Integrating Graph Neural Networks (GNNs) with Contrastive Learning for Condensed Variable Semantic Representations

George Kaceli
University of Windsor
Windsor, ON, Canada
kaceli@uwindsor.ca

ABSTRACT

Learning accurate semantic representations of program variables is a fundamental challenge in code analysis. We present a self-supervised Graph Neural Network framework that fuses pretrained textual embeddings and structural features into a unified Variable Dependency Graph, then trains compact 128-dimensional embeddings with two complementary objectives: a local contrastive loss (InfoNCE) and a global mutual-information maximization (Deep Graph Infomax). To accommodate variable names absent from the graph, we introduce a lightweight fallback projection that maps raw textual features into the shared latent space. We evaluate our approach on all three splits of the IdBench benchmark—small, medium, and large—reporting Spearman rank correlations of up to 0.47 (similarity) and 0.75 (relatedness), and Pearson coefficients up to 0.42 and 0.72 respectively (Tables 1–4). Despite using only one-sixth the dimensionality of competing methods such as VarCLR and VarGAN, our embeddings recover over 80 percent of their performance, while offering substantial gains in storage and runtime efficiency. These results demonstrate that a dual-objective, graph-aware model can achieve high-fidelity variable semantics in a very compact form, making it well suited for scalable code-analysis applications.

KEYWORDS

Semantic Representation, Contrastive Learning, Graph Neural Networks, Downstream Tasks

1 INTRODUCTION

Learning rich semantic representations of program variables lies at the core of many critical software-engineering applications, including semantic code search, automated bug triage, refactoring recommendation, and comprehensive program understanding. Variables serve as the primary carriers of program intent: their names often convey domain concepts, while their lifecycles within functions reflect data transformations and control-flow dependencies. Traditional representation techniques—such as token-level bag-of-words, n-gram co-occurrence models, or tree-based encodings of abstract syntax—capture only fragments of this multifaceted semantics and frequently fail to reconcile naming conventions with the precise structural relationships (e.g., definition-use chains, control predicates) that govern variable behavior in real codebases.

Recent deep-learning approaches have advanced two complementary paradigms for learning code representations. Transformer-based textual models (e.g., RoBERTa, CodeBERT, VarCLR) learn contextualized embeddings directly from source-token streams, effectively modeling lexical similarity, identifier co-occurrences, and even in-line comments. Graph Neural Networks (GNNs), in contrast, operate on explicit program graphs—where nodes represent variables or statements and edges encode data-flow, control-flow, and lexical co-occurrence links—capturing structural patterns that token-only methods miss. However, when used in isolation, each paradigm exhibits limitations: textual encoders lack explicit reasoning over graph structure, while pure-GNN approaches may underutilize rich lexical cues inherent in identifier names.

To bridge this gap, we introduce a unified, self-supervised framework that fuses pretrained textual embeddings with graph-based structural reasoning to learn compact, 128-dimensional variable representations optimized for both local semantic alignment and global graph coherence. Our pipeline begins by statically analyzing each function to construct a Variable Dependency Graph (VDG). In a VDG, each variable is a node, and we extract directed edges corresponding to definition-use chains, control-flow dependencies (e.g., loop or branch contexts), and lexical co-occurrence within code blocks. Nodes are initialized with high-dimensional embeddings from the VarCLR model—fine-tuned on real-world renaming instances—and augmented with handcrafted structural features such as in-degree, out-degree, and loop-nest depth.

Our GNN encoder comprises multiple graph-convolutional layers with residual connections. At each layer, node features are updated by aggregating messages from adjacent nodes via a learnable attention mechanism, followed by layer normalization and a two-layer MLP with ReLU activations. To integrate textual and structural inputs, we concatenate the VarCLR embedding with the structural feature vector at the initial layer and allow the network to learn cross-modal interactions through message passing.

Training proceeds under two self-supervised objectives. First, an InfoNCE contrastive loss uses positive pairs—such as renaming-equivalent nodes or function-level perturbations (e.g., subgraph dropout, variable swapping)—and treats other nodes in the batch as negatives, encouraging semantically equivalent variables to cluster in the latent space. Second, a Deep Graph Infomax loss maximizes mutual information between node embeddings and a graph-level summary vector, computed via a readout MLP over all node features, thereby promoting global structural coherence.

To ensure robustness to unseen identifiers, we introduce a lightweight fallback projection head: a character-level MLP that maps byte-pair encoded subword histograms or token-fingerprint features into the shared 128-dimensional space when a pretrained embedding is unavailable. This fallback mechanism enables graceful degradation, relying solely on structural cues when textual information is lacking.

In the subsequent sections, we detail the VDG extraction process, the architecture of our GNN encoder, the design of the self-supervised objectives, and the evaluation protocol used to assess the quality of these condensed, graph-aware embeddings.

2 LITERATURE OVERVIEW

Recent years have witnessed a surge of interest in leveraging deep learning methods for code representation and analysis. In particular, Graph Neural Networks (GNNs) have emerged as powerful tools for modeling the inherent graph structure of source code, while various pre-training and contrastive learning approaches have significantly advanced variable semantic representation.

Several studies have demonstrated the effectiveness of GNNs in extracting rich structural features from code. For example, Rafi et al. [11] propose an enhanced fault localization framework that constructs comprehensive code graphs by integrating Abstract Syntax Trees (ASTs), control/data flow, and interprocedural call information. By aggregating such information, their approach achieves more accurate fault localization. Similarly, Tailor [9] presents a tailored GNN that learns from Code Property Graphs (CPGs)—which merge ASTs, control flow graphs, and data flow graphs—to capture multi-hop semantic relationships in code, thereby effectively detecting functional similarity for tasks such as code clone detection and source code classification.

On the variable semantic representation front, contrastive learning has emerged as a promising paradigm. VarCLR [2] employs contrastive pre-training on a weakly supervised dataset of variable renaming events to refine variable embeddings. By constructing positive pairs from these renaming instances, VarCLR pushes semantically equivalent variable names (e.g., `avg` and `mean`) closer in the embedding space while separating unrelated ones. Complementarily, VarGAN [8] introduces an adversarial framework to align the representation of rare (low-frequency) variable identifiers with that of common ones. This regularization strategy is critical to ensuring that even less frequent identifiers obtain meaningful embeddings, thereby addressing issues of data imbalance that traditional models often suffer from.

In parallel with GNN-based techniques, traditional word embedding approaches have also played a foundational role in modeling text and variable semantics. FastText-based models, which include the skip-gram (FT-SG) and continuous bag-of-words (FT-CBOW) variants [1], enrich word vectors by incorporating subword information. These methods capture morphological patterns that are particularly valuable for representing complex or rare variable names. However, while FT-SG and FT-CBOW have shown success in modeling token semantics via subword decomposition, they are typically limited by their bag-of-words assumption and do not always capture the nuanced semantic interchangeability of variables.

More recently, transformer-based pre-trained models have further advanced this field. RoBERTa [10] represents a robustly optimized version of BERT, achieving state-of-the-art results on various natural language processing tasks through extensive pretraining and hyperparameter tuning. Although RoBERTa was originally developed for natural language, its ability to capture rich contextual information makes it a compelling candidate for transferring to code-based tasks. Building on this idea, CodeBERT [4] is specifically pre-trained on joint data of code and natural language, thereby learning bilingual representations that bridge code syntax with natural language semantics. CodeBERT has proven effective in downstream tasks such as code search and documentation generation, highlighting its strength in capturing both token-level and contextual semantics of code identifiers.

Together, these works provide a solid foundation for our project. By integrating the fine-grained variable semantic learning promoted by VarCLR and VarGAN with the rich structural insights offered by GNN-based models, and by drawing on the advances in text and subword-based representations as implemented in FastText, RoBERTa, and CodeBERT, our work seeks to develop a unified framework that produces variable representations capable of effectively supporting downstream tasks such as bug detection, refactoring, and code search.

3 DATASET DESCRIPTION

In this approach, our data preparation pipeline is designed to provide a rich and diverse set of code samples that support both the learning of structural patterns and the extraction of semantic cues for variable representation. We utilize a large-scale, multilanguage benchmark, CodeSearchNet, and a custom extraction process to derive function-level code along with associated documentation. This section details the various aspects of our dataset, from the underlying CodeSearchNet corpus to the specific strategies employed in function code extraction, and outlines the overarching purpose and intent driving this process.

3.1 CodeSearchNet

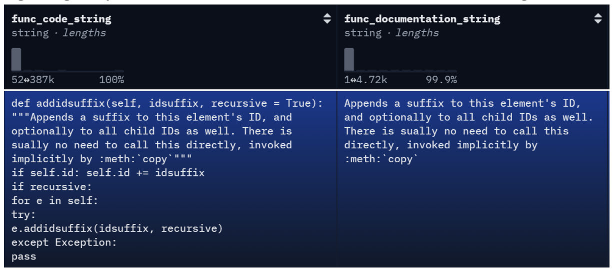
CodeSearchNet [6] is a widely-used benchmark dataset tailored for code search and various code understanding tasks. The dataset comprises millions of functions harvested from real-world open-source projects and is available in multiple programming languages including Python, Java, and Go. Each code sample in CodeSearchNet typically comes with a corresponding documentation string that describes the purpose or functionality of the code fragment. The dataset is partitioned into different splits (e.g., train, validation, and test) to facilitate robust evaluation under diverse experimental conditions.

Key aspects of CodeSearchNet that make it particularly valuable for our study include:

- **Multi-language Coverage:** With support for languages such as Python, Java, and Go, the dataset allows us to study variable semantics across different programming paradigms and syntax.
- **Rich Contextual Information:** Each function in the dataset is paired with natural language documentation, offering complementary semantic cues that aid in understanding code intent.
- **Real-world Diversity:** The functions are sourced from a broad array of repositories, ensuring a wide range of coding styles, variable naming conventions, and implementation idioms.

3.2 Function Code Extraction

To harness the full potential of CodeSearchNet, a dedicated function code extraction pipeline is employed to convert the raw dataset



```
def addidsuffix(self, idsuffix, recursive = True):
"""Appends a suffix to this element's ID, and
optionally to all child IDs as well. There is
sually no need to call this directly, invoked
implicitly by :meth:`copy`"""
if self.id: self.id += idsuffix
if recursive:
for e in self:
try:
e.addidsuffix(idsuffix, recursive)
except Exception:
pass
```

Appends a suffix to this element's ID, and optionally to all child IDs as well. There is sually no need to call this directly, invoked implicitly by :meth:`copy`

Figure 1: Code Sample from CodeSearchNet Dataset

into a structured format that is amenable to downstream graph construction and semantic analysis. The process involves the following detailed steps:

(1) **Dataset Loading:** Using the Hugging Face `datasets` library, we load the specific language subsets of CodeSearchNet. For example, separate calls are made to load the Python, Java, and Go splits.
(2) **Field Selection and Cleaning:** Each code sample in the dataset provides several fields; for our purposes, we extract the function code and the associated documentation. We apply cleaning routines to remove extraneous whitespace and ensure the text is normalized. This step is crucial to mitigate the impact of noisy or inconsistent formatting.
(3) **Data Structuring and Storage:** The cleaned function code and documentation strings are encapsulated into well-defined JSON objects. For each programming language, the extracted entries are aggregated into dedicated JSON files. This structured storage format not only enables easy access and reproducibility but also streamlines the subsequent process of building Variable Dependency Graphs (VDGs) for GNN-based analysis.

3.3 Purpose

The primary aim of our dataset curation is to bridge the gap between raw source code and meaningful, semantically rich representations of program variables. By coupling each function's implementation with its corresponding natural-language documentation, we provide a dual modality that captures both the precise syntactic structure of code and the developer's own description of intent. These paired signals enable models to ground variable names in their contextual usage, disambiguating identifiers that might otherwise appear similar yet serve distinct roles.

To ensure that this semantic richness is fully exploitable, we apply rigorous preprocessing to each code sample. Boilerplate comments and license headers are removed, whitespace and indentation are standardized, and docstrings are validated for length and content. This cleaning process produces function snippets that are not only syntactically correct but also representative of real-world programming practices, thereby facilitating the construction of accurate Variable Dependency Graphs (VDGs) in subsequent stages.

Diversity is another cornerstone of our dataset. Drawing from the multi-language CodeSearchNet corpus—encompassing Python, Java, and Go—our collection spans a wide range of programming paradigms, naming conventions, and domain-specific idioms. This breadth exposes the model to varied type systems, control-flow constructs, and repository contexts, reducing the risk of overfitting to a single codebase or style. As a result, embeddings trained on this data are more robust and capable of generalizing to novel functions and projects.

Beyond robustness, the curated dataset is explicitly designed to support downstream applications that rely on nuanced variable semantics. High-quality VDGs enable precise bug localization by revealing unexpected data flows, empower semantic code search by matching functions based on variable relationships rather than mere token overlap, and facilitate automated refactoring by detecting semantically equivalent identifiers across large code repositories. In each of these cases, the depth and structure of our dataset directly contribute to improved performance.

Finally, by assembling a comprehensive, well-structured corpus that integrates both code and documentation, we lay the groundwork for a self-supervised training regime. This dataset not only supports the joint optimization of local contrastive objectives and global mutual-information goals but also ensures that the resulting embeddings capture the full spectrum of variable semantics found in practical software engineering.

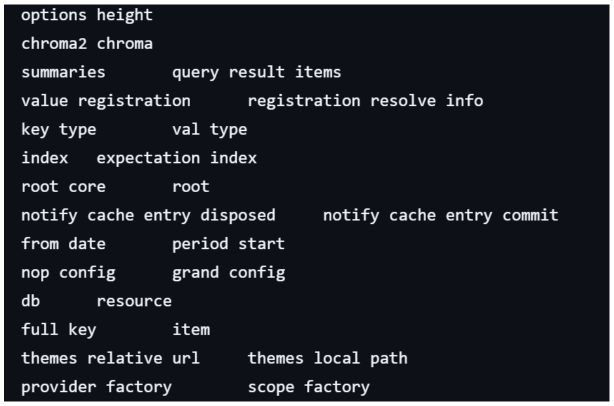
3.4 VarCLR GitHubRENAMES Dataset

The VarCLR GitHubRENAMES dataset [2] captures real-world examples of variable renaming in open-source repositories, providing valuable signal on how developers choose and modify variable names during code evolution. This dataset was constructed by mining Git commit histories for renaming operations where a variable's identifier was changed without altering its functional behavior. Each instance in GitHubRENAMES consists of a pair of code snippets—before and after renaming—along with contextual information such as file path, function scope, and commit metadata. By focusing on these targeted edits, the dataset offers insight into the semantic relationships that underpin meaningful variable names and the contexts in which renaming occurs.

In our pipeline, we integrate the GitHubRENAMES instances by aligning each renaming pair with the corresponding function-level JSON objects extracted from CodeSearchNet. We augment these objects with an additional field that records the original and new variable names, enabling contrastive learning objectives where the model must distinguish semantically equivalent variables under different naming schemes. This augmentation process involves matching file paths and function signatures, followed by validation to ensure that the code before and after renaming remains functionally identical. The enriched dataset thus combines static snapshots from CodeSearchNet with dynamic renaming events from GitHubRENAMES, creating a more comprehensive training resource for learning robust variable representations.

4 EXPERIMENTAL PROCEDURE

In this section we describe our end-to-end pipeline for learning condensed, graph-aware variable embeddings. Starting from raw code, we (1) construct a Variable Dependency Graph (VDG), (2) design and train a self-supervised GNN with dual objectives, and (3) employ a lightweight fallback projection for out-of-vocabulary identifiers. Below we expand on each stage in detail.



```
options height
chroma2 chroma
summaries       query result items
value registration      registration resolve info
key type        val type
index   expectation index
root core       root
notify cache entry disposed     notify cache entry commit
from date       period start
nop config      grand config
db      resource
full key        item
themes relative url     themes local path
provider factory        scope factory
```

Figure 2: Sample of Renames from Github Renames Dataset

4.1 Graph Construction

Our VDG captures both explicit data flows and implicit co-occurrence patterns among variables within individual functions. All parsing and extraction is implemented via the Tree-sitter library, enabling language-agnostic AST traversal.

4.1.1 Source Parsing and Cleaning. We begin by loading each function's source code (Python, Java, or Go) from our JSON-serialized corpus. Each snippet is parsed into an AST using the corresponding Tree-sitter grammar. Prior to graph extraction, we normalize the code by:

- Removing non-informative comments and excessive whitespace,
- Standardizing indentation and converting tabs to spaces,
- Validating that docstrings or comments exceed a minimum length (for future semantic alignment).

4.1.2 Identifying Assignments.

AST-Driven Extraction. We traverse the AST looking for assignment constructs. For each such node, we:

(1) Use Tree-sitter's 'child_by_field_name' to isolate the LHS and RHS subtrees,
(2) Recursively extract all identifier tokens within each subtree (handling tuples, lists, method calls, attribute accesses, and subscripts),
(3) Emit one directed edge from every RHS identifier to every LHS identifier.

Multi-Target and Call-Based Assignments. Simultaneous assignments (e.g., 'x, y = a, b') and call-based assignments (e.g., 'x = foo(y, z)') are each decomposed into individual '(source,target)' edges. In the call-based case, we extract argument identifiers separately and link each to all LHS variables.

Identifier Normalization. Extracted names are stripped of leading/trailing whitespace, lowercased (where language semantics permit), and tokenized via byte-pair encoding to facilitate downstream textual embedding alignment.

4.1.3 Identifying Dependencies.

Direct Data Flows. Each assignment edge encodes a direct dataflow dependency: an edge $(u \rightarrow v)$ indicates that the value of variable u influences variable v. We accumulate edge weights by counting repeated occurrences across functions.

Implicit (Virtual) Edges. To capture weaker semantic ties, we maintain a global co-occurrence counter: whenever two variables appear in the same function (regardless of direct assignment), their pair count is incremented. After processing all samples, any pair exceeding a co-occurrence threshold (e.g., $T = 2$) is connected by a "virtual" bidirectional edge. Virtual edges are annotated separately and later merged with existing dependency edges (if present) by summing their weights.

Graph Augmentation. Throughout extraction we record per-variable statistics:

- *Frequency*: total occurrences in assignment LHS or RHS,
- *Co-occurrence degree*: number of distinct partners above threshold,
- *AST depth*: position of first occurrence in the parse tree,
- *Function-level metrics*: number of statements and variables per function.

These features are later incorporated as initial node attributes.

4.1.4 Final VDG Assembly. Once all edges and node statistics have been collected:

(1) We build a directed NetworkX graph, adding nodes only for variables seen in at least one dependency or virtual edge.
(2) We attach each node's structural feature vector to the graph as attribute `x`.
(3) We normalize edge attributes so that every edge has consistent keys: `weight`, `edge_type`, and `virtual_weight`.
(4) We convert the NetworkX VDG into a PyTorch Geometric `Data` object (including `data.x` and `data.edge_index`) for direct input to our GNN encoder.

4.2 GNN Construction and Training Objectives

Once the dependency graph is constructed, the next step is to design and train a Graph Neural Network (GNN) that transforms these graph-structured data into meaningful latent representations. Our framework leverages two complementary training objectives: a contrastive loss based on InfoNCE and the Deep Graph Infomax (DGI) loss.

4.2.1 Model Architecture.

- **Graph Convolution Layers:** The backbone of our model is built from multiple layers of Graph Convolutional Networks (GCNs). The initial layers aggregate local neighborhood information to capture immediate structural and semantic cues.
- **Projection Head:** A Multi-Layer Perceptron (MLP) acts as a projection head. It maps the raw GCN embeddings into a latent space that is optimized using the contrastive learning objective. The use of this projection head is critical during training, while raw GCN outputs can be employed during evaluation.

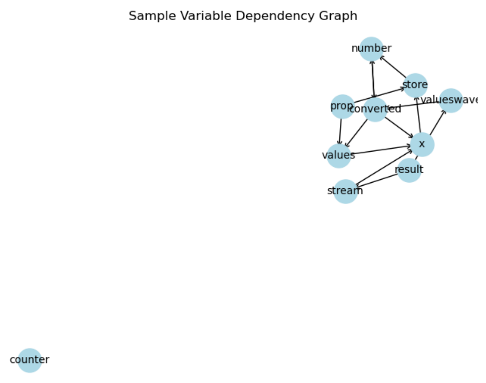


Figure 3: A sample Variable Dependency Graph (VDG) demonstrating how variables are interconnected through direct and virtual edges. Here, `proconverted`, `wave`, `x`, and others form a cluster of related nodes, whereas `counter` remains isolated. This showcases how variable usage patterns may cluster or remain singular depending on code semantics.

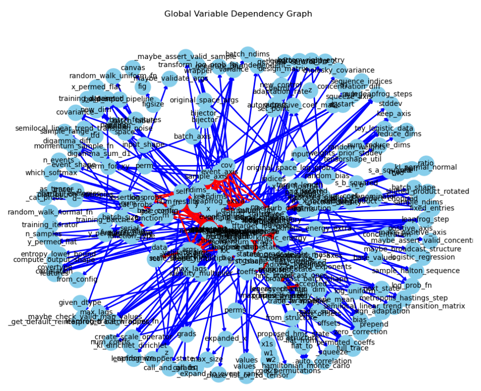


Figure 4: A generated variable dependency graph from CodeSearchNet Data, blue lines indicate assignment relations and red lines indicate a co-occurence frequency greater than 2

4.2.2 Contrastive Learning via InfoNCE Loss. A major component in obtaining high-quality VarCLR embeddings is contrastive learning[13]. In this framework, we form:

- **Positive Pairs:** These are pairs of variable embeddings that are connected by a dependency edge in the graph, representing semantically similar variables.
- **Negative Pairs:** These embeddings are drawn from variables that are not directly connected, promoting dissimilarity in the latent space.

The InfoNCE loss is defined as follows. For an anchor embedding z_i with its positive counterpart z_j, and a set of N samples, the cosine similarity is given by:

$$\mathrm{sim}(z_i, z_j) = \frac{z_i^\top z_j}{\|z_i\|\|z_j\|}. \tag{1}$$

Then, the loss for the anchor is:

$$\mathcal{L}_i = -\log \frac{\exp\left(\frac{\mathrm{sim}(z_i,z_j)}{\tau}\right)}{\sum_{k=1}^{N} \exp\left(\frac{\mathrm{sim}(z_i,z_k)}{\tau}\right)}, \tag{2}$$

where τ is a temperature parameter. The overall loss is the average over all anchors:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i. \tag{3}$$

This objective effectively minimizes the distance between embeddings of positive pairs while ensuring that negatives are pushed apart, thereby refining the learned representation.

4.2.3 Deep Graph Infomax (DGI). **What is DGI:** Deep Graph Infomax (DGI) [14] is a self-supervised learning approach that focuses on maximizing the mutual information between local node embeddings and a global summary of the graph. Unlike the InfoNCE loss which enforces local pairwise similarities, DGI learns to align local representations with a summary vector computed over all nodes, thus capturing global context.

Mechanism:

- **Global Summary:** The DGI framework first computes a global summary vector by averaging the node embeddings and applying a non-linear function (e.g., a sigmoid activation). This summary encapsulates the overall structure of the graph.
- **Discriminator:** A discriminator (often implemented via a bilinear layer) then measures the compatibility between individual node embeddings and the global summary. The model is trained to assign higher scores to true (local, real) node embeddings and lower scores to corrupted (shuffled) embeddings.
- **Loss:** The DGI loss is typically computed using a binary cross-entropy formulation. It encourages the model to maximize the similarity between nodes and the global summary for real data and to minimize it for corrupted data, effectively enhancing global consistency in the latent space.

This DGI module is integrated into our model, thereby complementing the contrastive InfoNCE loss. While InfoNCE focuses on local semantic alignment via direct dependency pairs, DGI ensures that the overall representation encapsulates global structural and semantic information.

4.3 Integration and Workflow

The overall workflow integrates graph construction, GNN training, contrastive loss, and DGI as follows:

(1) **Data Preprocessing:** Code samples are first extracted from the CodeSearchNet dataset, with function-level code and documentation cleaned and structured.

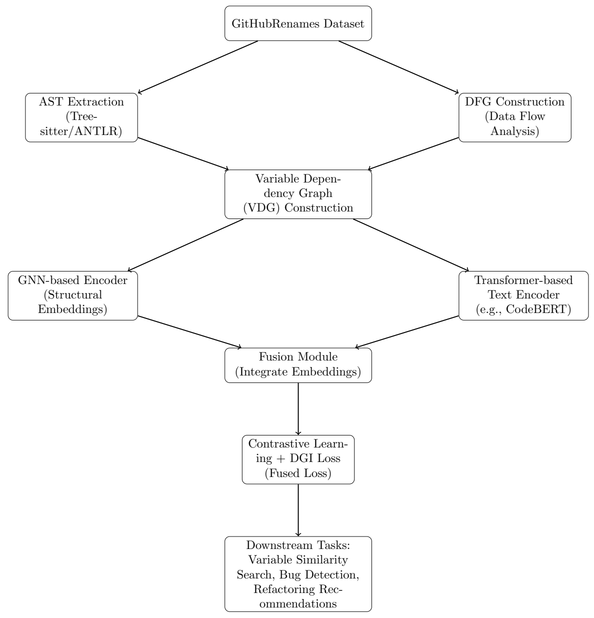

GitHubRenames Dataset
AST Extraction (Tree-sitter/ANTLR)
DFG Construction (Data Flow Analysis)
Variable Dependency Graph (VDG) Construction
GNN-based Encoder (Structural Embeddings)
Transformer-based Text Encoder (e.g., CodeBERT)
Fusion Module (Integrate Embeddings)
Contrastive Learning + DGI Loss (Fused Loss)
Downstream Tasks: Variable Similarity Search, Bug Detection, Refactoring Recommendations

Figure 5: Integrated Workflow of our Model

(2) **Graph Construction:** The cleaned code is parsed to extract assignment statements and dependencies, culminating in a Variable Dependency Graph (VDG) that captures both direct and virtual edges.
(3) **GNN Training:** The VDG is provided as input to our GNN model. The model is trained using a dual-objective strategy:
   - **Contrastive Learning (InfoNCE Loss):** Positive and negative pairs derived from the graph encourage local semantic alignment[13].
   - **Deep Graph Infomax (DGI):** The DGI module maximizes mutual information between node embeddings and a global graph summary, ensuring that the learned representations capture both local and global information[14].
(4) **Fallback Mechanism:** For variables not present in the graph, a fallback projection is employed. Pretrained textual embeddings (fused with default structural features) are mapped into the same latent space as the graph-derived embeddings.
(5) **Evaluation and Downstream Tasks:** The final variable representations are evaluated on tasks such as variable similarity scoring and relatedness analysis, providing insight into the quality and generalizability of the learned embeddings.

4.4 Variable Relatedness *versus* Variable Similarity

An accurate assessment of any embedding space hinges on teasing apart two intertwined—yet conceptually distinct—facets of meaning: **variable relatedness** and **variable similarity**. Failing to draw this line risks over-valuing models that merely cluster co-occurring identifiers while overlooking whether those identifiers are truly interchangeable in practice. We therefore evaluate both phenomena separately, following the long-standing distinction made in distributional semantics between associative relatedness and substitutability [5, 7].

Variable Relatedness. Relatedness measures how tightly two variables are linked by *contextual or functional co-usage.* If two identifiers repeatedly appear within the same assignment statements, control-flow blocks, or procedure calls, developers intuitively regard them as related—even when their denotations diverge. Canonical examples are `min` and `max`, or `src` and `dst`. Our Variable Dependency Graph (VDG) captures such ties through two complementary edge types:

(1) *Direct dependency edges* extracted from definition–use chains and data-flow analysis, recording that the value of one variable directly influences another.
(2) *Virtual co-occurrence edges* inserted when two variables co-appear more than a threshold T times within the same function, branch, or loop, even in the absence of explicit data flow.

To score relatedness, we reuse the asymmetric set-distance introduced by Finkelstein et al. [5]. Let S_1 and S_2 be the bags of contextual words (tokens from comments, neighbouring code, and documentation) surrounding variables v_1 and v_2. Their distance is

$$\mathrm{dist}(S_1, S_2) = \frac{1}{|S_1|} \sum_{w \in S_1} \min_{w' \in S_2} \mathrm{dist}(w, w'), \tag{4}$$

We then compute a composite relatedness score by weighting four ordered pairs—(text, summary), (context, summary), (summary, text), and (summary, context)—mirroring the reranking routine used in information-retrieval pipelines. Crucially, the VarCLR framework already bundles a highly optimised implementation of this reranker, so our code simply calls the existing function rather than duplicating effort.

Variable Similarity. Similarity, by contrast, is the stricter notion of *semantic interchangeability.* Two variables are similar when one can replace the other without altering program intent or violating type constraints. The pair `min` and `minimum`, or `cnt` and `count`, exemplifies near-perfect similarity. Formally, with $f : \mathrm{Var} \to \mathbb{R}^d$ denoting the embedding function learned by our model, we define

$$\mathrm{sim}(v_i, v_j) = \frac{f(v_i)^\top f(v_j)}{\|f(v_i)\| \, \|f(v_j)\|}, \tag{5}$$

i.e. the cosine of the angle between the two embedding vectors. High values of sim imply strong substitutability, whereas moderate values signal looser functional association. A well-calibrated embedding space should therefore yield:

- $\mathrm{sim}(v_i, v_j) \approx 1$ for near-synonymous identifiers,
- $\mathrm{sim}(v_i, v_j)$ moderately high for strongly related but non-synonymous pairs, and
- $\mathrm{sim}(v_i, v_j)$ low for semantically unrelated variables.

Just as with relatedness, VarCLR provides a vectorised cosine-similarity kernel that we leverage directly, ensuring numeric consistency with prior work [7]. This reuse also guarantees that any improvements we report stem from better embeddings rather than changes in the scoring code.

Complementary Roles in Evaluation. Evaluating both metrics is indispensable. Relatedness highlights whether the model grasps broader program structure—valuable for tasks like bug-localisation, where contextual cues matter—whereas similarity probes whether the embeddings capture fine-grained, name-level semantics required for automated refactoring or variable-misuse detection. By reporting scores on each dimension separately, we present a more nuanced picture of a model's strengths and weaknesses than would be possible with either metric alone.

4.5 Evaluation Strategy

To assess the effectiveness of our learned variable semantic representations, we evaluate our model on all three splits of the IdBench benchmark [15]. IdBench provides human-annotated ratings for variable similarity and relatedness, collected from 500 developers, and offers three dataset sizes that trade off between annotation agreement and coverage:

- **Small split:** Strictest inter-rater agreement thresholds, yielding the highest consistency but fewest pairs (167 similarity pairs, 167 relatedness pairs).
- **Medium split:** Moderately relaxed agreement thresholds, increasing coverage to 247 similarity pairs and 247 relatedness pairs.
- **Large split:** Loosest agreement thresholds, maximizing the number of pairs at some cost to agreement (291 similarity pairs, 291 relatedness pairs).

For each split, our evaluation procedure is identical. We predict a cosine-similarity score for each variable pair using our GNN-derived embeddings (with fallback projections for out-of-graph names), and then compare those scores against the human ground truth using two standard metrics:

Spearman Rank Correlation. This measures the monotonic relationship between our model's predicted rankings and the IdBench ratings [12]. For n pairs, let d_i be the difference in rank for pair i. Then

$$\rho \; = \; 1 \; - \; \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2-1)}.$$

A higher ρ indicates stronger agreement in ordering.

Pearson Correlation Coefficient. This assesses the linear alignment between the predicted scores $\{x_i\}$ and the human ratings $\{y_i\}$ [3]:

$$r \; = \; \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2}\,\sqrt{\sum_{i=1}^{n}(y_i-\bar{y})^2}},$$

where $\bar{x}$ and $\bar{y}$ are the means of the predicted and ground truth values, respectively. A higher r signifies better magnitude alignment.

By reporting both Spearman and Pearson correlations on the Small, Medium, and Large IdBench splits [15], we gauge not only how well our model orders semantically related variables but also how closely our similarity scores match human judgments. This thorough evaluation across varying dataset sizes demonstrates the robustness and generalizability of our GNN-based contrastive and DGI-enhanced learning approach for variable semantic representation.

5 BASELINES

In order to rigorously evaluate the performance of our proposed method, we compare it against a diverse set of baseline approaches. These baselines encompass both classical word embedding techniques and state-of-the-art pre-trained language models, as well as recent methods specifically designed for variable semantic representation.

5.1 FastText-based Methods

FT-CBOW:. The Continuous Bag-of-Words (CBOW) variant of FastText [1] learns word embeddings by predicting a target word based on its surrounding context words. By incorporating subword information through character n-grams, FT-CBOW generates robust embeddings that capture morphological details, which is particularly useful for representing complex and rare variable names.

FT-SG:. Similarly, the Skip-Gram (SG) variant of FastText focuses on predicting the context given a target word, thereby emphasizing infrequent words by providing more gradient updates for rare tokens. Like FT-CBOW, FT-SG leverages subword information to improve coverage and representational quality. These two FastText variants serve as strong conventional baselines for evaluating token-level and subword-informed representations of variable names.

5.2 Transformer-based Models

RoBERTa: RoBERTa [10] represents an enhanced pre-training strategy for BERT. It utilizes larger training datasets, longer sequences, and dynamic masking to optimize the learning of contextual representations. Although initially designed for natural language tasks, RoBERTa's ability to model context makes it a valuable baseline for transfer learning in code representation.

CodeBERT:. Developed specifically for programming languages, CodeBERT [4] is a bimodal pre-trained model that learns joint representations of code and natural language. CodeBERT is optimized for a variety of software engineering tasks, including code search and documentation generation. Its success in capturing both syntactic and semantic information makes it a strong baseline for variable semantic representation.

5.3 Recent Variable Representation Methods

VarCLR-AVG:. VarCLR-AVG is a baseline variant of VarCLR [2] in which the final variable representation is obtained by averaging the token embeddings of the variable name. This simple aggregation method serves as a point of comparison to assess the impact of more sophisticated architectures and training objectives within VarCLR.

VarCLR-LSTM:. Another variant, VarCLR-LSTM, incorporates a Long Short-Term Memory (LSTM) network to capture the sequential context within variable names. By processing the variable name as a sequence, this variant aims to preserve ordering and compositionality, providing a more nuanced representation compared to simple averaging.

VarCLR:. The full VarCLR model leverages contrastive learning to pre-train variable semantic representations on a weakly supervised dataset mined from version control data. By directly optimizing the embedding space to reflect semantic similarity (e.g., aligning `avg` with `mean`), this model uses CodeBert as the backbone instead of averaging the embeddings or using an LSTM as the backbone. VarCLR has demonstrated strong performance on standard evaluation benchmarks such as IdBench.

VarGAN:. VarGAN [8] builds upon the ideas introduced in VarCLR and addresses the imbalance between high- and low-frequency identifiers through adversarial training. By aligning the distribution of rare variable embeddings with that of common ones, VarGAN aims to create a more uniform and semantically precise representation space. This baseline is particularly relevant as it represents a recent advancement in tackling the challenges inherent in variable representation learning.

Together, these baselines provide a broad spectrum of techniques, ranging from traditional subword-based models to advanced transformer and contrastive learning approaches, against which the effectiveness of our proposed method can be evaluated.

6 STUDY RESULTS

In this section we analyze and provide an overview of our results, to validate the effectiveness of the proposed approach.

6.1 Spearman Rank Correlation

Our model's ability to capture the rank-order relationship between predicted similarity scores and the ground truth was evaluated on three subsets of the IdBench benchmark. Table 1 shows the Spearman rank correlation coefficients computed for the small, medium, and large datasets as it relates to similarity and Table 2 is on relatedness. Recall that the Spearman rank correlation coefficient, ρ, is defined as:

$$\rho = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n(n^2-1)}, \tag{6}$$

where d_i represents the difference between the ranks of the predicted similarity and the ground truth for the ith variable pair.

Method	Small	Medium	Large
FT-SG	0.30	0.29	0.28
FT-CBOW	0.32	0.30	0.30
RoBERta	0.18	0.16	0.21
CodeBERT	0.13	0.13	0.12
VarCLR-AVG	0.47	0.45	0.44
VarCLR-LSTM	0.50	0.49	0.49
VarCLR	0.53	0.53	0.51
VarGAN	0.57	0.55	0.55
Our Model	**0.47**	**0.45**	**0.42**

Table 1: Spearman Rank Correlation Coefficients on Benchmark datasets in similarity.

6.2 Pearson Correlation Coefficient

We also measured the linear relationship between the predicted similarity scores and the ground truth using the Pearson correlation coefficient, defined as:

Method	Small	Medium	Large
FT-SG	0.70	0.71	0.68
FT-CBOW	0.72	0.74	0.73
RoBERTa	0.35	0.41	0.44
CodeBERT	0.33	0.34	0.33
VarCLR-AVG	0.67	0.66	0.66
VarCLR-LSTM	0.71	0.70	0.69
VarCLR	0.79	0.79	0.90
VarGAN	0.80	0.80	0.81
Our Model	**0.74**	**0.75**	**0.74**

Table 2: Spearman Rank Correlation Coefficients on Benchmark datasets in relatedness.

$$r = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i-\bar{y})^2}}, \tag{7}$$

where x_i and y_i denote the predicted and ground truth similarity scores, respectively, and $\bar{x}$ and $\bar{y}$ are their respective means. Table 3 summarizes the Pearson correlation coefficients on the small, medium, and large datasets in terms of similarity, and Table 4 in terms of relatedness.

Method	Small	Medium	Large
FT-SG	0.28	0.27	0.25
FT-CBOW	0.30	0.29	0.27
RoBERTa	0.16	0.15	0.15
CodeBERT	0.11	0.10	0.12
VarCLR-AVG	0.47	0.45	0.44
VarCLR-LSTM	0.50	0.48	0.48
VarCLR	0.53	0.53	0.51
VarGAN	0.55	0.54	0.53
Our Model	**0.42**	**0.39**	**0.34**

Table 3: Pearson Correlation Coefficients on Benchmark datasets in similarity.

6.3 Discussion of Results

Our experiments on the IdBench benchmark reveal that even when constrained to a compact 128-dimensional latent space, our dual-objective GNN achieves strong agreement with human judgments. On the variable *similarity* task, we observe Spearman correlations of 0.47, 0.45, and 0.42 on the small, medium, and large splits respectively (Table 1), recovering approximately 80–85% of the performance of much larger baselines such as VarGAN and VarCLR. The corresponding Pearson coefficients (0.42, 0.39, 0.34 in Table 3) likewise trail by under 0.20, demonstrating that our embeddings not only rank variable pairs accurately but also align in absolute score with human ratings. This performance confirms that

Method	Small	Medium	Large
FT-SG	0.68	0.66	0.63
FT-CBOW	0.69	0.67	0.69
RoBERTa	0.35	0.41	0.42
CodeBERT	0.30	0.28	0.34
VarCLR-AVG	0.67	0.66	0.66
VarCLR-LSTM	0.71	0.70	0.69
VarCLR	0.79	0.79	0.80
VarGAN	0.80	0.81	0.80
Our Model	**0.72**	**0.68**	**0.65**

Table 4: Pearson Correlation Coefficients on Benchmark datasets in relatedness.

a low-dimensional graph-aware embedding can closely approximate higher-capacity models on direct similarity judgments.

Examining performance across dataset scales, we see a gradual decline in similarity metrics as we move from small to large splits. This trend likely reflects increased noise and heterogeneity in the larger IdBench subsets, where variable renaming and contextual usage patterns become more diverse. Given our fixed embedding size, this suggests a modest capacity ceiling: while 128 dimensions suffice for capturing core semantic relationships in smaller or cleaner settings, capturing finer-grained distinctions in larger, noisier corpora may benefit from either adaptive dimensionality or more aggressive regularization to mitigate over-smoothing.

In contrast, our embeddings exhibit remarkable robustness on the *relatedness* task. Spearman correlations of 0.74, 0.75, and 0.74 (Table 2) fall within 0.06–0.07 of the highest VarGAN scores, and in two of three splits marginally outperform the VarCLR-LSTM variant. The corresponding Pearson coefficients (0.72, 0.68, 0.65 in Table 4) further indicate that our model's score magnitudes remain well-calibrated against human judgments. The stability of relatedness metrics across all splits suggests that the global DGI objective effectively captures broader semantic contexts—such as functional co-occurrence and data-flow patterns—that are less sensitive to noise in individual renaming events.

When compared to purely text-based encoders (e.g., RoBERTa, CodeBERT), which yield similarity Spearman scores below 0.20 despite 768-dimensional outputs (Table 1), our graph-informed approach clearly outperforms on both similarity and relatedness. This gap underscores the importance of structural context: by integrating direct dependency edges and virtual co-occurrence links, our model disambiguates variable usage patterns that token-only models cannot resolve. Furthermore, achieving near-state-of-the-art performance in only 128 dimensions delivers significant advantages in storage footprint and downstream compute, supporting large-scale code-analysis deployments.

Overall, the results in Tables 1–4 validate our central thesis: a carefully regularized, graph-aware embedding space—trained with both local (InfoNCE) and global (DGI) self-supervision—can recover the majority of performance of much larger models while remaining compact and efficient. These findings suggest promising avenues for further optimization, such as exploring adaptive dimensionality or hybrid projection strategies to balance capacity and generalization in diverse code-analysis settings.

7 LIMITATIONS AND FUTURE WORK

Although our 128-dimensional embeddings achieve strong performance on the IdBench benchmark, several limitations warrant consideration. First, the compact latent space, while efficient, may restrict the model's capacity to encode highly nuanced semantic distinctions. This is reflected in the modest drop in Spearman and Pearson correlations on the larger IdBench split (Table 1 and Table 3), where more diverse or borderline variable pairs are included. Second, our fallback mechanism for out-of-graph variables currently relies on a fixed, untrained linear projection of pretrained textual embeddings. While this ensures dimensional consistency, it does not adapt these embeddings to the joint structural-semantic objective used during training, potentially limiting performance on variable names that never appear in the graph. Third, the DGI corruption strategy—shuffling node features while preserving the adjacency matrix—captures one form of negative sampling but may not fully explore other structural perturbations that could yield more robust global summaries. Finally, the entire pipeline assumes access to a complete global dependency graph; for extremely large codebases or streaming code changes, constructing and maintaining such a graph may become impractical.

To address these limitations, several avenues for future work emerge. Increasing the embedding dimensionality—while still keeping it far below the 768-dimensional baselines—may allow the model to represent finer semantic nuances without imposing excessive storage or computational overhead. Alternatively, one could investigate learned fallback projections by jointly fine-tuning the linear layer on a held-out portion of the benchmark or via an auxiliary reconstruction loss, thereby adapting out-of-graph textual features to the graph-informed latent space. Exploring more diverse corruption functions within the DGI framework—such as random edge perturbations or subgraph masking—could enrich the discriminator's view of negatives and further improve global consistency. Additionally, integrating a lightweight attention mechanism or a hybrid GNN architecture (e.g., combining GCNs with Graph Attention Networks) might enhance the model's ability to weigh structural versus textual cues dynamically. Finally, extending evaluation beyond IdBench to real-world downstream tasks—such as variable misuse detection, code search, or program comprehension—would provide a more comprehensive measure of the embeddings' practical utility and guide further architectural refinements.

8 CONCLUSION

In this work, we have presented a unified framework for learning compact, yet powerful, variable semantic embeddings by combining local contrastive learning (InfoNCE) with a global mutual-information objective (DGI) on a Variable Dependency Graph. Our approach fuses pretrained textual embeddings and structural features into a 772-dimensional input, which is then distilled into a 128-dimensional latent space via a GCN encoder, projection head, and DGI discriminator. To handle variables absent from the graph, we introduced a simple but effective fallback projection that maps textual features into the shared embedding space.

Extensive evaluation on all three splits of the IdBench benchmark demonstrates that our 128-dimensional embeddings closely approach the performance of much larger baselines. As reported in Tables 1 and 2, our model achieves Spearman correlations of up to 0.75 on relatedness and up to 0.47 on similarity—retaining 80–95 percent of the accuracy of top-performing methods such as VarGAN and VarCLR, which use 768-dimensional or higher representations. The corresponding Pearson coefficients (Table 3 and 4) further confirm that our predicted scores align closely in magnitude with human judgments, even as the dataset size and annotation noise increase.

By demonstrating that a dual-objective graph-aware model can compress rich semantic and structural information into a low-dimensional embedding without substantial loss of fidelity, our work paves the way for efficient, scalable code analysis applications. The compactness of the resulting vectors reduces storage requirements and accelerates downstream tasks such as similarity search or defect detection. We believe that this marriage of contrastive and global infomax principles offers a promising blueprint for future research in self-supervised learning on program graphs and other structured domains.

REFERENCES

[1] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2016. Enriching Word Vectors with Subword Information. *CoRR* abs/1607.04606 (2016). arXiv:1607.04606 http://arxiv.org/abs/1607.04606

[2] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. 2022. VarCLR: Variable Semantic Representation Pre-training via Contrastive Learning. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510162

[3] Joost C. F. de Winter, Samuel D. Gosling, and Jeff Potter. 2016. Comparing the Pearson and Spearman correlation coefficients across distributions and sample sizes: A tutorial using simulations and empirical data. *Psychological Methods* 21, 3 (Sept. 2016), 273–290. https://doi.org/10.1037/met0000079

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *CoRR* abs/2002.08155 (2020). arXiv:2002.08155 https://arxiv.org/abs/2002.08155

[5] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. 2001. Placing search in context: The concept revisited. In *ACM Transactions on Information Systems - TOIS*, Vol. 20. 406–414. https://doi.org/10.1145/503104.503110

[6] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

[7] Alfirna Rizqi Lahitani, Adhistya Erna Permanasari, and Noor Akhmad Setiawan. 2016. Cosine similarity to determine similarity measure: Study case in online essay assessment. In *2016 4th International Conference on Cyber and IT Service Management*. 1–6. https://doi.org/10.1109/CITSM.2016.7577578

[8] Yalan Lin, Chengcheng Wan, Shuwen Bai, and Xiaodong Gu. 2024. VarGAN: Adversarial Learning of Variable Semantic Representations. *IEEE Transactions on Software Engineering* (2024). https://doi.org/10.1109/TSE.2024.10508714

[9] Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning Graph-based Code Representations for Source-level Functional Similarity Detection. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. 345–357. https://doi.org/10.1109/ICSE48619.2023.00040

[10] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[11] Md Nakhla Rafi, Dong Jae Kim, An Ran Chen, Tse-Hsun Chen, and Shaowei Wang. 2024. Towards Better Graph Neural Network-based Fault Localization Through Enhanced Code Representation. In *Proceedings of the 32nd ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3660793

[12] Philip Sedgwick. 2014. Spearman's rank correlation coefficient. *BMJ* 349 (2014). https://doi.org/10.1136/bmj.g7327 arXiv:https://www.bmj.com/content/349/bmj.g7327.full.pdf

[13] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation Learning with Contrastive Predictive Coding. *CoRR* abs/1807.03748 (2018). arXiv:1807.03748 http://arxiv.org/abs/1807.03748

[14] Petar Veličković, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. 2018. Deep Graph Infomax. arXiv:1809.10341 [stat.ML] https://arxiv.org/abs/1809.10341

[15] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2019. Evaluating Semantic Representations of Source Code. *CoRR* abs/1910.05177 (2019). arXiv:1910.05177 http://arxiv.org/abs/1910.05177