A unified active learning framework for annotating graph data for regression task

Citation for the original published paper (version of record):
Samoaa, H., Aronsson, L., Longa, A. et al (2024). A unified active learning framework for annotating graph data for regression task. Engineering Applications of Artificial Intelligence, 138.
http://dx.doi.org/10.1016/j.engappai.2024.109383

(article starts on next page)

Engineering Applications of Artificial Intelligence 138 (2024) 109383

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

A unified active learning framework for annotating graph data for regression task

Peter Samoaa [a,*], Linus Aronsson [a], Antonio Longa [b], Philipp Leitner [c], Morteza Haghir Chehreghani [a]

[a] Chalmers University of Technology, Data Science and AI, Sweden
[b] University of Trento, Italy
[c] Chalmers University of Technology, Interaction Design and Software Engineering, Sweden

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.7792485

Keywords:
Graph neural networks (GNNs)
Active learning
Graphs-level regression

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 unlabelled 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.

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 (Settles, 2009). 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 (Bossér et al., 2021; Li and Oliva, 2021; Casanova et al., 2020; Sener and Savarese, 2018), recommender systems (Rubens et al., 2015), driver behaviour identification (Comuni et al., 2022), sound event detection

* Corresponding author.
E-mail address: samoaa@chalmers.se (P. Samoaa).

https://doi.org/10.1016/j.engappai.2024.109383
Received 9 August 2023; Received in revised form 25 April 2024; Accepted 23 September 2024
Available online 4 October 2024

(Shuyang et al., 2020), classification of driving time series (Jarl et al., 2022), reaction prediction in drug discovery (Viet Johansson et al., 2022), logged data analysis (Yan et al., 2018), medical analysis (Konyushkova et al., 2017; Li and Oliva, 2021), text processing (Vu et al., 2019), and person re-identification (Liu et al., 2019).

Although active learning has been extensively studied in the context of node-level classification tasks (Zhang et al., 2022; Hu et al., 2020; Wu et al., 2020; Cai et al., 2017), 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 (Luo et al., 2021; Guo et al., 2022; Wang et al., 2022) or to update the knowledge representation in knowledge graphs (Shin et al., 2015; Yang and Hou, 2023). 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 unlabelled 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 (Sener and Savarese, 2018), Query-by-Committee (Seung et al., 1992) and uncertainty selection based on Gaussian Processes (Kapoor et al., 2007). We find that no active learning query strategy consistently outperforms the others for all datasets, consistent with previous work on active learning (Konyushkova et al., 2017). 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 useable 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 labelling 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 Samoaa et al. (2023).

2. Background

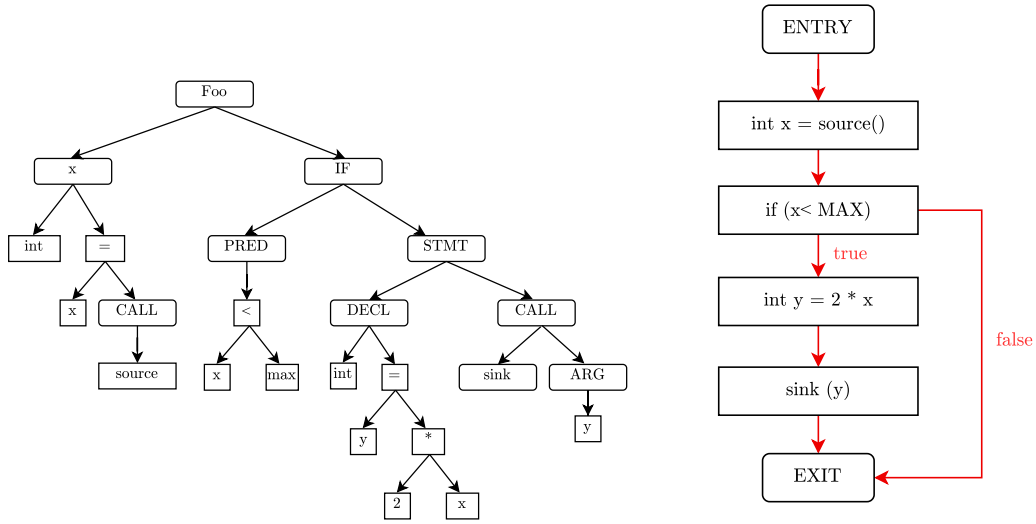
In this section, we provide an overview of the fundamental concepts underlying our approach. We first introduce the notion of graphs, then we present how source code can be represented as a graph. Later, those concepts are used to explain our framework and to evaluate the predicted execution time of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

2.1. Graphs

A graph is a mathematical structure used to model relational data across various domains such as social networks (Lachi et al., 2023; Nguyen et al., 2022; Scott, 2011), biological networks (Huber et al., 2007; Aittokallio and Schwikowski, 2006), interaction networks (Longa et al., 2024; Arregui-García et al., 2024; Longa 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 neighbourhood* 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 neighbours. The *density* of a directed graph is defined as 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.



(a) Abstract syntax tree (AST) for the code snippet in Listing 1 (Yamaguchi et al., 2014).

(b) Control Flow Graph (CFG) for the code snippet in Listing 1 (Yamaguchi et al., 2014).

Fig. 1. Example of Tree and Graph Representation for the code snippet in Listing 1 (Yamaguchi et al., 2014).

Listing 1: Simple example of C source code (from Yamaguchi et al., 2014).

```
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 Fig. 1(a) showing the AST for the code sample given in snippet 1 by Yamaguchi et al. (2014). 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. Fig. 1(b) displays the CFG for the code in snippet 1 by Yamaguchi et al. (2014). Control flow graphs are widely used in reverse engineering for program comprehension, although they lack data flow details despite depicting control flow.

3. Related work

Active Learning (AL) has been widely studied across various domains, including text (Shen et al., 2018) and image data (Gal et al., 2017), 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 (Li et al., 2022; Abel and Louzoun, 2019). 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, Cai et al. (2017) introduced AGE, an active graph embedding framework that operates at the node-level using uncertainty and representativeness as querying strategies. Similarly, Wu et al. (2020) 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 Hu et al. (2020), Zhang et al. (2022) applied active learning to graph data using reinforcement learning. Hu et al. (2020) 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, Zhang et al. (2022) 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 Gao et al. (2018), Chen et al. (2019) investigated multi-arm bandits in an active learning setting. Gao et al. (2018) proposed ANRMAB, which uses Information Entropy, Node Centrality, and Information Density as querying strategies for node-level labelling. Meanwhile, Chen et al. (2019) 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.

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

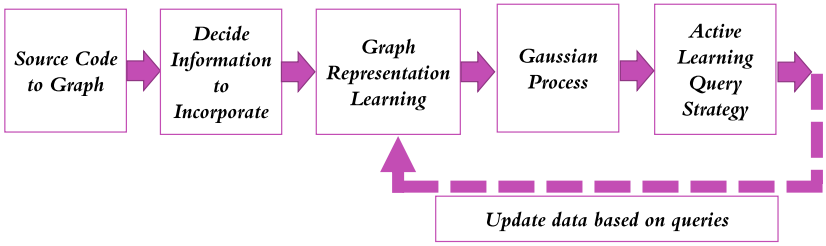


Fig. 2. Representation learning and Active Learning Strategies.

both supervised and unsupervised learning, and engaging with different graph sizes.

Our work is inspired by the study in Jarl et al. (2022), 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. Fig. 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 Fig. 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 unlabelled 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 Fig. 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 (Samoaa et al., 2022b).

[1] Note 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 labelled training dataset.

Our motivation for augmenting the AST comes from recent studies (Samoaa et al., 2022a), emphasizing 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.

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 Fig. 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:

[2] https://pypi.org/project/javalang/

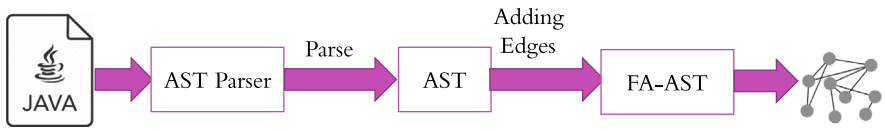


Fig. 3. Source Code to Graph Process.

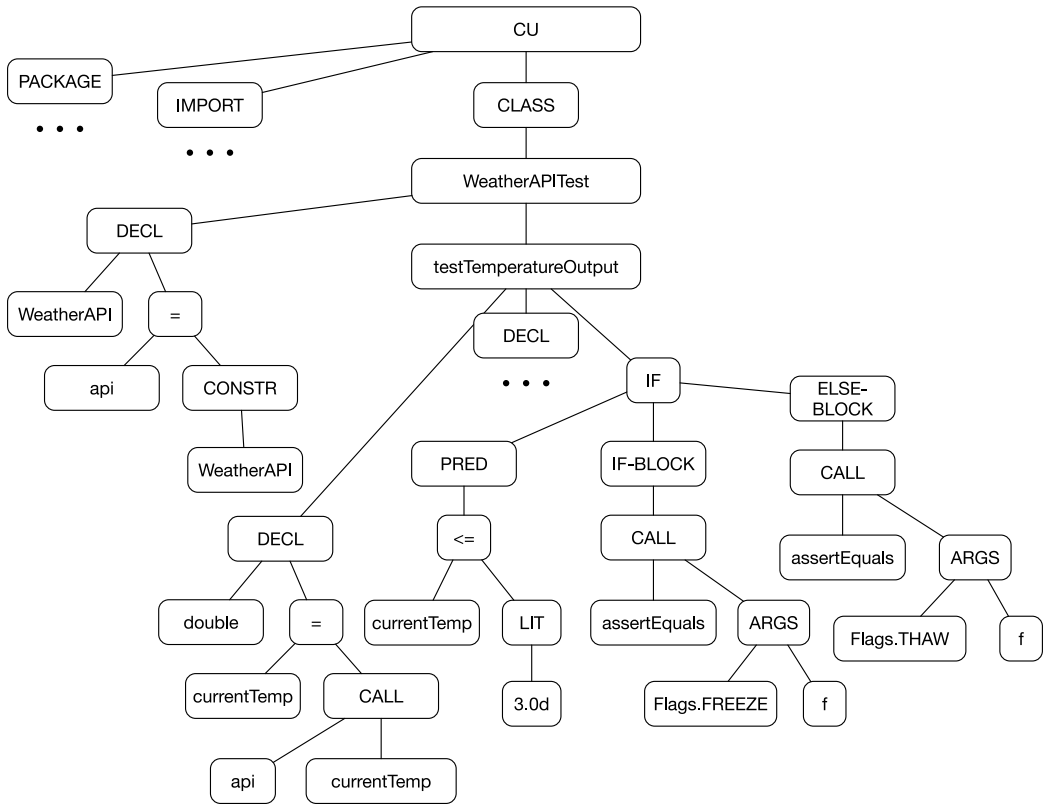


Fig. 4. Simplified abstract syntax tree (AST) representing the illustrative example presented in Listing 2. Package declarations, import statements, as well as the declaration in Line 15 are skipped for brevity.

- **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. 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 Fig. 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.

Fig. 5 shows an example augmenting the AST in Fig. 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 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 Fig. 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.

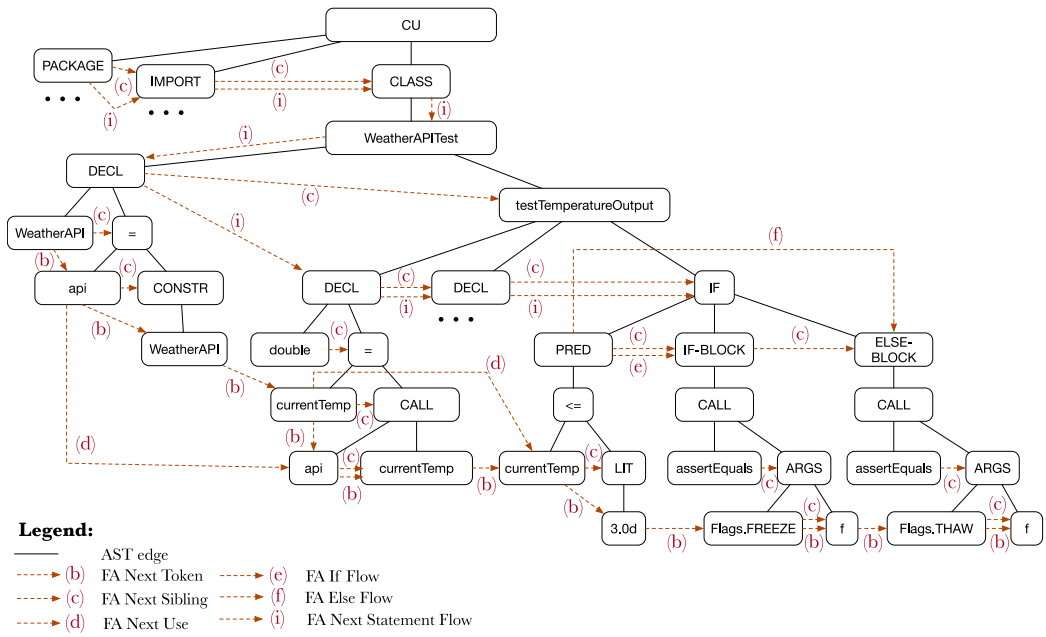


Fig. 5. Flow-Augmented AST (FA-AST) for the example presented in Listing 2. Solid lines represent AST parent and child edges, and dashed lines different types of flow augmentations.

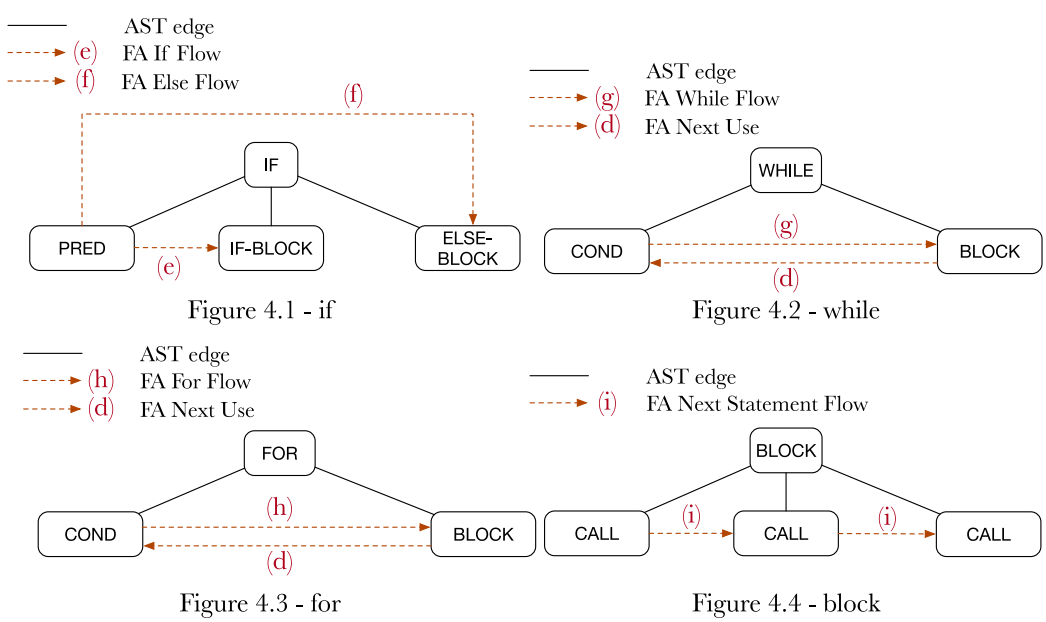


Fig. 6. Additional flow augmentations for different control flow constructs.

- **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. 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 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.

Fig. 7 illustrates the hierarchy of unsupervised embedding algorithms used. The hierarchy is inspired by Chami et al. (2022). We have two main types of shallow embedding approaches: matrix factorization and skip-gram. In matrix factorization, we use the Graph Representation (GR) approach (Cao et al., 2015) and Higher-Order Proximity Preserved Embedding (HOPE) (Wang et al., 2016), 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 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 (Perozzi et al., 2014), Node2Vec (Grover and Leskovec, 2016), 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 Chami et al. (2022), 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 (Kipf and Welling, 2017), GraphSAGE (Hamilton et al., 2017) and GraphConv (Defferrard et al., 2016). This is done using the well-known autoencoder neural network architecture (Kipf and Welling, 2016) (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 (Kipf and Welling, 2017), GraphSAGE (Hamilton et al., 2017), and GraphConv (Defferrard et al., 2016). 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.

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

  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.

$$h_i' = \sigma\left(W \cdot \mathrm{MEAN}(\{h_i\} \cup \{h_j, \forall j \in \mathcal{N}(i)\})\right) \tag{2}$$

  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.

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

  Where $H^{(l)}$ is the matrix of node features at layer l, U is the matrix of eigenvectors of the normalized graph Laplacian $L = I_N - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$, $\Lambda^{(l)}$ 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.

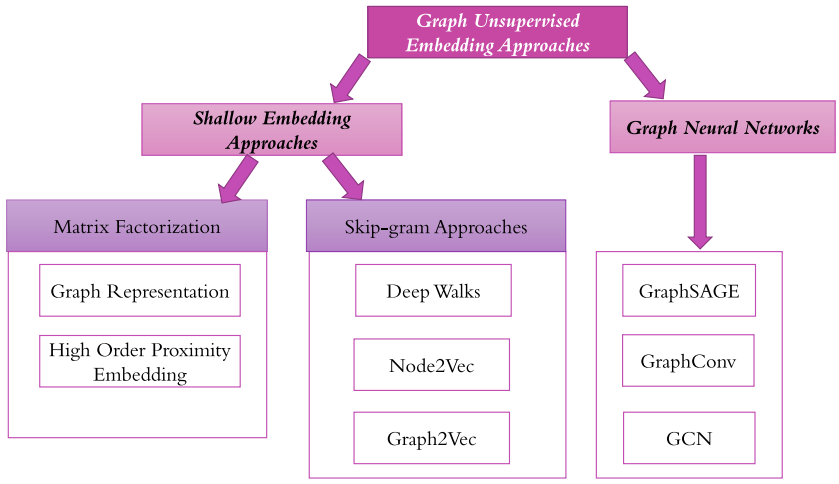


Fig. 7. Hierarchical structure of the different unsupervised graph embedding algorithms used in this study.

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. Fig. 8 shows a categorization of the extracted metrics. Below we list and explain each of the metrics.

1. **Integration Metrics** (Latora and Marchiori, 2001): 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** (Newman, 2002): 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** (Latora and Marchiori, 2001): 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)* (Watts and Strogatz, 1998): 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 Newman (2010). Five related metrics related to this category are listed below as the following:
   - *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 $\boldsymbol{X}_{\mathcal{A}}$ and $\boldsymbol{Y}_{\mathcal{A}}$ 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., $\boldsymbol{X}_{\mathcal{L}_i}$, $\boldsymbol{X}_{\mathcal{U}_i}$, $\boldsymbol{X}_{\mathcal{T}}$, $\boldsymbol{Y}_{\mathcal{L}_i}$, $\boldsymbol{Y}_{\mathcal{U}_i}$ and $\boldsymbol{Y}_{\mathcal{T}}$. As suggested by Munjal et al. (2020), 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 unlabelled data items $\boldsymbol{X}_{\mathcal{U}_i}$ (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 $\boldsymbol{X}_{\mathcal{L}_i}$ and $\boldsymbol{Y}_{\mathcal{L}_i}$). 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 $\boldsymbol{Y}_{\mathcal{T}}$, i.e., the labels of the test dataset.

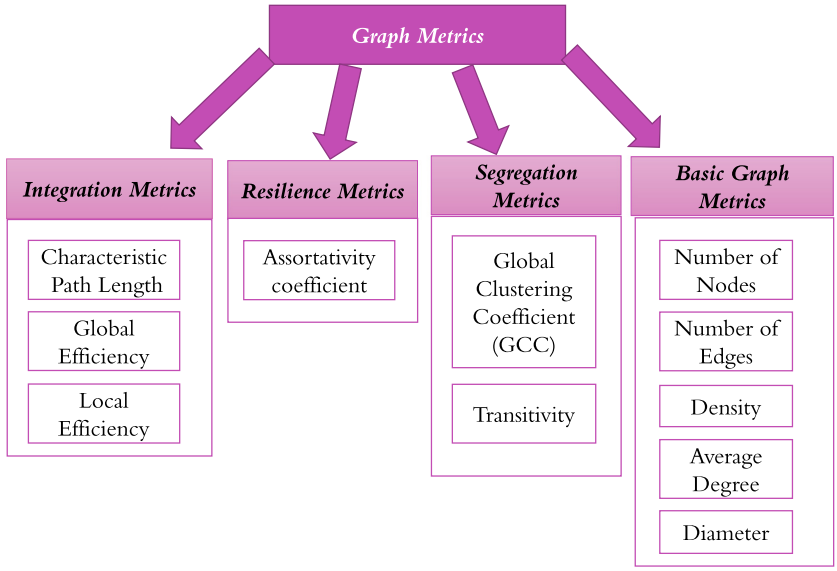


Fig. 8. Hierarchy of graph-based metrics.

- $\boldsymbol{X}_{\mathcal{L}_i}$, $\boldsymbol{X}_{\mathcal{U}_i}$ and $\boldsymbol{X}_{\mathcal{T}}$. 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.
- $\boldsymbol{X}_{\mathcal{L}_i}$ and $\boldsymbol{X}_{\mathcal{U}_i}$. 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 $\boldsymbol{X}_{\mathcal{L}_i}$ and $\boldsymbol{X}_{\mathcal{U}_i}$. 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 $\boldsymbol{X}_{\mathcal{L}_i}$ and $\boldsymbol{X}_{\mathcal{U}_i}$ 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).
- $\boldsymbol{X}_{\mathcal{L}_i}$ and $\boldsymbol{Y}_{\mathcal{L}_i}$. 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 $\boldsymbol{X}_{\mathcal{L}_i}$ and $\boldsymbol{Y}_{\mathcal{L}_i}$ (in each iteration i). Then, all data is projected into the latent space of the last layer of the GNN before performing the active learning.
- $\boldsymbol{X}_{\mathcal{L}_i}$, $\boldsymbol{Y}_{\mathcal{L}_i}$ and $\boldsymbol{X}_{\mathcal{U}_i}$. 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 (Rasmussen and Williams, 2005). 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 Rasmussen and Williams (2005) for the mathematical details of GPRs.

4.7. Query strategies for active learning

In this paper, we consider batch active learning (Ren et al., 2021). 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 Sener and Savarese (2018), 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 (Sener and Savarese, 2018).

- **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 $\sigma(\boldsymbol{x})$ correspond to the variance of some data item $\boldsymbol{x} \in \mathcal{U}_i$ (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 labelled and included in the labelled 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 $\mu_j(\mathbf{x})$ be the prediction of estimator j. We then compute the variance of the predictions as

$$\sigma_{\mathrm{QBC}}(\mathbf{x}) = \frac{1}{n}\sum_{j=1}^{n} \mu_j(\mathbf{x})^2 - \left(\frac{1}{n}\sum_{j=1}^{n} \mu_j(\mathbf{x})\right)^2. \tag{4}$$

  A batch is then selected as for the variance query strategy described above: $\mathcal{B}^* = \arg\max_{\mathcal{B}\subseteq\mathcal{U}_i,|\mathcal{B}|=B} \sum_{\mathbf{x}\in\mathcal{B}} \sigma_{\mathrm{QBC}}(\mathbf{x})$.

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 (Ren et al., 2021). However, the work in Kirsch et al. (2023) 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 $\epsilon \sim \mathrm{Gumbel}(0;1)$, 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?

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 Samoaa et al. (2022b)), 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.

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	110 651	3161	114 904	3205
	H2	Java SQL DB	194	1391	405 706	17 972	432 375	18 326
	Dubbo	Apache Remote Procedure Call framework	123	524	75 787	4499	77 142	4505
	RDF4J	Scalable RDF processing	478	1055	214 436	10 755	242 673	10 844
	Total		**922**	**4291**	**806 580**	**36 387**	**867 094**	**36 880**
HadoopTests	**Hadoop**	Apache framework for processing large datasets on clusters	**2895**	**24 348**	**4 314 360**	**135 408**	**5 090 798**	**138952**

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 Samoaa et al. (2023).

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 (Samoaa et al., 2022b).

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.,

$$tree - sim = \frac{|E| - (|V| - 1)}{(|V| - 1)(\frac{|V|}{2} - 1)} . \tag{5}$$

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.

From Table 2, it is easy to see that the input graph has a high diameter. In fact, if we generate a random graph (Batagelj and Brandes, 2005) 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.

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

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 48 GB of memory, two CPU Xeon(R) Gold 6338, and DDR4 RAM of 256 GB. However, the framework can be executed on less powerful machines with longer execution times as a consequence.

We used the Scikit-learn (Pedregosa et al., 2011) 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 (Kingma and Ba, 2014) 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 Appendix 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** ± 0.03	**0.74** ± 0.02	NA	NA
	GR	Mean	0.58 ± 0.03	0.50 ± 0.03	NA	NA
		Sum	0.47 ± 0.05	0.46 ± 0.04	NA	NA
	HOPE	Mean	0.16 ± 0.05	0.06 ± 0.03	NA	NA
		Sum	0.16 ± 0.05	0.37 ± 0.05	NA	NA
	DeepWalks	Mean	0.42 ± 0.05	0.47 ± 0.03	NA	NA
		Sum	0.41 ± 0.05	0.46 ± 0.04	NA	NA
	Node2Vec	Mean	0.30 ± 0.06	0.20 ± 0.03	NA	NA
		Sum	0.25 ± 0.07	0.40 ± 0.04	NA	NA
GNN	GCNConv		0.47 ± 0.05	0.52 ± 0.04	0.46 ± 0.04	0.50 ± 0.04
	GraphSAGE		0.44 ± 0.06	0.44 ± 0.04	0.42 ± 0.04	0.42 ± 0.04
	GraphConv		0.48 ± 0.05	0.52 ± 0.04	0.47 ± 0.05	0.51 ± 0.03

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

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 Appendix 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 $\boldsymbol{X}_{\mathcal{L}}$ and train labels $\boldsymbol{Y}_{\mathcal{L}}$ 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.

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 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.

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).

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** $\pm$ 0.03	**0.75** $\pm$ 0.02	NA	NA
	GR	Mean	0.45 $\pm$ 0.04	0.47 $\pm$ 0.02	NA	NA
		Sum	0.40 $\pm$ 0.05	0.43 $\pm$ 0.03	NA	NA
	HOPE	Mean	0.19 $\pm$ 0.07	0.06 $\pm$ 0.03	NA	NA
		Sum	0.20 $\pm$ 0.08	0.35 $\pm$ 0.04	NA	NA
	DeepWalks	Mean	0.37 $\pm$ 0.06	0.44 $\pm$ 0.02	NA	NA
		Sum	0.36 $\pm$ 0.06	0.43 $\pm$ 0.04	NA	NA
	Node2Vec	Mean	0.33 $\pm$ 0.06	0.42 $\pm$ 0.03	NA	NA
		Sum	0.36 $\pm$ 0.06	0.42 $\pm$ 0.04	NA	NA
GNN	GCNConv		0.41 $\pm$ 0.06	0.48 $\pm$ 0.03	0.44 $\pm$ 0.05	0.48 $\pm$ 0.03
	GraphSAGE		0.37 $\pm$ 0.06	0.42 $\pm$ 0.04	0.38 $\pm$ 0.04	0.45 $\pm$ 0.05
	GraphConv		0.43 $\pm$ 0.06	0.49 $\pm$ 0.03	0.44 $\pm$ 0.07	0.49 $\pm$ 0.03

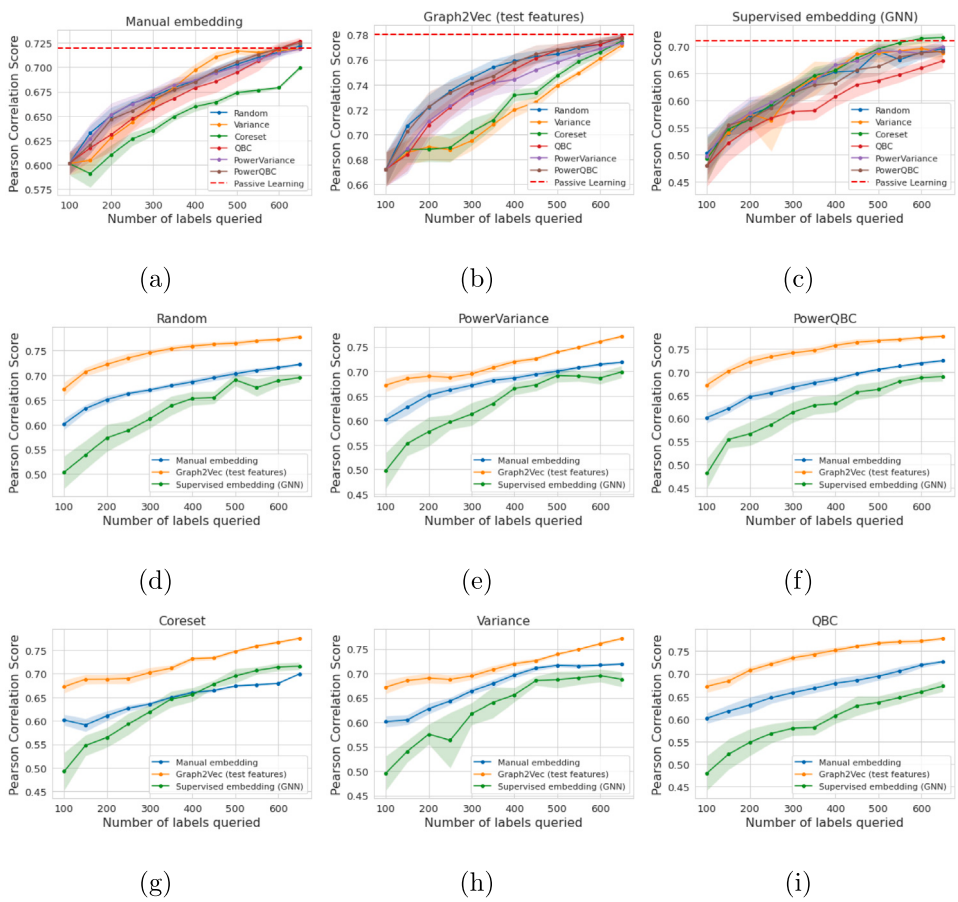


Fig. 9. Active learning results for all embeddings for the OSSBuilds dataset (File Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

Table 6
Results for Supervised and Manual Embedding for graphs of System Level Parsing.

		Train features	
		OSSBuilds	HadoopTests
Supervised Embedding (GNN)	GCNConv	0.59 $\pm$ 0.04	0.64 $\pm$ 0.03
	GraphSAGE	0.61 $\pm$ 0.04	**0.67** $\pm$ 0.02
	GraphConv	**0.65** $\pm$ 0.04	0.66 $\pm$ 0.02
Manual Embedding		0.60 $\pm$ 0.04	0.59 $\pm$ 0.03

File level parsing graphs. In Figs. 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 the best, but the supervised embedding

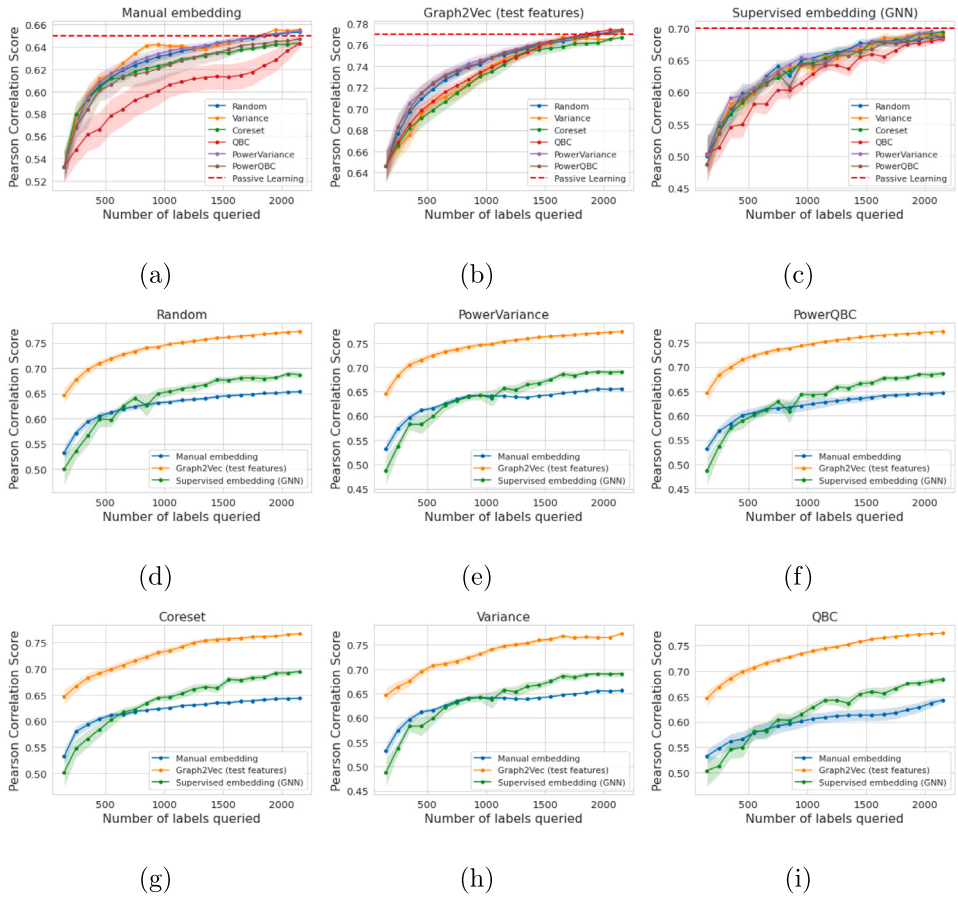


Fig. 10. Active learning results for all embeddings for the HadoopTests dataset (File Level Parsing) with $|\mathcal{L}_0| = 150$ and $|\mathcal{B}| = 100$.

outperforms the manual embedding in later iterations (when more labelled data is available).

System level parsing graphs. In Figs. 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 labelled 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.

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.

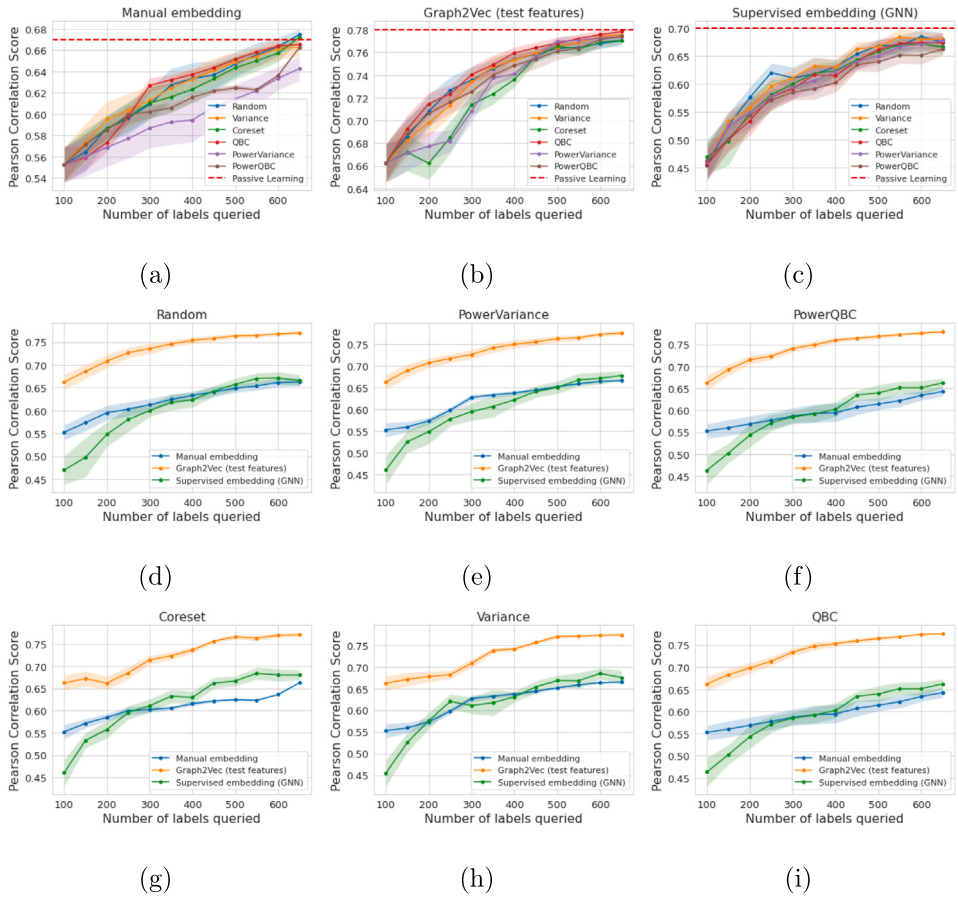


Fig. 11. Active learning results for all embeddings for the OSSBuilds dataset (System Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

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 (Konyushkova et al., 2017). 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 (Sener and Savarese, 2018); (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

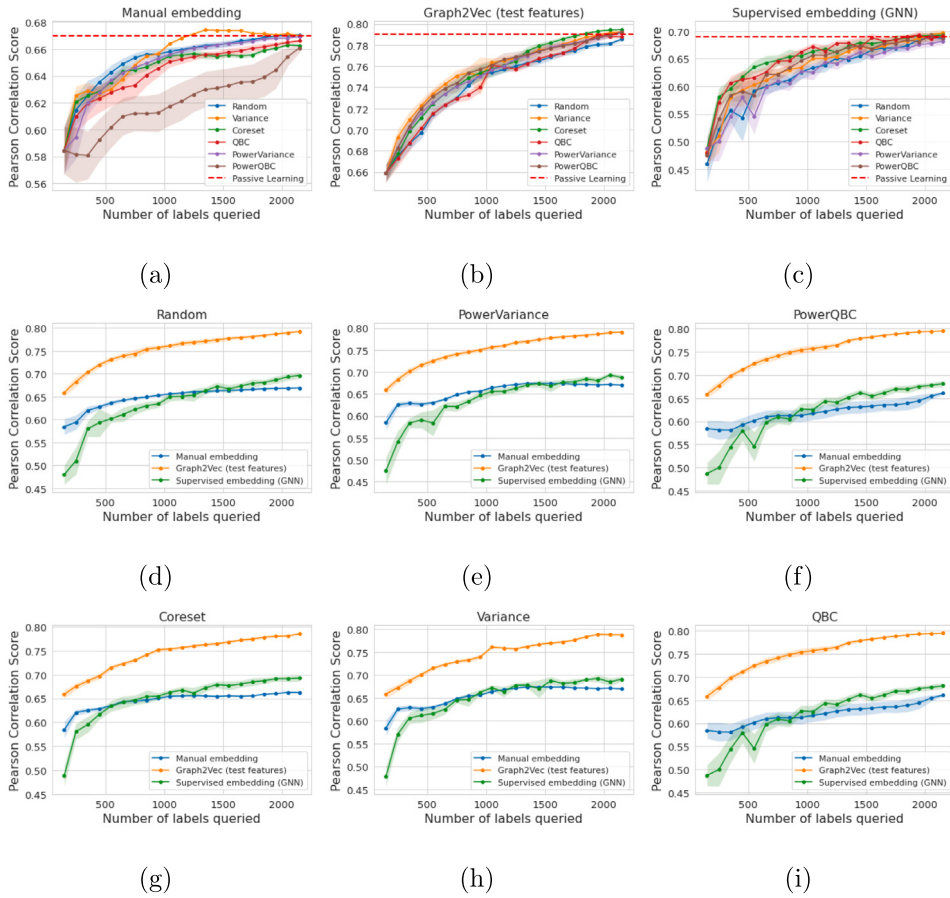


Fig. 12. Active learning results for all embeddings for the HadoopTests dataset (System Level Parsing) with $|\mathcal{L}_0| = 150$ and $|\mathcal{B}| = 100$.

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.

CRediT authorship contribution statement

Peter Samoaa: Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Linus Aronsson:** Formal analysis, Investigation, Methodology, Validation, Visualization, Writing – original draft, Writing – review & editing. **Antonio Longa:** Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Philipp Leitner:** Data curation, Funding acquisition, Supervision. **Morteza Haghir Chehreghani:** Conceptualization, Funding acquisition, Investigation, Methodology, Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data and used cade are available in the following link https://doi.org/10.5281/zenodo.7792485.

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 Haghir 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.

Appendix A. Graph analysis

In this section, we do a deeper investigation of the graph topology of our dataset.

A.1. Basic topology

Fig. A.13 displays node (Fig. A.13(a)) and edge (Fig. A.13(b)) distributions, respectively. The data indicate a minimal disparity between file and system levels in terms of both statistics.

The degree distribution, depicted in Fig. A.14, effectively captures the resemblance between the distributions of nodes and edges.

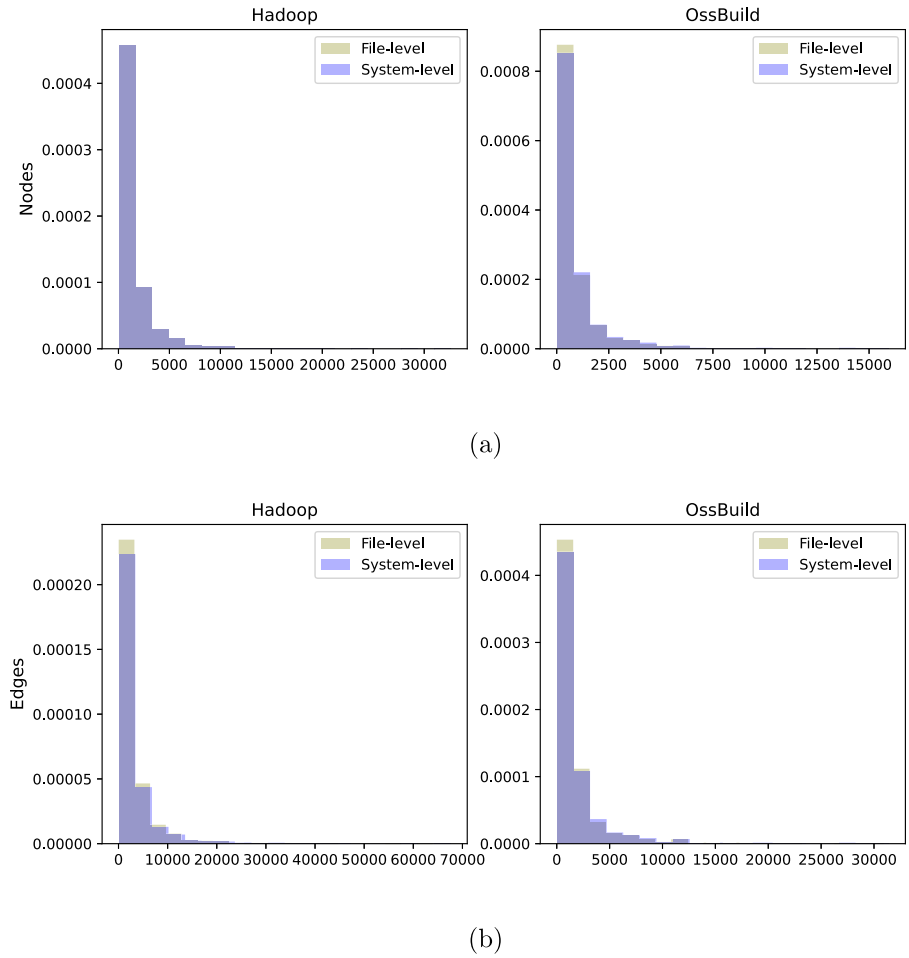


Fig. A.13. Distributions of the number of nodes and edges in Hadoop (left) and OssBuild (right) for both file-level and system-level settings.

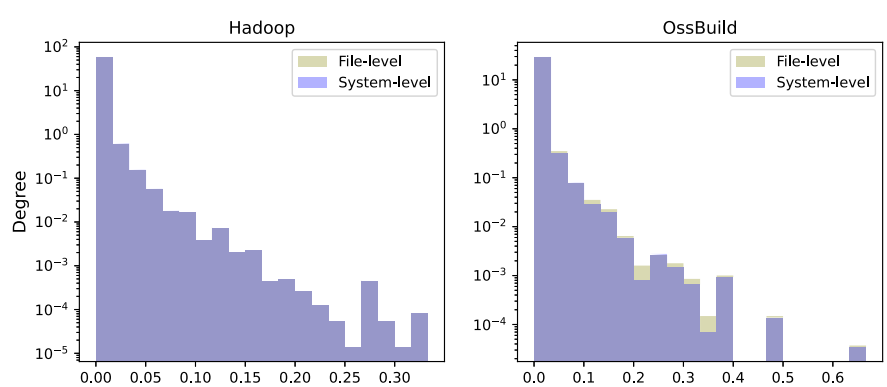


Fig. A.14. Degree distribution in logarithmic scale of Hadoop (left) and OssBuild (right) for both file-level and system-level.

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 the relationship between the graph and triangles through Transitivity (Watts and Strogatz, 1998) and Clustering Coefficient (Newman and Watts, 1999). Transitivity is defined as follows:

$$\text{Transitivity} = 3 \cdot \frac{\#\text{ of triangles}}{\#\text{ of triads}} \tag{A.1}$$

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:

$$C_u = \frac{2 \cdot T(u)}{(deg(u)) \cdot (deg(u) - 1)} \tag{A.2}$$

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

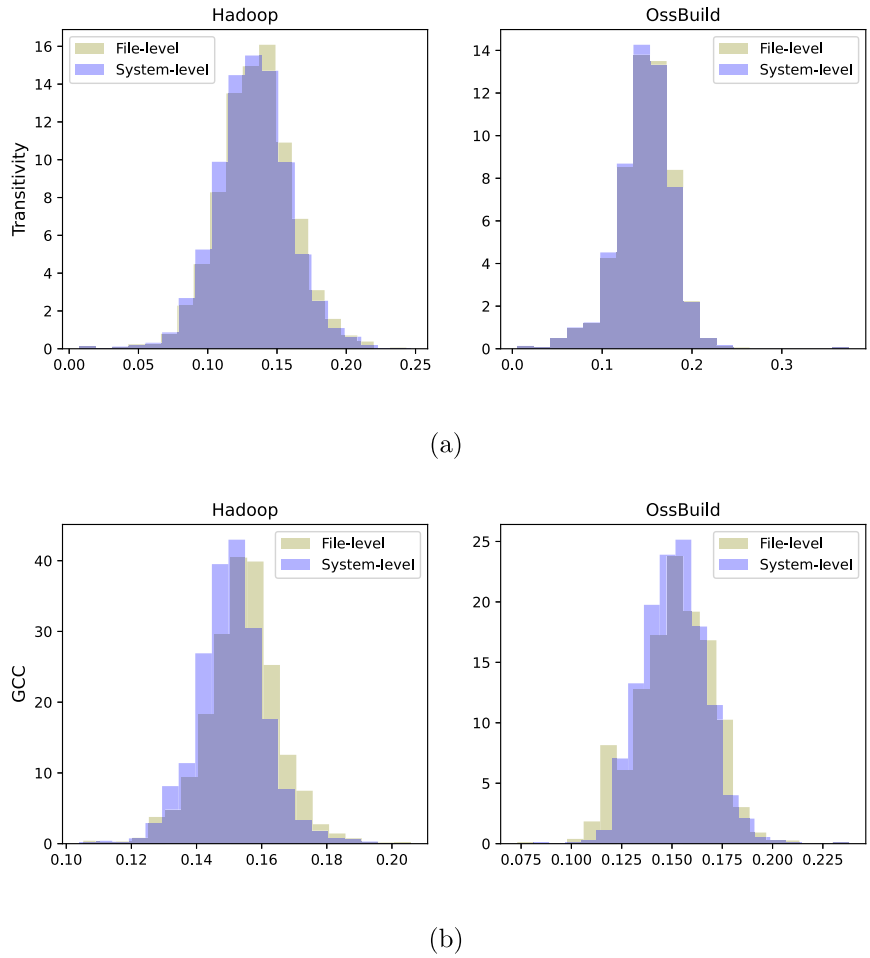


Fig. A.15. Distributions of the transitivity and global clustering coefficient (GCC) in Hadoop (left) and OssBuild (right) for both file-level and system-level settings.

specifically measures the density of connections between neighbouring nodes.

Fig. A.15 shows the transitivity (Fig. A.15(a)) and the global clustering coefficient (Fig. A.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 (Newman, 2002). In Fig. A.16 we report the degree assortativity that examines the correlation of node degrees between connected nodes.

Based on the observations in Fig. A.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.

Various centrality measures exist, where we have already evaluated the degree distributions (in Fig. A.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 (Freeman, 1977)

$$\text{Betweenness Centrality}_u = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)} \tag{A.3}$$

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 (Wasserman and Faust, 1994)

$$\text{Closeness Centrality}_u = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)} \tag{A.4}$$

Where here n is the number of nodes, and $d(v,u)$ is the shortest-path length between node v and node u.

Finally, the Page Rank (Ma et al., 2008) assigns importance to nodes based on the number and quality of incoming links. Nodes with higher Page Rank are considered more influential.

In Fig. A.17 we report the Betweenness, Closeness and Page Rank of the datasets. It is clear that the strongest difference between the file and

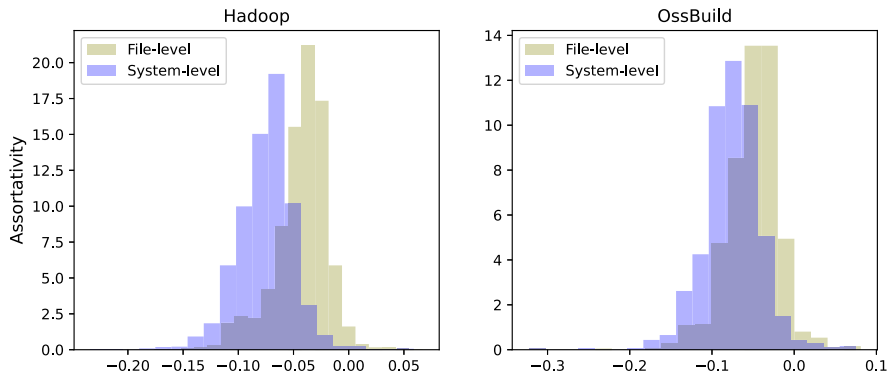
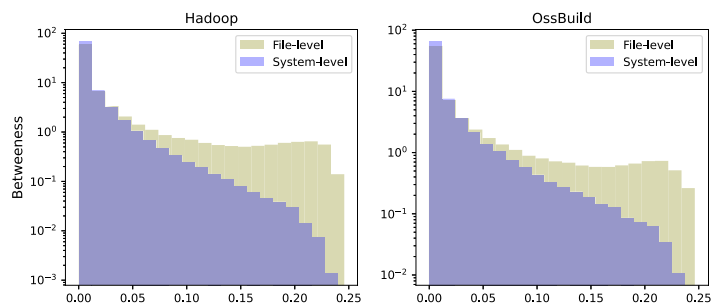
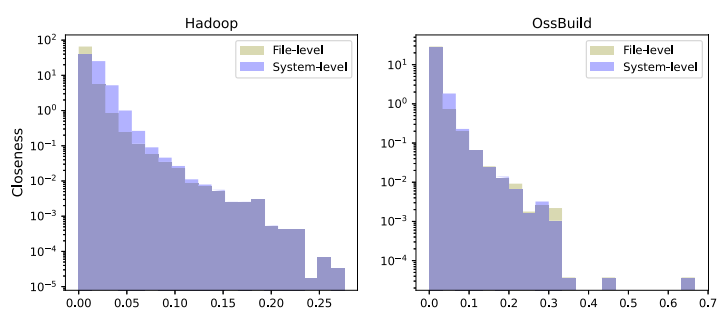


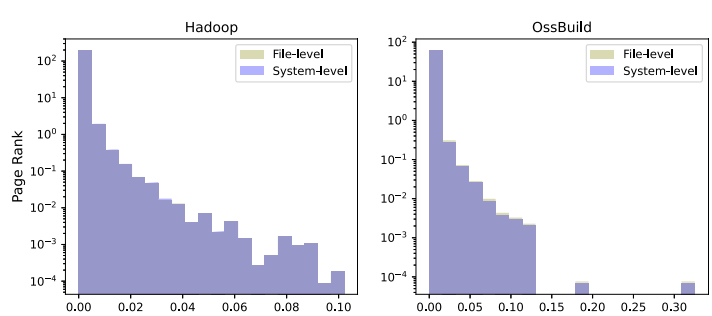
Fig. A.16. Distributions of the degree assortativity in Hadoop (left) and OssBuild (right) for both file-level and system-level.



(a)



(b)



(c)

Fig. A.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.

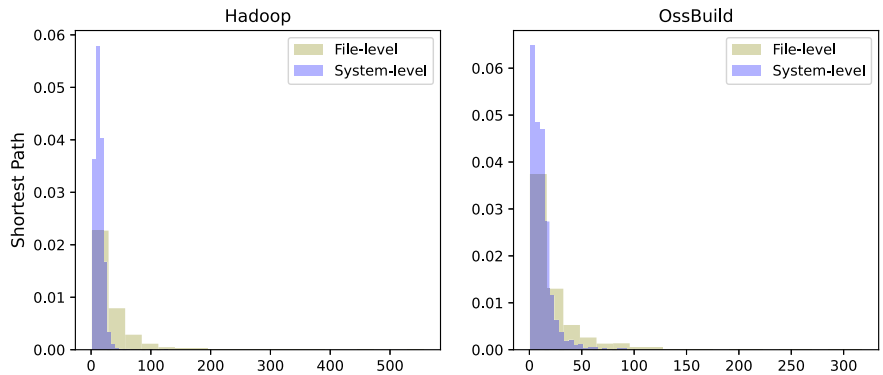


Fig. A.18. Shortest path length distributions.

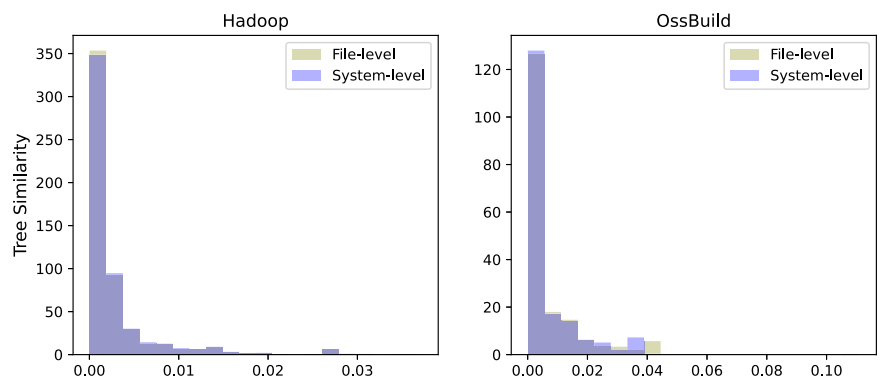


Fig. A.19. *Tree sim* distributions.

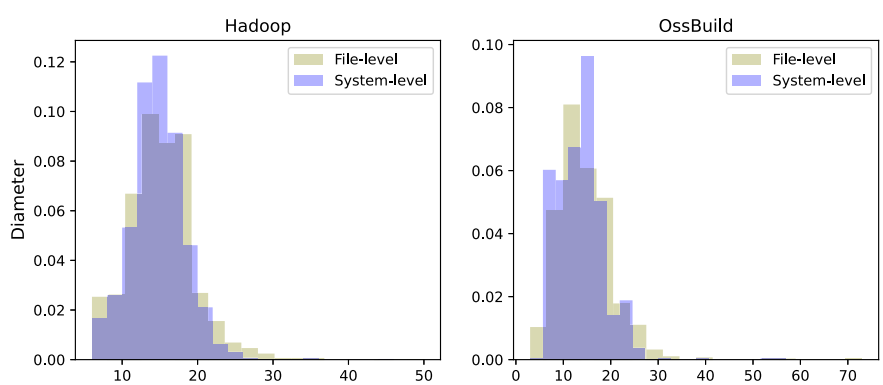


Fig. A.20. Diameter distributions.

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 (Newman, 2010), tree similarity, and the diameter (Newman, 2010) 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 Fig. A.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. (5), 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 Fig. A.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 Fig. A.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.

Table B.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

Appendix 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.

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 B.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 B.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 Fig. 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).

B.1.2. File-level parsing

The results for this setting are reported in Table B.8. In this setting, we still have better results than those obtained with 15 different splits for the dataset.

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.

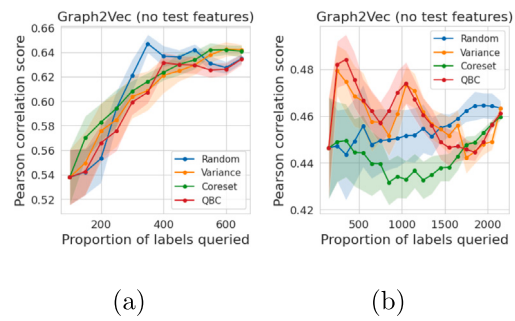


Fig. B.21. Active learning results for Graph2Vec When Test Features are not Used in embeddings for the OSSBuilds (left) and HadoopTest (right) datasets (System Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

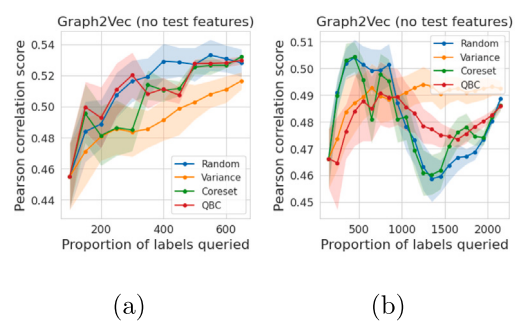


Fig. B.22. Active learning results for Graph2Vec When Test Features are not Used in embeddings for the OSSBuilds (left) and HadoopTest (right) datasets (File Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

B.2.1. System-level parsing

In Fig. B.21, the embedding performance based on Graph2Vec without test features for HadopTests 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.

B.2.2. File-level parsing

In Graph2Vec with no test features in Fig. B.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.

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 (see Figs. B.23–B.26).

Table B.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

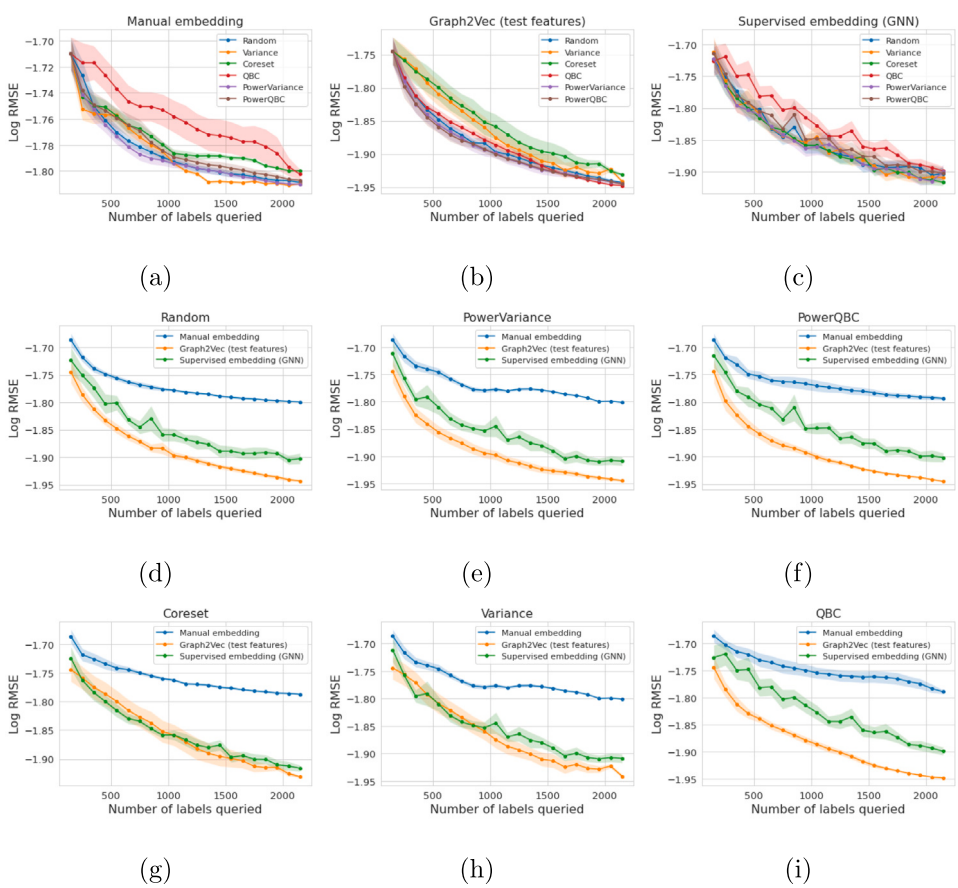


Fig. B.23. Active learning results for all embeddings for the HadoopTests dataset (File Level Parsing) with $|\mathcal{L}_0| = 150$ and $|\mathcal{B}| = 100$.

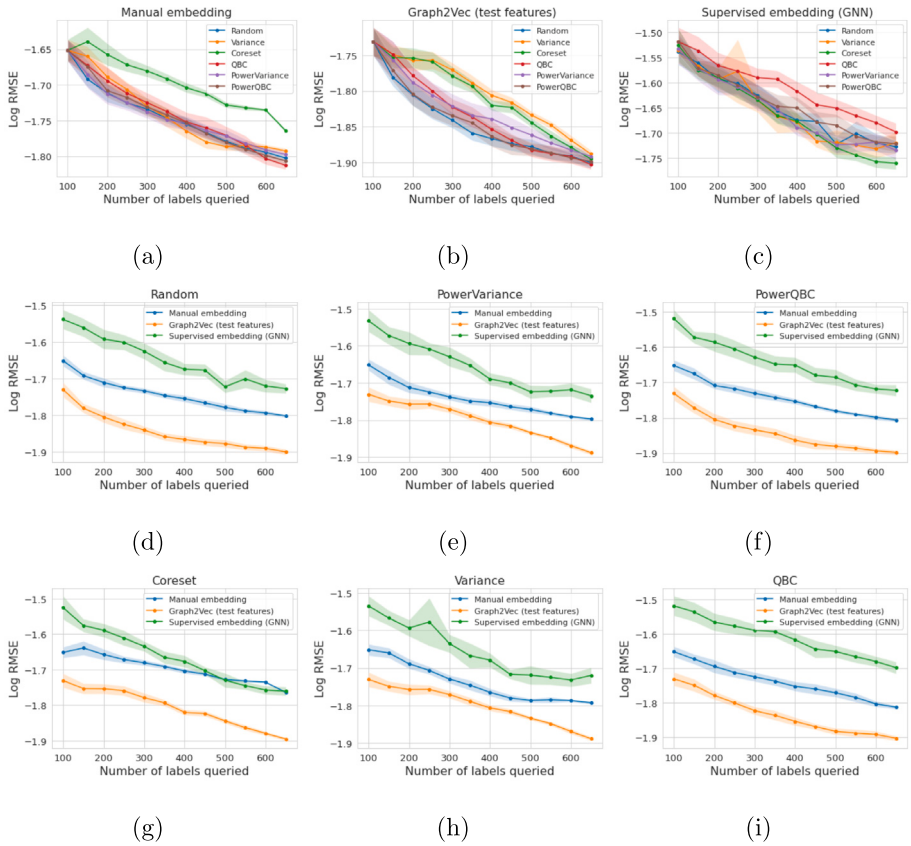


Fig. B.24. Active learning results for all embeddings for the OSSBuilds dataset (File Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

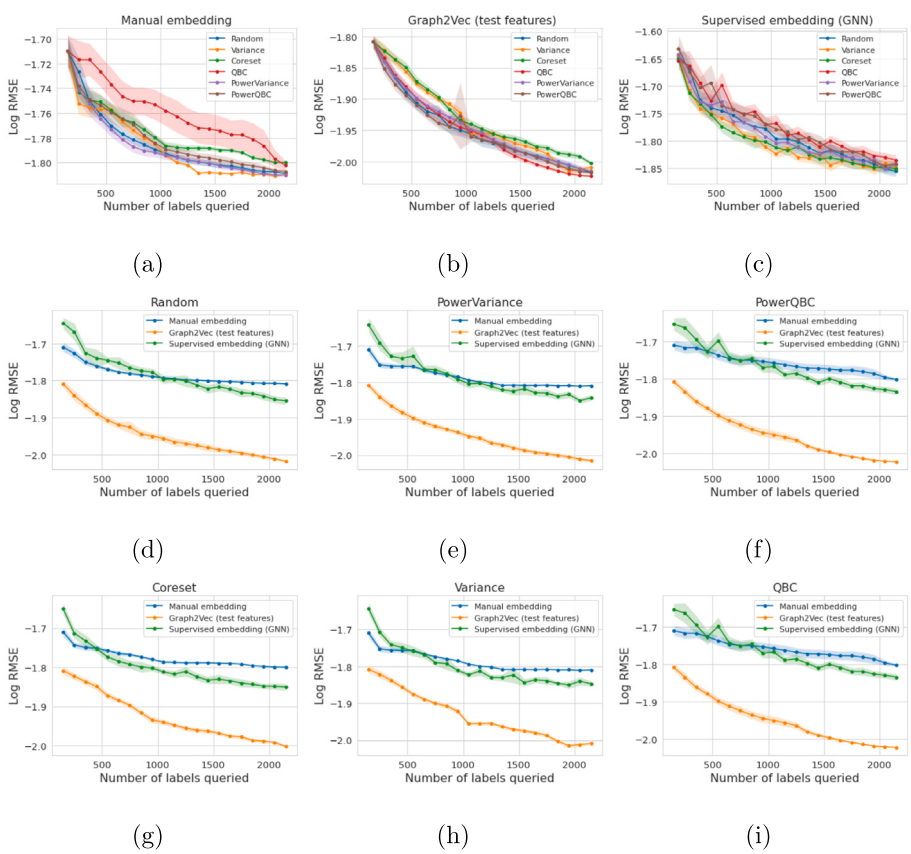


Fig. B.25. Active learning results for all embeddings for the HadoopTests dataset (System Level Parsing) with For the RMSE $|\mathcal{L}_0| = 150$ and $|\mathcal{B}| = 100$.

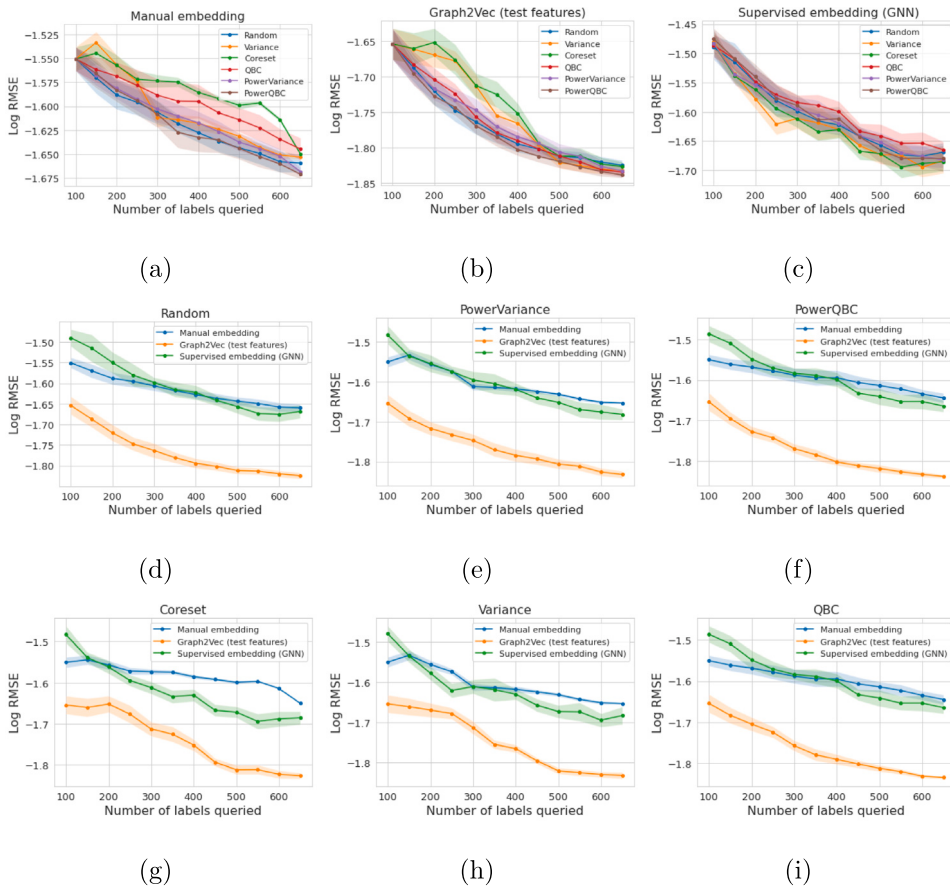


Fig. B.26. Active learning results for all embeddings for the OSSBuilds dataset (System Level Parsing) with $|\mathcal{L}_0| = 100$ and $|\mathcal{B}| = 50$.

References

Abel, R., Louzoun, Y., 2019. Regional based query in graph active learning. arXiv:1906.08541.

Aittokallio, T., Schwikowski, B., 2006. Graph-based methods for analysing networks in cell biology. Brief. Bioinform. 7 (3), 243–255.

Arregui-García, B., Longa, A., Lotito, Q.F., Meloni, S., Cencetti, G., 2024. Patterns in temporal networks with higher-order egocentric structures. Entropy 26 (3), 256.

Batagelj, V., Brandes, U., 2005. Efficient generation of large random networks. Phys. Rev. E 71 (3).

Bossér, J.D., Sörstadius, E., Chehreghani, M.H., 2021. Model-centric and data-centric aspects of active learning for deep neural networks. In: 2021 IEEE International Conference on Big Data (Big Data). pp. 5053–5062. http://dx.doi.org/10.1109/BigData52589.2021.9671795.

Cai, H., Zheng, V.W., Chang, K.C.-C., 2017. Active learning for graph embedding. arXiv:1705.05085.

Cao, S., Lu, W., Xu, Q., 2015. GraRep: Learning graph representations with global structural information. In: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management. CIKM '15, Association for Computing Machinery, New York, NY, USA, pp. 891–900. http://dx.doi.org/10.1145/2806416.2806512.

Cardia, M., Luca, M., Pappalardo, L., 2022. Enhancing crowd flow prediction in various spatial and temporal granularities. In: Companion Proceedings of the Web Conference 2022. pp. 1251–1259.

Casanova, A., Pinheiro, P.O., Rostamzadeh, N., Pal, C.J., 2020. Reinforced active learning for image segmentation. In: International Conference on Learning Representations. URL: https://openreview.net/forum?id=SkgC6TNFvr.

Chami, I., Abu-El-Haija, S., Perozzi, B., Ré, C., Murphy, K., 2022. Machine learning on graphs: A model and comprehensive taxonomy. J. Mach. Learn. Res. 23 (89), 1–64.

Chen, X., Yu, G., Wang, J., Domeniconi, C., Li, Z., Zhang, X., 2019. ActiveHNE: Active heterogeneous network embedding. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence. pp. 2123–2129. http://dx.doi.org/10.24963/ijcai.2019/294.

Comuni, F., Mészáros, C., Åkerblom, N., Haghir Chehreghani, M., 2022. Passive and active learning of driver behavior from electric vehicles. In: 25th IEEE International Conference on Intelligent Transportation Systems. ITSC, pp. 929–936.

Defferrard, M., Bresson, X., Vandergheynst, P., 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In: Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., Garnett, R. (Eds.), In: Advances in Neural Information Processing Systems, vol. 29.

Freeman, L.C., 1977. A set of measures of centrality based on betweenness. Sociometry 40 (1), 35–41, URL: http://www.jstor.org/stable/3033543.

Gal, Y., Islam, R., Ghahramani, Z., 2017. Deep bayesian active learning with image data. In: Precup, D., Teh, Y.W. (Eds.), Proceedings of the 34th International Conference on Machine Learning. In: Proceedings of Machine Learning Research, vol. 70, pp. 1183–1192, URL: https://proceedings.mlr.press/v70/gal17a.html.

Gao, L., Yang, H., Zhou, C., Wu, J., Pan, S., Hu, Y., 2018. Active discriminative network representation learning. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence. pp. 2142–2148. http://dx.doi.org/10.24963/ijcai.2018/296.

Grover, A., Leskovec, J., 2016. Node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '16, New York, NY, USA, pp. 855–864. http://dx.doi.org/10.1145/2939672.2939754.

Guo, Z., Guo, K., Nan, B., Tian, Y., Iyer, R.G., Ma, Y., Wiest, O., Zhang, X., Wang, W., Zhang, C., et al., 2022. Graph-based molecular representation learning. arXiv preprint arXiv:2207.04869.

Hamilton, W., Ying, Z., Leskovec, J., 2017. Inductive representation learning on large graphs. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (Eds.), Advances in Neural Information Processing Systems. 30.

Hu, S., Xiong, Z., Qu, M., Yuan, X., Côté, M.-A., Liu, Z., Tang, J., 2020. Graph policy network for transferable active learning on graphs. Adv. Neural Inf. Process. Syst. 33, 10174–10185.

Huber, W., Carey, V.J., Long, L., Falcon, S., Gentleman, R., 2007. Graphs in molecular biology. BMC Bioinform. 8 (6), 1–14. http://dx.doi.org/10.1186/1471-2105-8-S6-S8.

Jarl, S., Aronsson, L., Rahrovani, S., Chehreghani, M.H., 2022. Active learning of driving scenario trajectories. Eng. Appl. Artif. Intell. 113, 104972. http://dx.doi.org/10.1016/j.engappai.2022.104972.

Kapoor, A., Grauman, K., Urtasun, R., Darrell, T., 2007. Active learning with Gaussian processes for object categorization. In: 2007 IEEE 11th International Conference on Computer Vision. pp. 1–8. http://dx.doi.org/10.1109/ICCV.2007.4408844.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Kipf, T.N., Welling, M., 2016. Variational graph auto-encoders. arXiv preprint arXiv:1611.07308.

Kipf, T.N., Welling, M., 2017. Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations.

Kirsch, A., Farquhar, S., Atighehchian, P., Jesson, A., Branchaud-Charron, F., Gal, Y., 2023. Stochastic batch acquisition: A simple baseline for deep active learning. Trans. Machine Learn. Res. URL: https://openreview.net/forum?id=vcHwQyNBjW. Expert Certification.

Konyushkova, K., Raphael, S., Fua, P., 2017. Learning active learning from data. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. NIPS '17, Curran Associates Inc., Red Hook, NY, USA, pp. 4228–4238.

Lachi, V., Dimitri, G.M., Di Stefano, A., Liò, P., Bianchini, M., Mocenni, C., 2023. Impact of the covid 19 outbreaks on the italian twitter vaccination debat: a network based analysis. arXiv preprint arXiv:2306.02838.

Latora, V., Marchiori, M., 2001. Efficient behavior of small-world networks. Phys. Rev. Lett. 87, 198701. http://dx.doi.org/10.1103/PhysRevLett.87.198701.

Li, Y., Oliva, J., 2021. Active feature acquisition with generative surrogate models. In: Proceedings of the 38th International Conference on Machine Learning, ICML. pp. 6450–6459.

Li, X., Wu, Y., Rakesh, V., Lin, Y., Yang, H., Wang, F., 2022. 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, Association for Computing Machinery, New York, NY, USA, pp. 4199–4203. http://dx.doi.org/10.1145/3511808.3557701.

Liu, Z., Wang, J., Gong, S., Tao, D., Lu, H., 2019. Deep reinforcement active learning for human-in-the-loop person re-identification. In: International Conference on Computer Vision. IEEE, pp. 6121–6130.

Longa, A., Cencetti, G., Lehmann, S., Passerini, A., Lepri, B., 2024. Generating fine-grained surrogate temporal networks. Commun. Phys. 7 (1), 22.

Longa, A., Cencetti, G., Lepri, B., Passerini, A., 2022. An efficient procedure for mining egocentric temporal motifs. Data Min. Knowl. Discov. 1–24.

Luo, Y., Yan, K., Ji, S., 2021. Graphdf: A discrete flow model for molecular graph generation. In: International Conference on Machine Learning. PMLR, pp. 7192–7203.

Ma, N., Guan, J., Zhao, Y., 2008. Bringing PageRank to the citation analysis. Inf. Process. Manage. 44 (2), 800–810.

Mauro, G., Luca, M., Longa, A., Lepri, B., Pappalardo, L., 2022. Generating mobility networks with generative adversarial networks. EPJ Data Sci. 11 (1), 58.

Munjal, P., Hayat, N., Hayat, M., Sourati, J., Khan, S., 2020. Towards robust and reproducible active learning using neural networks. In: Conference on Computer Vision and Pattern Recognition. pp. 223–232.

Newman, M., 2002. Assortative mixing in networks. Phys. Rev. Lett. 89 (20), http://dx.doi.org/10.1103/physrevlett.89.208701.

Newman, M.E.J., 2010. Networks: an introduction. Oxford University Press, Oxford; New York.

Newman, M.E., Watts, D.J., 1999. Renormalization group analysis of the small-world network model. Phys. Lett. A 263 (4–6), 341–346.

Nguyen, A., Longa, A., Luca, M., Kaul, J., Lopez, G., 2022. Emotion analysis using multilayered networks for graphical representation of tweets. IEEE Access 10, 99467–99478.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. Scikit-learn: Machine learning in python. J. Mach. Learn. Res. 12, 2825–2830.

Perozzi, B., Al-Rfou, R., Skiena, S., 2014. DeepWalk: Online learning of social representations. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '14, Association for Computing Machinery, New York, NY, USA, pp. 701–710. http://dx.doi.org/10.1145/2623330.2623732.

Rasmussen, C.E., Williams, C.K.I., 2005. Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning). The MIT Press.

Ren, P., Xiao, Y., Chang, X., Huang, P.-Y., Li, Z., Gupta, B.B., Chen, X., Wang, X., 2021. A Survey of Deep Active Learning. vol. 54, (no. 9), Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3472291.

Rubens, N., Elahi, M., Sugiyama, M., Kaplan, D., 2015. Active learning in recommender systems. In: Recommender Systems Handbook. springer, pp. 809–846. http://dx.doi.org/10.1007/978-1-4899-7637-6_24.

Samoaa, P., Aronsson, L., Longa, A., Leitner, P., Chehreghani, M.H., 2023. A Unified Active Learning Framework for Annotating Graph Data with Application to Software Source Code Performance Prediction. Zenodo, http://dx.doi.org/10.5281/zenodo.7792485.

Samoaa, H.P., Bayram, F., Salza, P., Leitner, P., 2022a. A systematic mapping study of source code representation for deep learning in software engineering. IET Softw. 16 (4), 351–385. http://dx.doi.org/10.1049/sfw2.12064.

Samoaa, H.P., Longa, A., Mohamad, M., Chehreghani, M.H., Leitner, P., 2022b. TEP-gnn: Accurate execution time prediction of functional tests using graph neural networks. In: Taibi, D., Kuhrmann, M., Mikkonen, T., Klünder, J., Abrahamsson, P. (Eds.), Product-Focused Software Process Improvement. Springer International Publishing, Cham, pp. 464–479.

Scott, J., 2011. Social network analysis: developments, advances, and prospects. Social network analysis and mining 1, 21–26.

Sener, O., Savarese, S., 2018. Active learning for convolutional neural networks: A core-set approach. arXiv:1708.00489.

Settles, B., 2009. Active Learning Literature Survey. Computer Sciences Technical Report, 1648, University of Wisconsin–Madison.

Seung, H., Opper, M., Sompolinsky, H., 1992. Query by committee. In: Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory. In: Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory, Publ by ACM, pp. 287–294. http://dx.doi.org/10.1145/130385.130417, Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory ; Conference date: 27-07-1992 Through 29-07-1992.

Shen, Y., Yun, H., Lipton, Z.C., Kronrod, Y., Anandkumar, A., 2018. Deep active learning for named entity recognition. arXiv:1707.05928.

Shin, J., Wu, S., Wang, F., De Sa, C., Zhang, C., Ré, C., 2015. Incremental knowledge base construction using deepdive. In: Proceedings of the VLDB Endowment International Conference on Very Large Data Bases. 8, (11), NIH Public Access, p. 1310.

Shuyang, Z., Heittola, T., Virtanen, T., 2020. Active learning for sound event detection. IEEE/ACM Trans. Audio Speech Lang. Proc. 28, 2895–2905. http://dx.doi.org/10.1109/TASLP.2020.3029652.

Viet Johansson, S., Gummesson Svensson, H., Bjerrum, E., Schliep, A., Haghir Chehreghani, M., Tyrchan, C., Engkvist, O., 2022. Using active learning to develop machine learning models for reaction yield prediction. Mol. Inform. 41 (12), http://dx.doi.org/10.1002/minf.202200043.

Vu, T.-T., Liu, M., Phung, D., Haffari, G., 2019. Learning how to active learn by dreaming. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, pp. 4091–4101.

Wang, D., Cui, P., Zhu, W., 2016. Structural deep network embedding. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '16, Association for Computing Machinery, New York, NY, USA, pp. 1225–1234. http://dx.doi.org/10.1145/2939672.2939753.

Wang, Z., Liu, M., Luo, Y., Xu, Z., Xie, Y., Wang, L., Cai, L., Qi, Q., Yuan, Z., Yang, T., et al., 2022. Advanced graph and sequence neural networks for molecular property prediction and drug discovery. Bioinformatics 38 (9), 2579–2586.

Wasserman, S., Faust, K., 1994. Social network analysis: Methods and applications. Cambridge University Press.

Watts, D.J., Strogatz, S.H., 1998. Collective dynamics of 'small-world'networks. Nature 393 (6684), 440–442.

Wu, Y., Xu, Y., Singh, A., Dubrawski, A., Yang, Y., 2020. Active learning graph neural networks via node feature propagation.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. pp. 590–604. http://dx.doi.org/10.1109/SP.2014.44.

Yan, S., Chaudhuri, K., Javidi, T., 2018. Active learning with logged data. In: Dy, J.G., Krause, A. (Eds.), Proceedings of the 35th International Conference on Machine Learning. In: Proceedings of Machine Learning Research, 80, pp. 5517–5526.

Yang, S., Hou, M., 2023. Knowledge graph representation method for semantic 3D modeling of Chinese grottoes. Herit. Sci. 11 (1), 266.

Zhang, Y., Tong, H., Xia, Y., Zhu, Y., Chi, Y., Ying, L., 2022. Batch active learning with graph neural networks via multi-agent deep reinforcement learning. Proc. AAAI Conf. Artif. Intell. 36 (8), 9118–9126. http://dx.doi.org/10.1609/aaai.v36i8.20897.