Batch Mode Deep Active Learning for Regression on Graph Data

Peter Samoaa
Data Science and AI Division
Chalmers University of technology
Gothenburg, Sweden
samoaa@chalmers.se

Linus Aronsson
Data Science and AI Division
Chalmers University of technology
Gothenburg, Sweden
linaro@chalmers.se

Philipp Leitner
Interaction Design and Software Engineering
Chalmers University of technology
Gothenburg, Sweden
philipp.leitner@chalmers.se

Morteza Haghir Chehreghani
Data Science and AI Division
Chalmers University of technology
Gothenburg, Sweden
morteza.chehreghani@chalmers.se

***Abstract*—Acquiring labelled data for machine learning tasks, for example, for software performance prediction, remains a resource-intensive task. This study extends our previous work by introducing a batch-mode deep active learning approach tailored for regression in graph-structured data. Our framework leverages the source code conversion into Flow Augmented-AST graphs (FA-AST), subsequently utilizing both supervised and unsupervised graph embeddings. In contrast to single-instance querying, the batch-mode paradigm adaptively selects clusters of unlabeled data for labelling. We deploy an array of base kernels, kernel transformations, and selection methods, informed by both Bayesian and non-Bayesian strategies, to enhance the sample efficiency of neural network regression. Our experimental evaluation, conducted on multiple real-world software performance datasets, demonstrates the efficacy of the batch mode deep active learning approach in achieving robust performance with a reduced labelling budget. The methodology scales effectively to larger datasets and requires minimal alterations to existing neural network architectures.**

***Index Terms*—Active Learning, Graph Neural Network, Deep Learning, Kernels.**

I. Introduction

The rapid growth of machine learning (ML) applications across numerous domains is stifled by the limited availability of labelled data, including the domain of software engineering. Despite the abundance of source code files publicly hosted on platforms like GitHub, the absence of labels for these datasets remains a significant bottleneck. For tasks like performance prediction—which aims to forecast the execution time of software prior to execution—the cost of labelling is both computationally expensive and time-consuming. This conundrum gives rise to the need for Active Learning (AL) [26] techniques that efficiently identify the most informative samples for labelling. Active learning has been extensively investigated in various domains like text analysis [27], image data [5], [9], driving scenario trajectories [15], and drug design [28] to improve data annotation procedures. A particular challenge in deploying active learning for source code analysis arises from the representation of source code as graphs, coupled with the lack of a unified framework suitable for diverse learning tasks, such as regression.

Numerous studies have explored the use of active learning in graph-based models, particularly focusing on node classification tasks via Graph Neural Networks (GNNs) [2], [6], [17], [30]. These works primarily address active learning at the node level. Some research extends this by incorporating reinforcement learning into the active learning framework. For instance, Hu et al. [13] train a policy network on labeled source graphs and transferred this policy to unlabeled graphs for node labeling tasks. Zhang et al. [31], [32] examine batch settings in active learning, employing multi-agent reinforcement learning and meta Q-learning to facilitate node labeling for classification purposes. Additionally, multi-armed bandit approaches have also been used for active learning in graph settings [7], [10]. Despite these advances, the existing literature largely concentrates on node-level classification tasks. The application of active learning to graph-level regression tasks remains not widely explored.

To mitigate this challenge, our recent work [22] proposes a unified active learning framework tailored for graph representations of source code. Our framework employs enhanced Abstract Syntax Trees (ASTs), which we term FA-ASTs [24]. These FA-ASTs capture a rich tapestry of syntactical, semantic, and lexical source code information and serve as the data points for our active learning model. Despite the versatility in accommodating various regression techniques, our existing framework in [22] falls short in supporting diverse sample selection across batches. This limitation is critical [16] and becomes especially acute given the computational demands of retraining models—particularly neural networks—after each labelling iteration. Batch Mode Active Learning (BMAL) offers a solution by allowing the selection of multiple data points for labelling simultaneously. It is then called atch Mode Deep Active Learning (BMDAL) when the BMAL approach is employed with deep learning models for extracting expressive features [21]. Specifically, we consider pool-based

BMDAL, where the data points for labelling are chosen from a predefined pool.

Inspired by recent work [11] that employs BMDAL for regression on tabular data, we aim to extend this framework to accommodate graph-based source code data. In particular, to apply BMDAL to graph data, we investigate GNNs and Graph2Vec for graph learning and fully connected neural networks for the regression task (i.e., for performance prediction). Regression tasks inherently lack a natural measure of uncertainty, which is often straightforward in classification tasks through softmax layers. Computing uncertainties in regression, therefore, becomes less straightforward necessitating the use of kernel methods. Therefore, we admit a Gaussian Process (GP) framework [20] in order to investigate and utilize different notions of uncertainty. We conduct our experiments on a real-world dataset that we have collected for this study. Our experimental results indicate that utilising GNN within the BMDAL framework provides the most effective setting for active learning querying methods compared to Graph2Vec.

In summary, our contributions are threefold:

1) We extend the BMDAL framework to make it compatible with graph data.
2) To the best of our knowledge, we are the first to adapt BMDAL for graph representations of source code specifically for regression tasks.
3) We validate our approach using real-world datasets.
4) By addressing these gaps, we offer a novel approach to the problem of active learning in source code analysis, thereby contributing to more efficient labelling and, ultimately, broader application of machine learning in software engineering.

The code and data are publicly available at [1].

II. Source Code Representation

Listing 1: Simple example of Java source code

```
public static int factorial(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

This study aims to bring the power of ML to software engineering by enhancing performance prediction models. For that, understanding how source code can be effectively represented is crucial. As we detailed in our previous systematic literature review [23], program source code can be converted into various forms, ranging from tree-based and graph-based to token-based representations.

In this paper, we use a Java method calculating the factorial of a number as a concrete example for source code representation, specifically focusing on the Abstract Syntax Tree (AST), Data Flow Graph (DFG), and Control Flow Graph (CFG). These different representations serve unique purposes and offer different types of information about the code.

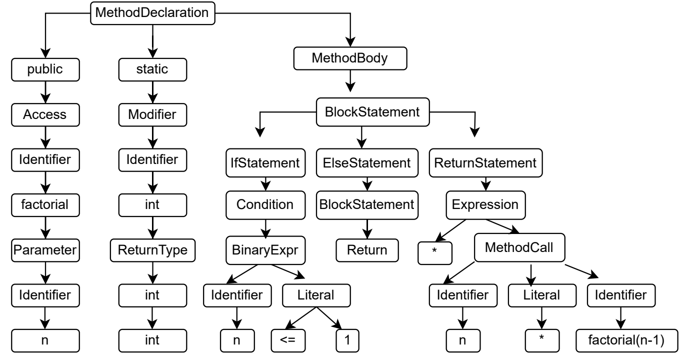
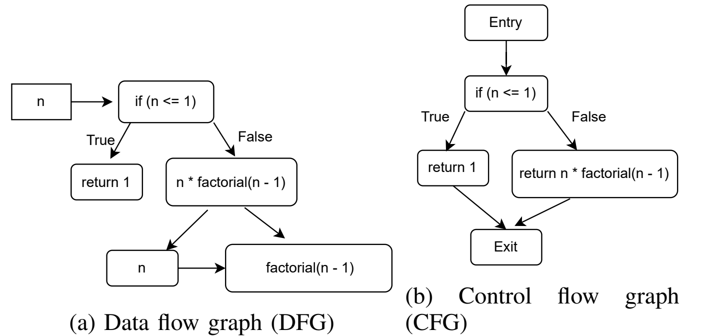


Fig. 1: Simplified abstract syntax tree (AST) for the code snippet in Listing 1



(a) Data flow graph (DFG)

(b) Control flow graph (CFG)

Fig. 2: Flow graphs representation for the code snippet in Listing 1

A. Abstract Syntax Tree (AST)

The AST representation is of particular interest due to the rich syntactical and lexical details it offers without the need for executing the code. An AST for our Java method is illustrated in Figure 1, where the tree structure provides an overview of the program's syntactic composition, including decision-making constructs like 'if' statements and expressions involving function calls and arithmetic operations. The AST is particularly beneficial for capturing the structural aspects of the code, which makes it well-suited for graph neural networks requiring many nodes and edges for meaningful feature extraction.

B. Data Flow Graph (DFG)

While the AST gives us valuable insights into the syntactic structure of the code, it does not capture how data moves or interacts within the program. This is where Data Flow Graphs (DFG) come into play. As demonstrated in Figure 2a, a DFG shows the flow of data between variables and computations, capturing the dependencies between different parts of the code.

C. Control Flow Graph (CFG)

To understand the runtime behaviour and possible paths that can be traversed during the code execution, Control Flow Graphs (CFG) are indispensable. Our Java method's CFG, shown in Figure 2b, presents a high-level overview of all

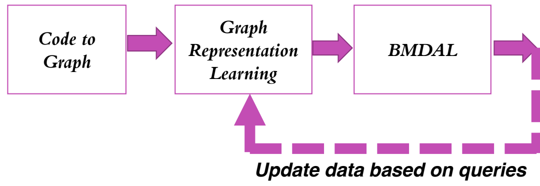


Fig. 3: BMDAL framework for graph data

possible routes the execution could take, from the initial method call to the return statements.

In summary, the combination of these source code representations enables us to comprehensively analyze and model the behaviour, structure, and data flow within a software system, which is particularly useful for ML-driven software engineering research.

III. Active Learning Approach

In this section, we outline the key components of our active learning framework. The process begins by converting source code into graph representations. These graphs are then embedded through unsupervised techniques or supervised. For supervised embeddings, we employ GNN in conjunction with active learning. This involves iteratively training the GNN based on newly added batches from the active learning process. During the active learning phase, we explore various selection methods, as well as kernels and their associated transformations.

A. Source Code to Graph

This section describes the methodology for constructing graphs from the source code, specifically Java files, as illustrated in Figure 4.

1) AST Parsing: We initially transform the source code into an AST as an intermediate representation. The AST representation can be extracted through source code parsing alone, without the need for executing the program. We use the pure Python Java parser javalang[1] to parse each test file and use the node types, values, and production rules in javalang to describe our ASTs. To encapsulate both semantic elements and syntactical attributes, we enhance the AST by integrating edges that capture data and control flow. This results in a Flow-Augmented AST (FA-AST) graph, a concept that was pioneered in our prior research [24].

The impetus for enriching the AST originates from contemporary research [23], underscoring the necessity for comprehensive code representations in applying deep learning techniques to software engineering. Given the intricate nature of performance prediction tasks, relying solely on the syntactic information derived from basic AST falls short of delivering high-fidelity outcomes. Therefore, we augment the tree-like architecture of the AST with additional semantic layers that signify both data and control flow, evolving it into a more elaborate graph. This enriched graph representation encodes a broader set of information than what is offered by the source code structure alone.

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

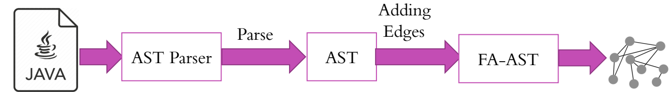


Fig. 4: Source Code to Graph Process

2) Capturing Ordering and Data Flow: To understand how the graphs are built, we will present each augmentation and then explain in detail how the FA-AST is built. We augment AST with different types of additional edges representing data flow and node order in the AST. Specifically, we use the following additional flow augmentation edges, in addition to the **AST child** and **AST parent** edges that are produced readily by AST parsing:

FA Next Token (b):
This type of edge connects a terminal node (leaf) in the AST to the next terminal node. Terminal nodes are nodes without children. In Figure 1, an FA Next Token edge would be added, for example, between `n` and `int`(the first leaves at the left bottom).

FA Next Sibling (c):
This connects each node (both terminal and non-terminal) to its next sibling and allows us to model the order of instructions in an otherwise unordered graph. In Figure 1, such an edge would be added, for example, connecting the `public` and with the `static` and `static` with `MethodBody` node.

FA Next Use (d):
This type of edge connects a node representing a variable to the place where this variable is next used. For example, the variable `n` is declared in the first line in Listing 1, and then used next in Lines 2 and 5.

3) Capturing Control Flow: In a second augmentation step, we now add further edges representing the control flow in the test cases. We currently support *if* statements, *while* and *for* loops, as well as *sequential execution*. We currently do not support *switch* statements or *do-while* loops, as these are less common. Java source code containing these elements will still be parsed successfully, but the FA-AST will not capture these control flow constructs. Specifically, the following further edges are added (see also Figure 5):

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

FA Else Flow (f):
Conversely, this edge type connects the predicate to the (optional) else code block.

FA While Flow (g):
A while loop essentially entails two elements - a condition and a code block that is executed as long as the condition remains `true`. We capture this through a FA While Flow (g) edge connecting the condition to the code block, and an FA

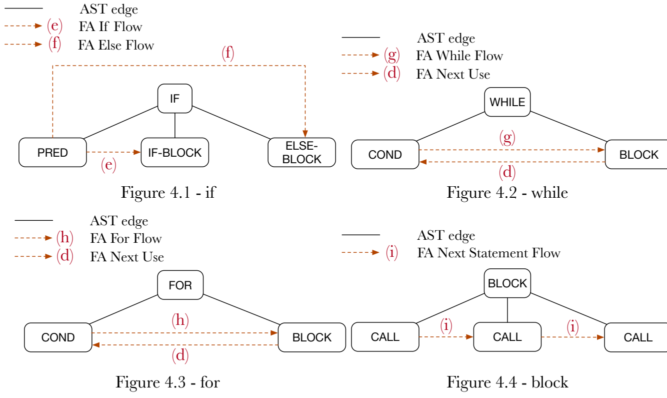


Fig. 5: Additional flow augmentations for different control flow constructs

Next Use (d) edge in the reverse direction. The latter is used to model the next usage of a loop counter.

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

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

B. Graph Representation Learning

The graph structure of the data items in $\mathcal{G}$ restricts the types of regression models that can be used, and thus, the types of query strategies to be employed for active learning. Therefore, we construct embeddings that can be used to project the graph data into a latent space where any regression model (and thus query strategy) can be utilized.

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

1) Unsupervised Embedding: In our previous study [22], we investigate a number of shallow graph embeddings based on matrix factorization or skip-gram-based embeddings. The obtained results (Table 3 and Table 5 in [22]) show that Graph2Vec [18] achieves the best results for graph-level embedding across all unsupervised graph embeddings. Thus, in this paper, we use Graph2Vec as the unsupervised embedding.

Algorithm 1 Pool-based BMDAL loop in *unsupervised setting*

Require: Graphs $\mathcal{G}$, BMDAL algorithm NEXTBATCH (see Algorithm 3), list $\mathcal{L}_{\text{batch}}$ of batch sizes
1: $\mathcal{X}$ = Graph2Vec($\mathcal{G}$)
2: Split $\mathcal{X}$ into $\mathcal{X}_{train}$, $\mathcal{X}_{\text{pool}}$, $\mathcal{X}_{\text{test}}$
3: **for** AL batch size $\mathcal{N}_{\text{batch}}$ in $\mathcal{L}_{\text{batch}}$ **do**
4: Train NN model f_θ on $\mathcal{X}_{\text{train}}$
5: Evaluate NN model f_θ on $\mathcal{X}_{\text{test}}$
6: $\mathcal{X}_{\text{batch}} \leftarrow \text{NEXTBATCH}(f_\theta, \mathcal{X}_{\text{train}}, \mathcal{X}_{\text{pool}}, \mathcal{N}_{\text{batch}})$
7: Move $\mathcal{X}_{\text{batch}}$ from $\mathcal{X}_{\text{pool}}$ to $\mathcal{X}_{\text{train}}$ and acquire labels $\mathcal{Y}_{\text{batch}}$ for $\mathcal{X}_{\text{batch}}$
8: **end for**
9: Train final model f_θ on $\mathcal{X}_{\text{train}}$
10: Evaluate final model f_θ on $\mathcal{X}_{\text{test}}$

2) Supervised Embedding: Similar to unsupervised settings we select the most accurate GNN model out of the state-of-the-art architectures (namely GCNConv, GraphSAGE, and GraphConv) that were used in our previous study. Thus, in our experiments, we use GraphConv [8] since it yields the most accurate results for our graph data.

C. Batch Mode Deep Active Learning

In this section, we will discuss how we use different selection methods and how kernels and kernel transformations are incorporated with the selection methods and neural networks. When constructing query strategies for the BMDAL framework, The following three criteria are generally considered for selecting batches [29]:

- *Informativeness*: The selection method should select samples where the model is mostly uncertain about the label.
- *Diversity*: The selection methods must ensure that the samples in the batch must be diverse and different from each other.
- *Representative*: The selection of the training set should be concentrated on the region where the pool data distribution has high density.

Algorithm 1 illustrates the general procedure for pool-based BMDAL utilizing unsupervised graph embedding. Initially, we derive the embeddings for the complete graph set $\mathcal{G}$ using Graph2Vec. The embeddings $\mathcal{X}$ are then partitioned into training $\mathcal{X}_{\text{train}}$, testing $\mathcal{X}_{\text{test}}$, and pooling $\mathcal{X}_{\text{pool}}$ subsets. Note that the test lables are never used in training or querying for active learning. Within the BMDAL loop, the neural network (NN) model is first trained on the initially labelled embeddings $\mathcal{X}_{\text{train}}$ and subsequently evaluated on $\mathcal{X}_{\text{test}}$. Next, a batch $\mathcal{X}_{\text{batch}} \subset \mathcal{X}_{\text{pool}}$ is selected using the *NEXTBATCH* method, which forms the core of BMDAL. The labeled set is updated by transferring the selected batch $\mathcal{X}_{\text{batch}}$ from $\mathcal{X}_{\text{pool}}$ to $\mathcal{X}_{\text{train}}$ and acquiring the labels $\mathcal{Y}_{\text{batch}}$ for it. The NN model is then retrained on the extended $\mathcal{X}_{\text{train}}$ and re-evaluated on $\mathcal{X}_{\text{test}}$. Finally, the model is trained on the complete $\mathcal{X}_{\text{train}}$ and evaluated on $\mathcal{X}_{\text{test}}$.

Algorithm 2 illustrates the general steps for pool-based BMDAL in supervised setting. The process is slightly different

Algorithm 2 Pool-based BMDAL loop in *supervised setting*

Require: Graph Data $\mathcal{G}$, initial labeled graphs training set $\mathcal{G}_{\text{train}}$, unlabeled graphs pool set $\mathcal{G}_{\text{pool}}$, test set $\mathcal{G}_{test}$, BMDAL algorithm NEXTBATCH (see Algorithm 3), list $\mathcal{L}_{\text{batch}}$ of batch sizes
1: **for** AL batch size $\mathcal{N}_{\text{batch}}$ in $\mathcal{L}_{\text{batch}}$ **do**
2: GNN = GraphConv ($\mathcal{G}_{train}$) {training the GNN model}
3: $\mathcal{X}$ = GNN.embedding($\mathcal{G}$)
4: Extract $\mathcal{X}_{train}$,$\mathcal{X}_{test}$,$\mathcal{X}_{pool}$ from the embedding set $\mathcal{X}$ based on the indices of $\mathcal{G}_{\text{train}}$, $\mathcal{G}_{\text{test}}$, $\mathcal{G}_{\text{pool}}$,
5: Train NN model f_θ on $\mathcal{X}_{\text{train}}$
6: Evaluate NN model f_θ on $\mathcal{X}_{\text{test}}$
7: $\mathcal{X}_{\text{batch}} \leftarrow \text{NEXTBATCH}(f_\theta, \mathcal{X}_{\text{train}}, \mathcal{X}_{\text{pool}}, \mathcal{N}_{\text{batch}})$
8: Move $\mathcal{X}_{\text{batch}}$ from $\mathcal{X}_{\text{pool}}$ to $\mathcal{X}_{\text{train}}$ and acquire labels $\mathcal{Y}_{\text{batch}}$ for $\mathcal{X}_{\text{batch}}$
9: Update $\mathcal{G}_{train}$
10: **end for**
11: Train final model f_θ on $\mathcal{X}_{\text{train}}$
12: Evaluate final model f_θ on $\mathcal{X}_{\text{test}}$

since the GNN model is incorporated into the active learning process because $\mathcal{X}_{train}$ is updated in each iteration in order to utilize the recently labelled data. Here, we first define the indices of training, test, and pool sets in advance. Then, in the active learning loop, we initially train the GNN model in order to obtain an initial embedding. Then, based on this embedding, we train the NN model and evaluate it on $\mathcal{X}_{test}$. Then, we select $\mathcal{X}_{batch}$ by *NEXTBATCH*. Next, we update the labelled set by moving the selected batch $\mathcal{X}_{batch}$ from $\mathcal{X}_{pool}$ to $\mathcal{X}_{train}$ and acquire the labels $\mathcal{Y}_{batch}$ for $\mathcal{X}_{batch}$. Thus, $\mathcal{X}_{train}$ is then extended, and we train the GNN again based on the extended training graph set to obtain a new embedding. The NN is then trained again on the extended embeddings set and so on. At the end of iteration, we train the final model on the full $\mathcal{X}_{train}$ and evaluate it on $\mathcal{X}_{test}$.

Algorithm 3 Kernel-based batch construction framework

1: **function** $\text{NEXTBATCH}(f_\theta, \mathcal{X}_{\text{train}}, X_{\text{pool}}, N_{\text{batch}})$
2: $k \leftarrow \text{BaseKernel}(f_\theta)$
3: $k \leftarrow \text{TransformKernel}(k, \mathcal{X}_{\text{train}})$
4: **return** $\text{SELECT}(k, \mathcal{X}_{\text{train}}, X_{\text{pool}}, N_{\text{batch}})$
5: **end function**

Kernels and Kernel Transformation in BMDAL: The usage of kernels and related transformations is inspired by the study in [11]. The authors formulate the use of kernels and kernel transformations within a general framework for BMDAL for tabular regression data.

The kernel-based batch construction framework outlined in Algorithm 3 serves as a fundamental component in Algorithms 1 and 2. This framework enables the manipulation of kernels and kernel transformations, fulfilling key functionalities.

A primary motivation for employing kernels in this framework is to emphasize *informativeness* as a crucial criterion for assessing the efficiency of selection methods. This is particularly vital for tasks involving uncertainty quantification. Whereas softmax layers commonly serve to measure uncertainty in classification, such methods are not directly applicable to regression tasks. In regression that yields scalar outputs—such as execution time in our case study—a straightforward uncertainty quantification mechanism is absent. This gap is bridged by using Gaussian Process (GP), a Bayesian technique that computes uncertainties via kernel methods.

In Gaussian Process, the selected kernel plays a critical role in determining the quality of the uncertainty estimates. In the context of Neural Networks, the base kernel (computed in line 2 of Algorithm 3) is used to approximate the NN by capturing similarities between data points in the feature space, which is obtained post-training. Kernels can be transformed to either enhance computational efficiency or better represent the relations between data points. The purpose of the kernel transformations (as introduced in [11]) is to formulate many existing BMDAL methods under one common framework.

After transforming the kernel, a selection method (*SELECT*) is invoked. This method utilizes the transformed kernel to guide the selection process, as detailed in Algorithm 4.

In our experiments, we use the neural tangent kernel (NTK) [14] as the base kernel. We use this kernel because it mimics the neural network and performs the best overall when used in conjunction with different selection methods according to the experiments of [11]. The NTK $\Theta(x, x')$ given two input vectors x and x' is defined as the Jacobian of the NN outputs with respect to the network parameters θ, evaluated at the initial parameters, and then taking their inner product (see Eq.1).

$$\Theta(x, x') = \sum_{i,j} \frac{\partial f_i(x)}{\partial \theta_j} \frac{\partial f_i(x')}{\partial \theta_j}, \tag{1}$$

Note that $f_i(x)$ is the ith output of the neural network for input x, and θ_j is the jth parameter of the network.

We consider four different kernel transformations in this paper. First, the GP posterior covariance after observing the training data $\mathcal{X}_{train}$ for a given base kernel k with the corresponding feature map ϕ which is defined in Eq.2.

$$k_{\rightarrow post(\mathcal{X}_{train}, \sigma^2)}(x, x') = \sigma^2 \phi(x)^T (\phi(\mathcal{X}_{train}^T \phi(\mathcal{X}_{train}) + \sigma^2 \mathrm{I})^{-1} \phi(x')) \tag{2}$$

Note that σ^2 is the variance of the observation noise in the underlying model.

Second, we use the scaling transformation where we employ a scaling factor $\lambda \in \mathbb{R}$ to form the scaled kernel $\lambda^2 k$ with the feature map $\lambda\phi$. This is particularly important when using a GP with $\lambda^2 k$ as its covariance function, as it quantifies the covariance between $f(x)$ and $f(\tilde{x})$ based on the prior over functions f.

$$k_{\rightarrow scale(\mathcal{X}_{train})}(x, x') = \lambda^2 k(x, x') \tag{3}$$

The third transformation is sketching, employed to approximate a high-dimensional kernel k with a lower-dimensional one for computational efficiency. We refer to Holzmüller et al. [11] for details.

Finally, we utilize two kernel transformations corresponding to two different ways of applying the ACS-FW method from Pinsler et al. [19] applied to GP regression. Thereby, we use *acs-rf* (kernel of Bayesian batch active learning as sparse subset approximation with p random features) and *acs-rf-hyper* (kernel of Bayesian batch active learning as sparse subset approximation with p random features and hyperprior on σ^2). We refer to Pinsler et al. for details of this method, and Holzmüller et al. [11] for details of the kernel transformation applied to GP regression.

Algorithm 4 Iterative Selection Algorithm Template with Customizable Function `NextSample`

Require: $k, \mathcal{X}_{\text{train}}, \mathcal{X}_{\text{pool}}, \mathcal{X}_{\text{batch}}, \text{mode} \in \{\text{P}, \text{TP}\}$
Ensure: $\mathcal{X}_{\text{batch}}$

1: **function** SELECT $(k, \mathcal{X}_{\text{train}}, \mathcal{X}_{\text{pool}}, \mathcal{X}_{\text{batch}}, \text{mode} \in \{\text{P}, \text{TP}\})$
2: $\mathcal{X}_{\text{batch}} \leftarrow \emptyset$
3: **if** mode = TP **then**
4: $\quad\mathcal{X}_{\text{mode}} \leftarrow \mathcal{X}_{\text{train}}$
5: **else**
6: $\quad\mathcal{X}_{\text{mode}} \leftarrow \emptyset$
7: **end if**
8: **for** $i = 1$ to N_{batch} **do**
9: $\quad\mathcal{X}_{\text{sel}} \leftarrow \mathcal{X}_{\text{mode}} \cup \mathcal{X}_{\text{batch}}$ {Currently ”selected” points}
10: $\quad\mathcal{X}_{\text{rem}} \leftarrow \mathcal{X}_{\text{pool}} \setminus \mathcal{X}_{\text{batch}}$ {Currently unselected points}
11: $\quad\mathcal{X}_{\text{batch}} \leftarrow \mathcal{X}_{\text{batch}} \cup \{\text{NextSample}(k, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}})\}$
12: **end for**
13: **return** $\mathcal{X}_{\text{batch}}$
14: **end function**

Selection Methods: We will now discuss a variety of kernel-based selection methods to be used for querying in active learning. Algorithm 4 shows the details of the selection method *SELECT* that was manipulated in Algorithm 3. To favour samples with high informativeness in an iterative active learning scheme that tries to enforce the diversity of the selected batch, two approaches can be used according to [11]:

- (P) Informativeness can be incorporated through the kernel. For example, $k \rightarrow \mathcal{X}_{train}(x, x)$ represents the posterior variance at x of a GP
- (TP) Informativeness can be incorporated implicitly by enforcing diversity of $\mathcal{X}_{train} \cup \mathcal{X}_{batch}$ instead of only enforcing diversity of $\mathcal{X}_{batch}$. In other words, a batch that is sufficiently different from the training set typically necessarily contains new information.

This explains the usage of *mode* parameter in *SELECT* in Algorithm 4 for different selection methods. It is worth mentioning that we use the same setting that was used in the experiments by Holzmüller et al. [11]. Algorithm 4 serves as a generalized mechanism for constructing sample batches. It takes as input a kernel k, the current training set $\mathcal{X}_{\text{train}}$, a pool of potential samples $\mathcal{X}_{\text{pool}}$, an initially empty batch $\mathcal{X}_{\text{batch}}$, and a mode parameter which can either be P or TP. The algorithm starts by initializing an empty set $\mathcal{X}_{\text{batch}}$ which will be incrementally populated with samples. Depending on the selected mode (P or TP), the algorithm initializes another set $\mathcal{X}_{\text{mode}}$ either as an empty set or as equivalent to the current training set $\mathcal{X}_{\text{train}}$. Then it loops for N_{batch} iterations, where in each iteration, the set of currently "selected" samples, denoted by $\mathcal{X}_{\text{sel}}$, is updated to be the union of $\mathcal{X}_{\text{mode}}$ and $\mathcal{X}_{\text{batch}}$. Additionally, the remaining samples $\mathcal{X}_{\text{rem}}$ are updated to consist of those samples in the pool $\mathcal{X}_{\text{pool}}$ which have not yet been added to $\mathcal{X}_{\text{batch}}$. The selection method (denoted by `NextSample` in the algorithm) then selects a new sample from the remaining set $\mathcal{X}_{\text{rem}}$, based on the kernel k and the set of currently selected samples $\mathcal{X}_{\text{sel}}$. This new sample is added to $\mathcal{X}_{\text{batch}}$. After N_{batch} iterations, the algorithm returns the selected batch $\mathcal{X}_{\text{batch}}$.

Table I shows the selection methods investigated in our experiments and the corresponding kernels and kernels transformation used. Note that many of the selection methods correspond to existing methods in the active learning literature, some of which were originally formulated for classification. Holzmüller et al. [11] formulate each of these selection methods under one common framework and adapt them to regression if needed. For simplicity, we use similar (but shortened) formulations, but we refer to [11] for details of each method.

- *Random Selection*. This corresponds to sampling a data point uniformly at random from the points in the pool $\mathcal{X}_{rem}$. The selection method is shown in Eq.4.

$$NextSample(K, \mathcal{X}_{sel}, \mathcal{X}_{rem}) \sim \mathcal{U}(\mathcal{X}_{rem}), \tag{4}$$

  where $\mathcal{U}(\mathcal{X}_{rem})$ is the uniform distribution over X_{rem}. For this method both P and TP are equivalent.
- *MAXDIAG*. This corresponds to Eq.5. It is shown in [11] that this is equivalent to BALD [12] in a regression setting.

$$NextSample(K, \mathcal{X}_{sel}, \mathcal{X}_{rem}) = \text{argmax}_{x \in \mathcal{X}_{\text{rem}}} K(x, x) \tag{5}$$

  According to Eq.5, MAXDIAG selects the maximum of the elements on the diagonal of the posterior covariance matrix. For this method both P and TP are equivalent.
- *MAXDET*. This corresponds to Eq.6. It is shown in [11] that this is equivalent to BatchBALD [16] in the regression setting. This only holds under certain conditions, see [11] for details.

$$NextSample(K, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}}) = \text{argmax}_{x \in \mathcal{X}_{\text{rem}}} \det(k(\mathcal{X}_{\text{sel}} \cup \{x\}, \mathcal{X}_{\text{sel}} \cup \{x\}) + \sigma^2 \text{I}) \tag{6}$$

  MAXDET. This is considered an improvement over *MAXDIAG* because it takes $\mathcal{X}_{\text{sel}}$ into account by conditioning the GP on $\mathcal{X}_{\text{sel}}$ when computing the posterior covariance.

- *BAIT*. This corresponds to the selection method introduced by Ash et al. [3]. BAIT potentially improves on the previous selection methods by also considering how well the selected batch represents the current pool set. It is shown in [11] that the original formulation from [3] is equivalent to Eq.7.

$$\text{NextSample}(K, \mathcal{X}_{\text{sel}}, \mathcal{X}_{\text{rem}}) = \text{argmin}_{x \in \mathcal{X}_{\text{rem}}} \Sigma_{x' \in \mathcal{X}_{train} \cup \mathcal{X}_{pool}} k \rightarrow_{post} (\mathcal{X}_{sel} \cup x, \sigma^2)(x', x') \tag{7}$$

  Note that [3] introduces two versions of BAIT: forward and forward/backward. Eq.7 corresponds to the forward version of BAIT, which we use in our experiments due to superior performance.
- *FRANKWOLFE*. This method approximates the kernel mean embedding using a Frank-Wolfe optimization algorithm. To ensure that $\mathcal{X}_{\text{batch}}$ accurately represents the pool set, Pinsler et al. [19] recommend constructing $\mathcal{X}_{\text{batch}}$ in a manner that closely approximates $\Sigma_{x \in \mathcal{X}_{\text{pool}}} \phi(x)$ by $\Sigma_{x \in \mathcal{X}_{\text{batch}}} w_x \phi(x)$, where w_x's are non-negative weights. Specifically, they advocate the use of the Frank-Wolfe optimization algorithm to solve the related optimization problem, enabling an iterative selection of elements into $\mathcal{X}_{\text{batch}}$. This method aims to approximate the distribution of $\mathcal{X}_{\text{pool}}$ through $\mathcal{X}_{\text{batch}}$ by mimicking the empirical kernel mean embedding $N_{\text{pool}}^{-1} \Sigma_{x \in \mathcal{X}_{\text{pool}}} k(x, .)$ using $\mathcal{X}_{\text{batch}}$. The strategy can be executed in either the kernel or feature space. Due to the quadratic scaling with N_{pool} in the kernel space, Pinsler et al. [19] opt for the feature space approach when handling large pool sets, a choice we also adopt in our experiments. Unlike the original method which allows for repeated selection of the same $x \in \mathcal{X}_{\text{pool}}$, we disallow this to ensure batch sizes remain consistent for a fair comparison with other techniques.
- *MAXDIST*. This corresponds to greedily selecting data points that maximize the distance to those already selected. The selection method is shown in Eq.8.

$$NextSample(k, \mathcal{X}_{sel}, \mathcal{X}_{rem}) = \text{argmax}_{x \in \mathcal{X}_{\text{rem}}} \min_{x' \in \mathcal{X}_{sel} d_k(x, x')} \tag{8}$$

  This method is equivalent to Coreset [25] for a particular configuration of the kernel (see [11] for details).
- *KMEANSPP*. This is defined in Eq.9 and is related to BADGE [4] (see [11] for details).

$$NextSample(k, \mathcal{X}_{sel}, \mathcal{X}_{rem}) = \frac{\min_{x' \in \mathcal{X}_{sel}} d_k(x, x')^2}{\Sigma_{x' \in \mathcal{X}_{rem}} \min_{x \in \mathcal{X}_{sel}} d_k(x, x')^2} \tag{9}$$

  Much like MaxDet, MaxDist ensures both Informativeness and Diversity but falls short on Representativity. To address this, one can consider batch selection as a clustering problem. In Eq.9, the optimization task essentially reformulates the k-medoids problem, blending the k-means clustering objective with the stipulation that cluster centroids must be selected from the clustered dataset.
- *LCMD*. As a deterministic counterpart to the stochastic k-meansPP method, Holzmüller et al. [11] introduce a method known as LCMD (Largest Cluster Maximum Distance). This selection method considers representativity by restricting selections to the largest cluster, while also promoting diversity by selecting the data point that is furthest away within that cluster. In this context, $x' \in \mathcal{X}_{sel}$ denotes cluster centroids, $c(x)$ signifies the associated center for each $x \in \mathcal{X}_{rem}$, and $S(x')$ represents the size of the cluster. According to Eq.10, the data point with the greatest distance from the largest cluster is selected.

$$NextSample(K, \mathcal{X}_{sel}, \mathcal{X}_{rem}) = \text{argmax}_{x \in \mathcal{X}_{\text{rem}} : s(c(x)) = max_{x' \in X_{sel}} s(x')} d_K(x, c(x)) \tag{10}$$
$$c(x) = \text{argmin}_{x' \in \mathcal{X}_{sel}} d_k(x, x')$$
$$s(x') = \Sigma_{x \in \mathcal{X}_{rem} : c(x) = x'} d_k(x, x')^2$$

TABLE I: An overview of the used kernel and kernel transformation for each selection method

Selection Method	Kernel	Kernel Transformation	Mode
BAIT MAXDIST MAXDET	NTK	$sketch \rightarrow scale \rightarrow post$	P
KNEANSPP MAXDIAG		$sketch \rightarrow$ acs-rf	P -
FRANKWOLFE		$sketch \rightarrow$ acs-rf-hyper	P
LCMD		$sketch$	TP
Random	-	-	-

IV. Experiment

A. Collection of Data

To bolster the dependability of our experiments, two distinct real-world datasets consisting of performance metrics are utilized. The first, called *OSSBuild*, consists of actual build data acquired from the continuous integration frameworks of four distinct open-source projects. The second, termed *HadoopTests*, is a more expansive dataset that we gathered ourselves by running the Hadoop open-source system's unit tests in a well-regulated setting. A summarization of both datasets can be found in Table II. Further details about each dataset are elaborated in the subsequent subsections.

1) OSSBuild Dataset: Initially employed in the work of Samoaa et al. [24], this dataset includes data related to test run times in the build systems of four open-source softwares: systemDS, H2, Dubbo, and RDF4J. All of these projects make use of public continuous integration platforms and provide publicly available build details, which we used to gather data on test execution times in the summer of 2021. Refer to

TABLE II: Overview of the OSSBuilds and HadoopTests datasets.

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

Table II (top) for essential statistics about these projects. The term "Files" refers to the unit test files we monitored for execution durations, while "Runs" signifies the aggregated execution count for these files. "Nodes" and "Vocabulary Size" denote the graphs. Prior to parsing, we exclude code comments to minimize the graph nodes. We observe 867094 nodes and 36880 vocabulary entries.

2) HadoopTests Dataset: In order to address the limitations of the OSSBuild dataset, particularly the confined file counts per project, a second dataset was generated. We selected the Apache Hadoop project due to its extensive collection of test files (2895) with adequate complexity. We executed all of the unit tests in the project five times and recorded each test file's execution time, as reported by the JUnit framework. We utilized a dedicated virtual machine equipped with two virtualized CPUs and 8 GBytes of RAM for this data collection, and non-essential services were disabled to ensure consistent performance. Statistics for the HadoopTests dataset are outlined in Table II (bottom). The dataset has an enlarged node count with 5090798 nodes and 138952 vocabulary terms.

B. Experiment Setting

To systematically investigate different combinations of kernels, kernel transformations, and selection methods as outlined in Table 1, we subject our datasets to these various selection techniques. For the HadoopTests dataset, the initial training size, denoted by $\mathcal{N}_{\text{train}}$, is set at 256, while for OssBuilds, it is 88. We then proceed to obtain 16 batches, each having a size of $\mathcal{N}_{\text{batch}}$ equalling 128 for HadoopTests and 45 for OssBuilds, applying the corresponding BMAL method. This entire process is repeated 10 times, each time with unique initialization seeds for the neural network (NN) and different partitions of the data into training, pool, and test subsets. The evaluation metric we consider is the root mean squared error (RMSE) calculated on the test dataset after each BMAL iteration. The logarithm of the RMSE error metric is then averaged over 10 repetitions and, depending on the specific experiment, over 16 steps for each dataset and embedding.

The GNN model is configured with three layers of GraphCov layers. In contrast, for the NN model, we employ a fully connected architecture consisting of three layers, each having 512 neurons in both of the hidden layers. The activation function chosen for both networks is 'relu'. The training of both GNN and NN is executed using the Adam optimizer, spanning 256 epochs with a batch size of 32. The embedding dimension in the supervised and unsupervised settings is 90.

C. Experimental Results

In this section, we will present the average RMSE values. The mean log RMSE for each embedding is illustrated in different subfigures. We assess the performance of the configurations outlined in Table I.

a) ***BMDAL for HadoopTests:*** Figure 6 illustrates the performance of various selection methods in the context of HadoopTests. It breaks down the results by depicting the average log RMSE in both supervised and unsupervised embeddings. In unsupervised embedding, shown in Figure 6a, Random selection consistently underperforms relative to other methods. MAXDIST stands out as the most effective, particularly as the labelled data grow. Moving to the supervised embedding results in Figure 6b, the Random selection method still performs the poorest, while BAIT and MAXDET consistently outperform the rest across various training set sizes. Interestingly, the typical best-performing methods (BAIT and MAXDET) do not maintain their lead when the size of the labelled data is restricted to around 256 samples. In this specific context, MAXDIST is the most effective method. Overall, the results demonstrate the effectiveness of active learning, i.e., the benefits of non-random selection methods, especially MAXDIST.

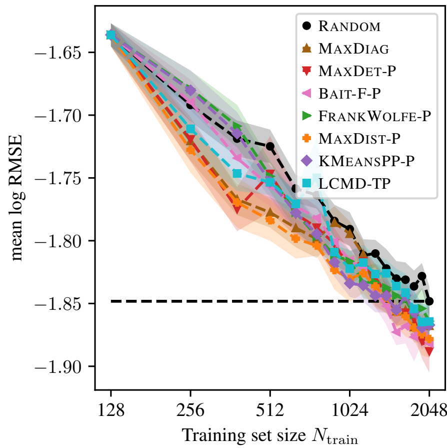
b) ***BMDAL for OssBuilds:*** The evaluations for OssBuilds reveal a noticeably higher variance in each selection method for OSSBuilds compared to the HadoopTest dataset. This increased variability is likely due to having fewer samples of OSSBuilds, which comprises graphs from four distinct projects.

In the unsupervised embedding setting, as indicated by Figure 7a, both Random and FRANKWOLFE methods generally underperform. Intriguingly, LCMD exhibits a sudden and significant improvement, becoming the best-performing method when the training batch size reaches approximately 256. However, this performance gain is ephemeral, as its RMSE error escalates once again beyond this point.

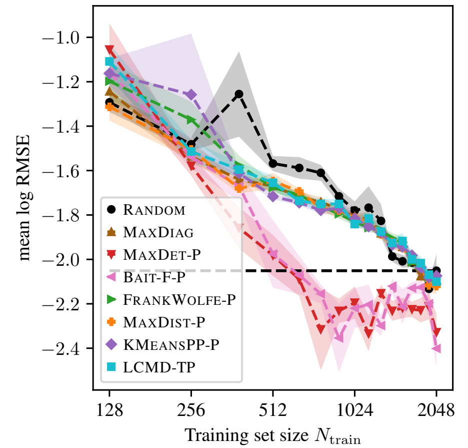
Despite Random being the least effective method in supervised settings as in Figure 7b, certain variations appear at smaller training set sizes. Specifically, in the first 64 labelled training samples, FRANKWOLFE underperforms most notably. For the same training sample size, MAXDIST emerges as the best performer, consistent with the HadoopTest dataset.

D. Further Discussions

In this section, we discuss further the results from various perspectives.

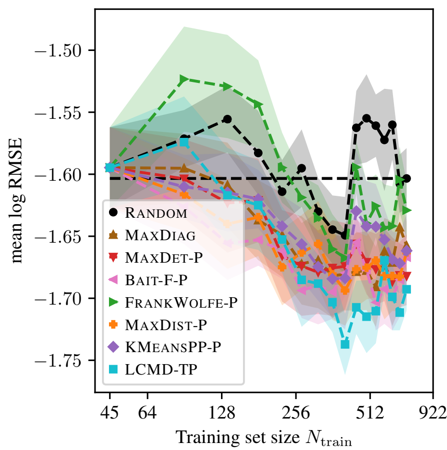


(a) Unsupervised using Graph2Vec

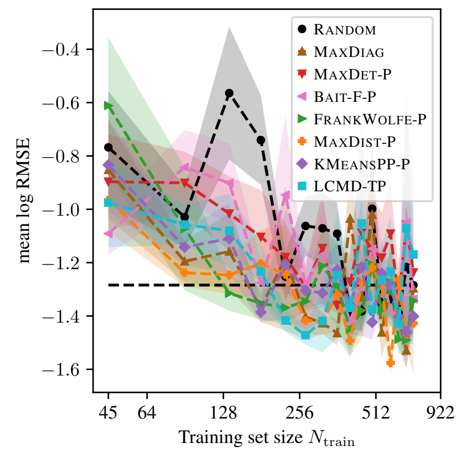


(b) Supervised using GNN

Fig. 6: Root mean square error for BMDAL in both embedding settings on HadooptTests.



(a) Unsupervised using Graph2Vec



(b) Supervised using GNN

Fig. 7: Root mean square error for BMDAL in both embedding settings on OssBuilds.

- **Observations on data variability**: Our results indicate smoother and less variable performance for HadoopTests compared to OssBuilds. This difference is primarily due to the source of the graphs. While Hadoop's graphs originate from a single project, OssBuilds features graphs from various domain projects (as detailed in Table II). Additionally, the larger number of graphs in Hadoop contributes to this stability.
- **Performance w.r.t. embedding types**: Upon examining the mean log RMSE values, it is clear that supervised embeddings offer the most effective setting for the selection methods. This is evidenced by the lower mean log RMSE and higher delta (mean log RMSE) — 1.4 for HadoopTests and 1 for OssBuilds—compared to 0.25 for unsupervised embedding. However, caution is warranted in generalizing these findings, as they may require validation with more diverse graph data from various projects.
- **Computational considerations**: It is worth noting that the supervised setting comes with increased computational demands. This is because each active learning iteration involves not only training an NN based on the embeddings but also training the GNN to obtain those embeddings.
- **Graph characteristics and implications**: Our analysis performed on graph data in our previous study [22] (Table 2) reveals that the graphs in our study are characterized by high diameter and sparsity, adding complexity to the task. Furthermore, these graphs are augmented versions of Abstract Syntax Trees (ASTs).
- **Comparison with previous work**: Interestingly, our current findings diverge from our previous paper where batch and kernel components were not utilized. This highlights the crucial role both the active learning and the quality of embeddings play in influencing the results.

V. Conclusion

In this study, we employed Batch Mode Deep Active Learning (BMDAL) for graph data within a regression framework. The algorithm integrates kernels and kernel transformations with active learning selection methods. Specifically, the Neural Tangent Kernel (NTK) serves as the base kernel, while the Gaussian Process (GP) posterior variance is primarily utilized for kernel transformation. Supervised and unsupervised embedding are investigated to adapt the graph data to this framework. Our experimental results indicate that supervised embedding provides the most effective setting for selection methods. While identifying a universally optimal selection method across different experimental settings proved challenging, MAXDET and MAXDIST consistently emerged as top performers. Conversely, the Random method, used as a baseline, consistently ranked as the least effective, indicating the advantage of active learning for data labelling.

Acknowledgment

This work received financial support from the Swedish Research Council VR under grant number 2018-04127. The work of Linus Aronsson was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundations.

References

[1] *Batch Mode Deep Active Learning for Regression on Graph Data*. Zenodo, Sept. 2023. https://doi.org/10.5281/zenodo.8352242.

[2] R. Abel and Y. Louzoun. Regional based query in graph active learning, 2019.

[3] J. Ash, S. Goel, A. Krishnamurthy, and S. Kakade. Gone fishing: Neural active learning with fisher embeddings. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 8927–8939. Curran Associates, Inc., 2021.

[4] J. T. Ash, C. Zhang, A. Krishnamurthy, J. Langford, and A. Agarwal. Deep batch active learning by diverse, uncertain gradient lower bounds, 2020.

[5] J. D. Bossér, E. Sörstadius, and M. H. Chehreghani. Model-centric and data-centric aspects of active learning for deep neural networks. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 5053–5062, 2021.

[6] H. Cai, V. W. Zheng, and K. C.-C. Chang. Active learning for graph embedding, 2017.

[7] X. Chen, G. Yu, J. Wang, C. Domeniconi, Z. Li, and X. Zhang. Activehne: Active heterogeneous network embedding. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 2123–2129. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[8] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[9] Y. Gal, R. Islam, and Z. Ghahramani. Deep bayesian active learning with image data. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1183–1192. PMLR, 06–11 Aug 2017.

[10] L. Gao, H. Yang, C. Zhou, J. Wu, S. Pan, and Y. Hu. Active discriminative network representation learning. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2142–2148. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[11] D. Holzmüller, V. Zaverkin, J. Kästner, and I. Steinwart. A framework and benchmark for deep batch active learning for regression. *Journal of Machine Learning Research*, 24(164):1–81, 2023.

[12] N. Houlsby, F. Huszár, Z. Ghahramani, and M. Lengyel. Bayesian active learning for classification and preference learning, 2011.

[13] S. Hu, Z. Xiong, M. Qu, X. Yuan, M.-A. Côté, Z. Liu, and J. Tang. Graph policy network for transferable active learning on graphs, 2020.

[14] A. Jacot, F. Gabriel, and C. Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[15] S. Jarl, L. Aronsson, S. Rahrovani, and M. H. Chehreghani. Active learning of driving scenario trajectories. *Engineering Applications of Artificial Intelligence*, 113:104972, 2022.

[16] A. Kirsch, J. van Amersfoort, and Y. Gal. Batchbald: Efficient and diverse batch acquisition for deep bayesian active learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[17] X. Li, Y. Wu, V. Rakesh, Y. Lin, H. Yang, and F. Wang. Smartquery: An active learning framework for graph neural networks through hybrid uncertainty reduction. In *Proceedings of the 31st ACM International Conference on Information; Knowledge Management*, CIKM '22, page 4199–4203, New York, NY, USA, 2022. Association for Computing Machinery.

[18] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal. graph2vec: Learning distributed representations of graphs, 2017.

[19] R. Pinsler, J. Gordon, E. Nalisnick, and J. M. Hernández-Lobato. Bayesian batch active learning as sparse subset approximation. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[20] C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006.

[21] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, and X. Wang. A survey of deep active learning, 2021.

[22] P. Samoaa, L. Aronsson, A. Longa, P. Leitner, and M. H. Chehreghani. A unified active learning framework for annotating graph data with application to software source code performance prediction, 2023.

[23] P. Samoaa, F. Bayram, P. Salza, and P. Leitner. A systematic mapping study of source code representation for deep learning in software engineering. *IET Software*, 16(4):351–385, 2022.

[24] P. Samoaa, A. Longa, M. Mohamad, M. H. Chehreghani, and P. Leitner. Tep-gnn: Accurate execution time prediction of functional tests using graph neural networks. In D. Taibi, M. Kuhrmann, T. Mikkonen, J. Klünder, and P. Abrahamsson, editors, *Product-Focused Software Process Improvement*, pages 464–479, Cham, 2022. Springer International Publishing.

[25] O. Sener and S. Savarese. Active learning for convolutional neural networks: A core-set approach. In *International Conference on Learning Representations*, 2018.

[26] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[27] Y. Shen, H. Yun, Z. C. Lipton, Y. Kronrod, and A. Anandkumar. Deep active learning for named entity recognition, 2018.

[28] S. Viet Johansson, H. Gummesson Svensson, E. Bjerrum, A. Schliep, M. Haghir Chehreghani, C. Tyrchan, and O. Engkvist. Using active learning to develop machine learning models for reaction yield prediction. *Molecular Informatics*, 41(12):2200043, 2022.

[29] D. Wu. Pool-based sequential active learning for regression. *IEEE Transactions on Neural Networks and Learning Systems*, 30(5):1348–1359, 2019.

[30] Y. Wu, Y. Xu, A. Singh, Y. Yang, and A. Dubrawski. Active learning for graph neural networks via node feature propagation, 2019.

[31] Y. Zhang, H. Tong, Y. Xia, Y. Zhu, Y. Chi, and L. Ying. Batch active learning with graph neural networks via multi-agent deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(8):9118–9126, Jun. 2022.

[32] Y. Zhang, Y. Xia, Y. Zhu, Y. Chi, L. Ying, and H. Tong. Active heterogeneous graph neural networks with per-step meta-q-learning. In *2022 IEEE International Conference on Data Mining (ICDM)*, pages 1329–1334, 2022.