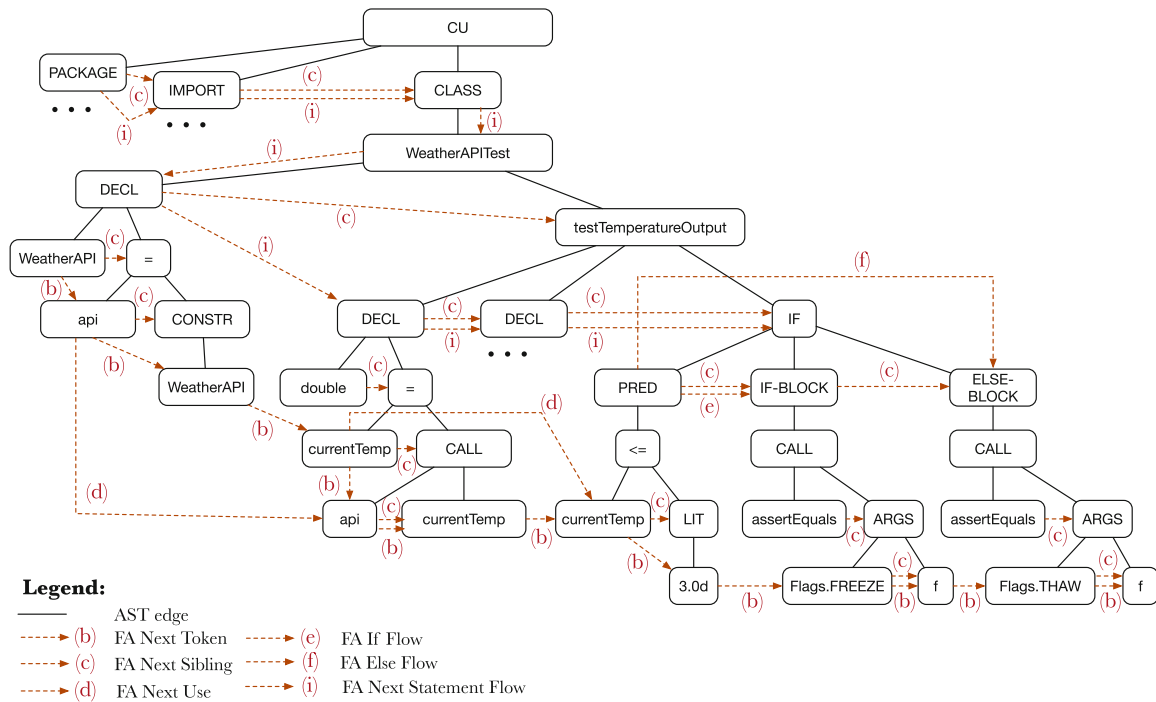


Fig. 3. Flow-Augmented AST (FA-AST) for the example presented in Listing 1.1. Solid lines represent AST parent and child edges, and dashed lines different types of flow augmentations. (Color figure online)

data and control flow in the FA-AST, with letter codes indicating the edge type. Terminal nodes are connected with FA Next Token edges (b), modelling the order of terminals in the test case. Similarly, the ordering of siblings is modelled using FA Next Sibling edges (c). Finally, data flow is modelled by connecting each variable to their next usage via FA Next Use edges (d). Edge types (e), (f), and (i) represent a control flow statement, which will be discussed in the following. Multiple edges of different types are possible between the same nodes. For example, the terminal nodes `Flags.FREEZE` and `f` are connected via both, an FA Next Token (b) and an FA Next Sibling (c) edge.

Capturing Control Flow. In a second augmentation step, we now add further edges representing the control flow in the test cases. We currently support *if* statements, *while* and *for* loops, as well as *sequential execution*. We currently do not support *switch* statements or *do-while* loops, as these are less common in test cases. Test files containing these elements will still be parsed successfully, but these control flow constructs will not be captured by the FA-AST. Specifically, the following further edges are added: An overview over the additional edges introduced by these control flow statements is given in Fig. 4.

FA If Flow (e):

This type of edge connects the predicate (condition) of the if-statement with the code block that is executed if the condition evaluates to `true`. Every if statement contains exactly one such edge by construction.

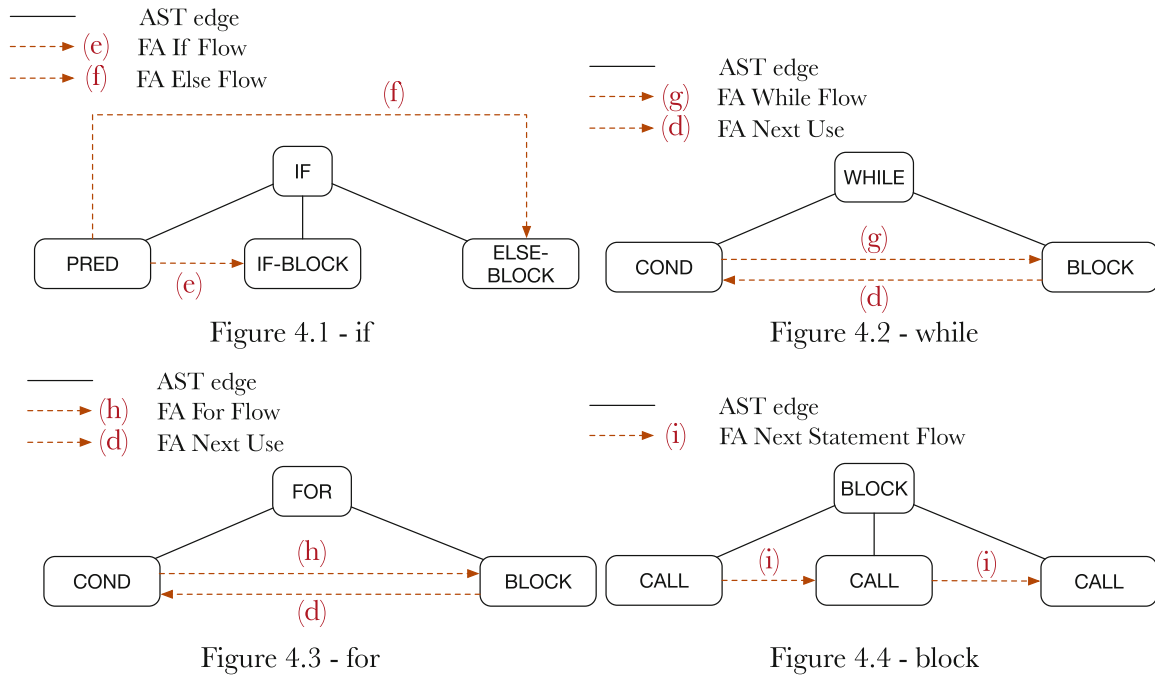


Fig. 4. Additional flow augmentations for different control flow constructs (Color figure online)

FA Else Flow (f):

Conversely, this edge type connects the predicate to the (optional) else code block.

FA While Flow (g):

A while loop essentially entails two elements - a condition and a code block that is executed as long as the condition remains `true`. We capture this through a FA While Flow (g) edge connecting the condition to the code block, and an FA Next Use (d) edge in the reverse direction. The latter is used to model the next usage of a loop counter.

FA For Flow (h):

For loops are conceptually similar to while loops. We use FA For Flow (h) edges to connect the condition to the code block, and an FA Next Use (d) edge in the reverse direction. Similar to the modelling of while-loops, FA Next Use (d) relates to the usage (typically incrementing) of a loop counter.

FA Next Statement Flow (i):

In addition to the control flow constructs discussed so far, Java of course also supports the simple sequential execution of multiple statements in a sequence within a code block. FA Next Statement Flow edges (i) are used to represent this case. Different from the constructs discussed so far, a code block can contain an arbitrary number of children, and the FA Next Statement Flow edge is always used to connect each statement to the one directly following it.

Referring back to Fig. 3, two types of control flow annotations are visible - the modelling of the if-statement in lines 16 to 19 of the test case on the right-hand side, and various sequential executions (FA Next Statement flow (i)) edges.

Further note how flow annotation adds a large number of edges to even a very small AST, transforming the syntax tree into a densely connected graph. This rich additional information can be used in the next step by our GNN model to predict highly accurate test execution times.

2.4 GNN Model for Test Execution Time Prediction

Once the FA-AST graph has been built for a test file using the three steps discussed above, we use a higher order GNN model to predict the execution time of the Java code. As Fig. 5 shows, we use a 3-layer higher order graph convolution neural network to predict the execution time. Each layer is followed by a ReLU activation function. Since GNN learns node embedding, we use global max pooling to compute a graph embedding. Finally, the graph embedding goes into two Linear layers with a ReLU and a sigmoid activation function to perform the prediction of the test execution time. To train our model we use the mean square error loss.

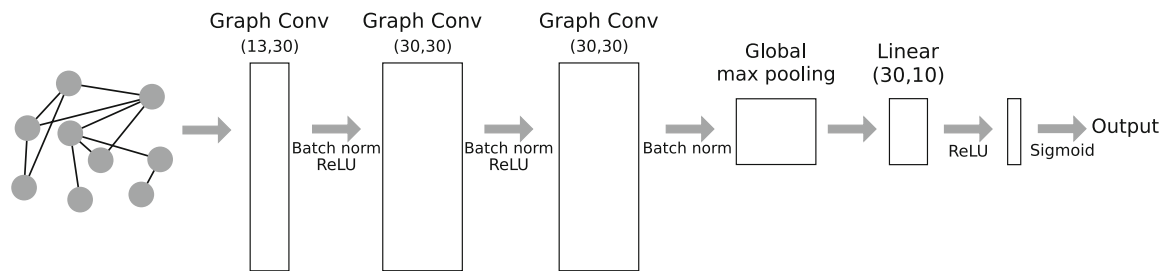


Fig. 5. Architecture of the GNN Model used in TEP-GNN.

3 Evaluation

We now present the results of an experimental evaluation of TEP-GNN based on open source Java projects. As training and test data we make use of existing test suite execution traces from the study subjects' build systems. A replication package containing the scripts used to implement the TEP-GNN approach, all data used in the evaluation, as well as all analysis scripts, are available [8].

3.1 Dataset

Related studies in performance engineering frequently collect their own performance data, for example by repeated execution of the projects on a researcher's laptop [26], in cloud virtual machines [13], or on controlled hardware [27]. To increase the realism of the study we have chosen a different approach – we harvest existing execution traces from an open source build system (GitHub), and extract test execution times from this public data. This data represents actual, real-life test execution times. However, we do not have the option to collect more data on-demand, and we do not know what precise hardware has been used to collect the data.

To collect the data, we searched for projects to serve as study subjects. We applied the following selection criteria: *(i)* projects written in Java; *(ii)* available on GitHub; *(iii)* include test results published on GitHub; and *(iv)* use GitHub shared runners as build system.

Table 1. Overview of study subjects.

Project	Description	Files	Runs	Nodes	Vocabulary size
systemDS	Apache Machine Learning system for data science lifecycle	127	1321	110651	3161
H2	Java SQL database	194	1391	405706	17972
Dubbo	Apache Remote Procedure Call framework	123	524	75787	4499
RDF4J	Scalable RDF processing for Java	478	1055	214436	10755
Total		922	4291	806580	36387

Based on these criteria, we selected four projects of diverse application domains, i.e., databases, web servers, and data science life-cycle (systemDS, H2, Dubbo, and RDF4J). These are depicted in Table 1. The first column shows the project’s name, the second provides a brief description of the project. The third column shows the number of distinct test files extracted from the project. As for the fourth column, it shows the total number of runs performed in the testing job. The last two columns show the total number of tokens in the entire project test files and the vocabulary size (the number of distinct nodes in the graphs). We observe that RDF4J, a triplestore database used in semantic web contexts [23], contains more test files than the other projects. For the H2 relational database and systemDS we were able to collect the most test runs. Finally, it should be noted that H2 has the highest code density as measured by the number of nodes and the resulting vocabulary size by a wide margin. This indicates that H2 tests are generally larger and more complex than the test cases in the other study subjects.

All data was extracted from GitHub-hosted runners, which are virtual machines hosted by GitHub with the GitHub Actions runner application installed. All shared runners can be assumed to use the same hardware resources, which is available at GitHub’s website⁶ and each job runs in a fresh instance of the virtual machine. Additionally, all jobs from which the data is extracted uses the same operating system, specifically Ubuntu 18.04. This allows us to minimize bias introduced by variations in execution environment or hardware.

For collecting test execution traces we looked at the latest successful action workflow run for each project. We then extracted the run times from the test report in the workflow, and mined the corresponding source code files from the respective project repositories in order to feed them to the parser. For H2,

⁶ <https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners#supported-runners-and-hardware-resources>.

some test cases are run several times during the same build job. In these cases, we recorded the average of the run times. As the execution times of tests can vary dramatically between and within projects, to increase the efficiency of the model training, we normalize each execution time to interval $[0; 1]$. Hence, our final dataset includes distinct test files, each associated with one runtime value between 0 and 1. Then after model training, we denormalize the runtime value and present the results based on the original values.

Table 2. Occurrences of control flow nodes in each project

Control flow statement	systemDS	H2	Dubbo	RDF4J
If Statement	166	1322	53	161
While Loop	2	222	3	22
ForStatement	196	1114	42	158
Block Statement	293	2900	116	395
Total	707	5612	214	736

Table 2 indicates how prevalent different control flow nodes were in the test cases of our study subjects. For all projects, block statements are the most frequent control flow construct, since sequential executions widely exist in nearly all programs. For loops are substantially more common than while loops, and if statements are also frequent. Do-while loops and switch statements, which are currently unsupported by TEP-GNN, are both quite rare in the tests of our subjects (not shown in the table).

3.2 Results

In this section, we investigate the results of applying TEP-GNN to our dataset, answering RQ1 and RQ2 introduced in Sect. 1.

RQ1: Quality of Predictions. In order to answer the first research question, we combine the collected data for all projects into one dataset entailing 922 code fragments and associated normalized execution times. After that, we apply TEP-GNN as discussed in Sect. 2. For model training, we split the dataset into train and test sets using 80% and 20%, respectively. Each network is trained for 100 epochs. As optimizer we use Adam [10] with a learning rate = 0.001. To evaluate the results of our model, we use a Pearson correlation metric, a measure of linear correlation between two sets of data. In addition, as a loss function, we use mean squared error, which is the average squared difference between the estimated and actual values. All experiments have been executed in a machine equipped with a GeForce 940MX graphics card and 16 GB of RAM.

Results illustrate that our model trained on FA-AST is able to predict test execution times with a very high accuracy, as can be seen in the Pearson correlation (between predicted and actual execution times in the test data set) of

0.789, and a mean squared error of 0.02. These results substantially outperform the accuracy values reported in previous studies that attempted to predict absolute software performance counters [19,21]. We argue that the key innovation that enables this high accuracy is the combination of FA-AST as a powerful code representation model and GraphConv as a modern GNN.

RQ2: Comparison of TEP-GNN Against a Baseline GNN. To validate the suitability of our approach and the selected GNN model, we compare it to a commonly used GNN baseline, called Gated Graph Neural Networks (GGNN) [16]. GGNNs are widely used in studies that attempt to learn code semantics [1, 5]. We compare the methods at two levels – for the entire dataset (similar to the analysis presented for RQ1) and at the level of individual projects.

Comparison for the Entire Dataset. We first apply both TEP-GNN and the baseline method to the dataset consisting of all projects. Figure 6 depicts the respective results. Our model outperforms the baseline, with a Pearson correlation that is higher almost up to 0.1 (i.e., 0.789 versus 0.697). Hence, we conclude that our model and GNN architecture is indeed more appropriate to predict the execution time of test cases than a more standard GGNN approach.

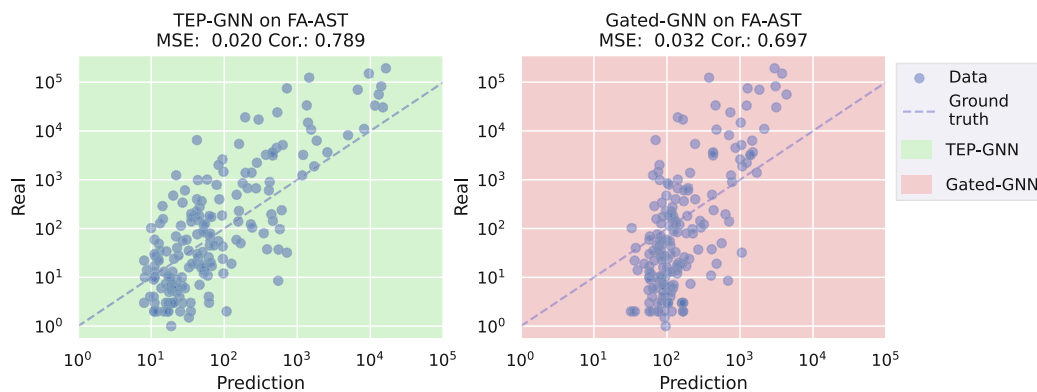


Fig. 6. Comparison of TEP-GNN and a baseline (applying GGNN to the same FA-AST graphs). Dot points show real (y axes) and predicted (x axes) denormalized (original) execution times produced by our model. The dash line refers to the perfect prediction.

Analyzing the results, we observe that TEP-GNN is able to achieve highly accurate predictions in most cases. However, there are rare outliers where our prediction model misses by approximately 20%. The baseline GGNN method, on the other hand, has a tendency to predict fairly uniform execution times between 10^2 and 10^3 , almost independent of what the actually observed test execution time is. Hence, it suffers from lower accuracy scores.

Comparison for Individual Projects. In the next step, we conduct a similar analysis, but focused on individual projects. This study answers the question of how well TEP-GNN works if trained on and used by a single project. Thus, we train and test TEP-GNN and the baseline on each of the four projects individually. The results for each project are depicted in Fig. 7.

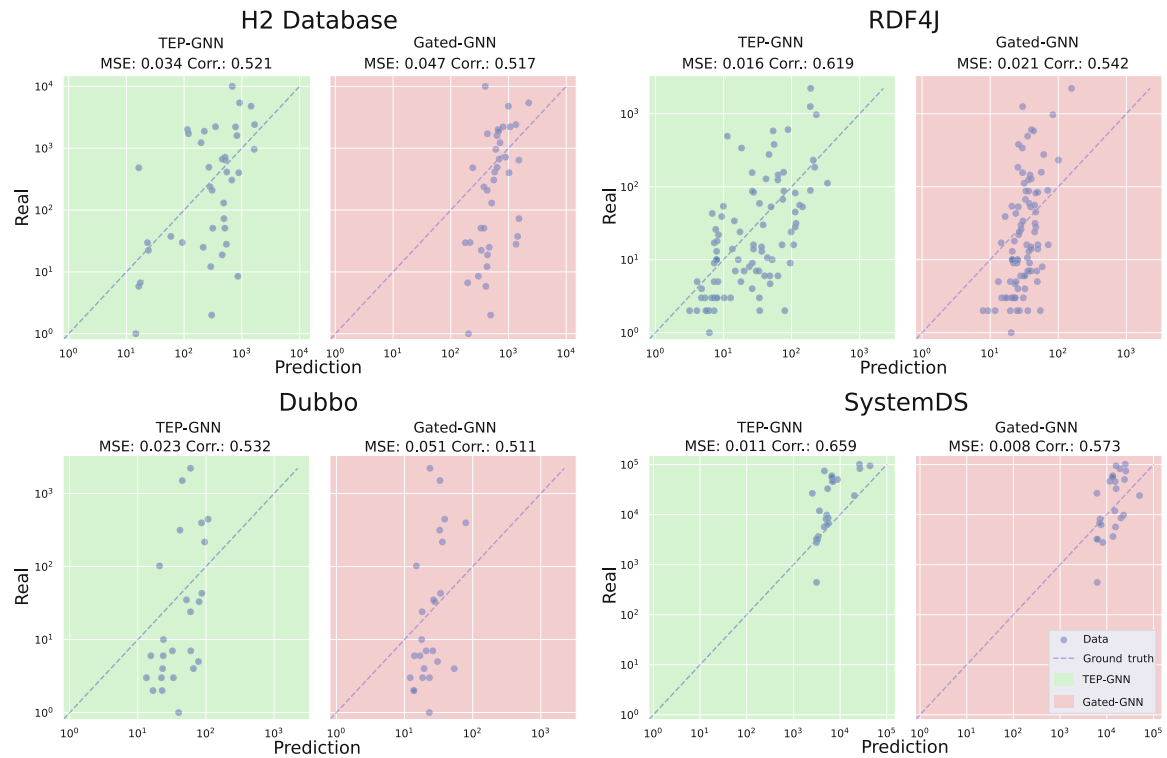


Fig. 7. Overview of TEP-GNN and the Gated-GNN baseline trained for each individual project.

We observe that in general the prediction quality is substantially lower if the model is trained on individual projects, both for TEP-GNN and the baseline. TEP-GNN still outperforms the baseline for each project, but only with negligible prediction performance differences in the case of H2 and Dubbo. For RDF4J, which contains the largest number of test cases (and, consequently, the largest number of graphs to learn from), the difference between our approach and the baseline remains larger.

From these results we conclude that (a) TEP-GNN indeed outperforms the baseline in all the settings we tested, but (b) our approach works best if sufficient training graphs are available in comparison to the size of the graphs and vocabulary (if graphs are complex and/or training data is sparse the difference between our approach and the baseline is insignificant); (c) finally, we conclude that both approaches appear to learn some transferable knowledge even when training on graphs that originate from a different project.

4 Discussion

Our study results show that the accurate prediction of execution times of test suites is possible. This gives developers an early indication of the time required to run the cases in the build process, deciding in the process if techniques such as test case selection are required.

4.1 Lessons Learned

FA-ASTs are a promising approach to represent source code for performance prediction. Unlike previous work [19, 21, 31], our goal in this study was to treat performance prediction as a regression rather than a classification (slow or fast) problem. Our results in Sect. 3.2 indicate that using flow augmentation we are able to achieve good prediction quality. Furthermore, more information could be added to the FA-AST, such as program dependency graphs. We speculate that this approach is also promising to predict the performance of more complex, arbitrary code; however, more specific experiments in this direction need to be carried out as future research.

GraphConv substantially outperforms the more common GGNN models in performance prediction as long as sufficient data is available. As discussed in Sect. 3.2, our GraphConv based GNN model substantially outperforms GGNN, which is a currently commonly used graph neural network model in software engineering research [1, 5]. However, this is only true if sufficient data is available – when training models for individual projects, we observed that, due to the limited amount of training data available in these cases, the performance difference between our GraphConv based model and the GGNN baseline was minimal. We conclude that, as long as sufficient data is available, GraphConv should also be investigated in other software engineering contexts that make use of GNNs.

4.2 Threats to Validity

Internal Validity Threats. A key design choice in our study was the usage of existing, real-world data from GitHub’s build system, rather than collecting performance data ourselves (e.g., on a dedicated experiment machine). This has obvious advantages with regards to the realism of our approach, but raises the threat that our training and test data may be subject to confounding factors outside of our knowledge. In particular, prior research has shown that even identically configured cloud virtual machines can vary significantly in performance [14]. However, the high accuracy achieved by our prediction models indicates that this is not a major concern with the data we used.

Another design choice was that we predict execution times for entire test classes (files). More fine-grained predictions (e.g., for individual test cases) would of course be doable. However, individual test cases often have very short execution times in relation to the precision with which build systems typically measure execution times, and the resulting graphs would be very small. We argue that our choice of test class granularity constitutes a good trade-off that is still useful for developers.

External Validity Threats. An obvious question raised by our work is how well the results reported in Sect. 3.2 would generalize to other projects. To mitigate this threat, we have chosen four relatively different Java projects as study subjects following a diversity sampling strategy [2]. However, our study does not allow us to conclude whether the TEP-GNN approach would generalize to other programming languages or closed-source software.

5 Related Work

Predicting Software Performance. Predicting the absolute value of performance, such as execution time, based on the source code alone is challenging. Hence, existing studies often struggle with poor prediction accuracy [19, 21]. One way to simplify the problem (and hence make it more tractable) is to convert it into a classification problem. Examples of this approach include Zhou et al. [31], who predict if a program from a programming competition website exceeds the time limit, Ramadan et al. [22], who predict whether a performance change is introduced by a code structure change, or Laaber et al. [12], who have shown that a categorical classification of benchmarks into high- or low-variability is feasible.

However, recent research has shown that predicting absolute performance values can be feasible in more specialized contexts like Guo et al. in the context of configurable system [6, 7], and Samoaa and Leitner in the context of benchmark with a specific workload configuration [24].

Graph Neural Networks for Software Engineering. Graph Neural Networks (GNNs) constitute an up-and-coming machine learning model in the context of software engineering research [25]. Li et al. [16] use a GRU cell in gated graph neural networks (GGNNs) for updating the nodes' states. To evaluate their model they run the model on a basic program and try to detect null pointers.

Phan et al. [29] use graph convolutional networks (GCNs) based on compiled assembly code to detect defects on control flow graphs in C. Another application of control flow graphs is using graph matching networks (GMN) between two graphs of binary functions proposed by Li et al. [15]. Other researchers propose the creation of program graphs based on the AST. Allamanis et al. [1] and Brockschmidt et al. [3] use GGNN in C# for naming variables and generating program expressions for code completion respectively.

6 Conclusion and Future Work

In this work, we provide the developers with predictions of the execution times of their test cases, and consequently give them an early indication of the time required to run the cases in the build process. We presented TEP-GNN, an effective method for predicting the execution time of Java test files. Our approach leverages explicitly capturing control and data flow information as augmentations to the program AST. Further, our approach applies high order convolution graph neural networks over this flow-augmented AST (FA-AST). By building FA-AST using original ASTs and flow edges, our approach can directly capture the syntax and semantic structure of test classes. Experimental results on four diverse test subjects demonstrate that by combining graph neural networks and control/data flow information, we can predict absolute test execution times with high accuracy.

As the future work, we plan to further extent the FA-AST model currently used by TEP-GNN, as well as explore other ways of program representation

to capture more syntactic and semantic code features. Additionally, we plan to apply our approach to the execution time of general-purpose programs rather than test cases. Finally, we would like to extend our current labeled data set by applying active learning to systematically increase the amount of training data.

Acknowledgements. This work received financial support from the Swedish Research Council VR under grant number 2018-04127 (Developer-Targeted Performance Engineering for Immersed Release and Software Engineering).

References

1. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs (2017). <https://arxiv.org/abs/1711.00740>
2. Baltes, S., Ralph, P.: Sampling in software engineering research: a critical review and guidelines. *EMSE* **94**(27) (2022)
3. Brockschmidt, M., Allamanis, M., Gaunt, A.L., Polozov, O.: Generative code modeling with graphs (2018). <https://arxiv.org/abs/1805.08490>
4. de Oliveira Neto, F.G., Ahmad, A., Leifler, O., Sandahl, K., Enoiu, E.: Improving continuous integration with similarity-based test case selection. In: Proceedings of the 13th International Workshop on Automation of Software Test, pp. 39–45 (2018)
5. Fernandes, P., Allamanis, M., Brockschmidt, M.: Structured neural summarization (2018). <https://arxiv.org/abs/1811.01824>
6. Guo, J., Czarnecki, K., Apel, S., Siegmund, N., Wařowski, A.: Variability-aware performance prediction: a statistical learning approach. In: ASE, pp. 301–311 (2013)
7. Guo, J., et al.: Data-efficient performance learning for configurable systems. *EMSE* **23**(3), 1826–1867 (2018)
8. Samoaa, H.P., Longa, A., Mohamed, M., Chehreghani, M.H., Leitner, P.: TEP-GNN: accurate execution time prediction of functional tests using graph neural networks. Zenodo, August 2022. <https://doi.org/10.5281/zenodo.7003881>
9. Huber, W., Carey, V.J., Long, L., Falcon, S., Gentleman, R.: Graphs in molecular biology. *BMC Bioinform.* **8**(6), 1–14 (2007)
10. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization (2014). <https://arxiv.org/abs/1412.6980>
11. Knauss, E., Staron, M., Meding, W., Söder, O., Nilsson, A., Castell, M.: Supporting continuous integration by code-churn based test selection. In: 2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering, pp. 19–25. IEEE (2015)
12. Laaber, C., Basmaci, M., Salza, P.: Predicting unstable software benchmarks using static source code features. *EMSE* **26**(6) (2021)
13. Laaber, C., Scheuner, J., Leitner, P.: Software microbenchmarking in the cloud. How bad is it really? *EMSE* **24**(4), 2469–2508 (2019)
14. Leitner, P., Cito, J.: Patterns in the Chaos - a study of performance variation and predictability in public IaaS clouds. *ACM TOIT* **16**(3), 15:1–15:23 (2016)
15. Li, Y., Gu, C., Dullien, T., Vinyals, O., Kohli, P.: Graph matching networks for learning the similarity of graph structured objects. In: Proceedings of the 36th International Conference on Machine Learning, vol. 97. PMLR (2019)

16. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.: Gated graph sequence neural networks (2015). <https://arxiv.org/abs/1511.05493>
17. Longa, A., Cencetti, G., Lepri, B., Passerini, A.: An efficient procedure for mining egocentric temporal motifs. *Data Min. Knowl. Disc.* **36**(1), 355–378 (2022)
18. Marijan, D., Gotlieb, A., Liaaen, M.: A learning algorithm for optimizing continuous integration development and testing practice. *Softw. Pract. Exp.* **49**(2), 192–213 (2019)
19. Meng, K., Norris, B.: Mira: a framework for static performance analysis. In: *CLUSTER* (2017)
20. Morris, C., et al.: Weisfeiler and leman go neural: higher-order graph neural networks. In: *AAAI*, vol. 33 (2019)
21. Narayanan, S.H.K., Norris, B., Hovland, P.D.: Generating performance bounds from source code. In: *International Conference on Parallel Processing Workshops*, pp. 197–206 (2010)
22. Ramadan, T., Islam, T.Z., Phelps, C., Pinnow, N., Thiagarajan, J.J.: Comparative code structure analysis using deep learning for performance prediction. In: *ISPASS*, Los Alamitos, CA, USA. IEEE Computer Society, March 2021
23. Samoaa, H., Catania, B.: A pipeline for measuring brand loyalty through social media mining. In: Bureš, T., et al. (eds.) *SOFSEM 2021*. LNCS, vol. 12607, pp. 489–504. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67731-2_36
24. Samoaa, H., Leitner, P.: An exploratory study of the impact of parameterization on JMH measurement results in open-source projects. In: *ICPE*. Association for Computing Machinery (2021)
25. Samoaa, H.P., Bayram, F., Salza, P., Leitner, P.: A systematic mapping study of source code representation for deep learning in software engineering. *IET Softw.* (2022)
26. Sandoval Alcocer, J.P., Bergel, A., Valente, M.T.: Learning from source code history to identify performance failures. In: *ICPE*. Association for Computing Machinery (2016)
27. Schulz, H., Okanović, D., van Hoorn, A., Tuma, P.: Context-tailored workload model generation for continuous representative load testing. In: *ICPE*. Association for Computing Machinery (2021)
28. Spieker, H., Gotlieb, A., Marijan, D., Mossige, M.: Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: *ISSTA*, pp. 12–22 (2017)
29. Viet Phan, A., Le Nguyen, M., Thu Bui, L.: Convolutional neural networks over control flow graphs for software defect prediction. In: *ICTAI* (2017)
30. Wang, W., Li, G., Ma, B., Xia, X., Jin, Z.: Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: *SANER* (2020)
31. Zhou, M., Chen, J., Hu, H., Yu, J., Li, Z., Hu, H.: DeepTLE: learning code-level features to predict code performance before it runs. In: *APSEC* (2019)

Paper 4

A Unified Active Learning Framework for Annotating Graph Data For Regression Task

Peter Samoaa, Linus Aronsson, Antonio Longa, Philipp Leitner, Morteza Haghiri
Chehreghani

Journal of Engineering Applications of Artificial Intelligence (EAAI), 2024

A UNIFIED ACTIVE LEARNING FRAMEWORK FOR ANNOTATING GRAPH DATA FOR REGRESSION TASK

Peter Samoaa

Chalmers University of Technology
Data Science and AI
samoaa@chalmers.se

Linus Aronsson

Chalmers University of Technology
Data Science and AI
linaro@chalmers.se

Antonio Longa

University of Trento, Italy
Department of Information Engineering and Computer Science
antonio.longa@unitn.it

Philipp Leitner

Chalmers University of Technology
Interaction Design and Software Engineering
philipp.leitner@chalmers.se

Morteza Haghiri Chehreghani

Chalmers University of Technology
Data Science and AI
morteza.chehreghani@chalmers.se

ABSTRACT

In many domains, effectively applying machine learning models requires a large number of annotations and labelled data, which might not be available in advance. Acquiring annotations often requires significant time, effort, and computational resources, making it challenging. Active learning strategies are pivotal in addressing these challenges, particularly for diverse data types such as graphs. Although active learning has been extensively explored for node-level classification, its application to graph-level learning, especially for regression tasks, is not well-explored. We develop a unified active learning framework specializing in graph annotating and graph-level learning for regression tasks on both standard and expanded graphs, which are more detailed representations. We begin with graph collection and construction. Then, we construct various graph embeddings (unsupervised and supervised) into a latent space. Given such an embedding, the framework becomes task agnostic and active learning can be performed using any regression method and query strategy suited for regression. Within this framework, we investigate the impact of using different levels of information for active and passive learning, e.g., partially available labels and unlabeled test data. Despite our framework being domain agnostic, we validate it on a real-world application of software performance prediction, where the execution time of the source code is predicted. Thus, the graph is constructed as an intermediate source code representation. We support our methodology with a real-world dataset to underscore the applicability of our approach. Our real-world experiments reveal that satisfactory performance can be achieved by querying labels for only a small subset of all the data. A key finding is that Graph2Vec (an unsupervised embedding approach for graph data) performs the best, but only when all train and test features are used. However, Graph Neural Networks (GNNs) are the most flexible embedding techniques when used for different levels of information with and without label access. In addition, we find that the benefit of active learning increases for larger datasets (more graphs) and when the graphs are more complex, which is arguably when active learning is the most important.

Keywords Graph Neural Networks · Active Learning · Graph Representation Learning

1 Introduction

The effectiveness of machine learning applications often depends on the availability of a large quantity of high-quality annotated and labelled data. Obtaining such data can be challenging and resource-intensive, particularly in scenarios involving complex data structures like graphs. However, graph data presents unique challenges due to its non-linear and interconnected nature, which complicates the annotation process. Annotating graphs often requires significant human expertise and effort, particularly in large-scale or complex graph structures, which can be a bottleneck in the application of machine learning.

Active learning strategies offer a solution to these challenges by focusing on the most informative and uncertain data points for annotation, thereby reducing the amount of data that needs to be manually labelled while maintaining high-quality learning outcomes [1]. This targeted approach can lead to more efficient use of resources and time, making machine learning more accessible and feasible in real-world applications, for instance in image processing [2, 3, 4, 5], recommender systems [6], driver behaviour identification [7], sound event detection [8], classification of driving time series [9], reaction prediction in drug discovery [10], logged data analysis [11], medical analysis [12, 3], text processing [13], and person re-identification [14].

Although active learning has been extensively studied in the context of node-level classification tasks [15, 16, 17, 18], its application to graph-level learning, particularly for regression tasks, is not explored. Graph-level learning involves understanding the properties and relationships of entire graphs, as opposed to individual nodes and is critical for tasks such as graph regression, where predictions are made at the graph-level rather than at the node-level.

In many different domains, graphs can be expanded by adding more nodes and edges to improve the properties in molecular graphs [19, 20, 21] or to update the knowledge representation in knowledge graphs [22, 23]. Despite the importance of expanded graphs and their applications, the literature does not explore the resilience of active learning for expanded graphs.

In this paper, we present a unified active learning framework tailored to graph-level learning for regression tasks on both the standard graphs and an extension of them. The framework begins with the collection and construction of graphs, followed by the generation of graph embeddings (both supervised and unsupervised) into a latent space. This approach renders the framework task-agnostic, allowing for the application of any regression method and active learning query strategy available in the literature. We explore the impact of utilizing different levels of information for active and passive learning, such as partially available labels and unlabeled test data, as well as the training and testing features. Although our framework is designed to be domain-agnostic, we validate its effectiveness on a real-world application: software performance prediction. In this context, the execution time of the source code is predicted, with the graph constructed as an intermediate representation of the source code. Our approach is supported by real-world experimental results, which demonstrate that querying labels for only a small subset of the data can yield respectable performance.

As key findings, Graph2Vec outperforms all the other unsupervised and supervised embedding when the training and testing features are used without accessing the labels. However, GNNs tend to be the more flexible and can be used for all levels of information (i.e., it can utilize both labels and features of any available dataset). When the graphs are expanded, Graph2Vec shows consistent effectiveness, whereas for GNNs we observe marginally worse performance. As for active learning, we investigate common query strategies from the literature such as Coreset [5], Query-by-Committee [24] and uncertainty selection based on Gaussian Processes [25]. We find that no active learning query strategy consistently outperforms the others for all datasets, consistent with previous work on active learning [12]. In addition, we find that the benefit of active learning increases for larger datasets, in particular for the expanded versions of the graphs (i.e., when the data is more complex). Arguably, this is when active learning is the most important.

The aforementioned key findings highlight the potential of our framework in improving the efficiency and accuracy of machine learning applications for graph-level regression tasks. Our contributions are manifold and address several gaps in the current landscape of graph-based learning methodologies:

1. **Development of Specialized Graph Datasets:** We propose new graph datasets designed to be directly usable by researchers, facilitating further exploration and validation of graph learning techniques.
2. **Novel Active Learning Framework on the Graph-Level:** We introduce a flexible framework for active learning applied to graph data in regression tasks. This approach is distinct in its focus on graph-level dynamics rather than node-level interactions, filling a gap in existing literature.
3. **Expanded Graphs Handling:** Our framework is designed to efficiently handle expanded graphs, making it particularly suitable for complex, large-scale graph structures.

4. **Investigation of the Impact of Additional Information:** Our research extensively investigates how various types of additional information can enhance the active learning process. This exploration is crucial for understanding and maximizing the efficacy of active learning in complex scenarios.
5. **Application to Software Performance Prediction:** We utilize our active learning framework for real-world software performance prediction. This novel approach not only propels AI forward in the domain of software performance engineering, but it also provides an efficient and practical method for annotating and labeling source code data.
6. **Open-Source Active Learning Framework:** We provide the research community with an open-source implementation of our framework. This tool is versatile, supporting various settings and graph configurations, thereby enhancing its utility for a broad range of applications. The code and the data are publicly available at [26].

2 Background

In this section, we provide an overview of the fundamental concepts underlying our approach. We first introduce the notion of graphs, then we present how source code can be represented as a graph. Later, those concepts are used to explain our framework and to evaluate the predicted execution time of source code

2.1 Graphs

A graph is a mathematical structure used to model relational data across various domains such as social networks [27, 28, 29], biological networks [30, 31], interaction networks [32, 33, 34], and mobility networks [35, 36]. It is represented as a pair (V, E) where V is the set of vertices or nodes and E is the set of edges between the nodes, $E \subseteq \{(u, v) \mid u, v \in V\}$. The graph can be undirected if it lacks self-loops and has a symmetric adjacency matrix, or directed otherwise. A *path* $P = \{v_1, \dots, v_k\}$ is an ordered sequence of connected nodes, with its length being the number of nodes it contains, and the shortest path between two nodes is the path with the minimal length connecting them. The *node neighborhood* of a node v in graph $G = (V, E)$ is the set of nodes adjacent to v , and the *degree* of a node is the number of its neighbors. The *density* of a directed graph is defined as $\text{Density} = \frac{|E|}{|V|(|V|-1)}$. A *triad* in a graph is a subset of three connected nodes, classified as *closed* if it forms a triangle with three edges, otherwise *open*.

2.2 Source Code Representation

Different representations of code have been crafted for program analysis, aiming to understand program properties and optimize them. While mainly used for analysis and optimization, these representations also help characterize code, as explored in this study. Specifically, we delve into two fundamental representations: Abstract Syntax Trees (AST) and Control Flow Graphs (CFG), which form the basis for our approach to predict the execution time.

Listing 1: Simple example of C source code (from [37]).

```
void foo () {
    int x = source ();
    if ( x < MAX ) {
        int y = 2*x;
        sink (y);
    }
}
```

Abstract Syntax Tree (AST): Abstract syntax trees capture the nested structure of statements and expressions in programs, abstracting away specific syntax. For example, in C, a comma-separated list of declarations yields the same tree as two consecutive declarations. They are ordered trees with inner nodes representing operators and leaf nodes representing operands. As an example, consider Figure 1(a) showing the AST for the code sample given in snippet 1 by [37]. While useful for basic transformations and identifying similar code, they lack explicit representation of control flow and data dependencies, limiting their use in advanced code analysis tasks like detecting dead code or uninitialized variables.

Control Flow Graphs (CFG): A Control Flow Graph precisely outlines the sequence of code execution and the conditions required for specific execution paths. Nodes represent statements and conditions, connected by directed edges to signify control transfer. Unlike abstract syntax trees, these edges do not require a specific order. Predicate nodes have two edges representing true or false outcomes. Figure 1(b) displays the CFG for the code in snippet 1 by [37]. Control flow graphs are widely used in reverse engineering for program comprehension, although they lack data flow details despite depicting control flow.

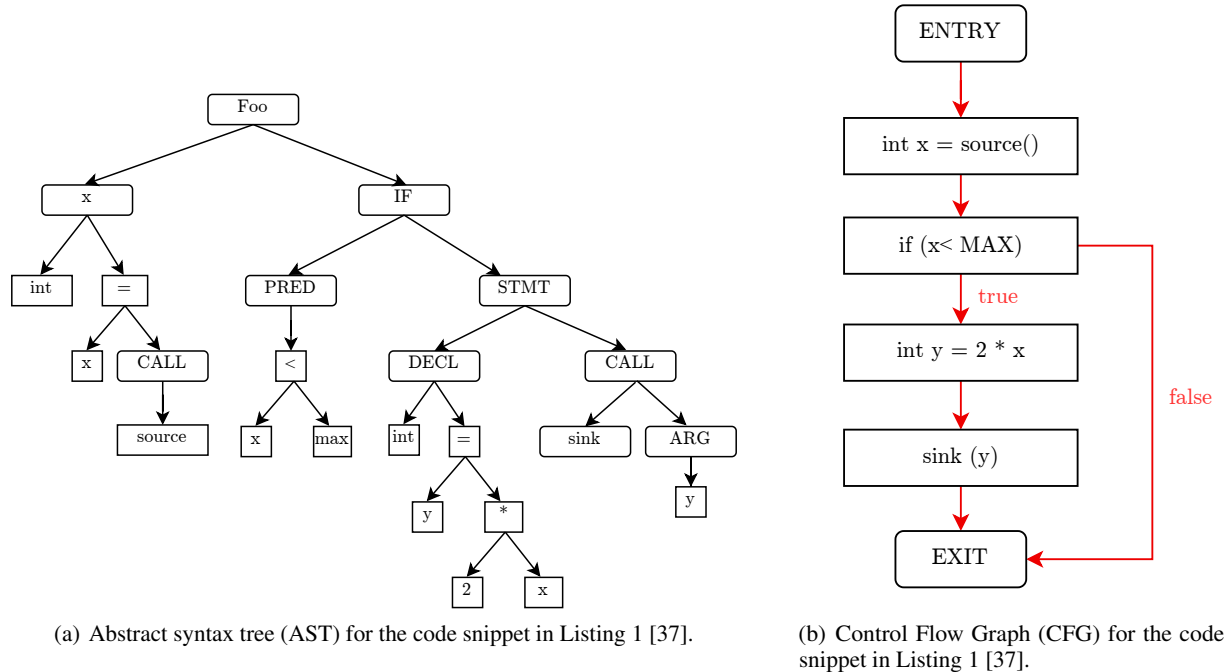


Figure 1: Example of Tree and Graph Representation for the code snippet in Listing 1 [37].

3 Related Work

Active Learning (AL) has been widely studied across various domains, including text [38] and image data [39], to enhance data annotation processes and enable more practical AI applications.

Graph data presents a distinct challenge for active learning, especially within densely connected networks [40, 41]. However, the application of AL to graph-level tasks remains an unresolved area of research. Several approaches have been proposed to address AL on node-level tasks. For instance, [18] introduced AGE, an active graph embedding framework that operates at the node-level using uncertainty and representativeness as querying strategies. Similarly, [17] developed a generic active learning framework that employs distance-based clustering. Both studies relied on Graph Convolutional Networks (GCN) for node representation learning.

Reinforcement learning has also been leveraged to enhance the selection of informative nodes in graph-based active learning. For example, the works in [16, 15] applied active learning to graph data using reinforcement learning. [16] proposed a Graph Policy Network (GPA) for transferable active learning on graphs, formalizing the process as a Markov decision process (MDP) and using reinforcement learning to identify the optimal query strategy. Conversely, [15] presented BIGENE, a batch active learning method formulated as a cooperative multi-agent reinforcement learning problem.

Multi-arm bandit strategies offer another perspective on active learning, optimizing node selection through strategic exploration. For example, the works in [42, 43] investigated multi-arm bandits in an active learning setting. [42] proposed ANRMAB, which uses Information Entropy, Node Centrality, and Information Density as querying strategies for node-level labeling. Meanwhile, [43] introduced ActiveHNE, a heterogeneous network embedding method that combines Network Centrality, Convolutional Information Entropy, and Convolutional Information Density as selection strategies based on uncertainty and representativeness.

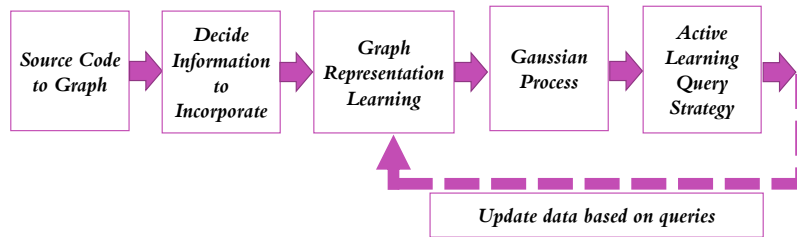


Figure 2: Representation learning and Active Learning Strategies

Despite these advances, a few limitations remain common across these studies: they primarily used benchmark datasets such as Citeseer, Cora, and Pubmed for validation; they employed semi-supervised learning; and they focused on the node-level. Our approach diverges by utilizing real-world datasets, operating at the graph-level, incorporating both supervised and unsupervised learning, and engaging with different graph sizes.

Our work is inspired by the study in [9], which introduced a flexible active learning framework for time series data. This framework embeds the data into a latent space, allowing for the use of any machine learning model and active learning strategy. Similarly, our research adopts this structure for graph data. However, we also conduct experimental studies to systematically assess the performance of this framework at various levels of information. For more details about our framework, see Section 4.

4 Learning Framework

In this section, we provide a detailed overview of our active learning framework. Figure 2 presents our framework for active learning. Section 4.1 begins by explaining the general setup for active and passive learning given a graph dataset. The remaining sections will then explain each of the components visualized in Figure 2.

4.1 Active and Passive Learning Procedure

We are given a dataset \mathcal{D} of N source code files (represented as graphs, see next section). For active learning, we then split this dataset into three parts, the initially labelled dataset \mathcal{L}_0 , the initially unlabeled dataset \mathcal{U}_0 and a test set \mathcal{T} . The purpose of the test set is to be able to evaluate the active learning procedure. Active learning can be seen as an iterative procedure where in each iteration i , one begins by training some regressor \mathcal{R}_i based on the currently available information, i.e., \mathcal{L}_i , \mathcal{U}_i and possibly \mathcal{T} .¹ Then, the current regressor \mathcal{R}_i is evaluated using the test set \mathcal{T} . Then, a query strategy is used to select the most informative batch $\mathcal{B} \subseteq \mathcal{U}_i$ of data items from \mathcal{U}_i based on information in the following components: \mathcal{R}_i , \mathcal{L}_i , \mathcal{U}_i and \mathcal{T} . Finally, the datasets are updated by setting $\mathcal{L}_{i+1} := \mathcal{L}_i \cup \mathcal{B}$ and $\mathcal{U}_{i+1} := \mathcal{U}_i \setminus \mathcal{B}$. This is repeated until a stopping criterion is met (e.g., if the labelling budget has been reached). In addition to active learning, we conduct experiments in the passive setting, which corresponds to setting $\mathcal{L} = \mathcal{L}_0$ and $\mathcal{U}_0 = \emptyset$. Then, one trains a regressor \mathcal{R} on \mathcal{L} and makes predictions on \mathcal{T} (i.e., the traditional supervised machine learning).

4.2 Transforming Source Code to Graphs

This section explains how to build the graphs from the source codes. As shown in Figure 3, we investigate Java source code files. We represent the source code as an AST intermediate representation. To compress both semantic and syntactical information, we augment the AST by adding edges that preserve both data and control the flow of the graphs. Hence, we arrive at a flow-augmented AST (FA-AST) graph, a concept that we introduced in our earlier work [44].

Our motivation for augmenting the AST comes from recent studies [45], emphasising the importance of rich code representation when using deep learning in software engineering. Hence, and given the complexity of predicting performance, prediction based on the syntactical information extracted from ASTs alone is not sufficient to achieve high-quality predictions. The AST's basic structural information is enriched with semantic information representing data and control flow. Consequently, the tree structure of the AST is generalized to a (substantially richer) graph, encoding more information than the code structure alone.

¹Note that here we assume for the test data only the data features might be available to be utilized, not the labels. Assume, for example, a photographer has taken two sets of photos from the same objects. For the first set (i.e., the training dataset) she has the image labels, but for the second set (i.e., the test dataset) only the images (without labels) are available. When training a classifier, she may then use the test images as well, in addition to labeled training dataset.

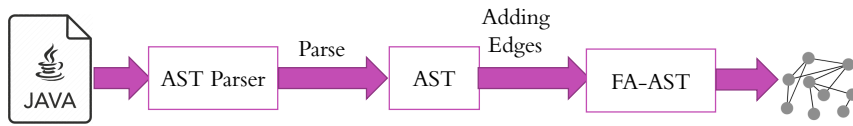


Figure 3: Source Code to Graph Process

4.2.1 Motivation Example

To understand how the graphs are built, we will present an example for a Java code file and then explain in detail how the FA-AST is built (see Listing 2).

Listing 2: A Simple JUnit 5 Test Case

```
package org.myorg.weather.tests;

import static
    org.junit.jupiter.api.Assertions.assertEquals;
import org.myorg.weather.WeatherAPI;
import org.myorg.weather.Flags;

public class WeatherAPITest {

    WeatherAPI api = new WeatherAPI();

    @Test
    public void testTemperatureOutput() {
        double currentTemp = api.currentTemp();
        Flags f = api.getFreezeFlag();
        if (currentTemp <= 3.0d)
            assertEquals(Flags.FREEZE, f);
        else
            assertEquals(Flags.THAW, f);
    }
}
```

AST Parsing In this example, a single test case `testTemperatureOutput()` is presented that tests a feature of an (imaginary) API. As common for test cases, the example is short and structurally relatively simple. Much of the body of the test case consists of invocations to the system-under-test and calls of JUnit standard methods, such as `assertEquals`.

A (slightly simplified) AST for this illustrative example is depicted in Figure 4. The produced AST does not contain purely syntactical elements, such as comments, brackets, or code location information. We make use of the pure Python Java parser `javalang`² to parse each test file and use the node types, values, and production rules in `javalang` to describe our ASTs.

Capturing Ordering and Data Flow In the next step, we augment this AST with different types of additional edges representing data flow and node order in the AST. Specifically, we use the following additional flow augmentation edges, in addition to the **AST child** and **AST parent** edges that are produced readily by AST parsing:

- **FA Next Token (b)**: This type of edge connects a terminal node (leaf) in the AST to the next terminal node. Terminal nodes are nodes without children. In Figure 4, an FA Next Token edge would be added, for example, between `WeatherAPI` and `api`.
- **FA Next Sibling (c)**: This connects each node (both terminal and non-terminal) to its next sibling and allows us to model the order of instructions in an otherwise unordered graph. In Figure 4, such an edge would be added, for example, connecting the first usage of `api` and with the `CONSTR` node (representing a Java constructor call).
- **FA Next Use (d)**: This type of edge connects a node representing a variable to the place where this variable is next used. For example, the variable `api` is declared in Line 10 in Listing 2, and then used next in Line 14.

Figure 5 shows an example augmenting the AST in Figure 4 (and, consequently, the example test case in Listing 2). Solid black lines indicate the AST parent and child relationships (for simplicity indicated through a single arrow, read

²<https://pypi.org/project/javalang/>

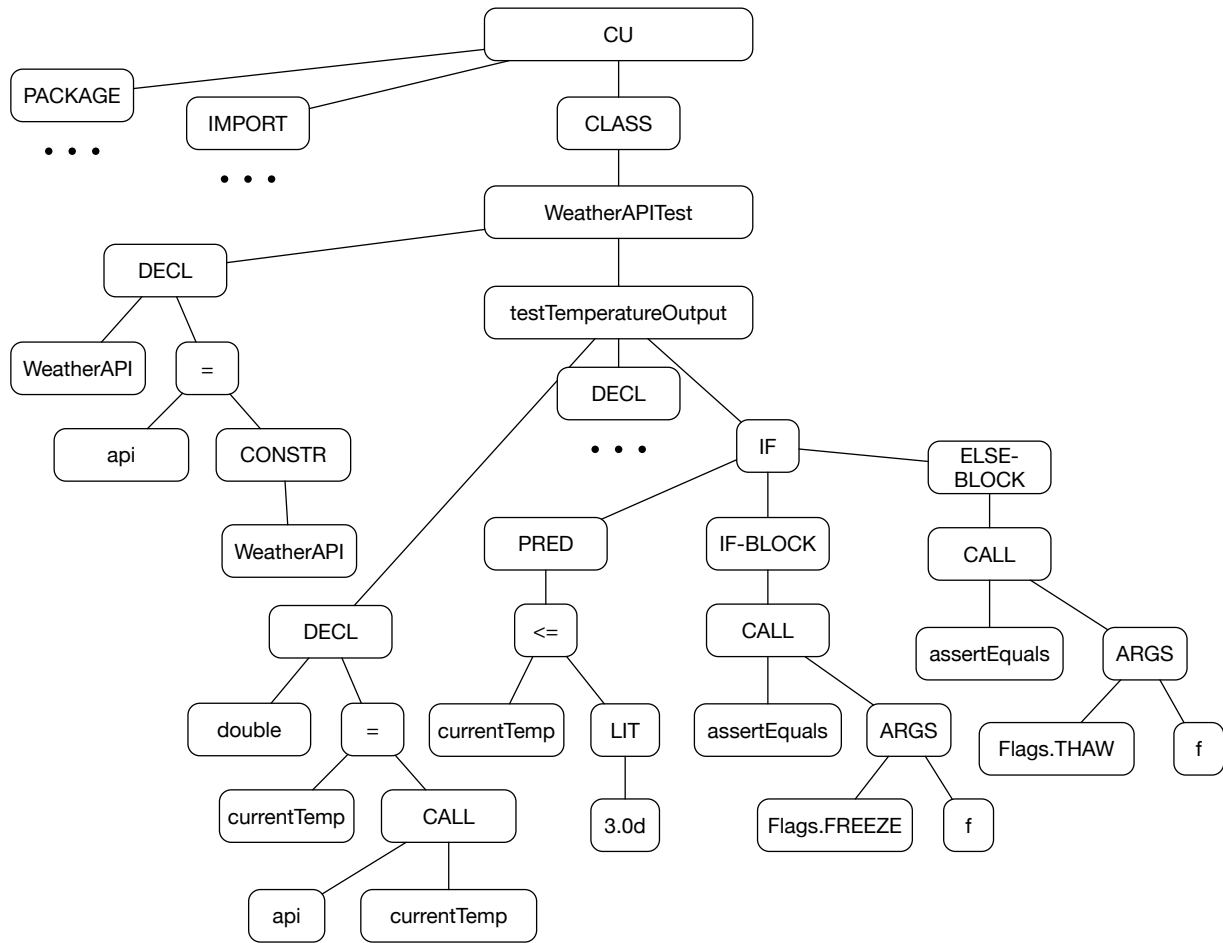


Figure 4: Simplified abstract syntax tree (AST) representing the illustrative example presented in Listing 2. Package declarations, import statements, as well as the declaration in Line 15 are skipped for brevity.

from top to bottom). Red dashed arrows refer to the new edges added to represent the data and control flow in the FA-AST, with letter codes indicating the edge type. Terminal nodes are connected with FA Next Token edges (b), modelling the order of terminals in the test case. Similarly, the ordering of siblings is modelled using FA Next Sibling edges (c). Finally, data flow is modelled by connecting each variable to their next usage via FA Next Use edges (d). Edge types (e), (f), and (i) represent a control flow statement, which will be discussed in the following. Multiple edges of different types are possible between the same nodes. For example, the terminal nodes `Flags.FREEZE` and `f` are connected via both, an FA Next Token (b) and an FA Next Sibling (c) edge.

Capturing Control Flow In a second augmentation step, we now add further edges representing the control flow in the test cases. We currently support *if* statements, *while* and *for* loops, as well as *sequential execution*. We currently do not support *switch* statements or *do-while* loops, as these are less common. Java source code containing these elements will still be parsed successfully, but these control flow constructs will not be captured by the FA-AST. Specifically, the following further edges are added (see also Figure 6):

- **FA If Flow (e):** This type of edge connects the predicate (condition) of the if-statement with the code block that is executed if the condition evaluates to `true`. Every if-statement contains exactly one such edge by construction.
- **FA Else Flow (f):** Conversely, this edge type connects the predicate to the (optional) else code block.
- **FA While Flow (g):** A while loop essentially entails two elements - a condition and a code block that is executed as long as the condition remains `true`. We capture this through a FA While Flow (g) edge connecting the condition to the code block, and an FA Next Use (d) edge in the reverse direction. The latter is used to model the next usage of a loop counter.

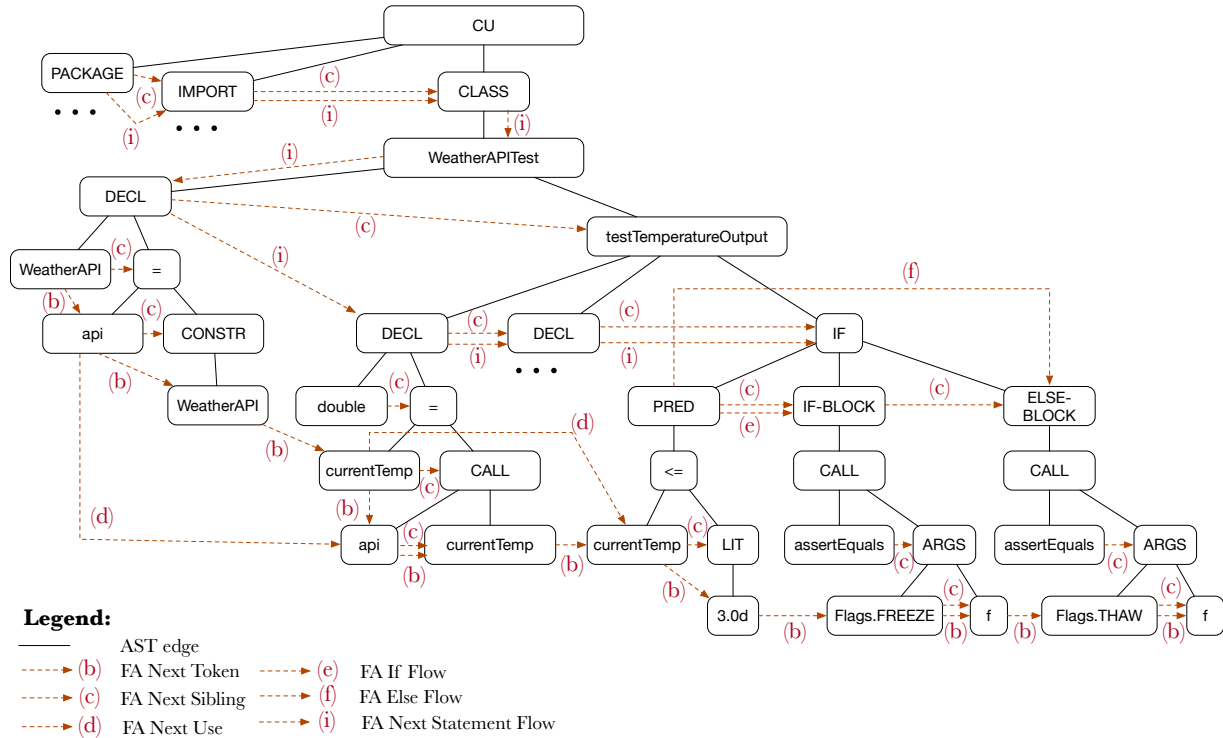


Figure 5: Flow-Augmented AST (FA-AST) for the example presented in Listing 2. Solid lines represent AST parent and child edges, and dashed lines different types of flow augmentations.

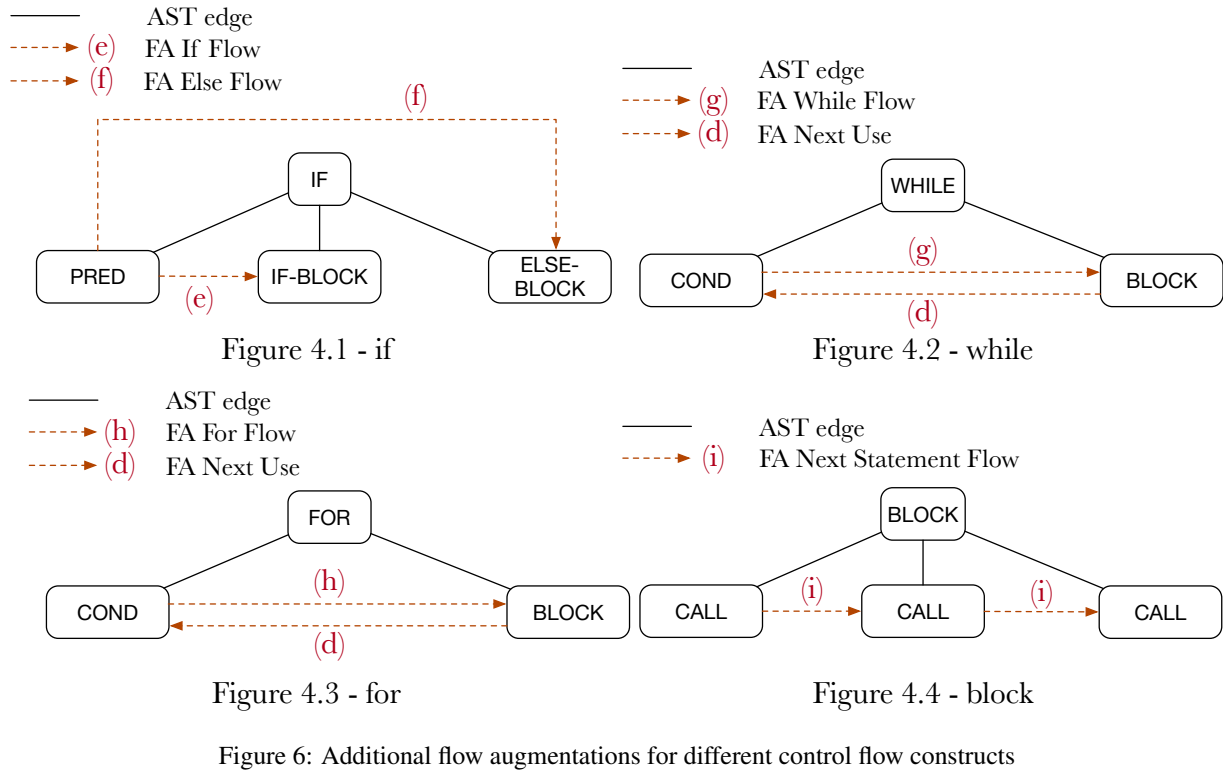
- **FA For Flow (h):** For loops are conceptually similar to while loops. We use FA For Flow (h) edges to connect the condition to the code block, and an FA Next Use (d) edge in the reverse direction. Similar to the modelling of while-loops, FA Next Use (d) relates to the usage (typically incrementing) of a loop counter.
- **FA Next Statement Flow (i):** In addition to the control flow constructs discussed so far, Java of course also supports the simple sequential execution of multiple statements in a sequence within a code block. FA Next Statement Flow edges (i) are used to represent this case. Different from the constructs discussed so far, a code block can contain an arbitrary number of children, and the FA Next Statement Flow edge is always used to connect each statement to the one directly following it.

Referring back to Figure 5, two types of control flow annotations are visible: the modelling of the if-statement in lines 16 to 19 of the test case on the right-hand side and various edges representing sequential executions (FA Next Statement flow (i)). Further note how flow annotation adds a large number of edges to even a very small AST, transforming the syntax tree into a sparse graph. This rich additional information can be used in the next step by our GNN model to predict highly accurate test execution times.

4.3 Depth of FA-AST Parsing

One challenge with representing source code as graphs is that graphs tend to become very large. We address this challenge by limiting how deeply we parse the AST. We investigate two alternatives:

- **File-Level Parsing:** in the first alternative, we parse the AST only on the level of individual Java source files. References to Java constructs (e.g., classes, functions, etc.) not implemented in this file are turned into leaf nodes (and not resolved further). This leads to graphs of manageable size and has the added benefit of simplifying parsing, but evidently much expressive information is lost.
- **System-Level Parsing:** in the second alternative, the parser has access to all source code files of the study subject system (e.g., all source code files of Hadoop when constructing FA-ASTs for Hadoop), and the all references to classes or functions that are implemented in the study subject are resolved fully. External dependencies or calls to the Java system library are not resolved, these remain represented as leaf nodes. This



parsing strategy leads to substantially larger and more complex graphs, but has the benefit that more knowledge about the performance of methods of the study subject is represented in the graph.

4.4 Graph Representation Learning

The graph structure of the data items in \mathcal{D} yields a restriction on the types of regression models that can be used, and thus the types of query strategies to use for active learning. Therefore, we investigate a number of unsupervised and supervised approaches to constructing embeddings that can be used to project the graph data into a latent space where any regression model (and thus query strategy) can be used. In this section, we outline each of the embeddings that we investigate in this work.

Since our focus is on directed graphs, we use embedding algorithms compatible with directed graphs where the adjacency matrix is not symmetric. For this purpose, we explore three main approaches: unsupervised embeddings (based on Graph Neural Networks (GNNs) and shallow embedding algorithms), supervised embeddings (based on GNNs) and manual embeddings (based on manually extracted graph features). Each of these categories are listed and explained below.

4.4.1 Unsupervised embeddings.

Figure 7 illustrates the hierarchy of unsupervised embedding algorithms used. The hierarchy is inspired by [46]. We have two main types of shallow embedding approaches: matrix factorization and skip-gram. In matrix factorization, we use the Graph Representation (GR) approach [47] and Higher-Order Proximity Preserved Embedding (HOPE) [48], both of which are compatible with directed graphs.

GR operationalizes matrix factorization to capture both local and global structural information within graphs. It does this by first constructing k-step probability transition matrices for different lengths of walks in the graph, essentially encoding the connectivity patterns at various scales. GR then applies matrix factorization to these transition matrices, enabling the extraction of node embeddings that reflect the composite of these patterns.

HOPE, on the other hand, employs matrix factorization to preserve high-order proximities between nodes in a graph. It constructs a similarity matrix based on certain measures of node similarity (such as the Katz Index or rooted PageRank) that encapsulates higher-order connections beyond immediate neighbours. By factorizing this similarity matrix, HOPE

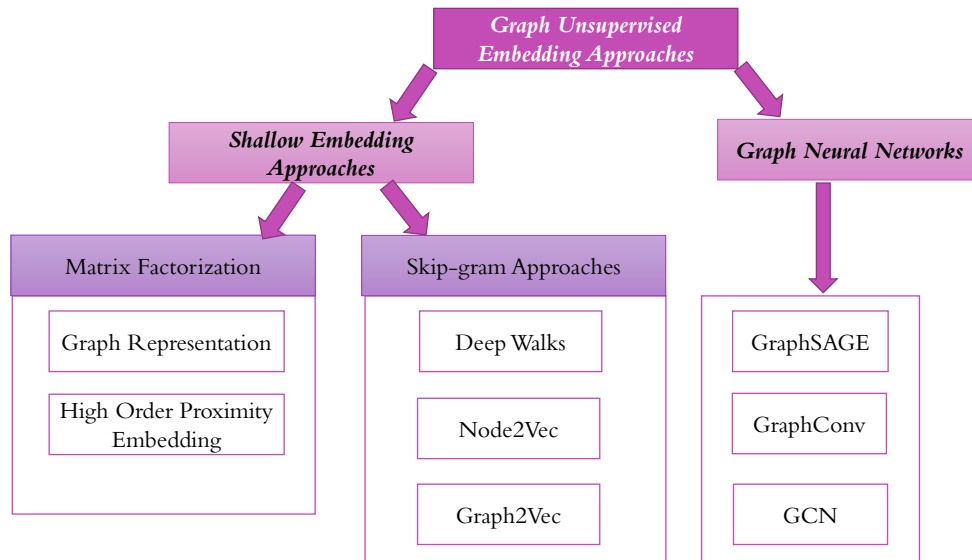


Figure 7: Hierarchical structure of the different unsupervised graph embedding algorithms used in this study.

efficiently generates node embeddings that maintain the asymmetric transitive relationships, especially useful in directed graphs, by focusing on scalable, low-rank approximations to handle large-scale graphs.

These algorithms operate at the node-level, resulting in an embedding array for each graph rather than a vector. Therefore, we aggregate the embedding using mean and sum aggregation to represent the graph embeddings as vectors. For skip-gram-related methods, we use DeepWalk [49], Node2Vec [50], both of which learn the embedding at the node-level, and Graph2Vec which is the only method for the shallow embedding category that returns a vector representing the embedding for the entire graph.

DeepWalk utilizes random walks to sample sequences of nodes from a graph analogously to sentences in a corpus. By treating these sequences as "sentences," DeepWalk applies the skip-gram model to learn node embeddings that preserve the neighbourhood structure of the graph. This approach effectively captures the local connectivity patterns around each node, embedding them into a low-dimensional space that reflects the structural similarities between nodes.

Node2Vec builds upon the DeepWalk framework by introducing a flexible notion of a node's neighbourhood. It achieves this by parameterizing the random walks to balance between breadth-first sampling (capturing immediate neighbourhood structures) and depth-first sampling (exploring more distant parts of the graph). This controlled exploration allows Node2Vec to learn embeddings that can reflect both homophily and structural equivalences, thereby providing a more nuanced representation of node relationships in the embedding space.

Graph2Vec creates Weisfeiler-Lehman tree features for nodes in graphs. A graph feature co-occurrence matrix is decomposed to generate graph representations using these features.

According to [46], shallow embedding methods are applied to a finite set of input graphs and cannot be applied to instances different from those used to train the model.

In addition to the shallow embeddings, we train GNNs (without labels) to compute unsupervised embeddings. We employ three state-of-the-art GNN architectures, namely GCNConv [51], GraphSAGE [52] and GraphConv [53]. This is done using the well-known autoencoder neural network architecture [54] (in combination with one of the mentioned GNNs). In short, this works by training the corresponding GNN to reconstruct the input graphs. After training, an embedding is extracted from the last layer of the corresponding GNN.

4.4.2 Supervised embeddings.

For supervised representation learning (embedding), we employ three state-of-the-art architectures, namely GCNConv [51], GraphSAGE [52], and GraphConv [53]. These methods are explained in detail below.

- GCNs leverage the concept of convolutional operations on graph-structured data. The model updates a node's representation by aggregating its neighbours' features.

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}) \quad (1)$$

Where $H^{(l)}$ is the matrix of node features at layer l , $\tilde{A} = A + I_N$ is the adjacency matrix A with added self-connections I_N , \tilde{D} is the degree matrix of \tilde{A} , $W^{(l)}$ is the weight matrix for layer l , and σ is a non-linear activation function.

- GraphSAGE (Graph Sample and Aggregation) generates embeddings by sampling and aggregating features from a node's local neighbourhood.

$$h'_i = \sigma(W \cdot \text{MEAN}(\{h_i\} \cup \{h_j, \forall j \in \mathcal{N}(i)\})) \quad (2)$$

Where h_i is the feature vector of node i , $\mathcal{N}(i)$ is the set of its neighbours, and W is the weight matrix associated with the aggregator function.

- GraphConv (Spectral Graph Convolution) employs spectral graph convolutions by leveraging the graph Laplacian's eigenbasis. This approach efficiently captures the graph structure at different scales.

$$H^{(l+1)} = \sigma(U \Lambda^{(l)} U^T H^{(l)} W^{(l)}) \quad (3)$$

Where $H^{(l)}$ is the matrix of node features at layer l , U is the matrix of eigenvectors of the normalized graph Laplacian $L = I_N - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$, $\Lambda^{(l)}$ is a diagonal matrix of spectral filters (parameters) at layer l , $W^{(l)}$ is the weight matrix for layer l , and σ is a non-linear activation function.

Given this embedding, the active and passive learning is performed using the regression model introduced in Section 4.6. The reasons for this is to be consistent with the unsupervised embeddings (that will use the same regression model) and because the performance turned out to be slightly better compared to the predictions made by the last (linear) layer of the GNN.

4.4.3 Manual embedding

We also consider a manually constructed embedding by extracting a set of graph metrics for each of the graphs (data items). Here, we represent each graph as a vector of metrics that are directly extracted from the graphs without learning. Figure 8 shows a categorization of the extracted metrics. Below we list and explain each of the metrics.

1. **Integration Metrics** [55]: those metrics capture the spreading of information within the network. In particular:
 - *Characteristic Path Length*: This metric represents the average shortest path length between all pairs of nodes in the graph.
 - *Global Efficiency*: It measures the average inverse shortest path length between all pairs of nodes in the graph.
 - *Local Efficiency*: Local efficiency is computed for each node as the global efficiency of its neighbourhood subgraph and then averaged over all nodes.
2. **Resilience Metrics** [56]: These metrics assess the robustness of a graph and its ability to maintain its structure and functionality despite changes or failures. In particular, we consider
 - *Assortativity Coefficient*: this metric measures the correlation between the degrees of a node and its neighbourhood.
3. **Segregation Metrics** [55]: they quantify the degree to which nodes in a graph tend to form tightly knit communities or clusters. Two metrics related to this category are listed below.
 - *Global Clustering Coefficient (GCC)* [57]: it is the number of closed triplets over the total number of triplets.

$$GCC = \frac{1}{n} \sum_{v \in G} \frac{2T(v)}{\deg(v)(\deg(v) - 1)}$$

where $T(v)$ is the number of triangles through node v .

- *Transitivity*: defined as $3 \frac{\#triangles}{\#triads}$.
4. **Basic Graph Metrics**: Basic graph metrics describe a graph's fundamental structure, size, and connectivity. In this category, we are inspired by [58]. Five related metrics related to this category are listed below as the following:

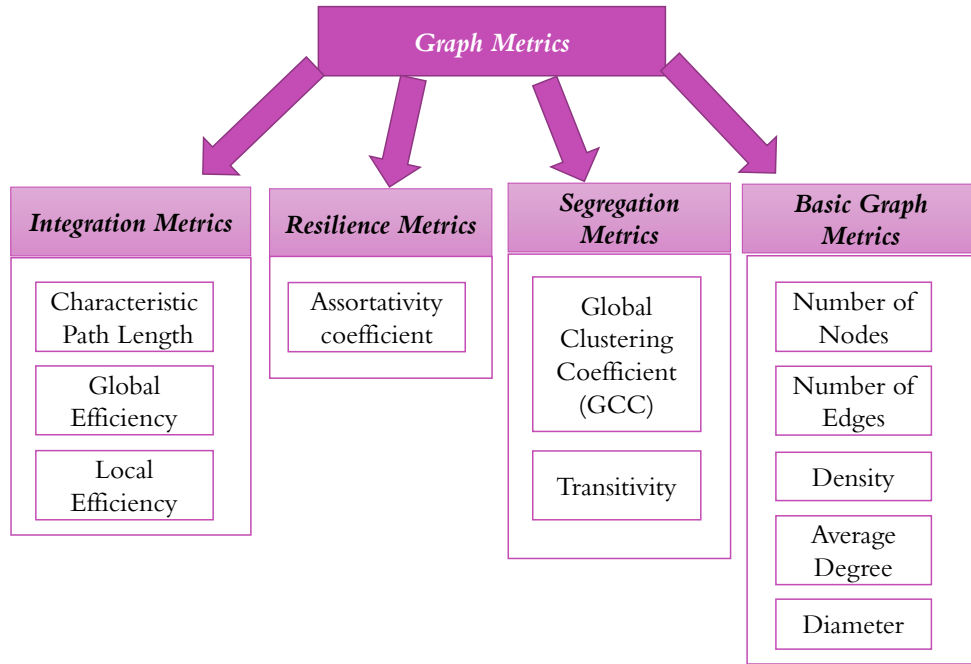


Figure 8: Hierarchy of graph-based metrics.

- *Number of Nodes*: The total number of nodes in the graph.
- *Number of Edges*: The total number of edges in the graph.
- *Diameter*: The diameter D is the shortest path length between the two most distant nodes in the network.
- *Edge Density*: The ratio of the actual number of edges to the maximum possible number of edges.
- *Average Degree*: The average number of degrees.

By considering these categories and their associated metrics, we can understand the graph's properties comprehensively, which can be valuable in various graph analysis and machine learning tasks.

4.5 Incorporating Different Information

When constructing the embeddings and performing the active/passive learning procedure outlined in Section 4.1, one can utilize different levels of information about the datasets. We describe how this is done for active learning and passive learning below. Let $\mathbf{X}_{\mathcal{A}}$ and $\mathbf{Y}_{\mathcal{A}}$ refer to the feature vectors and labels respectively of some generic dataset \mathcal{A} .

4.5.1 Active Learning

For active learning we have three datasets: \mathcal{L}_i , \mathcal{U}_i and \mathcal{T} . In principle, the information that can be used to construct the embeddings and perform the active learning are the labels and features of these datasets, i.e., $\mathbf{X}_{\mathcal{L}_i}$, $\mathbf{X}_{\mathcal{U}_i}$, $\mathbf{X}_{\mathcal{T}}$, $\mathbf{Y}_{\mathcal{L}_i}$, $\mathbf{Y}_{\mathcal{U}_i}$ and $\mathbf{Y}_{\mathcal{T}}$. As suggested by [59], it is important to separate the reported active learning results depending on what information is used. For example, if one notices improved performance when using the features of the unlabeled data items $\mathbf{X}_{\mathcal{U}_i}$ (through, e.g., semi-supervised learning) compared to not doing so, it is important not to fully credit this improvement to the query strategy used. Partial credit must be given to the learning algorithm used since it was able to effectively use the additional information. Note that for the active learning pipeline followed in this paper, both the construction of the embedding and the active learning can utilize different levels of information (separately). For simplicity, the active learning (given some embedding) is always done based on the training features and training labels only (i.e., supervised training based on $\mathbf{X}_{\mathcal{L}_i}$ and $\mathbf{Y}_{\mathcal{L}_i}$). However, for the construction of the embeddings, we considered four different levels of information, each of which are listed and explained below. Note that we never use $\mathbf{Y}_{\mathcal{T}}$, i.e., the labels of the test dataset.

- $\mathbf{X}_{\mathcal{L}_i}$, $\mathbf{X}_{\mathcal{U}_i}$ and $\mathbf{X}_{\mathcal{T}}$. This category is only applicable to the unsupervised embeddings (since the labels are not used). In this case, we simply construct the embedding using all available features and then embed \mathcal{L}_i , and \mathcal{U}_i and \mathcal{T} into the resulting latent space before performing the active learning.
- $\mathbf{X}_{\mathcal{L}_i}$ and $\mathbf{X}_{\mathcal{U}_i}$. This category is only applicable to the unsupervised embeddings (since the labels are not used). For the GNN based unsupervised embeddings it is straightforward. One begins by constructing an embedding using $\mathbf{X}_{\mathcal{L}_i}$ and $\mathbf{X}_{\mathcal{U}_i}$. Given the embedding, \mathcal{L}_i , \mathcal{U}_i and \mathcal{T} can be projected into the resulting latent space before doing the active learning. For the shallow embeddings this does not work since it is not possible to project new data items into the resulting latent space (i.e., only the data items that were used to construct the latent space can be accessed in the resulting latent space). Instead, we first construct an embedding based on $\mathbf{X}_{\mathcal{L}_i}$ and $\mathbf{X}_{\mathcal{U}_i}$ and access \mathcal{L}_i and \mathcal{U}_i in the resulting latent space. Then, we construct an embedding based on \mathcal{L}_i , \mathcal{U}_i and \mathcal{T} and access \mathcal{T} in the resulting latent space. It should be noted that in this case \mathcal{T} is in a different (but hopefully similar) feature space compared to \mathcal{L}_i and \mathcal{U}_i . Finally, we consider the manual embedding to belong to this category since it does not use the test features when it is constructed. However, it should be noted that it is not strictly the same, since for the manual embedding the feature representation of each graph is only based on information in the graph itself (i.e., it is independent of all other graphs).
- $\mathbf{X}_{\mathcal{L}_i}$ and $\mathbf{Y}_{\mathcal{L}_i}$. This category is only applicable to the supervised embedding approach (based on GNNs) since it uses the labels of \mathcal{L}_i . In this setting one simply performs supervised training of a GNN based on $\mathbf{X}_{\mathcal{L}_i}$ and $\mathbf{Y}_{\mathcal{L}_i}$. Then, all data is projected into the latent space of the last layer of the GNN before performing the active learning.
- $\mathbf{X}_{\mathcal{L}_i}$, $\mathbf{Y}_{\mathcal{L}_i}$ and $\mathbf{X}_{\mathcal{U}_i}$. This category is only applicable to the supervised embedding approach (based on GNNs) since it uses the labels of \mathcal{L}_i . This setting works identically to the previous category except that we also use pseudo-labels for data items in \mathcal{U}_i (i.e., semi-supervised learning). After some investigation, this category turned out to not lead to improved performance for our datasets and models, and is therefore not reported in the results.

4.5.2 Passive Learning

The passive learning is conducted in a corresponding fashion to the active learning described above by simply setting $\mathcal{L} = \mathcal{L}_0$ and $\mathcal{U}_0 = \emptyset$.

4.6 Regression model

Given some graph embedding, we require a regression model to make predictions (either for passive learning or active learning). Our framework is generic enough to utilize any regression model. In this project, we investigate Gaussian Process Regressors (GPR). The reason is that GPRs are both powerful regressors while also providing an explicit uncertainty model due to their probabilistic nature [60]. This uncertainty model allows us to define a natural acquisition functions that can be used in an active learning setting. This is discussed more in the next section. We refer to [60] for the mathematical details of GPRs.

4.7 Query Strategies for Active Learning

In this paper, we consider batch active learning [61]. In batch active learning, a batch of data points $\mathcal{B} \subseteq \mathcal{U}_i$ is selected in each iteration of the active learning procedure (instead of a single data point). This adds an extra level of complexity in the construction of query strategies, because the selected batch \mathcal{B} must contain data points that are jointly informative (i.e., not redundant). With this in mind, we list and explain all query strategies (acquisition functions) used in the active learning experiments below. All query strategies below are commonly investigated in active learning and are not specific to graph data. This highlights the benefit of our framework: given a graph embedding, we can utilize any model (GPR in our case) and any active learning query strategy suited for this model, none of which are specific to the graph data.

- **Random:** This corresponds to selecting a batch $\mathcal{B} \subseteq \mathcal{U}_i$ uniformly at random, which is a common baseline strategy.
- **Coreset:** This was originally introduced by [5], and has become a well established baseline method for batch active learning. Intuitively, it aims to select a batch $\mathcal{B} \subseteq \mathcal{U}_i$ that is maximally representative of \mathcal{U}_i while simultaneously being maximally different from the samples in \mathcal{L}_i (i.e., informative). In general, representativeness is quantified based on distances in feature space. In our case, that corresponds to distances in the latent space provided by the graph embeddings. We utilize the efficient k-Center-Greedy algorithm described in the original work [5].
- **Variance:** This is based on the uncertainty estimations provided by the GPR. Due to the probabilistic nature of GPRs, it can produce an estimate of the variance for every data point. Let $\sigma(\mathbf{x})$ correspond to the variance of

some data item $x \in \mathcal{U}_i$ (estimated by the GPR). A data point with large variance indicates the GPR is uncertain about this data point, and may therefore be informative if labeled and included in the labeled data set. We can then select the top- $|\mathcal{B}|$ data points from \mathcal{U}_i according to $\sigma(\mathbf{x})$: $\mathcal{B}^* = \arg \max_{\mathcal{B} \subseteq \mathcal{U}_i, |\mathcal{B}|=B} \sum_{\mathbf{x} \in \mathcal{B}} \sigma(\mathbf{x})$, where B is the batch size.

- **Query-by-committee (QBC):** In general, this corresponds to fitting n estimators to (potentially bootstrapped) subsets of the labelled data. Then, a prediction is made by each of the estimators for all the data items in \mathcal{U}_i . If the estimators disagree strongly about a data point $\mathbf{x} \in \mathcal{U}_i$, this indicates large uncertainty and thus informativeness. In this paper, we employ QBC by training 10 GPR estimators on different bootstrapped subsets of the training data \mathcal{L}_i . Let $\sigma_i(\mathbf{x})$ be the variance of estimator i . We then compute the average as $\sigma_{\text{avg}}(\mathbf{x}) = \frac{1}{10} \sum_{i=1}^{10} \sigma_i(\mathbf{x})$. A batch is then selected as for the variance query strategy described above: $\mathcal{B}^* = \arg \max_{\mathcal{B} \subseteq \mathcal{U}_i, |\mathcal{B}|=B} \sum_{\mathbf{x} \in \mathcal{B}} \sigma_{\text{avg}}(\mathbf{x})$.

Finally, **variance** and **QBC** are single-sample acquisition functions that do not explicitly consider the joint informativeness among the elements in a batch \mathcal{B} . This may lead to redundancy in the batch, but has the benefit of avoiding the combinatorial complexity of selecting an optimal batch, which is a common problem for batch active learning [61]. However, the work in [62] proposes a simple method for improving the batch diversity for single-sample acquisition functions using noise. For both variance and QBC we utilize the *power* acquisition method. For variance, this corresponds to modifying $\sigma(\mathbf{x}) := \log(\sigma(\mathbf{x})) + \epsilon$ where $\epsilon \sim \text{Gumbel}(0; 1)$, before selecting the top- $|\mathcal{B}|$ elements. This works analogously for QBC. The adjusted versions of variance and QBC will be referred to as **PowerVariance** and **PowerQBC**, respectively.

4.8 Limitations and Challenges

Despite the robustness of our framework through the usage of different embedding techniques, utilization of different levels of information, and the investment in different selection methods of active learning, our framework does face some challenges and limitations. This section outlines the main practical challenges and limitations of our proposed framework:

- **Access to Oracle:** A pivotal challenge arises from the reliance on oracles to acquire labels. In our setting, the oracle could correspond to software developers who execute code files in order to retrieve the execution time. This means that expensive computational resources must be available, which adds a monetary cost, in particular if cloud instances are utilized.
- **Variability in Oracle Costs:** In practice, we may have multiple oracles, i.e., multiple software developers with different levels of experience and different access to computational resources. This means that a query to each oracle may have different costs. However, in this paper, we assume we have only one oracle, where each query incurs the same cost (as is common in previous work on active learning).
- **Computational Resource Requirements:** The comprehensive nature of our framework demands significant computational resources for graph learning and executing active learning iterations. This is particularly pronounced in supervised settings where re-training of the GNN model is required with each update to the training set after label acquisition, thus intensifying time and resource consumption.

5 Experiments

In this section, we describe the experiments and present the results.

5.1 Research Objectives

This section outlines the principal research objectives explored through experiments on the proposed framework. Our primary goal is to explore the application of active learning in the context of graph learning on a graph-level, with a particular emphasis on directed sparse graphs. Nonetheless, it is posited that the framework holds potential applicability to a broader spectrum of graphs, contingent upon the adaptation of embedding techniques suitable for variants such as undirected graphs. In pursuit of these aims, the following research questions will guide our investigation:

- To what extent can active learning contribute to graph-level learning?
- Among the active learning query strategies evaluated, which demonstrate superior performance in conjunction with specific embedding techniques?
- Are the results obtained through the framework robust and consistent when applied to expanded graphs?

Table 1: Overview of the OSSBuilds and HadoopTests datasets.

	Project	Description	Files	Runs	File-Level Parsing		System-Level Parsing	
					Nodes	Vocab.	Nodes	Vocab.
OSSBuilds	systemDS	Apache Machine Learning system for data science lifecycle	127	1321	110651	3161	114904	3205
	H2	Java SQL DB	194	1391	405706	17972	432375	18326
	Dubbo	Apache Remote Procedure Call framework	123	524	75787	4499	77142	4505
	RDF4J	Scalable RDF processing	478	1055	214436	10755	242673	10844
	Total		922	4291	806580	36387	867094	36880
HadoopTests	Hadoop	Apache framework for processing large datasets on clusters	2895	24348	4314360	135408	5090798	138952

5.2 Dataset Collection

In our experiments, to increase reliability, we use two different real-world datasets of performance measurements. The first dataset (*OSSBuild*) is real build data collected from the continuous integration systems of four open-source systems. The second (*HadoopTests*) is a larger dataset we have collected ourselves by repeatedly executing the unit tests of the Hadoop open-source system in a controlled environment. A summary of both datasets is provided in Table 1. In the following subsections, we provide some additional information about each of the two datasets that we used in the experimental studies.

5.2.1 OSSBuild Dataset

In this dataset (originally used in [44]), information about test execution times in production build systems was collected for four open-source projects: systemDS, H2, Dubbo, and RDF4J. All four projects use public continuous integration servers containing (public) information about the project’s builds, which we harvested for test execution times as a proxy of performance in summer 2021. Basic statistics about the projects in this dataset are described in Table 1 (top). "Files" refers to the number of unit test files we collected execution times for, "Runs" is the (total) number of executions of files we extracted data for, whereas "Nodes" and "Vocabulary Size" indicate the resulting graphs (for both file and system-level parsing). Prior to parsing the test files, we remove code comments to reduce the number of nodes in each graph (by construction irrelevant). We note that we have 60514 more nodes for system-level parsing and 493 new vocabs.

5.2.2 HadoopTests Dataset

To address limitations with the OSSBuilds dataset (primarily the limited number of files for each individual project in the dataset), we additionally collected a second dataset for this study. We selected the Apache Hadoop framework since it entails a large number of test files (2895) of sufficient complexity. We then executed all unit tests in the project five times, recording the execution duration of each test file as reported by the JUnit framework (in millisecond granularity). As an execution environment for this data collection, we used a dedicated virtual machine running in a private cloud environment, with two virtualized CPUs and 8 GByte of RAM. Following performance engineering best practices, we deactivated all other non-essential services while running the tests. Statistics about the HadoopTests dataset are described in Table 1 (bottom).

Since we have more files in HadoopTests, we have more added nodes to the system-level parsing setting. Thus 776438 nodes are added to the graphs in the system-level parsing, and we get 3544 more vocabs.

5.2.3 Dataset Selection Rationale

The selection of this dataset was guided by several considerations, underscoring its suitability for our research objectives:

- The dataset’s real-world origin enhances the credibility and applicability of our research findings and the proposed framework.
- Its characteristics offer potential for generalization to diverse graph datasets.

- Notably, existing research on active learning for graphs predominantly focuses on node-level tasks (classification or regression). Our datasets provide the opportunity to investigate graph-level regression tasks, a field that, to our knowledge, has not been extensively explored in the existing literature.
- The variation in graph sizes is particularly important for our research. It encompasses graphs derived from file-level parsing, which can be further expanded through system-level parsing by incorporating additional nodes, and edges. This aspect, especially in the context of active learning, represents a novel research direction not explored in literature.

It is worth mentioning that our work provides a public and real-world graph dataset, enabling researchers to investigate and use it in research. The dataset is publicly available at [26].

5.3 Analysis of Graphs

We want to annotate each source code file with the corresponding scalar value related to execution time. The source code is represented as a graph. In particular, each graph represents a Java source code file (a JUnit test case). As aforementioned, the base structure is a tree that is then extended to a graph adding edges representing program control flow [44].

Table 2 shows the average statistics of the input graphs. In particular, we report the average number of nodes ($|V|$), the average number of edges ($|E|$), the density, the average global clustering coefficient (GCC), the average number of cycles and the average tree similarity. We define a simple function to measure how similar the graph is to a tree ($tree - sim$) as the number of edges that have to be removed to convert the graph into a tree, i.e.,

$$tree - sim = \frac{|E| - (|V| - 1)}{(|V| - 1) \left(\frac{|V|}{2} - 1 \right)}. \quad (4)$$

The formula has to be interpreted as the number of edges of the graphs minus the number of edges of a tree with N nodes, normalized. If the input graph is a tree, then we have that $tree - sim$ is equal to 0, while if the graph is complete, $tree - sim$ is equal to 1.

Table 2: Average statistics of the input graphs of System Level Parsing.

Dataset	type	$ V $	$ E $	Diameter	Density	GCC	$tree - sim$
OSSBuilds	File-level	875	1679	14	0.014	0.16	0.007
	System-level	940	1848	13	0.013	0.15	0.006
HadoopTests	File-level	1490	1848	15	0.005	0.15	0.003
	System-level	1734	3428	14	0.006	0.15	0.003

From Table 2, it is easy to see that the input graph has a high diameter. In fact, if we generate a random graph [63] with the same number of nodes and the same density as the original ones, we obtain an average diameter of 2 and 4 for OSSBuilds and HadoopTests, respectively. It is also easy to see that the input graphs are quite sparse. Finally, in both datasets, the $tree - sim$ is close to zero. Thus, we can conclude that input graphs are similar to trees. We report a detailed analysis of the input graphs in appendix A.

5.4 Experimental Setup

In this section, we describe the experimental setup. Each experiment has been executed on a computer with four GPU NVIDIA Tesla A40 with 48GB of memory, two CPU Xeon(R) Gold 6338, and DDR4 RAM of 256GB. However, the framework can be executed on less powerful machines with longer execution times as a consequence.

We used the Scikit-learn [64] implementation of Gaussian Process Regressors with a Matern kernel. In the passive setting, the hyperparameters of the Matern kernel were fine-tuned. For active learning, the hyperparameters of the Matern kernel were fine-tuned in each iteration based on the currently available labelled data in \mathcal{L}_i . The GNN models used for both supervised and unsupervised embeddings consist of three layers with 30 neurons each. Since each layer learns a node representation, we compute the graph representation by concatenating the sum, average, and max of the node representation, resulting in an embedding of 90 dimensions. The Adam optimizer [65] is employed with a learning rate of 0.001, and the loss used is the Mean Squared Error.

We measure the quality of the predictions by computing the Pearson correlation score between the predicted value and the real value. A larger Pearson correlation score implies better quality predictions. In B.3 we include results with the Root Mean Squared Error (RMSE) metric.

5.5 Results

In this section, we present the results of both the passive and active learning experiments. In Section 6 we discuss the conclusions from the results in detail.

5.5.1 Passive Learning

To perform passive learning, we utilize a training set \mathcal{L} and a test set \mathcal{T} . For each embedding, we train a Gaussian process (GP) using \mathcal{L} and then use it to predict the execution time of all test data items in \mathcal{T} . Additionally, all passive learning results correspond to the average of 15 runs with different seeds, where for each method, the mean and standard deviation (STD) values are reported.

We will show the results for file-level parsing and system-level parsing.

Table 3: Results for Unsupervised Embedding for graphs of File Level Parsing.

		Train and Test Features		Train Features		
		OSSBuilds	HadoopTests	OSSBuilds	HadoopTests	
Shallow Embedding	Graph2Vec	0.74 \pm 0.03	0.74 \pm 0.02	NA	NA	
	GR	mean	0.58 \pm 0.03	0.50 \pm 0.03	NA	NA
		sum	0.47 \pm 0.05	0.46 \pm 0.04	NA	NA
	HOPE	mean	0.16 \pm 0.05	0.06 \pm 0.03	NA	NA
		sum	0.16 \pm 0.05	0.37 \pm 0.05	NA	NA
	DeepWalks	mean	0.42 \pm 0.05	0.47 \pm 0.03	NA	NA
		sum	0.41 \pm 0.05	0.46 \pm 0.04	NA	NA
	Node2Vec	mean	0.30 \pm 0.06	0.20 \pm 0.03	NA	NA
		sum	0.25 \pm 0.07	0.40 \pm 0.04	NA	NA
	GNN	GCNConv	0.47 \pm 0.05	0.52 \pm 0.04	0.46 \pm 0.04	0.50 \pm 0.04
GraphSAGE		0.44 \pm 0.06	0.44 \pm 0.04	0.42 \pm 0.04	0.42 \pm 0.04	
GraphConv		0.48 \pm 0.05	0.52 \pm 0.04	0.47 \pm 0.05	0.51 \pm 0.03	

File-Level Parsing In Section 4.5, we explained how different levels of information can be used when constructing the embeddings. Table 3 displays the results for the unsupervised embeddings when they are constructed using: (i) both train and test features (i.e., $\mathbf{X}_{\mathcal{L}}$ and $\mathbf{X}_{\mathcal{T}}$, respectively); (ii) only train features ($\mathbf{X}_{\mathcal{L}}$). As explained in Section 4.5.1, the second option is not straightforward using shallow embeddings, as it will lead to the training data and test data being in different feature spaces. Because of this, we do not include it in the table (it is marked as NA). However, we show the results for this setting in B, with further explanation.

When utilizing both the training and testing features, Graph2Vec attains the highest scores with consistent average scores for both datasets—0.74 each.

Graph2Vec provides an embedding for the entire graph by default, but the remaining shallow embedding methods are on a node-level. Thus, in order to have the embedding for the entire graph, the embedding is aggregated using *mean* and *sum* aggregation functions. For the shallow embeddings that operate on a node-level, we observe that GR performs significantly better compared to the other methods for both datasets, where HOPE is the worst performing overall.

The results of shallow embeddings are more stable for HadoopTests since the STD is in the range of [0.02,0.05], which is not the case for OSSBuilds when the STD range is [0.03,0.07]. This is reasonable because by looking at Table 1, we can see that OSSBuilds contains four different projects for four different domains, which is not the case for HadoopTests, where all code files are related to one project.

The performance of the GNN-based methods is slightly better when the test features are used in the embedding. GraphConv is the best GNN model for both datasets in both cases. The unsatisfactory performance of GNNs is not surprising, as unsupervised graph representation learning by GNNs requires vast data.

The results for the supervised embeddings based on the train features $\mathbf{X}_{\mathcal{L}}$ and train labels $\mathbf{Y}_{\mathcal{L}}$ are presented in Table 4. The Pearson correlation obtained with GNNs is shown in the first rows, while the results obtained using the manual embedding are reported in the last row. It is evident from the table that the performance of the GNN-based approaches is superior to that of the manual embeddings for both datasets (except the GCN for OssBuilds, which is slightly worse than manual embedding). Thus, GraphConv performs the best for OSSBuilds, with an average correlation score of 0.67 and STD of 0.02. In contrast, for HadoopTests, GraphSAGE and GraphConv have the highest average correlation score of 0.68 and STD of 0.02 and 0.01, respectively.

Table 4: Results for Supervised and Manual Embedding for graphs of File Level Parsing.

		OSSBuilds	HadoopTests
Supervised Embedding (GNN)	GCNConv	0.61 ± 0.04	0.66 ± 0.02
	GraphSAGE	0.64 ± 0.03	0.68 ± 0.02
	GraphConv	0.67 ± 0.02	0.68 ± 0.01
Manual Embedding		0.64 ± 0.05	0.61 ± 0.02

Overall, for passive learning, Graph2Vec with test features achieves the best score for both datasets and settings. The reason why Graph2Vec performs well could be because our input graphs are similar to trees (see Section 5.3). In fact, Graph2Vec explores a much deeper path within the input graph compared to GNN. On the other hand, GNNs in a supervised setting deliver reasonable results for both datasets (unlike the unsupervised GNN embedding). This is likely because the labels are utilized. The manual embedding also yields an acceptable score compared to the shallow embeddings (except Graph2Vec).

System-Level Parsing This section examines the passive learning outcomes for System-Level parsing, where graphs are expanded from their File-Level counterparts.

Table 5: Results for Unsupervised Embedding for graphs of System Level Parsing.

		Train and Test Features		Train Features		
		OSSBuilds	HadoopTests	OSSBuilds	HadoopTests	
Shallow Embedding	Graph2Vec	0.73 ± 0.03	0.75 ± 0.02	NA	NA	
	GR	mean	0.45 ± 0.04	0.47 ± 0.02	NA	NA
		sum	0.40 ± 0.05	0.43 ± 0.03	NA	NA
	HOPE	mean	0.19 ± 0.07	0.06 ± 0.03	NA	NA
		sum	0.20 ± 0.08	0.35 ± 0.04	NA	NA
	DeepWalks	mean	0.37 ± 0.06	0.44 ± 0.02	NA	NA
		sum	0.36 ± 0.06	0.43 ± 0.04	NA	NA
	Node2Vec	mean	0.33 ± 0.06	0.42 ± 0.03	NA	NA
		sum	0.36 ± 0.06	0.42 ± 0.04	NA	NA
	GNN	GCNConv	0.41 ± 0.06	0.48 ± 0.03	0.44 ± 0.05	0.48 ± 0.03
GraphSAGE		0.37 ± 0.06	0.42 ± 0.04	0.38 ± 0.04	0.45 ± 0.05	
GraphConv		0.43 ± 0.06	0.49 ± 0.03	0.44 ± 0.07	0.49 ± 0.03	

Table 5 displays the results for the unsupervised embeddings based on both train and test features, as well as only train features for GNN for System-Level parsing. Thus, looking at the results of Tables 5, we notice that Graph2Vec attains the highest scores of 0.73 and 0.75 for the OSSBuilds and HadoopTests datasets, respectively, which is consistent with the results obtained for File-Level Parsing. For both datasets, GR, DeepWalks, and Node2Vec with both aggregation functions achieve a reasonable Pearson correlation score. On the other hand, HOPE remains the worst-performing approach in terms of embedding quality. The results of shallow embeddings are more stable for HadoopTests since the STD is in the range of [0.02,0.04], which is not the case for OSSBuilds when the STD range is [0.03,0.08].

For GNNs, the average score decreases by a small margin (especially for HadoopTests graphs) with/without test features compared to the original graphs in File-Level Parsing.

Table 6: Results for Supervised and Manual Embedding for graphs of System Level Parsing.

		Train Features	
		OSSBuilds	HadoopTests
Supervised Embedding (GNN)	GCNConv	0.59 ± 0.04	0.64 ± 0.03
	GraphSAGE	0.61 ± 0.04	0.67 ± 0.02
	GraphConv	0.65 ± 0.04	0.66 ± 0.02
Manual Embedding		0.60 ± 0.04	0.59 ± 0.03

As for supervised results in Table 6, the results for all GNN-based models are slightly worse compared to the original graphs in File-Level Parsing. The same is true regarding Manual Embedding. The reason for this might be that we have

more nodes and edges with System-level parsing, which means more sparsity as well as more layers needed by the GNN models to get more information from the new nodes.

5.5.2 Active Learning

Given an embedding, the active learning experiments were conducted as outlined in Section 4.1. We investigate different sizes of the initially labelled dataset $|\mathcal{L}_0|$ and the batch size $|\mathcal{B}|$. Additionally, all active learning results correspond to the average of 15 runs with different seeds, where the variance of the runs is indicated by a shaded colour.

The active learning experiments investigate three different graph embeddings (based on the passive learning results): manual embedding, Graph2Vec (with test features) and GraphConv as the supervised (GNN) embedding. For each embedding, we use the six query strategies outlined in Section 4.7 (i.e., random, coreset, variance, QBC, PowerVariance and PowerQBC).

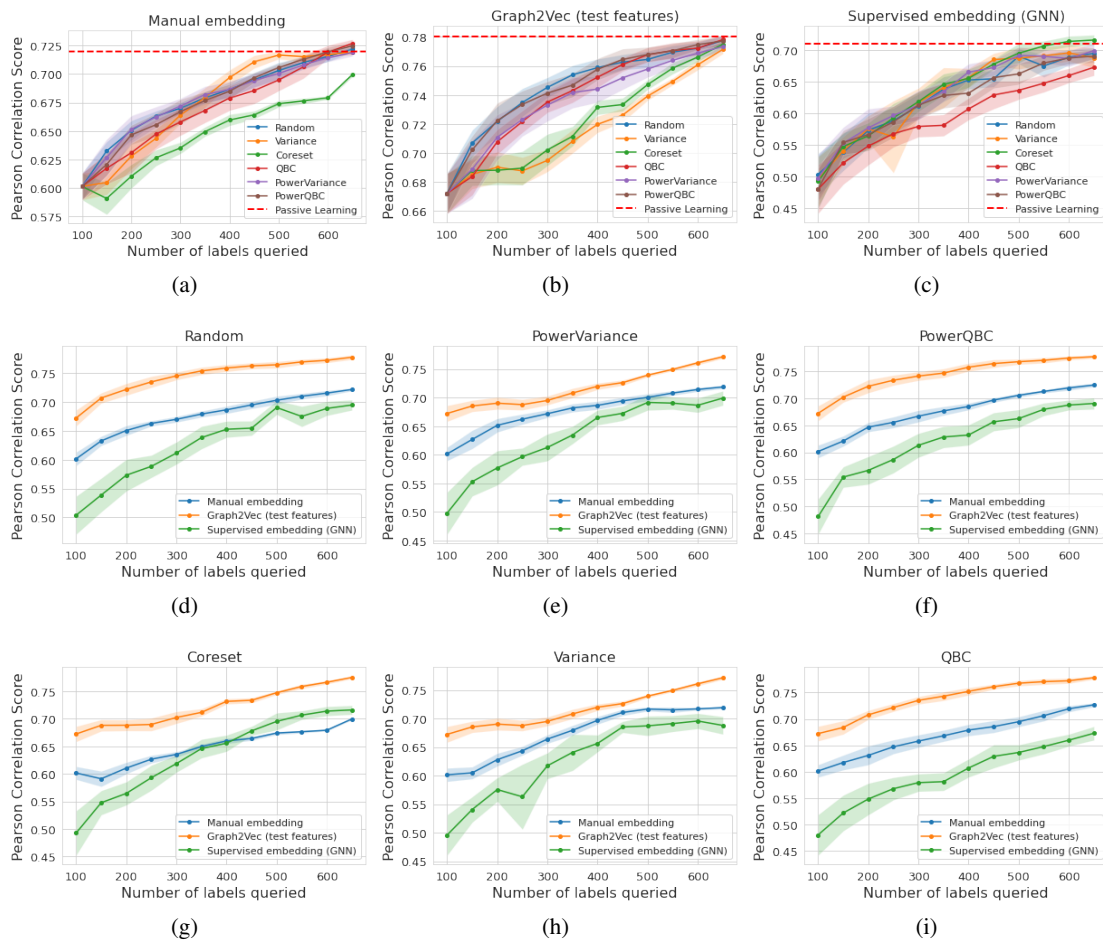


Figure 9: Active learning results for all embeddings for the OSSBuilds dataset (File Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

File Level Parsing Graphs In Figures 9 and 10 we show the active learning results for file level parsing for all embeddings for the OSSBuilds and Hadoop datasets, respectively. We observe that random selection is a strong baseline for both datasets. However, we see some benefit of the other query strategies indicating the usefulness of active learning. This benefit is more clear for system level parsing (see below). In particular, we see the usefulness of PowerVariance and PowerQBC (compared to their non-power versions). In terms of the embeddings, we see that the ranking is consistent for all query strategies at all iterations of the active learning procedure. For OSSBuilds, Graph2Vec is the best, manual embedding second best, and supervised embedding the worst. One exception to this is for the coreset query strategy, where the supervised embedding outperforms the manual embedding in later iterations. For Hadoop, Graph2Vec is still

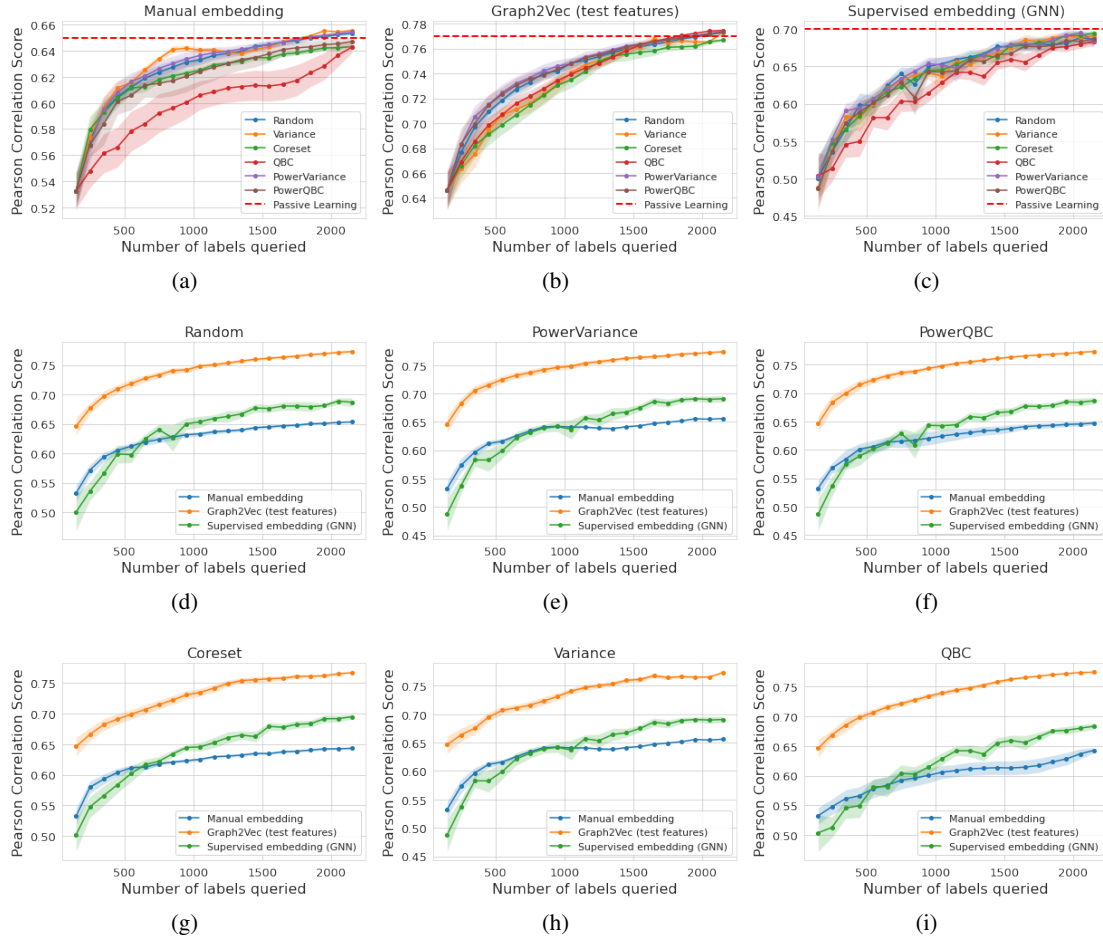


Figure 10: Active learning results for all embeddings for the HadoopTests dataset (File Level Parsing) with $|\mathcal{L}_0| = 150$ and $|\mathcal{B}| = 100$.

the best, but the supervised embedding outperforms the manual embedding in later iterations (when more labeled data is available).

System Level Parsing Graphs In Figures 11 and 12 we show the active learning results for file level parsing for all embeddings for the OSSBuilds and Hadoop datasets, respectively. For the OSSBuilds dataset, we observe that QBC and Variance perform slightly better than random. For Hadoop, we see that Variance significantly outperforms random (in particular in later iterations) for the manual embedding. For Graph2Vec and the supervised embedding, we see that random is consistently outperformed by the other query strategies. For the embeddings, we observe that Graph2Vec is the best for both datasets. In addition, we observe that the manual embedding is better in early iterations, whereas the supervised embedding eventually becomes better than the manual embedding (once sufficient labeled data is available).

5.6 Experiment Limitation

In utilizing real-world graphs for source code representation, we posit that our framework is applicable across various domains of directed graph data, including social networks and pharmacological graphs and others. While our framework is primarily tailored for directed graphs, adaptations for undirected graph scenarios, particularly within supervised embedding contexts, are conceivable. It is imperative, however, to acknowledge that the efficacy and relevance of our findings may vary across different graph datasets. This variance can be attributed to inherent differences in graph structure and characteristics. Our experimental graphs, as delineated in Table 2, are sparse, large, and complex. These attributes may not be universally representative, suggesting that certain embedding techniques and query strategies optimized for our dataset might not directly translate to or yield comparable results in dissimilar graph environments.

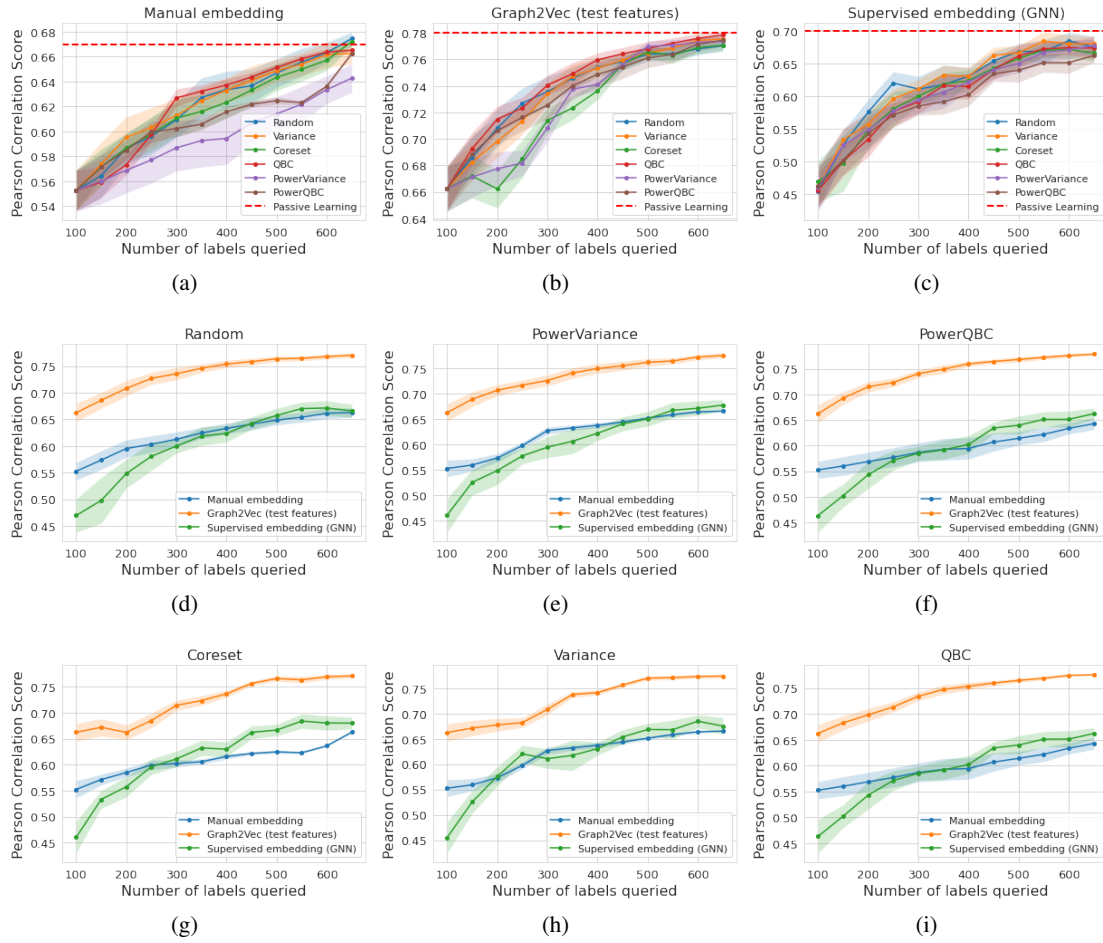


Figure 11: Active learning results for all embeddings for the OSSBuilds dataset (System Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

6 Discussion

In this section, we comment on the results for both passive and active learning.

6.1 Passive Learning

In this section, we assess the resilience of embedding techniques as graphs in System-Level parsing evolve by incorporating additional nodes and edges, thus providing insights into how these techniques perform under conditions of increased graph complexity and size.

The resilience of unsupervised embedding techniques to the expanded version of graphs varies across the methods tested. Graph2Vec exhibits strong resilience, showing minimal performance change despite increased graph complexity, which suggests its effectiveness in scalable applications. GR and HOPE demonstrate some sensitivity to scale, with slight to moderate performance declines, indicating potential limitations in more complex graph environments. DeepWalks maintain performance levels but do not show improvements, suggesting stability rather than adaptability to larger scales. The embedding quality for Node2Vec increased compared to the original graphs in File-Level parsing, and the opposite for GR. That explains why Node2Vec performs better on graph data with more nodes and edges. Lastly, GNN models (GCNConv, GraphSAGE, GraphConv) show a moderate decrease in performance in extended graphs compared to the original graphs in File-Level parsing, suggesting that while they handle increased complexity, their efficacy slightly diminishes as graph complexity increases. This analysis underlines the importance of carefully selecting embedding techniques based on anticipated graph structure and complexity for optimal performance in scalable environments.

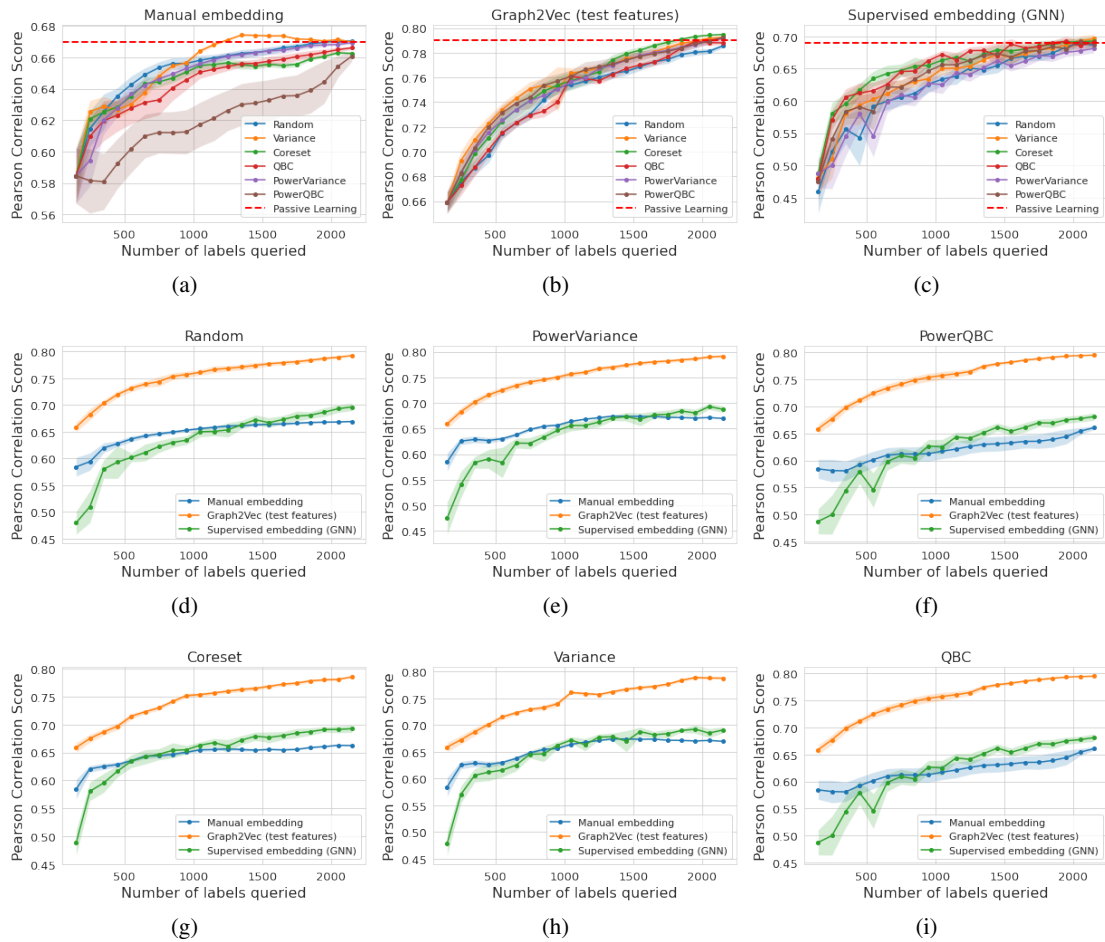


Figure 12: Active learning results for all embeddings for the HadoopTests dataset (System Level Parsing) with $|\mathcal{L}_0| = 150$ and $|\mathcal{B}| = 100$.

As graphs expanded from File-Level to System-Level parsing, supervised and manual embedding techniques exhibited a slight decline in performance. GCNConv, GraphSAGE, and GraphConv demonstrate robustness with minor reductions, suggesting they manage increased complexity well, though effectiveness slightly diminishes in more complex settings. Manual Embedding shows a more noticeable performance drop, indicating a greater sensitivity to graph complexity. This trend highlights the need for cautious application as graph size and intricacy grow.

6.2 Active Learning

There are three main observations from the active learning results: (i) There is no single query strategy that consistently outperforms the others across all settings, which is consistent with observations in other AL works [12]. However, there is some indication that the coreset performs particularly well in conjunction with the supervised embedding (based on deep GNN). This is logical considering that coreset was originally introduced for deep batch active learning [5]; (ii) The benefit of the different query strategies over random selection improves for the Hadoop dataset (compared to OSSBuilds), and in particular for system level parsing. The reason is likely because Hadoop contains more data points compared to OSSBuilds. In addition, for system-level parsing, we obtain more complex graphs. In other words, we observe the increased benefit of (batch) active learning for larger and more complex datasets. In contrast, for small and simple datasets, random selection becomes a very strong baseline. However, it can be argued that for small and simple datasets, the use of active learning is not as important; (iii) The supervised embedding (based on GNN) is worse than the manual embedding in early iterations but exceeds it in later iterations. This reflects our intuition since out of the three embeddings considered for active learning, only the supervised embedding will update its latent space iteratively as more labels become available. However, Graph2Vec still outperforms the supervised embedding when all labels are

available. The main reason for this is likely (as discussed for the passive learning results) that Graph2Vec uses the test features when constructing its latent space.

7 Conclusion

Our investigation of a unified active learning framework for annotating graphs at the graph-level has yielded several significant insights. We found that unsupervised embedding techniques like Graph2Vec exhibit robust performance when leveraging both training and testing features. However, supervised embeddings like GNNs offer greater flexibility across various levels of information accessibility. Specifically, active learning strategies excel in environments with larger, more complex datasets, underscoring the potential for these techniques in scaling to more extensive graph structures. Reflecting on our research objectives, this study successfully demonstrates the application of active learning to graph-level regression tasks, a relatively unexplored area. The ability of our framework to adapt to expanded graphs and efficiently utilize computational resources highlights its practical relevance and potential for broad application. The implications of our findings are profound for the domain of graph data analysis, particularly in enhancing the efficiency of data annotation processes without compromising quality of the machine learning models trained on this data. This is particularly relevant in fields where data complexity and volume pose significant challenges. However, the following limitations of our work should be mentioned. First, the framework can be computationally demanding, in particular when used in conjunction with GNN embeddings, since a GNN must be trained from scratch in each iteration. Second, the obtained results are specific to the considered datasets and active learning strategies. Consequently, for future work, we recommend further investigation into the scalability of the proposed active learning framework and the investigation of more diverse datasets to broaden the applicability of our findings.

Acknowledgements

This work received financial support from the Swedish Research Council VR under grant number 2018-04127 (Developer-Targeted Performance Engineering for Immersed Release and Software Engineers). The work of Linus Aronsson and Morteza Haghiri Chehreghani was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundations. Antonio Longa acknowledges the support of the MUR PNRR project FAIR—Future AI Research (PE00000013) funded by the NextGenerationEU. Finally, the computations and data handling was enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC), partially funded by the Swedish Research Council through grant agreement no. 2022-06725 and no. 2018-05973.

References

- [1] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.
- [2] John Daniel Bossér, Erik Sörstadius, and Morteza Haghiri Chehreghani. Model-centric and data-centric aspects of active learning for deep neural networks. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 5053–5062, 2021.
- [3] Yang Li and Junier Oliva. Active feature acquisition with generative surrogate models. In *Proceedings of the 38th International Conference on Machine Learning, ICML*, pages 6450–6459, 2021.
- [4] Arantxa Casanova, Pedro O. Pinheiro, Negar Rostamzadeh, and Christopher J. Pal. Reinforced active learning for image segmentation. In *International Conference on Learning Representations*, 2020.
- [5] Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach, 2018.
- [6] Neil Rubens, Mehdi Elahi, Masashi Sugiyama, and Dain Kaplan. *Active Learning in Recommender Systems*, pages 809–846. Springer, 2015.
- [7] Federica Comuni, Christopher Mészáros, Niklas Åkerblom, and Morteza Haghiri Chehreghani. Passive and active learning of driver behavior from electric vehicles. In *25th IEEE International Conference on Intelligent Transportation Systems, ITSC*, pages 929–936, 2022.
- [8] Zhao Shuyang, Toni Heittola, and Tuomas Virtanen. Active learning for sound event detection. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 28:2895–2905, nov 2020.
- [9] Sanna Jarl, Linus Aronsson, Sadegh Rahrovani, and Morteza Haghiri Chehreghani. Active learning of driving scenario trajectories. *Engineering Applications of Artificial Intelligence*, 113:104972, 2022.

- [10] Simon Viet Johansson, Hampus Gummesson Svensson, Esben Bjerrum, Alexander Schliep, Morteza Haghiri Chehreghani, Christian Tyrchan, and Ola Engkvist. Using active learning to develop machine learning models for reaction yield prediction. *Molecular Informatics*, 41(12), 2022.
- [11] Songbai Yan, Kamalika Chaudhuri, and Tara Javidi. Active learning with logged data. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5517–5526, 2018.
- [12] Ksenia Konyushkova, Sznitman Raphael, and Pascal Fua. Learning active learning from data. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4228–4238, Red Hook, NY, USA, 2017. Curran Associates Inc.
- [13] Thuy-Trang Vu, Ming Liu, Dinh Phung, and Gholamreza Haffari. Learning how to active learn by dreaming. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4091–4101. Association for Computational Linguistics, 2019.
- [14] Zimo Liu, Jingya Wang, Shaogang Gong, Dacheng Tao, and Huchuan Lu. Deep reinforcement active learning for human-in-the-loop person re-identification. In *International Conference on Computer Vision*, pages 6121–6130. IEEE, 2019.
- [15] Yuheng Zhang, Hanghang Tong, Yinglong Xia, Yan Zhu, Yuejie Chi, and Lei Ying. Batch active learning with graph neural networks via multi-agent deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(8):9118–9126, Jun. 2022.
- [16] Shengding Hu, Zheng Xiong, Meng Qu, Xingdi Yuan, Marc-Alexandre Côté, Zhiyuan Liu, and Jian Tang. Graph policy network for transferable active learning on graphs. *Advances in Neural Information Processing Systems*, 33:10174–10185, 2020.
- [17] Yuexin Wu, Yichong Xu, Aarti Singh, Artur Dubrawski, and Yiming Yang. Active learning graph neural networks via node feature propagation, 2020.
- [18] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. Active learning for graph embedding, 2017.
- [19] Youzhi Luo, Keqiang Yan, and Shuiwang Ji. Graphdf: A discrete flow model for molecular graph generation. In *International conference on machine learning*, pages 7192–7203. PMLR, 2021.
- [20] Zhichun Guo, Kehan Guo, Bozhao Nan, Yijun Tian, Roshni G Iyer, Yihong Ma, Olaf Wiest, Xiangliang Zhang, Wei Wang, Chuxu Zhang, et al. Graph-based molecular representation learning. *arXiv preprint arXiv:2207.04869*, 2022.
- [21] Zhengyang Wang, Meng Liu, Youzhi Luo, Zhao Xu, Yaochen Xie, Limei Wang, Lei Cai, Qi Qi, Zhuoning Yuan, Tianbao Yang, et al. Advanced graph and sequence neural networks for molecular property prediction and drug discovery. *Bioinformatics*, 38(9):2579–2586, 2022.
- [22] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. Incremental knowledge base construction using deepdive. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, volume 8, page 1310. NIH Public Access, 2015.
- [23] Su Yang and Miaole Hou. Knowledge graph representation method for semantic 3d modeling of chinese grottoes. *Heritage Science*, 11(1):266, 2023.
- [24] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory, pages 287–294. Publ by ACM, 1992. Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory ; Conference date: 27-07-1992 Through 29-07-1992.
- [25] Ashish Kapoor, Kristen Grauman, Raquel Urtasun, and Trevor Darrell. Active learning with gaussian processes for object categorization. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007.
- [26] Peter Samoaa, Linus Aronsson, Antonio Longa, Philipp Leitner, and Morteza Haghiri Chehreghani. A Unified Active Learning Framework for Annotating Graph Data with Application to Software Source Code Performance Prediction, 2023.
- [27] Veronica Lachi, Giovanna Maria Dimitri, Alessandro Di Stefano, Pietro Liò, Monica Bianchini, and Chiara Mocenni. Impact of the covid 19 outbreaks on the italian twitter vaccination debat: a network based analysis. *arXiv preprint arXiv:2306.02838*, 2023.
- [28] Anna Nguyen, Antonio Longa, Massimiliano Luca, Joe Kaul, and Gabriel Lopez. Emotion analysis using multilayered networks for graphical representation of tweets. *IEEE Access*, 10:99467–99478, 2022.
- [29] John Scott. Social network analysis: developments, advances, and prospects. *Social network analysis and mining*, 1:21–26, 2011.

- [30] Wolfgang Huber, Vincent J Carey, Li Long, Seth Falcon, and Robert Gentleman. Graphs in molecular biology. *BMC bioinformatics*, 8(6):1–14, 2007.
- [31] Tero Aittokallio and Benno Schwikowski. Graph-based methods for analysing networks in cell biology. *Briefings in bioinformatics*, 7(3):243–255, 2006.
- [32] Antonio Longa, Giulia Cencetti, Sune Lehmann, Andrea Passerini, and Bruno Lepri. Generating fine-grained surrogate temporal networks. *Communications Physics*, 7(1):22, 2024.
- [33] Beatriz Arregui-García, Antonio Longa, Quintino Francesco Lotito, Sandro Meloni, and Giulia Cencetti. Patterns in temporal networks with higher-order egocentric structures. *Entropy*, 26(3):256, 2024.
- [34] Antonio Longa, Giulia Cencetti, Bruno Lepri, and Andrea Passerini. An efficient procedure for mining egocentric temporal motifs. *Data Mining and Knowledge Discovery*, pages 1–24, 2022.
- [35] Giovanni Mauro, Massimiliano Luca, Antonio Longa, Bruno Lepri, and Luca Pappalardo. Generating mobility networks with generative adversarial networks. *EPJ data science*, 11(1):58, 2022.
- [36] Marco Cardia, Massimiliano Luca, and Luca Pappalardo. Enhancing crowd flow prediction in various spatial and temporal granularities. In *Companion Proceedings of the Web Conference 2022*, pages 1251–1259, 2022.
- [37] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.
- [38] Yanyao Shen, Hyokun Yun, Zachary C. Lipton, Yakov Kronrod, and Animashree Anandkumar. Deep active learning for named entity recognition, 2018.
- [39] Yarin Gal, Riashat Islam, and Zoubin Ghahramani. Deep bayesian active learning with image data. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1183–1192, 06–11 Aug 2017.
- [40] Xiaoting Li, Yuhang Wu, Vineeth Rakesh, Yusan Lin, Hao Yang, and Fei Wang. Smartquery: An active learning framework for graph neural networks through hybrid uncertainty reduction. In *Proceedings of the 31st ACM International Conference on Information; Knowledge Management, CIKM '22*, page 4199–4203, New York, NY, USA, 2022. Association for Computing Machinery.
- [41] Roy Abel and Yoram Louzoun. Regional based query in graph active learning, 2019.
- [42] Li Gao, Hong Yang, Chuan Zhou, Jia Wu, Shirui Pan, and Yue Hu. Active discriminative network representation learning. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 2142–2148, 7 2018.
- [43] Xia Chen, Guoxian Yu, Jun Wang, Carlotta Domeniconi, Zhao Li, and Xiangliang Zhang. Activehne: Active heterogeneous network embedding. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 2123–2129, 7 2019.
- [44] Hazem Peter Samoaa, Antonio Longa, Mazen Mohamad, Morteza Haghiri Chehrehani, and Philipp Leitner. Tep-gnn: Accurate execution time prediction of functional tests using graph neural networks. In Davide Taibi, Marco Kuhmann, Tommi Mikkonen, Jil Klünder, and Pekka Abrahamsson, editors, *Product-Focused Software Process Improvement*, pages 464–479, Cham, 2022. Springer International Publishing.
- [45] Hazem Peter Samoaa, Firas Bayram, Pasquale Salza, and Philipp Leitner. A systematic mapping study of source code representation for deep learning in software engineering. *IET Software*, 16(4):351–385, 2022.
- [46] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy. *Journal of Machine Learning Research*, 23(89):1–64, 2022.
- [47] Shaosheng Cao, Wei Lu, and Qionkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, CIKM '15*, page 891–900, New York, NY, USA, 2015. Association for Computing Machinery.
- [48] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16*, page 1225–1234, New York, NY, USA, 2016. Association for Computing Machinery.
- [49] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14*, page 701–710, New York, NY, USA, 2014. Association for Computing Machinery.
- [50] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16*, page 855–864, New York, NY, USA, 2016.

- [51] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.
- [52] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, 2017.
- [53] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29, 2016.
- [54] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.
- [55] Vito Latora and Massimo Marchiori. Efficient behavior of small-world networks. *Phys. Rev. Lett.*, 87:198701, Oct 2001.
- [56] M. E. J. Newman. Assortative mixing in networks. *Physical Review Letters*, 89(20), oct 2002.
- [57] Duncan J Watts and Steven H Strogatz. Collective dynamics of ‘small-world’ networks. *nature*, 393(6684):440–442, 1998.
- [58] M. E. J. Newman. *Networks: an introduction*. Oxford University Press, Oxford; New York, 2010.
- [59] Prateek Munjal, Nasir Hayat, Munawar Hayat, Jamshid Sourati, and Shadab Khan. Towards robust and reproducible active learning using neural networks. *Conference on Computer Vision and Pattern Recognition*, pages 223–232, 2020.
- [60] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.
- [61] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B. Gupta, Xiaojiang Chen, and Xin Wang. A survey of deep active learning. 54(9), oct 2021.
- [62] Andreas Kirsch, Sebastian Farquhar, Parmida Atighehchian, Andrew Jesson, Frédéric Branchaud-Charron, and Yarin Gal. Stochastic batch acquisition: A simple baseline for deep active learning. *Transactions on Machine Learning Research*, 2023. Expert Certification.
- [63] Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3), 2005.
- [64] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [65] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [66] Mark EJ Newman and Duncan J Watts. Renormalization group analysis of the small-world network model. *Physics Letters A*, 263(4-6):341–346, 1999.
- [67] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.
- [68] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*. 1994.
- [69] Nan Ma, Jiancheng Guan, and Yi Zhao. Bringing pagerank to the citation analysis. *Information Processing & Management*, 44(2):800–810, 2008.

A Graph analysis

In this section, we do a deeper investigation of the graph topology of our dataset.

A.1 Basic topology

Figure 13 displays node (Figure 13(a)) and edge (Figure 13(b)) distributions, respectively. The data indicate a minimal disparity between file and system levels in terms of both statistics.

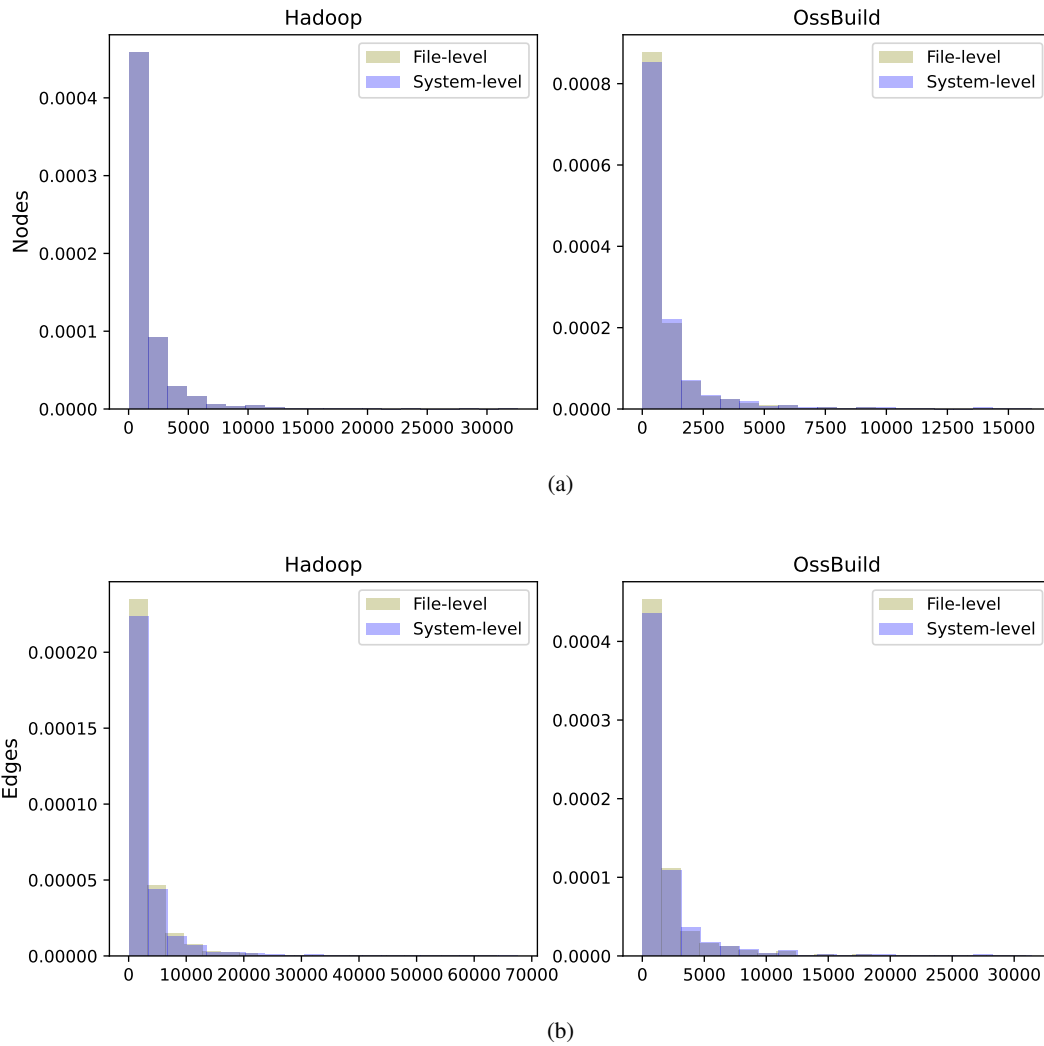


Figure 13: Distributions of the number of nodes and edges in Hadoop (left) and OssBuild (right) for both file-level and system-level settings.

The degree distribution, depicted in Figure 14, effectively captures the resemblance between the distributions of nodes and edges.

A.2 Triangles

In network science, the concept of triangle closure, also known as the "friendship paradox", is a well-established and widely recognized phenomenon. It has garnered significant attention and has been extensively studied in various research works, highlighting its relevance and importance in numerous real-world applications. In particular, we explore

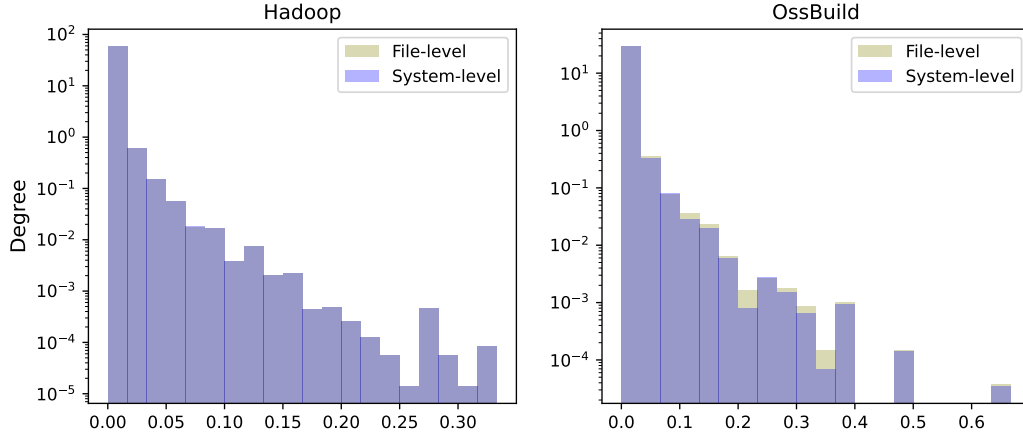


Figure 14: Degree distribution in logarithmic scale of Hadoop (left) and OssBuild (right) for both file-level and system-level.

the relationship between the graph and triangles through Transitivity[57] and Clustering Coefficient[66]. Transitivity is defined as follows:

$$\text{Transitivity} = 3 \cdot \frac{\# \text{ of triangles}}{\# \text{ of triads}} \quad (5)$$

On the other hand, the clustering coefficient is a metric associated with a given node u , and it refers to the degree to which nodes in a graph tend to cluster together. The clustering coefficient of a node u is defined as follows:

$$C_u = \frac{2 \cdot T(u)}{(deg(u)) \cdot (deg(u) - 1)} \quad (6)$$

Where $T(u)$ is the number of triangles through node u , and $deg(u)$ is the degree of node u . The Global Clustering Coefficient (GCC) is the average among the clustering coefficient of all nodes. In summary, while both transitivity and clustering coefficient capture the local clustering patterns in a network, transitivity focuses on the presence of triangles and overall network connectivity, whereas the clustering coefficient specifically measures the density of connections between neighboring nodes.

Figure 15 shows the transitivity (Figure 15(a)) and the global clustering coefficient (Figure 15(b)) distributions. Based on the results, it is apparent that both transitivity and GCC exhibit higher values in the file-level dataset compared to the system-level dataset. However, this distinction is not as pronounced in the OssBuild dataset.

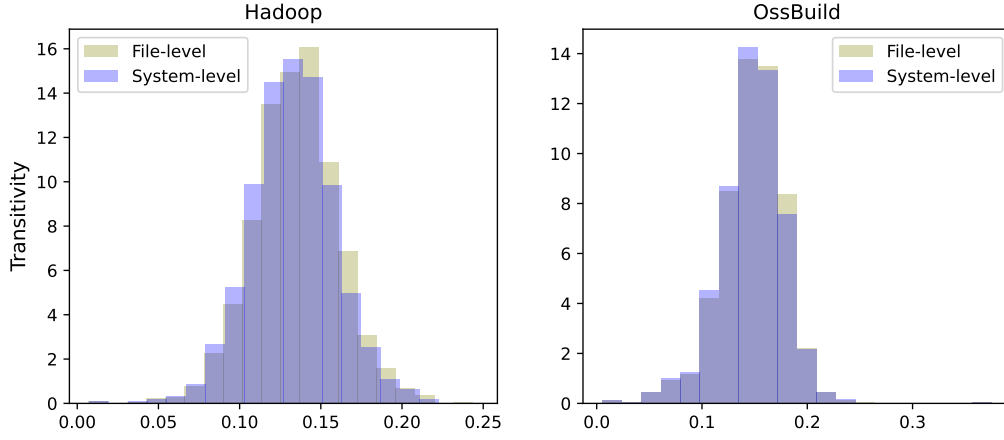
A.3 Assortativity

Assortativity, in network theory, refers to the tendency of nodes in a network to connect with similar nodes. It measures the degree of homophily or assortative mixing in a network based on node attributes or characteristics. Assortativity can be quantified using various metrics, such as degree assortativity, attribute assortativity, or assortativity coefficient[56]. In Figure 16 we report the degree assortativity that examines the correlation of node degrees between connected nodes.

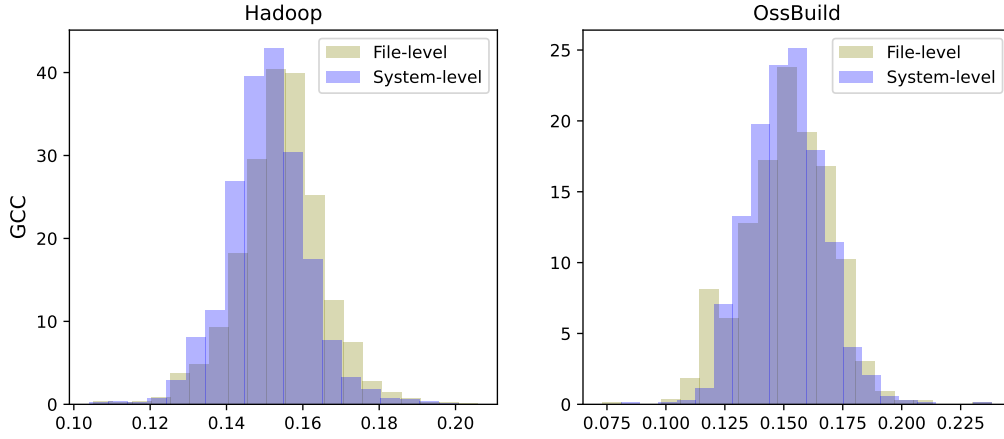
Based on the observations in Figure 16, it is challenging to determine whether the graphs exhibit positive assortativity (where nodes with similar degrees tend to connect) or negative assortativity (indicating connections between nodes with differing degrees). However, upon examining the histograms, it appears that in both scenarios, the System-level dataset tends to connect nodes to other nodes with differing degrees.

A.4 Centralities

Centrality in network analysis refers to the importance or prominence of nodes within a network. It measures the extent to which a node is influential, well-connected, or positioned strategically within the network structure. Centrality measures help identify key nodes that play crucial roles in information flow, influence propagation, and network dynamics.



(a)



(b)

Figure 15: Distributions of the transitivity and global clustering coefficient (GCC) in Hadoop (left) and OssBuild (right) for both file-level and system-level settings.

Various centrality measures exist, where we have already evaluated the degree distributions (in Figure 14). Here we dig deeper into Betweenness Centrality, Closeness Centrality, and Page Rank. The Betweenness Centrality measures the control a node has over the flow of information in the network. Formally, it is defined as[67]

$$\text{Betweenness Centrality}_u = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)} \quad (7)$$

where, $\sigma(s,t)$ is the number of shortest paths between node s and node t , while $\sigma(s,t|u)$ is the number of shortest paths between node s and node t passing through node u .

Closeness Centrality measures the proximity of a node to all other nodes in the network. Formally, it is defined as[68]

$$\text{Closeness Centrality}_u = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)} \quad (8)$$

Where here n is the number of nodes, and $d(v,u)$ is the shortest-path length between node v and node u .

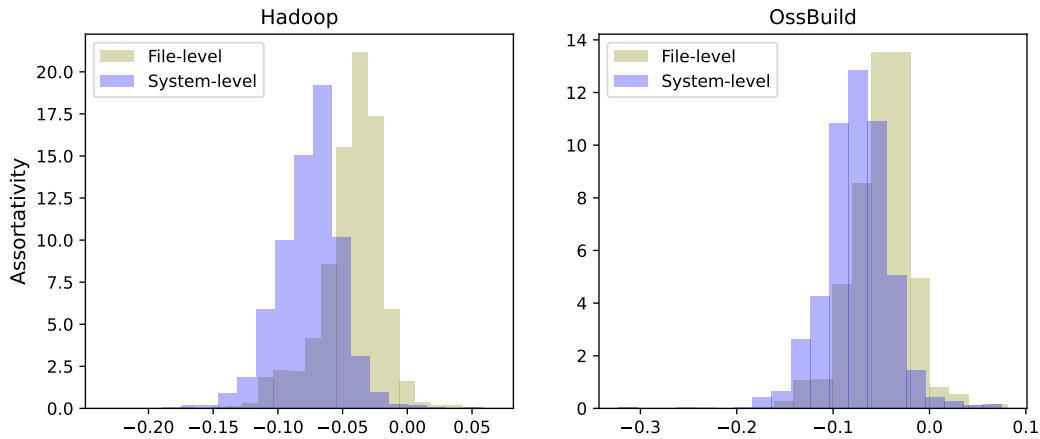


Figure 16: Distributions of the degree assortativity in Hadoop (left) and OssBuild (right) for both file-level and system-level.

Finally, the Page Rank[69] assigns importance to nodes based on the number and quality of incoming links. Nodes with higher Page Rank are considered more influential.

In Figure 17 we report the Betweenness, Closeness and Page Rank of the datasets. It is clear that the strongest difference between the file and system level settings relies on the Betweenness. This is not surprising at all, since in the file-level setting there are fewer edges, thus the number of edges with a higher Betweenness is greater.

A.5 Meso-scale

In conclusion, we explore the meso-scale characteristics of the network topology by employing measures such as shortest-path analysis[58], tree similarity, and the diameter[58] of the input network.

The shortest path is defined in Definition 2, and it reports the smaller path connection between two given nodes. The distribution is reported in Figure 18. The figure clearly indicates that the system-level network exhibits shorter shortest paths compared to the file-level network. This observation is expected, as the system-level networks contain a higher number of edges in comparison to the file-level networks.

The *tree-sim* metric, defined in Eq. 4, is a custom measure that quantifies the similarity between the input graph and its corresponding tree structure. It is important to note that this metric should not be confused with Treewidth. In our study, we introduced the tree-sim metric as an alternative to overcome the computational complexity associated with calculating Treewidth. The distribution of the tree-sim metric for each dataset is presented in Figure 19. However, no significant insights or noteworthy patterns were observed from the analysis of these distributions, where, as expected, both follow power-law distribution.

Lastly, in Figure 20, we present the distribution of diameters for each graph. As expected, the system-level networks exhibit a smaller diameter compared to the file-level networks.

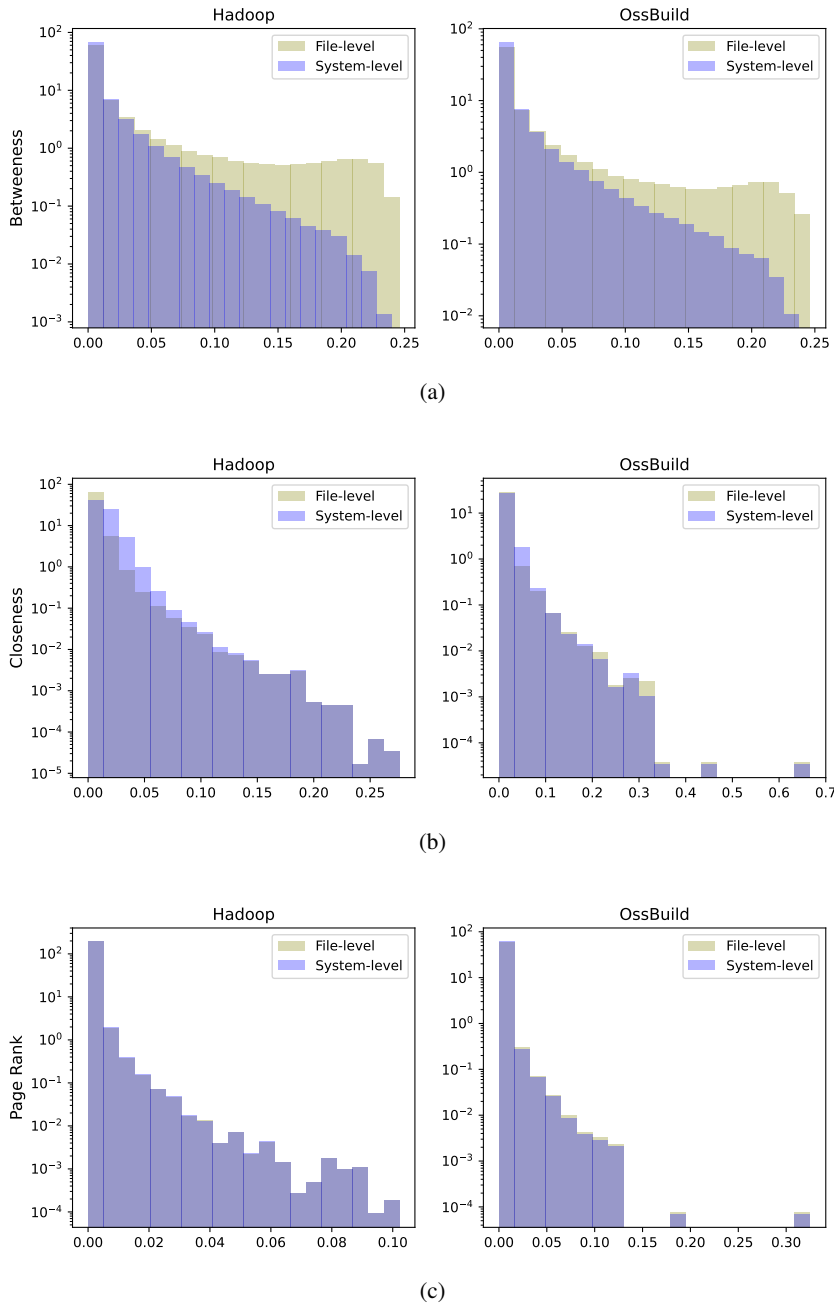


Figure 17: Distributions of the Betweenness, Closeness and Page Rank (in log scale) in Hadoop (left) and OssBuild (right) for both file-level and system-level settings.

B Shallow Embedding Results When Test Features are Not Used

As we mentioned in Section 5.5.1, computing the embedding using only the training features without manipulating the test features in embedding is not possible for unsupervised shallow embedding because eventually, we will have different features space for both training and testing data. In this section, we will prove the aforementioned statement for both passive and active learning.

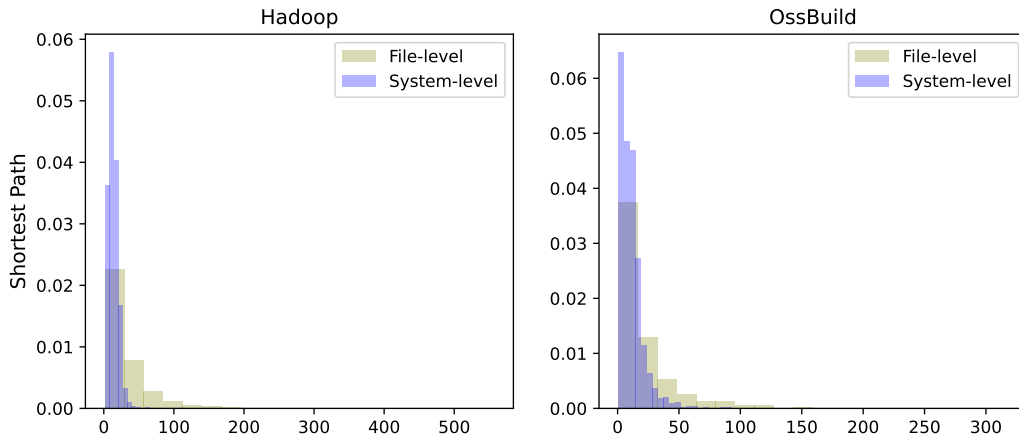


Figure 18: Shortest path length distributions.

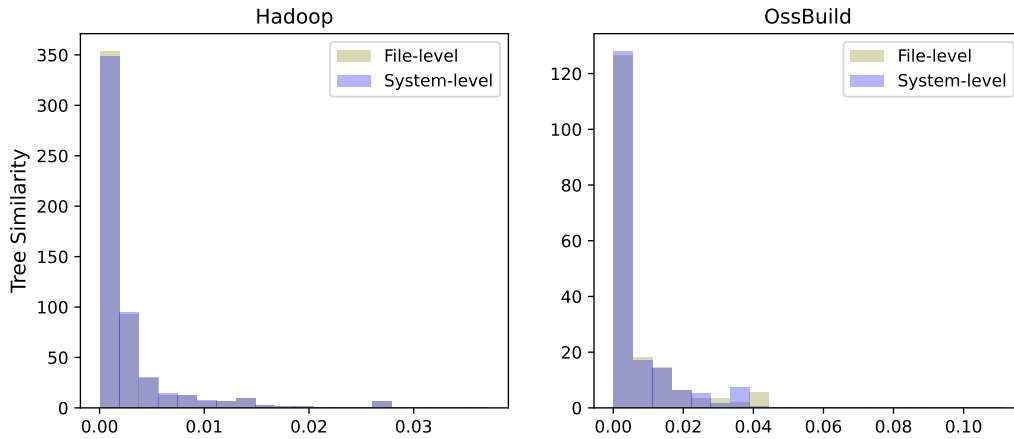


Figure 19: *Tree sim* distributions.

B.1 Passive Learning

We are experimenting with computing the embedding only for one seed for the dataset. We compute the embedding for the entire dataset (when test features are included) and then we get the first 80% of the dataset and compute the embedding only for this portion of the dataset(so here, the last 20% are excluded). Here we either retrain the model based on the last 20%, which leads to poor results or alternatively, we get the benefit of the embedding of the entire dataset since the training data is included. Then using the same split index that we did when we got the training data, we get the last 20% of the embedding.

B.1.1 System-Level Parsing

Table 7, shows the results for shallow embedding with and without test features with one split for the data. These results are significantly better than the ones we averaged for 15 different splits when we used the test features. However, with more splits, the results are more reliable. Looking at Table 7, using only the train data features leads to a substantial decline in the performance of Graph2Vec, DeepWalk, and Node2Vec (except for sum aggregation in the HadoopTest dataset). These methods are all shallow embedding techniques based on skip-gram, as shown in Figure 7. Conversely, there are generally slight improvements for the other shallow methods based on Matrix Factorization, such as HOPE and GR (except for mean aggregation in the HadoopTests dataset).

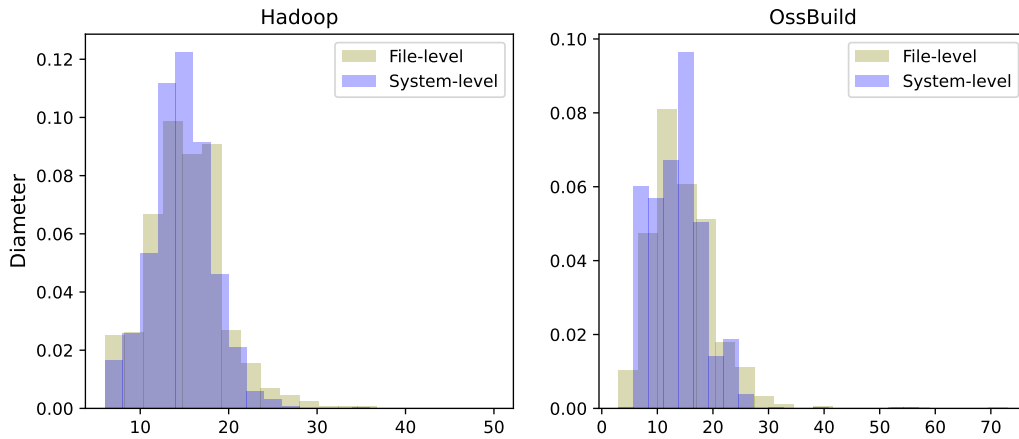


Figure 20: Diameter distributions.

Table 7: Results for Unsupervised Embedding for graphs of System Level Parsing.

			Train and Test Features		Train Features	
			OSSBuilds	HadoopTests	OSSBuilds	HadoopTests
Shallow Embedding	Graph2Vec		0.78	0.74	0.65	0.46
	GR	mean	0.44	0.49	0.50	0.45
		sum	0.45	0.42	0.45	0.48
	HOPE	mean	0.14	0.015	0.16	0.04
		sum	0.13	0.36	0.16	0.39
	DeepWalks	mean	0.41	0.47	0.38	0.4
		sum	0.43	0.45	0.32	0.39
	Node2Vec	mean	0.39	0.42	0.29	0.32
		sum	0.44	0.42	0.33	0.43

B.1.2 File-Level Parsing

The results for this setting are reported in Table 8. In this setting, we still have better results than those obtained with 15 different splits for the dataset.

Table 8: Results for Unsupervised Embedding for graphs of File Level Parsing.

			Train and Test Features		Train Features	
			OSSBuilds	HadoopTests	OSSBuilds	HadoopTests
Shallow Embedding	Graph2Vec		0.78	0.74	0.53	0.49
	GR	mean	0.57	0.46	0.59	0.41
		sum	0.51	0.42	0.49	0.37
	HOPE	mean	0.17	0.034	0.06	0.07
		sum	0.15	0.35	0.07	0.3
	DeepWalks	mean	0.45	0.43	0.34	0.24
		sum	0.42	0.41	0.39	0.02
	Node2Vec	mean	0.33	0.2	0.39	0.15
		sum	0.33	0.36	0.31	0.32

Nevertheless, when we exclude the test features, the correlation score for Graph2Vec is drastically reduced to 0.53 for OssBuilds and 0.49 for HadoopTests which remains the best for such dataset when we only use the train features. Conversely, GR with mean aggregation is the best for the same setting for OssBuilds.

B.2 Active Learning

To understand the impact of different feature spaces embedding we will present the active learning results for Graph2Vec when test features are not included.

B.2.1 System-Level Parsing

In Figure 21, the embedding performance based on Graph2Vec without test features for HadoopTests only improves slightly at the start but then stays fairly constant. The reason for this is likely because the resulting latent graph representation is not rich enough for this embedding past 500 labels. We have the same issue for the OSSBuilds dataset for random and QBC.

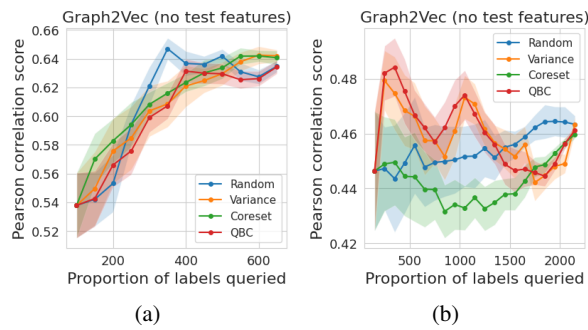


Figure 21: Active learning results for Graph2Vec When Test Features are not Used in embeddings for the OSSBuilds (left) and HadoopTest (right) datasets (System Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

B.2.2 File-Level Parsing

In Graph2Vec with no test features in Figure 22, for the HadoopTests dataset, coreset and random are the best choice when we have up to 1000 samples but the quality of labelling drastically reduces after that threshold when variance remains the best as it performs reliably after 500 samples. Variance is the worst option for the OSSBuilds dataset.

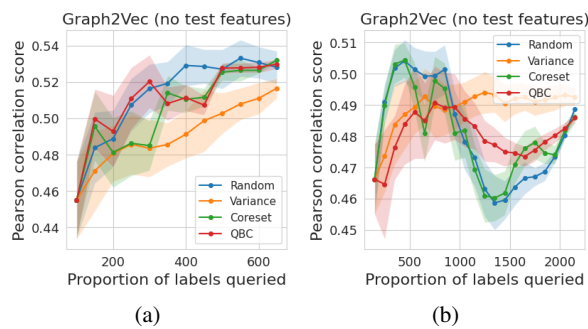


Figure 22: Active learning results for Graph2Vec When Test Features are not Used in embeddings for the OSSBuilds (left) and HadoopTest (right) datasets (File Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

B.3 Root Mean Square Error

In this section, we present the active learning results using the (log) RMSE metric for all datasets for both file level and system level parsing. In all cases, we observe consistent results with the Pearson correlation score from the main paper.

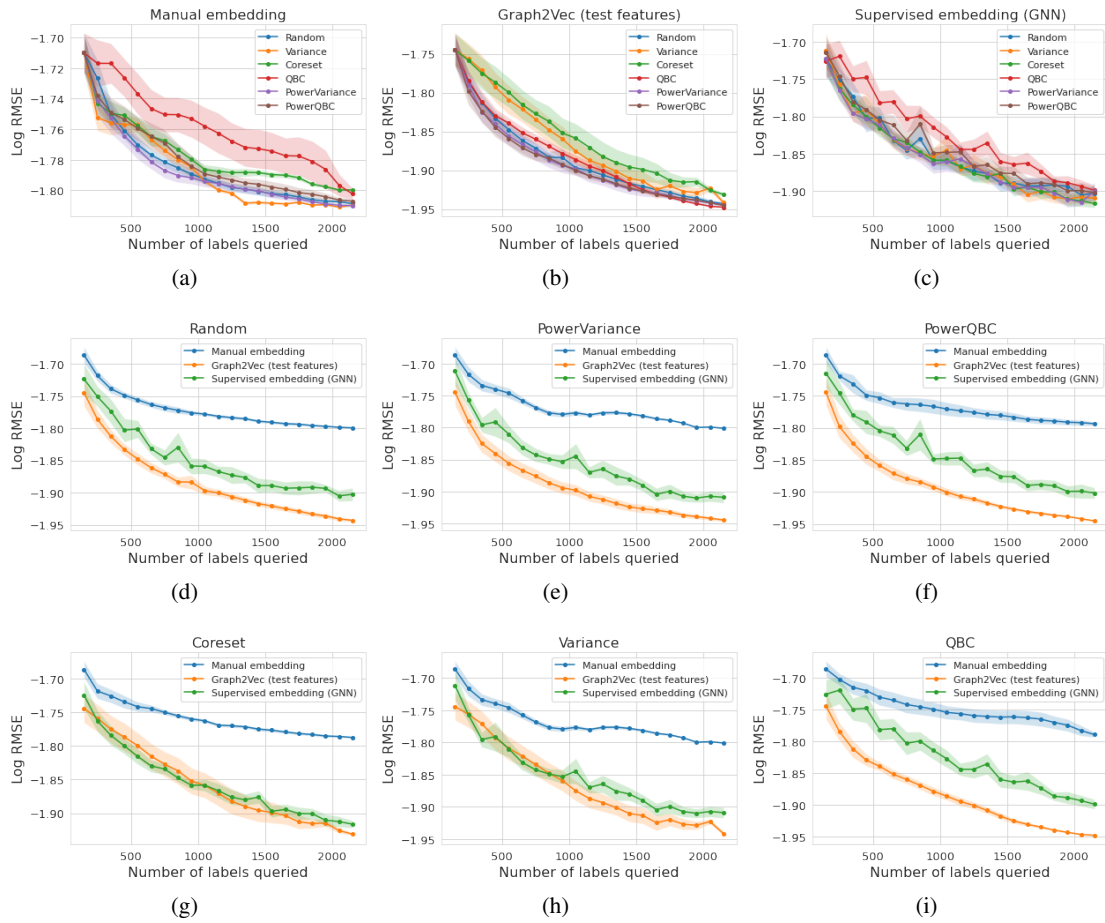


Figure 23: Active learning results for all embeddings for the HadoopTests dataset (File Level Parsing) with $|\mathcal{L}_0| = 150$ and $|\mathcal{B}| = 100$.

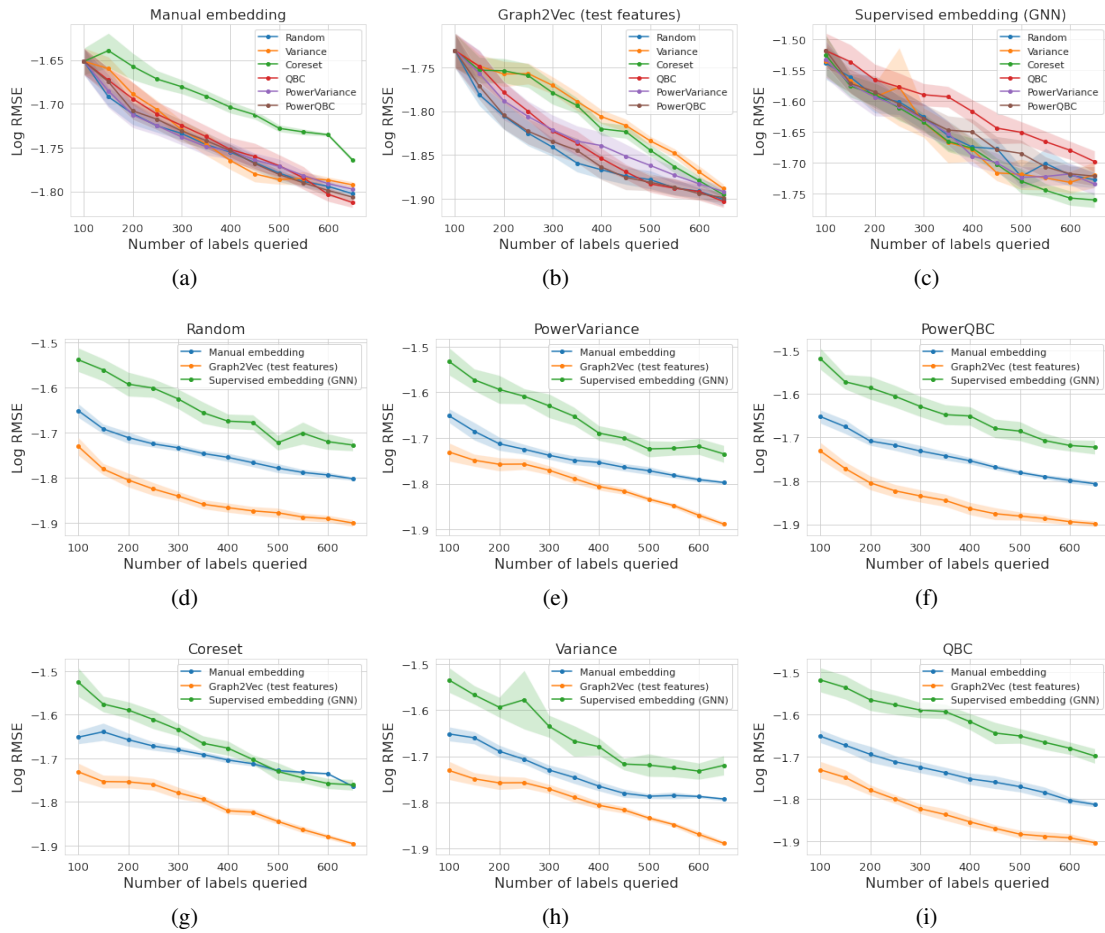


Figure 24: Active learning results for all embeddings for the OSSBuilds dataset (File Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

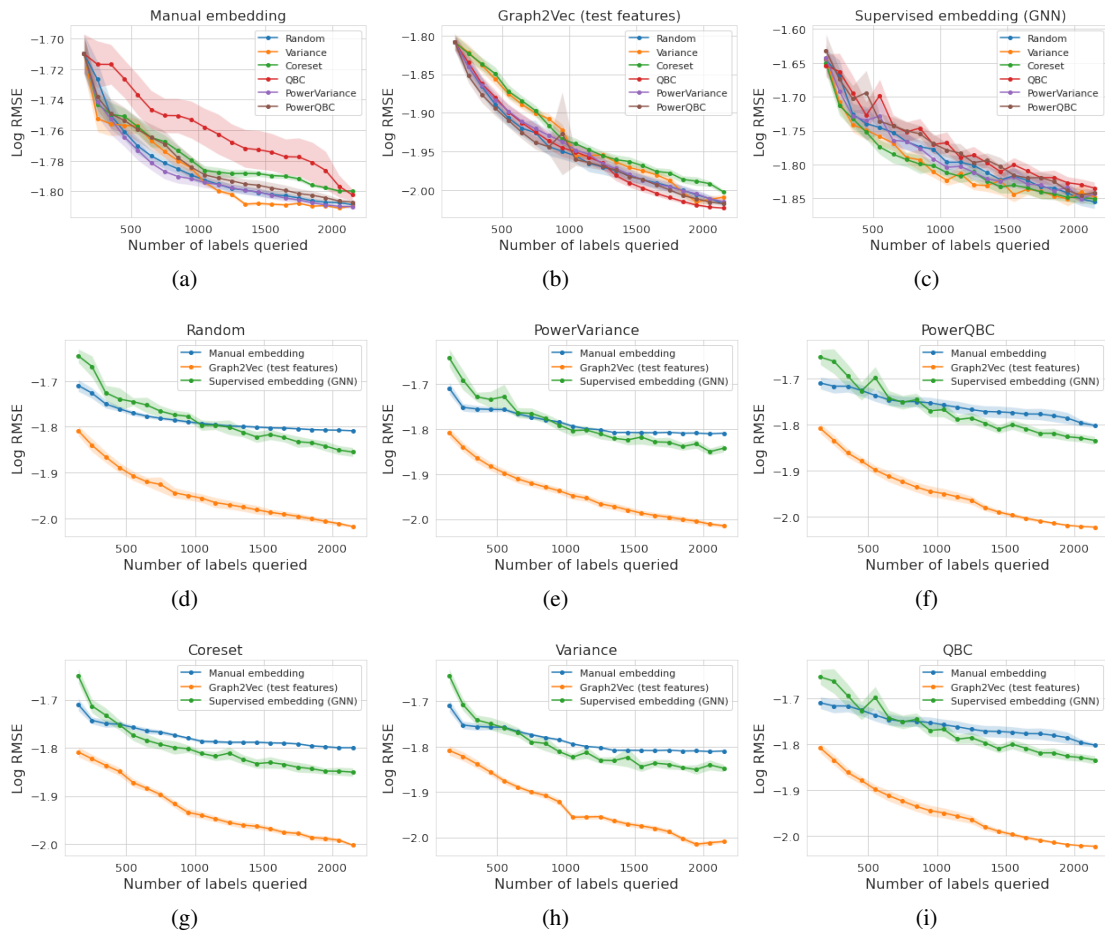


Figure 25: Active learning results for all embeddings for the HadoopTests dataset (System Level Parsing) with For the RMSE $|\mathcal{L}_0| = 150$ and $|\mathcal{B}| = 100$.

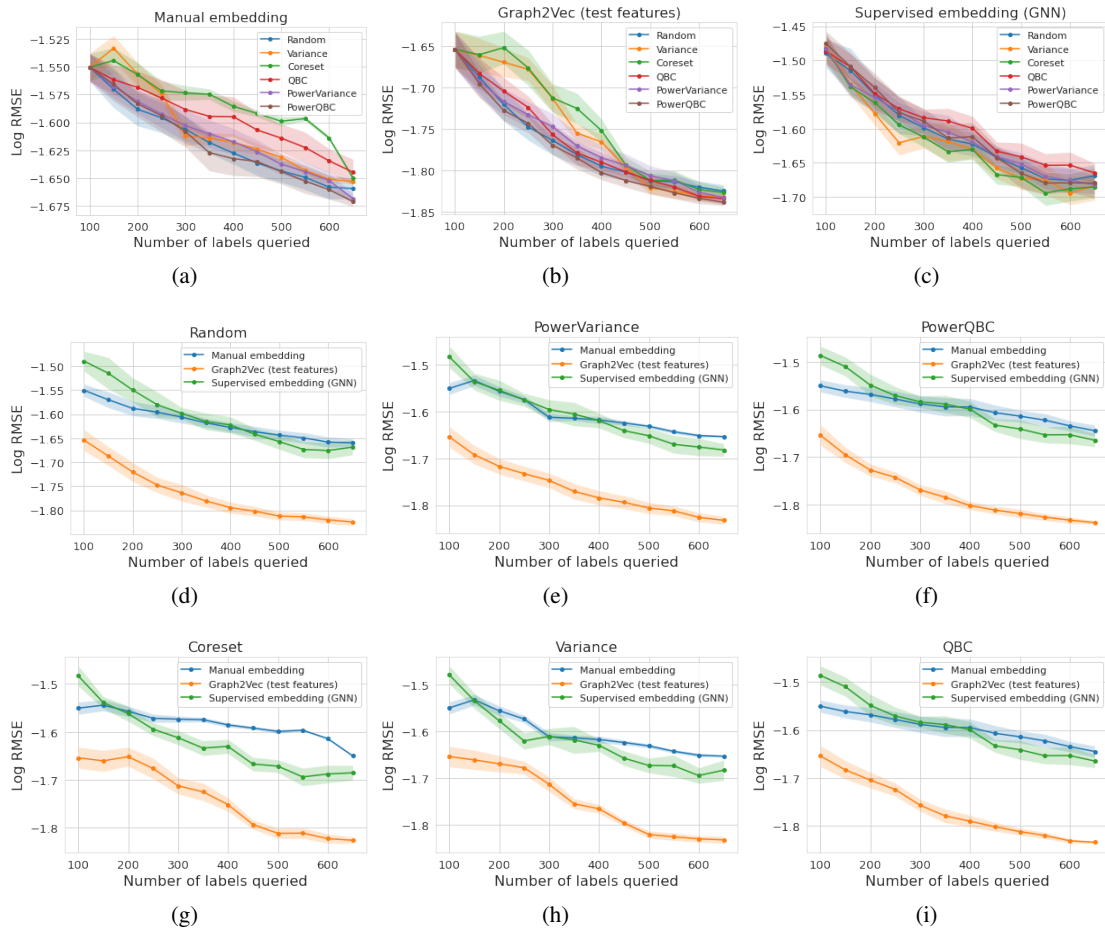


Figure 26: Active learning results for all embeddings for the OSSBuilds dataset (System Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

Paper 5

Batch Mode Deep Active Learning for Regression on Graph Data

Peter Samoaa, Linus Aronsson, Philipp Leitner, Morteza Haghiri Chehreghani

International Conference on Big Data (BigData), 2023

Batch Mode Deep Active Learning for Regression on Graph Data

Peter Samoaa

*Data Science and AI Division
Chalmers University of technology
Gothenburg, Sweden
samoaa@chalmers.se*

Linus Aronsson

*Data Science and AI Division
Chalmers University of technology
Gothenburg, Sweden
linaro@chalmers.se*

Philipp Leitner

*Interaction Design and Software Engineering
Chalmers University of technology
Gothenburg, Sweden
philipp.leitner@chalmers.se*

Morteza Haghiri Chehreghani

*Data Science and AI Division
Chalmers University of technology
Gothenburg, Sweden
morteza.chehreghani@chalmers.se*

Abstract—Acquiring labelled data for machine learning tasks, for example, for software performance prediction, remains a resource-intensive task. This study extends our previous work by introducing a batch-mode deep active learning approach tailored for regression in graph-structured data. Our framework leverages the source code conversion into Flow Augmented-AST graphs (FA-AST), subsequently utilizing both supervised and unsupervised graph embeddings. In contrast to single-instance querying, the batch-mode paradigm adaptively selects clusters of unlabeled data for labelling. We deploy an array of base kernels, kernel transformations, and selection methods, informed by both Bayesian and non-Bayesian strategies, to enhance the sample efficiency of neural network regression. Our experimental evaluation, conducted on multiple real-world software performance datasets, demonstrates the efficacy of the batch mode deep active learning approach in achieving robust performance with a reduced labelling budget. The methodology scales effectively to larger datasets and requires minimal alterations to existing neural network architectures.

Index Terms—Active Learning, Graph Neural Network, Deep Learning, Kernels.

I. INTRODUCTION

The rapid growth of machine learning (ML) applications across numerous domains is stifled by the limited availability of labelled data, including the domain of software engineering. Despite the abundance of source code files publicly hosted on platforms like GitHub, the absence of labels for these datasets remains a significant bottleneck. For tasks like performance prediction—which aims to forecast the execution time of software prior to execution—the cost of labelling is both computationally expensive and time-consuming. This conundrum gives rise to the need for Active Learning (AL) [26] techniques that efficiently identify the most informative samples for labelling. Active learning has been extensively investigated in various domains like text analysis [27], image data [5], [9], driving scenario trajectories [15], and drug design [28] to improve data annotation procedures. A particular challenge in deploying active learning for source code analysis arises from the representation of source code as graphs, coupled with the

lack of a unified framework suitable for diverse learning tasks, such as regression.

Numerous studies have explored the use of active learning in graph-based models, particularly focusing on node classification tasks via Graph Neural Networks (GNNs) [2], [6], [17], [30]. These works primarily address active learning at the node level. Some research extends this by incorporating reinforcement learning into the active learning framework. For instance, Hu et al. [13] train a policy network on labeled source graphs and transferred this policy to unlabeled graphs for node labeling tasks. Zhang et al. [31], [32] examine batch settings in active learning, employing multi-agent reinforcement learning and meta Q-learning to facilitate node labeling for classification purposes. Additionally, multi-armed bandit approaches have also been used for active learning in graph settings [7], [10]. Despite these advances, the existing literature largely concentrates on node-level classification tasks. The application of active learning to graph-level regression tasks remains not widely explored.

To mitigate this challenge, our recent work [22] proposes a unified active learning framework tailored for graph representations of source code. Our framework employs enhanced Abstract Syntax Trees (ASTs), which we term FA-ASTs [24]. These FA-ASTs capture a rich tapestry of syntactical, semantic, and lexical source code information and serve as the data points for our active learning model. Despite the versatility in accommodating various regression techniques, our existing framework in [22] falls short in supporting diverse sample selection across batches. This limitation is critical [16] and becomes especially acute given the computational demands of retraining models—particularly neural networks—after each labelling iteration. Batch Mode Active Learning (BMAL) offers a solution by allowing the selection of multiple data points for labelling simultaneously. It is then called atch Mode Deep Active Learning (BMDAL) when the BMAL approach is employed with deep learning models for extracting expressive features [21]. Specifically, we consider pool-based

BMDAL, where the data points for labelling are chosen from a predefined pool.

Inspired by recent work [11] that employs BMDAL for regression on tabular data, we aim to extend this framework to accommodate graph-based source code data. In particular, to apply BMDAL to graph data, we investigate GNNs and Graph2Vec for graph learning and fully connected neural networks for the regression task (i.e., for performance prediction). Regression tasks inherently lack a natural measure of uncertainty, which is often straightforward in classification tasks through softmax layers. Computing uncertainties in regression, therefore, becomes less straightforward necessitating the use of kernel methods. Therefore, we admit a Gaussian Process (GP) framework [20] in order to investigate and utilize different notions of uncertainty. We conduct our experiments on a real-world dataset that we have collected for this study. Our experimental results indicate that utilising GNN within the BMDAL framework provides the most effective setting for active learning querying methods compared to Graph2Vec.

In summary, our contributions are threefold:

- 1) We extend the BMDAL framework to make it compatible with graph data.
- 2) To the best of our knowledge, we are the first to adapt BMDAL for graph representations of source code specifically for regression tasks.
- 3) We validate our approach using real-world datasets.
- 4) By addressing these gaps, we offer a novel approach to the problem of active learning in source code analysis, thereby contributing to more efficient labelling and, ultimately, broader application of machine learning in software engineering.

The code and data are publicly available at [1].

II. SOURCE CODE REPRESENTATION

Listing 1: Simple example of Java source code

```

public static int factorial(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}

```

This study aims to bring the power of ML to software engineering by enhancing performance prediction models. For that, understanding how source code can be effectively represented is crucial. As we detailed in our previous systematic literature review [23], program source code can be converted into various forms, ranging from tree-based and graph-based to token-based representations.

In this paper, we use a Java method calculating the factorial of a number as a concrete example for source code representation, specifically focusing on the Abstract Syntax Tree (AST), Data Flow Graph (DFG), and Control Flow Graph (CFG). These different representations serve unique purposes and offer different types of information about the code.

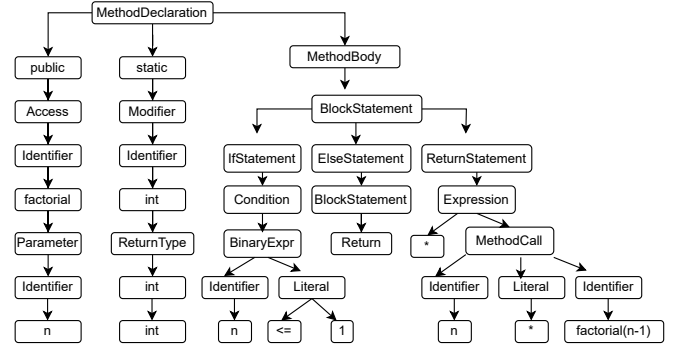


Fig. 1: Simplified abstract syntax tree (AST) for the code snippet in Listing 1

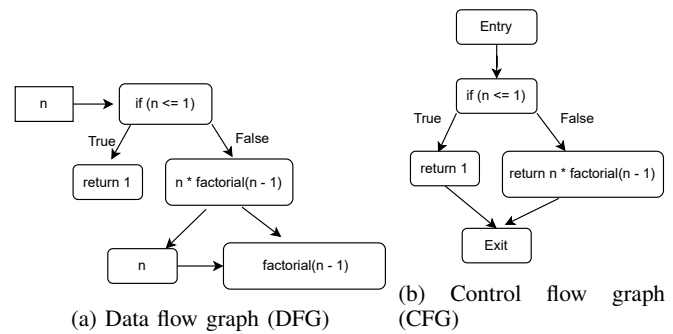


Fig. 2: Flow graphs representation for the code snippet in Listing 1

A. Abstract Syntax Tree (AST)

The AST representation is of particular interest due to the rich syntactical and lexical details it offers without the need for executing the code. An AST for our Java method is illustrated in Figure 1, where the tree structure provides an overview of the program's syntactic composition, including decision-making constructs like 'if' statements and expressions involving function calls and arithmetic operations. The AST is particularly beneficial for capturing the structural aspects of the code, which makes it well-suited for graph neural networks requiring many nodes and edges for meaningful feature extraction.

B. Data Flow Graph (DFG)

While the AST gives us valuable insights into the syntactic structure of the code, it does not capture how data moves or interacts within the program. This is where Data Flow Graphs (DFG) come into play. As demonstrated in Figure 2a, a DFG shows the flow of data between variables and computations, capturing the dependencies between different parts of the code.

C. Control Flow Graph (CFG)

To understand the runtime behaviour and possible paths that can be traversed during the code execution, Control Flow Graphs (CFG) are indispensable. Our Java method's CFG, shown in Figure 2b, presents a high-level overview of all

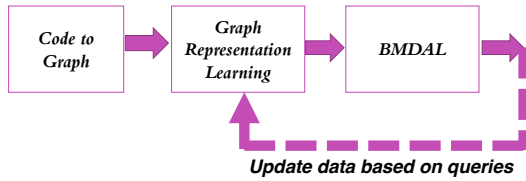


Fig. 3: BMDAL framework for graph data

possible routes the execution could take, from the initial method call to the return statements.

In summary, the combination of these source code representations enables us to comprehensively analyze and model the behaviour, structure, and data flow within a software system, which is particularly useful for ML-driven software engineering research.

III. ACTIVE LEARNING APPROACH

In this section, we outline the key components of our active learning framework. The process begins by converting source code into graph representations. These graphs are then embedded through unsupervised techniques or supervised. For supervised embeddings, we employ GNN in conjunction with active learning. This involves iteratively training the GNN based on newly added batches from the active learning process. During the active learning phase, we explore various selection methods, as well as kernels and their associated transformations.

A. Source Code to Graph

This section describes the methodology for constructing graphs from the source code, specifically Java files, as illustrated in Figure 4.

1) *AST Parsing*: We initially transform the source code into an AST as an intermediate representation. The AST representation can be extracted through source code parsing alone, without the need for executing the program. We use the pure Python Java parser `javalang`¹ to parse each test file and use the node types, values, and production rules in `javalang` to describe our ASTs. To encapsulate both semantic elements and syntactical attributes, we enhance the AST by integrating edges that capture data and control flow. This results in a Flow-Augmented AST (FA-AST) graph, a concept that was pioneered in our prior research [24].

The impetus for enriching the AST originates from contemporary research [23], underscoring the necessity for comprehensive code representations in applying deep learning techniques to software engineering. Given the intricate nature of performance prediction tasks, relying solely on the syntactic information derived from basic AST falls short of delivering high-fidelity outcomes. Therefore, we augment the tree-like architecture of the AST with additional semantic layers that signify both data and control flow, evolving it into a more elaborate graph. This enriched graph representation encodes a

¹<https://pypi.org/project/javalang/>

broader set of information than what is offered by the source code structure alone.



Fig. 4: Source Code to Graph Process

2) *Capturing Ordering and Data Flow*: To understand how the graphs are built, we will present each augmentation and then explain in detail how the FA-AST is built. We augment AST with different types of additional edges representing data flow and node order in the AST. Specifically, we use the following additional flow augmentation edges, in addition to the **AST child** and **AST parent** edges that are produced readily by AST parsing:

FA Next Token (b):

This type of edge connects a terminal node (leaf) in the AST to the next terminal node. Terminal nodes are nodes without children. In Figure 1, an FA Next Token edge would be added, for example, between `n` and `int` (the first leaves at the left bottom).

FA Next Sibling (c):

This connects each node (both terminal and non-terminal) to its next sibling and allows us to model the order of instructions in an otherwise unordered graph. In Figure 1, such an edge would be added, for example, connecting the `public` and with the `static` and `static` with `MethodBody` node.

FA Next Use (d):

This type of edge connects a node representing a variable to the place where this variable is next used. For example, the variable `n` is declared in the first line in Listing 1, and then used next in Lines 2 and 5.

3) *Capturing Control Flow*: In a second augmentation step, we now add further edges representing the control flow in the test cases. We currently support *if* statements, *while* and *for* loops, as well as *sequential execution*. We currently do not support *switch* statements or *do-while* loops, as these are less common. Java source code containing these elements will still be parsed successfully, but the FA-AST will not capture these control flow constructs. Specifically, the following further edges are added (see also Figure 5):

FA If Flow (e):

This type of edge connects the predicate (condition) of the *if*-statement with the code block that is executed if the condition evaluates to `true`. Every *if*-statement contains exactly one such edge by construction.

FA Else Flow (f):

Conversely, this edge type connects the predicate to the (optional) *else* code block.

FA While Flow (g):

A while loop essentially entails two elements - a condition and a code block that is executed as long as the condition remains `true`. We capture this through a FA While Flow (g) edge connecting the condition to the code block, and an FA

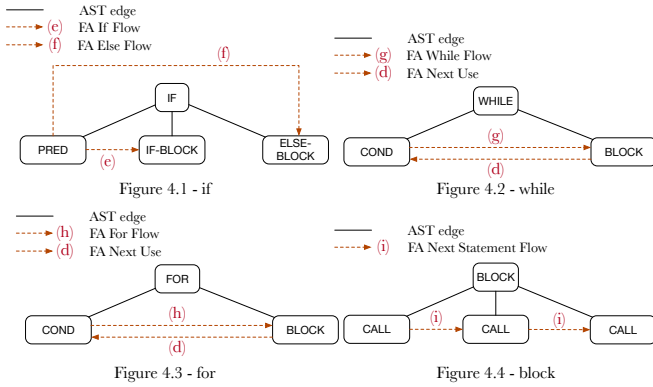


Fig. 5: Additional flow augmentations for different control flow constructs

Next Use (d) edge in the reverse direction. The latter is used to model the next usage of a loop counter.

FA For Flow (h):

For loops are conceptually similar to while loops. We use FA For Flow (h) edges to connect the condition to the code block, and an FA Next Use (d) edge in the reverse direction. Similar to the modelling of while-loops, FA Next Use (d) relates to the usage (typically incrementing) of a loop counter.

FA Next Statement Flow (i):

In addition to the control flow constructs discussed so far, Java of course also supports the simple sequential execution of multiple statements in a sequence within a code block. FA Next Statement Flow edges (i) are used to represent this case. Different from the constructs discussed so far, a code block can contain an arbitrary number of children, and the FA Next Statement Flow edge is always used to connect each statement to the one directly following it.

B. Graph Representation Learning

The graph structure of the data items in \mathcal{G} restricts the types of regression models that can be used, and thus, the types of query strategies to be employed for active learning. Therefore, we construct embeddings that can be used to project the graph data into a latent space where any regression model (and thus query strategy) can be utilized.

Since we focus on directed graphs, we use embedding algorithms compatible with directed graphs where the adjacency matrix is not symmetric. For this purpose, we explore three main approaches: unsupervised embeddings based on shallow embedding methods and supervised embeddings (based on GNNs). Each of these categories is listed and explained below.

1) *Unsupervised Embedding*: In our previous study [22], we investigate a number of shallow graph embeddings based on matrix factorization or skip-gram-based embeddings. The obtained results (Table 3 and Table 5 in [22]) show that Graph2Vec [18] achieves the best results for graph-level embedding across all unsupervised graph embeddings. Thus, in this paper, we use Graph2Vec as the unsupervised embedding.

Algorithm 1 Pool-based BMDAL loop in *unsupervised setting*

Require: Graphs \mathcal{G} , BMDAL algorithm NEXTBATCH (see Algorithm 3), list $\mathcal{L}_{\text{batch}}$ of batch sizes

- 1: $\mathcal{X} = \text{Graph2Vec}(\mathcal{G})$
- 2: Split \mathcal{X} into $\mathcal{X}_{\text{train}}$, $\mathcal{X}_{\text{pool}}$, $\mathcal{X}_{\text{test}}$
- 3: **for** AL batch size $\mathcal{N}_{\text{batch}}$ in $\mathcal{L}_{\text{batch}}$ **do**
- 4: Train NN model f_{θ} on $\mathcal{X}_{\text{train}}$
- 5: Evaluate NN model f_{θ} on $\mathcal{X}_{\text{test}}$
- 6: $\mathcal{X}_{\text{batch}} \leftarrow \text{NEXTBATCH}(f_{\theta}, \mathcal{X}_{\text{train}}, \mathcal{X}_{\text{pool}}, \mathcal{N}_{\text{batch}})$
- 7: Move $\mathcal{X}_{\text{batch}}$ from $\mathcal{X}_{\text{pool}}$ to $\mathcal{X}_{\text{train}}$ and acquire labels $\mathcal{Y}_{\text{batch}}$ for $\mathcal{X}_{\text{batch}}$
- 8: **end for**
- 9: Train final model f_{θ} on $\mathcal{X}_{\text{train}}$
- 10: Evaluate final model f_{θ} on $\mathcal{X}_{\text{test}}$

2) *Supervised Embedding*: Similar to unsupervised settings we select the most accurate GNN model out of the state-of-the-art architectures (namely GCNConv, GraphSAGE, and GraphConv) that were used in our previous study. Thus, in our experiments, we use GraphConv [8] since it yields the most accurate results for our graph data.

C. Batch Mode Deep Active Learning

In this section, we will discuss how we use different selection methods and how kernels and kernel transformations are incorporated with the selection methods and neural networks. When constructing query strategies for the BMDAL framework, The following three criteria are generally considered for selecting batches [29]:

- *Informativeness*: The selection method should select samples where the model is mostly uncertain about the label.
- *Diversity*: The selection methods must ensure that the samples in the batch must be diverse and different from each other.
- *Representative*: The selection of the training set should be concentrated on the region where the pool data distribution has high density.

Algorithm 1 illustrates the general procedure for pool-based BMDAL utilizing unsupervised graph embedding. Initially, we derive the embeddings for the complete graph set \mathcal{G} using Graph2Vec. The embeddings \mathcal{X} are then partitioned into training $\mathcal{X}_{\text{train}}$, testing $\mathcal{X}_{\text{test}}$, and pooling $\mathcal{X}_{\text{pool}}$ subsets. Note that the test labels are never used in training or querying for active learning. Within the BMDAL loop, the neural network (NN) model is first trained on the initially labelled embeddings $\mathcal{X}_{\text{train}}$ and subsequently evaluated on $\mathcal{X}_{\text{test}}$. Next, a batch $\mathcal{X}_{\text{batch}} \subset \mathcal{X}_{\text{pool}}$ is selected using the NEXTBATCH method, which forms the core of BMDAL. The labeled set is updated by transferring the selected batch $\mathcal{X}_{\text{batch}}$ from $\mathcal{X}_{\text{pool}}$ to $\mathcal{X}_{\text{train}}$ and acquiring the labels $\mathcal{Y}_{\text{batch}}$ for it. The NN model is then retrained on the extended $\mathcal{X}_{\text{train}}$ and re-evaluated on $\mathcal{X}_{\text{test}}$. Finally, the model is trained on the complete $\mathcal{X}_{\text{train}}$ and evaluated on $\mathcal{X}_{\text{test}}$.

Algorithm 2 illustrates the general steps for pool-based BMDAL in supervised setting. The process is slightly different

Algorithm 2 Pool-based BMDAL loop in *supervised setting*

Require: Graph Data \mathcal{G} , initial labeled graphs training set \mathcal{G}_{train} , unlabeled graphs pool set \mathcal{G}_{pool} , test set \mathcal{G}_{test} , BMDAL algorithm NEXTBATCH (see Algorithm 3), list \mathcal{L}_{batch} of batch sizes

- 1: **for** AL batch size \mathcal{N}_{batch} in \mathcal{L}_{batch} **do**
- 2: GNN = GraphConv(\mathcal{G}_{train}) {training the GNN model}
- 3: $\mathcal{X} = \text{GNN.embedding}(\mathcal{G})$
- 4: Extract $\mathcal{X}_{train}, \mathcal{X}_{test}, \mathcal{X}_{pool}$ from the embedding set \mathcal{X} based on the indices of $\mathcal{G}_{train}, \mathcal{G}_{test}, \mathcal{G}_{pool}$,
- 5: Train NN model f_θ on \mathcal{X}_{train}
- 6: Evaluate NN model f_θ on \mathcal{X}_{test}
- 7: $\mathcal{X}_{batch} \leftarrow \text{NEXTBATCH}(f_\theta, \mathcal{X}_{train}, \mathcal{X}_{pool}, \mathcal{N}_{batch})$
- 8: Move \mathcal{X}_{batch} from \mathcal{X}_{pool} to \mathcal{X}_{train} and acquire labels \mathcal{Y}_{batch} for \mathcal{X}_{batch}
- 9: Update \mathcal{G}_{train}
- 10: **end for**
- 11: Train final model f_θ on \mathcal{X}_{train}
- 12: Evaluate final model f_θ on \mathcal{X}_{test}

since the GNN model is incorporated into the active learning process because \mathcal{X}_{train} is updated in each iteration in order to utilize the recently labelled data. Here, we first define the indices of training, test, and pool sets in advance. Then, in the active learning loop, we initially train the GNN model in order to obtain an initial embedding. Then, based on this embedding, we train the NN model and evaluate it on \mathcal{X}_{test} . Then, we select \mathcal{X}_{batch} by *NEXTBATCH*. Next, we update the labelled set by moving the selected batch \mathcal{X}_{batch} from \mathcal{X}_{pool} to \mathcal{X}_{train} and acquire the labels \mathcal{Y}_{batch} for \mathcal{X}_{batch} . Thus, \mathcal{X}_{train} is then extended, and we train the GNN again based on the extended training graph set to obtain a new embedding. The NN is then trained again on the extended embeddings set and so on. At the end of iteration, we train the final model on the full \mathcal{X}_{train} and evaluate it on \mathcal{X}_{test} .

Algorithm 3 Kernel-based batch construction framework

- 1: **function** NEXTBATCH($f_\theta, \mathcal{X}_{train}, \mathcal{X}_{pool}, \mathcal{N}_{batch}$)
- 2: $k \leftarrow \text{BaseKernel}(f_\theta)$
- 3: $k \leftarrow \text{TransformKernel}(k, \mathcal{X}_{train})$
- 4: **return** SELECT($k, \mathcal{X}_{train}, \mathcal{X}_{pool}, \mathcal{N}_{batch}$)
- 5: **end function**

Kernels and Kernel Transformation in BMDAL: The usage of kernels and related transformations is inspired by the study in [11]. The authors formulate the use of kernels and kernel transformations within a general framework for BMDAL for tabular regression data.

The kernel-based batch construction framework outlined in Algorithm 3 serves as a fundamental component in Algorithms 1 and 2. This framework enables the manipulation of kernels and kernel transformations, fulfilling key functionalities.

A primary motivation for employing kernels in this framework is to emphasize *informativeness* as a crucial criterion

for assessing the efficiency of selection methods. This is particularly vital for tasks involving uncertainty quantification. Whereas softmax layers commonly serve to measure uncertainty in classification, such methods are not directly applicable to regression tasks. In regression that yields scalar outputs—such as execution time in our case study—a straightforward uncertainty quantification mechanism is absent. This gap is bridged by using Gaussian Process (GP), a Bayesian technique that computes uncertainties via kernel methods.

In Gaussian Process, the selected kernel plays a critical role in determining the quality of the uncertainty estimates. In the context of Neural Networks, the base kernel (computed in line 2 of Algorithm 3) is used to approximate the NN by capturing similarities between data points in the feature space, which is obtained post-training. Kernels can be transformed to either enhance computational efficiency or better represent the relations between data points. The purpose of the kernel transformations (as introduced in [11]) is to formulate many existing BMDAL methods under one common framework.

After transforming the kernel, a selection method (*SELECT*) is invoked. This method utilizes the transformed kernel to guide the selection process, as detailed in Algorithm 4.

In our experiments, we use the neural tangent kernel (NTK) [14] as the base kernel. We use this kernel because it mimics the neural network and performs the best overall when used in conjunction with different selection methods according to the experiments of [11]. The NTK $\Theta(x, x')$ given two input vectors x and x' is defined as the Jacobian of the NN outputs with respect to the network parameters θ , evaluated at the initial parameters, and then taking their inner product (see Eq.1).

$$\Theta(x, x') = \sum_{i,j} \frac{\partial f_i(x)}{\partial \theta_j} \frac{\partial f_i(x')}{\partial \theta_j}, \quad (1)$$

Note that $f_i(x)$ is the i^{th} output of the neural network for input x , and θ_j is the j^{th} parameter of the network.

We consider four different kernel transformations in this paper. First, the GP posterior covariance after observing the training data \mathcal{X}_{train} for a given base kernel k with the corresponding feature map ϕ which is defined in Eq.2.

$$k_{\rightarrow post(\mathcal{X}_{train}, \sigma^2)}(x, x') = \sigma^2 \phi(x)^T (\phi(\mathcal{X}_{train}^T \phi(\mathcal{X}_{train}) + \sigma^2 \mathbf{I})^{-1} \phi(x')) \quad (2)$$

Note that σ^2 is the variance of the observation noise in the underlying model.

Second, we use the scaling transformation where we employ a scaling factor $\lambda \in \mathbb{R}$ to form the scaled kernel $\lambda^2 k$ with the feature map $\lambda \phi$. This is particularly important when using a GP with $\lambda^2 k$ as its covariance function, as it quantifies the covariance between $f(x)$ and $f(\bar{x})$ based on the prior over functions f .

$$k_{\rightarrow scale(\mathcal{X}_{train})}(x, x') = \lambda^2 k(x, x') \quad (3)$$

The third transformation is sketching, employed to approximate a high-dimensional kernel k with a lower-dimensional one for computational efficiency. We refer to Holzmüller et al. [11] for details.

Finally, we utilize two kernel transformations corresponding to two different ways of applying the ACS-FW method from Pinsler et al. [19] applied to GP regression. Thereby, we use *acs-rf* (kernel of Bayesian batch active learning as sparse subset approximation with p random features) and *acs-rf-hyper* (kernel of Bayesian batch active learning as sparse subset approximation with p random features and hyperprior on σ^2). We refer to Pinsler et al. for details of this method, and Holzmüller et al. [11] for details of the kernel transformation applied to GP regression.

Algorithm 4 Iterative Selection Algorithm Template with Customizable Function `NextSample`

Require: $k, \mathcal{X}_{\text{train}}, \mathcal{X}_{\text{pool}}, \mathcal{X}_{\text{batch}}, \text{mode} \in \{\text{P}, \text{TP}\}$
Ensure: $\mathcal{X}_{\text{batch}}$

- 1: **function** `SELECT` ($k, \mathcal{X}_{\text{train}}, \mathcal{X}_{\text{pool}}, \mathcal{X}_{\text{batch}}, \text{mode} \in \{\text{P}, \text{TP}\}$)
- 2: $\mathcal{X}_{\text{batch}} \leftarrow \emptyset$
- 3: **if** $\text{mode} = \text{TP}$ **then**
- 4: $\mathcal{X}_{\text{mode}} \leftarrow \mathcal{X}_{\text{train}}$
- 5: **else**
- 6: $\mathcal{X}_{\text{mode}} \leftarrow \emptyset$
- 7: **end if**
- 8: **for** $i = 1$ to N_{batch} **do**
- 9: $\mathcal{X}_{\text{sel}} \leftarrow \mathcal{X}_{\text{mode}} \cup \mathcal{X}_{\text{batch}}$ {Currently "selected" points}
- 10: $\mathcal{X}_{\text{rem}} \leftarrow \mathcal{X}_{\text{pool}} \setminus \mathcal{X}_{\text{batch}}$ {Currently unselected points}
- 11: $\mathcal{X}_{\text{batch}} \leftarrow \mathcal{X}_{\text{batch}} \cup \{\text{NextSample}(k, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}})\}$
- 12: **end for**
- 13: **return** $\mathcal{X}_{\text{batch}}$
- 14: **end function**

Selection Methods: We will now discuss a variety of kernel-based selection methods to be used for querying in active learning. Algorithm 4 shows the details of the selection method *SELECT* that was manipulated in Algorithm 3. To favour samples with high informativeness in an iterative active learning scheme that tries to enforce the diversity of the selected batch, two approaches can be used according to [11]:

- (*P*) Informativeness can be incorporated through the kernel. For example, $k \rightarrow \mathcal{X}_{\text{train}}(x, x)$ represents the posterior variance at x of a GP
- (*TP*) Informativeness can be incorporated implicitly by enforcing diversity of $\mathcal{X}_{\text{train}} \cup \mathcal{X}_{\text{batch}}$ instead of only enforcing diversity of $\mathcal{X}_{\text{batch}}$. In other words, a batch that is sufficiently different from the training set typically necessarily contains new information.

This explains the usage of *mode* parameter in *SELECT* in Algorithm 4 for different selection methods. It is worth mentioning that we use the same setting that was used in the experiments by Holzmüller et al. [11]. Algorithm 4 serves as a generalized mechanism for constructing sample batches.

It takes as input a kernel k , the current training set $\mathcal{X}_{\text{train}}$, a pool of potential samples $\mathcal{X}_{\text{pool}}$, an initially empty batch $\mathcal{X}_{\text{batch}}$, and a mode parameter which can either be *P* or *TP*. The algorithm starts by initializing an empty set $\mathcal{X}_{\text{batch}}$ which will be incrementally populated with samples. Depending on the selected mode (*P* or *TP*), the algorithm initializes another set $\mathcal{X}_{\text{mode}}$ either as an empty set or as equivalent to the current training set $\mathcal{X}_{\text{train}}$. Then it loops for N_{batch} iterations, where in each iteration, the set of currently "selected" samples, denoted by \mathcal{X}_{sel} , is updated to be the union of $\mathcal{X}_{\text{mode}}$ and $\mathcal{X}_{\text{batch}}$. Additionally, the remaining samples \mathcal{X}_{rem} are updated to consist of those samples in the pool $\mathcal{X}_{\text{pool}}$ which have not yet been added to $\mathcal{X}_{\text{batch}}$. The selection method (denoted by `NextSample` in the algorithm) then selects a new sample from the remaining set \mathcal{X}_{rem} , based on the kernel k and the set of currently selected samples \mathcal{X}_{sel} . This new sample is added to $\mathcal{X}_{\text{batch}}$. After N_{batch} iterations, the algorithm returns the selected batch $\mathcal{X}_{\text{batch}}$.

Table I shows the selection methods investigated in our experiments and the corresponding kernels and kernels transformation used. Note that many of the selection methods correspond to existing methods in the active learning literature, some of which were originally formulated for classification. Holzmüller et al. [11] formulate each of these selection methods under one common framework and adapt them to regression if needed. For simplicity, we use similar (but shortened) formulations, but we refer to [11] for details of each method.

- *Random Selection.* This corresponds to sampling a data point uniformly at random from the points in the pool \mathcal{X}_{rem} . The selection method is shown in Eq.4.

$$\text{NextSample}(K, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}}) \sim \mathcal{U}(\mathcal{X}_{\text{rem}}), \quad (4)$$

where $\mathcal{U}(\mathcal{X}_{\text{rem}})$ is the uniform distribution over \mathcal{X}_{rem} . For this method both P and TP are equivalent.

- *MAXDIAG.* This corresponds to Eq.5. It is shown in [11] that this is equivalent to BALD [12] in a regression setting.

$$\text{NextSample}(K, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}}) = \operatorname{argmax}_{x \in \mathcal{X}_{\text{rem}}} K(x, x) \quad (5)$$

According to Eq.5, MAXDIAG selects the maximum of the elements on the diagonal of the posterior covariance matrix. For this method both P and TP are equivalent.

- *MAXDET.* This corresponds to Eq.6. It is shown in [11] that this is equivalent to BatchBALD [16] in the regression setting. This only holds under certain conditions, see [11] for details.

$$\text{NextSample}(K, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}}) = \operatorname{argmax}_{x \in \mathcal{X}_{\text{rem}}} \det(k(\mathcal{X}_{\text{sel}} \cup \{x\}, \mathcal{X}_{\text{sel}} \cup \{x\}) + \sigma^2 \mathbf{I}) \quad (6)$$

MAXDET. This is considered an improvement over *MAXDIAG* because it takes \mathcal{X}_{sel} into account by conditioning the GP on \mathcal{X}_{sel} when computing the posterior covariance.

- **BAIT**. This corresponds to the selection method introduced by Ash et al. [3]. BAIT potentially improves on the previous selection methods by also considering how well the selected batch represents the current pool set. It is shown in [11] that the original formulation from [3] is equivalent to Eq.7.

$$\text{NextSample}(K, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}}) = \underset{x \in \mathcal{X}_{\text{rem}}}{\text{argmin}} \sum_{x' \in \mathcal{X}_{\text{train}} \cup \mathcal{X}_{\text{pool}}} k_{\rightarrow \text{post}}(\mathcal{X}_{\text{sel}} \cup x, \sigma^2)(x', x') \quad (7)$$

Note that [3] introduces two versions of BAIT: forward and forward/backward. Eq.7 corresponds to the forward version of BAIT, which we use in our experiments due to superior performance.

- **FRANKWOLFE**. This method approximates the kernel mean embedding using a Frank-Wolfe optimization algorithm. To ensure that $\mathcal{X}_{\text{batch}}$ accurately represents the pool set, Pinsler et al. [19] recommend constructing $\mathcal{X}_{\text{batch}}$ in a manner that closely approximates $\sum_{x \in \mathcal{X}_{\text{pool}}} \phi(x)$ by $\sum_{x \in \mathcal{X}_{\text{batch}}} w_x \phi(x)$, where w_x 's are non-negative weights. Specifically, they advocate the use of the Frank-Wolfe optimization algorithm to solve the related optimization problem, enabling an iterative selection of elements into $\mathcal{X}_{\text{batch}}$. This method aims to approximate the distribution of $\mathcal{X}_{\text{pool}}$ through $\mathcal{X}_{\text{batch}}$ by mimicking the empirical kernel mean embedding $N_{\text{pool}}^{-1} \sum_{x \in \mathcal{X}_{\text{pool}}} k(x, \cdot)$ using $\mathcal{X}_{\text{batch}}$. The strategy can be executed in either the kernel or feature space. Due to the quadratic scaling with N_{pool} in the kernel space, Pinsler et al. [19] opt for the feature space approach when handling large pool sets, a choice we also adopt in our experiments. Unlike the original method which allows for repeated selection of the same $x \in \mathcal{X}_{\text{pool}}$, we disallow this to ensure batch sizes remain consistent for a fair comparison with other techniques.
- **MAXDIST**. This corresponds to greedily selecting data points that maximize the distance to those already selected. The selection method is shown in Eq.8.

$$\text{NextSample}(k, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}}) = \underset{x \in \mathcal{X}_{\text{rem}}}{\text{argmax}} \min_{x' \in \mathcal{X}_{\text{sel}}} d_k(x, x') \quad (8)$$

This method is equivalent to Coreset [25] for a particular configuration of the kernel (see [11] for details).

- **KMEANSPP**. This is defined in Eq.9 and is related to BADGE [4] (see [11] for details).

$$\text{NextSample}(k, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}}) = \frac{\min_{x' \in \mathcal{X}_{\text{sel}}} d_k(x, x')^2}{\sum_{x' \in \mathcal{X}_{\text{rem}}} \min_{x \in \mathcal{X}_{\text{sel}}} d_k(x, x')^2} \quad (9)$$

Much like MaxDet, MaxDist ensures both Informativeness and Diversity but falls short on Representativity. To address this, one can consider batch selection as a clustering problem. In Eq.9, the optimization task essentially reformulates the k-medoids problem, blending

the k-means clustering objective with the stipulation that cluster centroids must be selected from the clustered dataset.

- **LCMD**. As a deterministic counterpart to the stochastic k-meansPP method, Holzmüller et al. [11] introduce a method known as LCMD (Largest Cluster Maximum Distance). This selection method considers representativity by restricting selections to the largest cluster, while also promoting diversity by selecting the data point that is furthest away within that cluster. In this context, $x' \in \mathcal{X}_{\text{sel}}$ denotes cluster centroids, $c(x)$ signifies the associated center for each $x \in \mathcal{X}_{\text{rem}}$, and $S(x')$ represents the size of the cluster. According to Eq.10, the data point with the greatest distance from the largest cluster is selected.

$$\begin{aligned} \text{NextSample}(K, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}}) = \\ \underset{x \in \mathcal{X}_{\text{rem}}: s(c(x)) = \max_{x' \in \mathcal{X}_{\text{sel}}} s(x')}{\text{argmax}} d_K(x, c(x)) \quad (10) \\ c(x) = \underset{x' \in \mathcal{X}_{\text{sel}}}{\text{argmin}} d_k(x, x') \\ s(x') = \sum_{x \in \mathcal{X}_{\text{rem}}: c(x) = x'} d_k(x, x')^2 \end{aligned}$$

TABLE I: An overview of the used kernel and kernel transformation for each selection method

Selection Method	Kernel	Kernel Transformation	Mode
BAIT	NTK	<i>sketch</i> \rightarrow <i>scale</i> \rightarrow <i>post</i>	P
MAXDIST			P
MAXDET		<i>sketch</i> \rightarrow acs-rf	-
KNEANSPP		<i>sketch</i> \rightarrow acs-rf-hyper	P
MAXDIAG			-
FRANKWOLFE			
LCMD		<i>sketch</i>	TP
Random	-	-	-

IV. EXPERIMENT

A. Collection of Data

To bolster the dependability of our experiments, two distinct real-world datasets consisting of performance metrics are utilized. The first, called *OSSBuild*, consists of actual build data acquired from the continuous integration frameworks of four distinct open-source projects. The second, termed *HadoopTests*, is a more expansive dataset that we gathered ourselves by running the Hadoop open-source system's unit tests in a well-regulated setting. A summarization of both datasets can be found in Table II. Further details about each dataset are elaborated in the subsequent subsections.

1) *OSSBuild Dataset*: Initially employed in the work of Samoa et al. [24], this dataset includes data related to test run times in the build systems of four open-source softwares: systemDS, H2, Dubbo, and RDF4J. All of these projects make use of public continuous integration platforms and provide publicly available build details, which we used to gather data on test execution times in the summer of 2021. Refer to

TABLE II: Overview of the OSSBuilds and HadoopTests datasets.

	Proj.	Desc.	Files	Runs	Nodes	Vocab.
OSS	sysDS	Apache ML for Science Data lifecycle	127	1321	114904	3205
	H2	Java SQL DB	194	1391	432375	18326
	Dubbo	Apache Remote Procedure Call framework	123	524	77142	4505
	RDF4J	Scalable RDF	478	1055	242673	10844
	Tot.		922	4291	867094	36880
Hadoop	Hadoop	Apache framework for big data	2895	24348	5090798	138952

Table II (top) for essential statistics about these projects. The term "Files" refers to the unit test files we monitored for execution durations, while "Runs" signifies the aggregated execution count for these files. "Nodes" and "Vocabulary Size" denote the graphs. Prior to parsing, we exclude code comments to minimize the graph nodes. We observe 867094 nodes and 36880 vocabulary entries.

2) *HadoopTests Dataset*: In order to address the limitations of the OSSBuild dataset, particularly the confined file counts per project, a second dataset was generated. We selected the Apache Hadoop project due to its extensive collection of test files (2895) with adequate complexity. We executed all of the unit tests in the project five times and recorded each test file's execution time, as reported by the JUnit framework. We utilized a dedicated virtual machine equipped with two virtualized CPUs and 8 GBytes of RAM for this data collection, and non-essential services were disabled to ensure consistent performance. Statistics for the HadoopTests dataset are outlined in Table II (bottom). The dataset has an enlarged node count with 5090798 nodes and 138952 vocabulary terms.

B. Experiment Setting

To systematically investigate different combinations of kernels, kernel transformations, and selection methods as outlined in Table 1, we subject our datasets to these various selection techniques. For the HadoopTests dataset, the initial training size, denoted by $\mathcal{N}_{\text{train}}$, is set at 256, while for OSSBuilds, it is 88. We then proceed to obtain 16 batches, each having a size of $\mathcal{N}_{\text{batch}}$ equalling 128 for HadoopTests and 45 for OSSBuilds, applying the corresponding BMAL method. This entire process is repeated 10 times, each time with unique initialization seeds for the neural network (NN) and different partitions of the data into training, pool, and test subsets. The evaluation metric we consider is the root mean squared error (RMSE) calculated on the test dataset after each BMAL iteration. The logarithm of the RMSE error metric is then

averaged over 10 repetitions and, depending on the specific experiment, over 16 steps for each dataset and embedding.

The GNN model is configured with three layers of Graph-Cov layers. In contrast, for the NN model, we employ a fully connected architecture consisting of three layers, each having 512 neurons in both of the hidden layers. The activation function chosen for both networks is 'relu'. The training of both GNN and NN is executed using the Adam optimizer, spanning 256 epochs with a batch size of 32. The embedding dimension in the supervised and unsupervised settings is 90.

C. Experimental Results

In this section, we will present the average RMSE values. The mean log RMSE for each embedding is illustrated in different subfigures. We assess the performance of the configurations outlined in Table I.

a) *BMDAL for HadoopTests*: Figure 6 illustrates the performance of various selection methods in the context of HadoopTests. It breaks down the results by depicting the average log RMSE in both supervised and unsupervised embeddings. In unsupervised embedding, shown in Figure 6a, Random selection consistently underperforms relative to other methods. MAXDIST stands out as the most effective, particularly as the labelled data grow. Moving to the supervised embedding results in Figure 6b, the Random selection method still performs the poorest, while BAIT and MAXDET consistently outperform the rest across various training set sizes. Interestingly, the typical best-performing methods (BAIT and MAXDET) do not maintain their lead when the size of the labelled data is restricted to around 256 samples. In this specific context, MAXDIST is the most effective method. Overall, the results demonstrate the effectiveness of active learning, i.e., the benefits of non-random selection methods, especially MAXDIST.

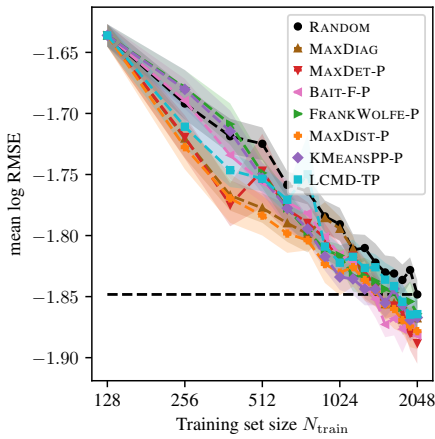
b) *BMDAL for OssBuilds*: The evaluations for OssBuilds reveal a noticeably higher variance in each selection method for OSSBuilds compared to the HadoopTest dataset. This increased variability is likely due to having fewer samples of OSSBuilds, which comprises graphs from four distinct projects.

In the unsupervised embedding setting, as indicated by Figure 7a, both Random and FRANKWOLFE methods generally underperform. Intriguingly, LCMD exhibits a sudden and significant improvement, becoming the best-performing method when the training batch size reaches approximately 256. However, this performance gain is ephemeral, as its RMSE error escalates once again beyond this point.

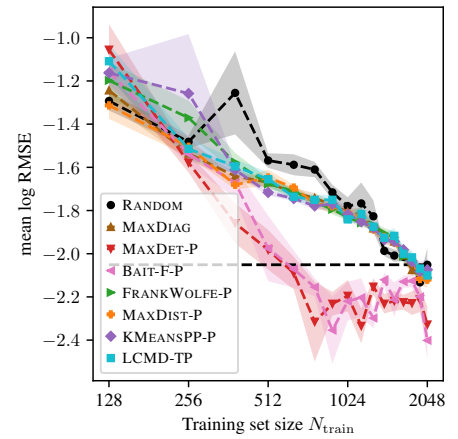
Despite Random being the least effective method in supervised settings as in Figure 7b, certain variations appear at smaller training set sizes. Specifically, in the first 64 labelled training samples, FRANKWOLFE underperforms most notably. For the same training sample size, MAXDIST emerges as the best performer, consistent with the HadoopTest dataset.

D. Further Discussions

In this section, we discuss further the results from various perspectives.

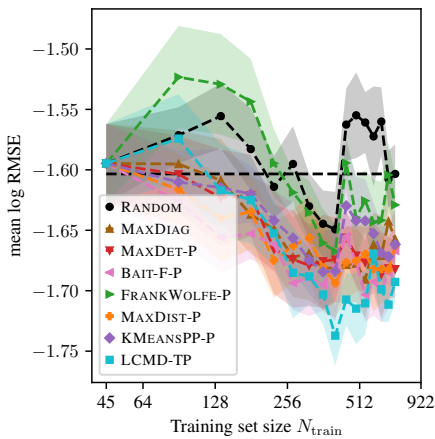


(a) Unsupervised using Graph2Vec

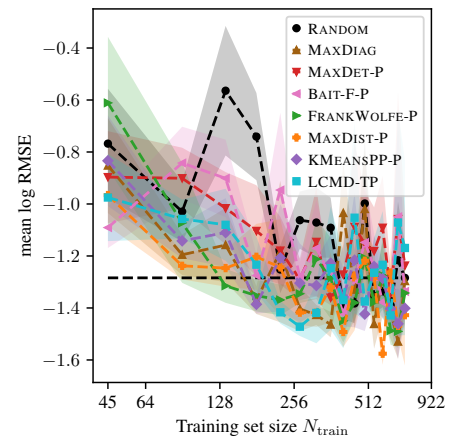


(b) Supervised using GNN

Fig. 6: Root mean square error for BMDAL in both embedding settings on HadoopTests.



(a) Unsupervised using Graph2Vec



(b) Supervised using GNN

Fig. 7: Root mean square error for BMDAL in both embedding settings on OssBuilds.

- **Observations on data variability:** Our results indicate smoother and less variable performance for HadoopTests compared to OssBuilds. This difference is primarily due to the source of the graphs. While Hadoop’s graphs originate from a single project, OssBuilds features graphs from various domain projects (as detailed in Table II). Additionally, the larger number of graphs in Hadoop contributes to this stability.
- **Performance w.r.t. embedding types:** Upon examining the mean log RMSE values, it is clear that supervised embeddings offer the most effective setting for the selection methods. This is evidenced by the lower mean log RMSE and higher delta (mean log RMSE) — 1.4 for HadoopTests and 1 for OssBuilds—compared to 0.25 for unsupervised embedding. However, caution is warranted in generalizing these findings, as they may require validation with more diverse graph data from various projects.

- **Computational considerations:** It is worth noting that the supervised setting comes with increased computational demands. This is because each active learning iteration involves not only training an NN based on the embeddings but also training the GNN to obtain those embeddings.
- **Graph characteristics and implications:** Our analysis performed on graph data in our previous study [22] (Table 2) reveals that the graphs in our study are characterized by high diameter and sparsity, adding complexity to the task. Furthermore, these graphs are augmented versions of Abstract Syntax Trees (ASTs).
- **Comparison with previous work:** Interestingly, our current findings diverge from our previous paper where batch and kernel components were not utilized. This highlights the crucial role both the active learning and the quality of embeddings play in influencing the results.

V. CONCLUSION

In this study, we employed Batch Mode Deep Active Learning (BMDAL) for graph data within a regression framework. The algorithm integrates kernels and kernel transformations with active learning selection methods. Specifically, the Neural Tangent Kernel (NTK) serves as the base kernel, while the Gaussian Process (GP) posterior variance is primarily utilized for kernel transformation. Supervised and unsupervised embedding are investigated to adapt the graph data to this framework. Our experimental results indicate that supervised embedding provides the most effective setting for selection methods. While identifying a universally optimal selection method across different experimental settings proved challenging, MAXDET and MAXDIST consistently emerged as top performers. Conversely, the Random method, used as a baseline, consistently ranked as the least effective, indicating the advantage of active learning for data labelling.

ACKNOWLEDGMENT

This work received financial support from the Swedish Research Council VR under grant number 2018-04127. The work of Linus Aronsson was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundations.

REFERENCES

- [1] *Batch Mode Deep Active Learning for Regression on Graph Data*. Zenodo, Sept. 2023. <https://doi.org/10.5281/zenodo.8352242>.
- [2] R. Abel and Y. Louzoun. Regional based query in graph active learning, 2019.
- [3] J. Ash, S. Goel, A. Krishnamurthy, and S. Kakade. Gone fishing: Neural active learning with fisher embeddings. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 8927–8939. Curran Associates, Inc., 2021.
- [4] J. T. Ash, C. Zhang, A. Krishnamurthy, J. Langford, and A. Agarwal. Deep batch active learning by diverse, uncertain gradient lower bounds, 2020.
- [5] J. D. Bossér, E. Sörstadius, and M. H. Chehreghani. Model-centric and data-centric aspects of active learning for deep neural networks. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 5053–5062, 2021.
- [6] H. Cai, V. W. Zheng, and K. C.-C. Chang. Active learning for graph embedding, 2017.
- [7] X. Chen, G. Yu, J. Wang, C. Domeniconi, Z. Li, and X. Zhang. Activehne: Active heterogeneous network embedding. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 2123–2129. International Joint Conferences on Artificial Intelligence Organization, 7 2019.
- [8] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.
- [9] Y. Gal, R. Islam, and Z. Ghahramani. Deep bayesian active learning with image data. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1183–1192. PMLR, 06–11 Aug 2017.
- [10] L. Gao, H. Yang, C. Zhou, J. Wu, S. Pan, and Y. Hu. Active discriminative network representation learning. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2142–2148. International Joint Conferences on Artificial Intelligence Organization, 7 2018.
- [11] D. Holzmüller, V. Zaverkin, J. Kästner, and I. Steinwart. A framework and benchmark for deep batch active learning for regression. *Journal of Machine Learning Research*, 24(164):1–81, 2023.
- [12] N. Houthby, F. Huszár, Z. Ghahramani, and M. Lengyel. Bayesian active learning for classification and preference learning, 2011.
- [13] S. Hu, Z. Xiong, M. Qu, X. Yuan, M.-A. Côté, Z. Liu, and J. Tang. Graph policy network for transferable active learning on graphs, 2020.
- [14] A. Jacot, F. Gabriel, and C. Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.
- [15] S. Jarl, L. Aronsson, S. Rahrovani, and M. H. Chehreghani. Active learning of driving scenario trajectories. *Engineering Applications of Artificial Intelligence*, 113:104972, 2022.
- [16] A. Kirsch, J. van Amersfoort, and Y. Gal. Batchbald: Efficient and diverse batch acquisition for deep bayesian active learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
- [17] X. Li, Y. Wu, V. Rakesh, Y. Lin, H. Yang, and F. Wang. Smartquery: An active learning framework for graph neural networks through hybrid uncertainty reduction. In *Proceedings of the 31st ACM International Conference on Information; Knowledge Management, CIKM '22*, page 4199–4203, New York, NY, USA, 2022. Association for Computing Machinery.
- [18] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal. graph2vec: Learning distributed representations of graphs, 2017.
- [19] R. Pinsler, J. Gordon, E. Nalisnick, and J. M. Hernández-Lobato. Bayesian batch active learning as sparse subset approximation. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
- [20] C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006.
- [21] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, and X. Wang. A survey of deep active learning, 2021.
- [22] P. Samooa, L. Aronsson, A. Longa, P. Leitner, and M. H. Chehreghani. A unified active learning framework for annotating graph data with application to software source code performance prediction, 2023.
- [23] P. Samooa, F. Bayram, P. Salza, and P. Leitner. A systematic mapping study of source code representation for deep learning in software engineering. *IET Software*, 16(4):351–385, 2022.
- [24] P. Samooa, A. Longa, M. Mohamad, M. H. Chehreghani, and P. Leitner. Tep-gnn: Accurate execution time prediction of functional tests using graph neural networks. In D. Taibi, M. Kuhmann, T. Mikkonen, J. Klünder, and P. Abrahamsson, editors, *Product-Focused Software Process Improvement*, pages 464–479, Cham, 2022. Springer International Publishing.
- [25] O. Sener and S. Savarese. Active learning for convolutional neural networks: A core-set approach. In *International Conference on Learning Representations*, 2018.
- [26] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.
- [27] Y. Shen, H. Yun, Z. C. Lipton, Y. Kronrod, and A. Anandkumar. Deep active learning for named entity recognition, 2018.
- [28] S. Viet Johansson, H. Gummesson Svensson, E. Bjerrum, A. Schliep, M. Hagher Chehreghani, C. Tyrchan, and O. Engkvist. Using active learning to develop machine learning models for reaction yield prediction. *Molecular Informatics*, 41(12):2200043, 2022.
- [29] D. Wu. Pool-based sequential active learning for regression. *IEEE Transactions on Neural Networks and Learning Systems*, 30(5):1348–1359, 2019.
- [30] Y. Wu, Y. Xu, A. Singh, Y. Yang, and A. Dubrawski. Active learning for graph neural networks via node feature propagation, 2019.
- [31] Y. Zhang, H. Tong, Y. Xia, Y. Zhu, Y. Chi, and L. Ying. Batch active learning with graph neural networks via multi-agent deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(8):9118–9126, Jun. 2022.
- [32] Y. Zhang, Y. Xia, Y. Zhu, Y. Chi, L. Ying, and H. Tong. Active heterogeneous graph neural networks with per-step meta-q-learning. In *2022 IEEE International Conference on Data Mining (ICDM)*, pages 1329–1334, 2022.