VarCLR: Variable Semantic Representation Pre-training via Contrastive Learning

Qibin Chen
qibinc@cs.cmu.edu
Carnegie Mellon University

Jeremy Lacomis
jlacomis@cs.cmu.edu
Carnegie Mellon University

Edward J. Schwartz
eschwartz@cert.org
Carnegie Mellon University Software Engineering Institute

Graham Neubig
gneubig@cs.cmu.edu
Carnegie Mellon University

Bogdan Vasilescu
bogdanv@cs.cmu.edu
Carnegie Mellon University

Claire Le Goues
clegoues@cs.cmu.edu
Carnegie Mellon University

ABSTRACT

Variable names are critical for conveying intended program behavior. Machine learning-based program analysis methods use variable name representations for a wide range of tasks, such as suggesting new variable names and bug detection. Ideally, such methods could capture semantic relationships between names beyond syntactic similarity, e.g., the fact that the names `average` and `mean` are similar. Unfortunately, previous work has found that even the best of previous representation approaches primarily capture "relatedness" (whether two variables are linked at all), rather than "similarity" (whether they actually have the same meaning).

We propose VarCLR, a new approach for learning semantic representations of variable names that effectively captures variable similarity in this stricter sense. We observe that this problem is an excellent fit for *contrastive learning*, which aims to minimize the distance between explicitly similar inputs, while maximizing the distance between dissimilar inputs. This requires labeled training data, and thus we construct a novel, weakly-supervised variable renaming dataset mined from GitHub edits. We show that VarCLR enables the effective application of sophisticated, general-purpose language models like BERT, to variable name representation and thus also to related downstream tasks like variable name similarity search or spelling correction. VarCLR produces models that significantly outperform the state-of-the-art on IdBench, an existing benchmark that explicitly captures variable similarity (as distinct from relatedness). Finally, we contribute a release of all data, code, and pre-trained models, aiming to provide a drop-in replacement for variable representations used in either existing or future program analyses that rely on variable names.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; • **Computing methodologies → Learning latent representations**; *Natural language processing*; *Neural networks.*

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510162

ACM Reference Format:
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 *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510162

1 INTRODUCTION

Variable names convey key information about code structure and developer intention. They are thus central for code comprehension, readability, and maintainability [7, 46]. A growing array of automatic techniques make use of variable names in the context of tasks like but not limited to bug finding [64, 68] or specification mining [88]. Beyond leveraging the information provided by names in automated tools, recent work has increasingly attempted to directly suggest good or improved names, such as in reverse engineering [35, 45] or refactoring [3, 5, 52].

Developing (and evaluating) such automated techniques (or *name-based analyses* [75]) relies in large part on the ability to model and reason about the relationships between variable names. For concreteness, consider an analysis for automatically suggesting names in decompiled code. Given a compiled program (such that variable names are discarded) that is then decompiled (resulting in generic names like `a1`,`a2`), a renaming tool seeks to replace the generic decompiler-provided identifiers with more informative variable names for the benefit of reverse engineers aiming to understand it. Good names in this context are presumably closely related to the names used in the original program (before the developer-provided names were discarded). A variable originally named `max`, for example, and then decompiled to `a2`, should be replaced with a name at least close to `max`, like `maximum`. Modeling this relationship well is key for both constructing and evaluating such analyses.

Accurately capturing and modeling these relationships is difficult. A longstanding approach has used syntactic difference — like various measures of string edit distance — to estimate the relationship between two variables (such as for spellchecking [18]). However, syntactic distance is quite limited in capturing underlying name semantics. For example, the pairs (`minimum`, `maximum`) and (`minimum`, `minimal`) are equidistant syntactically — with a Levenshtein distance of two — but `maximum` and `minimum` are antonyms.

More recent work has sought to instead encode variable name semantics using neural network embeddings, informing a variety of name-based analyses [29, 65, 78]. Unfortunately, although state-of-the-art techniques for variable name representation better capture *relatedness*, they still struggle to accurately capture variable name *similarity*, in terms of how interchangeable two names are. Variables may be related for a variety of reasons. While **`maximum`** and **`minimum`** are highly related, they certainly cannot be substituted for one another in a code base. **`minimum`** and **`minimal`**, on the other hand, are both related and very similar. In recent work, Wainakh et al. [75] presented a novel dataset, IdBench, based on a human survey on variable similarity and interchangeability, and used it to evaluate state-of-the-art embedding approaches. They empirically established that there remains significant room for improvement in terms of capturing similarity rather than merely relatedness.

In this paper, we formulate the *variable semantic representation learning problem* as follows: given a set of variable data, learn a function f that maps a variable name string to a low-dimensional dense vector that can be used in a variety of tasks (like the types of name-based analyses discussed above). To be useful, such a mapping function should effectively encode *similarity*, i.e., whether two variables have the same meaning. That is, $f(\texttt{minimum})$ and $f(\texttt{minimal})$ should be close to one another. Importantly, however, the function should also ensure that variable names that are *not* similar (regardless of relatedness!) are *far* from one another. That is, $f(\texttt{minimum})$ and $f(\texttt{maximum})$ should be distant.

Our first key insight is that this problem is well suited for a contrastive-learning approach [14, 30, 63, 82]. Conceptually, contrastive learning employs encoder networks to encode instances (in this task, variables) into representations (i.e., hidden vectors), with a goal of minimizing the distance between (the representation of) similar instances and maximizing the distance between (the representation of) dissimilar instances. Contrastive learning requires as input a set of "positive pair" examples—of similar variables, in our case—for training.

Our second key insight is that we can construct a suitable weakly-supervised dataset of examples of similar variables by taking advantage of large amounts of source control information on GitHub. Following the definition of "similarity" from prior work [60, 75], we consider two variable names are similar if they have the same meaning, or are *interchangeable*. We therefore automatically mine source control edits to identify historical changes where developers renamed a variable but did not otherwise overly modify the code in which it was used. Although potentially noisy, this technique matches an intuitive understanding of variable name similarity in terms of interchangeability, and allows for the collection of a large dataset, which we call GitHubRenames.

Finally, we observe that the variable semantic representation learning problem requires more powerful neural architectures than `word2vec`-based approaches [8, 59, 75].[1] Such approaches are limited both empirically (as Wainakh et al. showed) and conceptually; note for example that they cannot capture component ordering, such as the difference between **`idx_to_word`** and **`word_to_idx`**. Meanwhile, Pre-trained Language Models (PLMs) [9, 20, 67] based on the powerful Transformer architecture [74] have achieved the state-of-the-art on a wide range of natural language processing tasks, including text classification [20], question answering and summarization [48], and dialog systems [1]. PLMs tailored specifically for programming languages such as CodeBERT [22] and Codex [12] are useful in a variety of tasks such as code completion, repair, and generation [12, 55], though not yet for variable name representation. Encouragingly, previous work shows that contrastive learning can strongly improve BERT sentence embeddings for textual similarity tasks [24]. And, contrastive learning has been shown to benefit from deeper and wider network architectures [13].

We combine these insights to produce VarCLR, a novel machine learning method based on contrastive learning for learning general-purpose variable semantic representation encoders. In VarCLR, the contrastive learning element serves as a pre-training step for a traditional encoder. While powerful modern approaches like CodeBERT perform poorly on the variable representation problem off-the-shelf, we show that VarCLR-trained models dramatically outperform the previous state-of-the-art on capturing both variable similarity and relatedness. VarCLR is designed to be general to a variety of useful downstream tasks; we demonstrate its effectiveness for both the basic variable similarity/relatedness task (using the IdBench dataset as a gold standard baseline) as well as for variable similarity search, and spelling error correction.

To summarize, our main contributions are as follows:

(1) VarCLR, a novel method based on contrastive learning that learns general-purpose variable semantic representations suitable for a variety of downstream tasks.
(2) A new weakly supervised dataset, GitHubRenames, for better variable representation learning consisting of similar variable names collected from real-world GitHub data.
(3) Experimental results demonstrating that VarCLR's models significantly outperform state-of-the-art representation approaches on IdBench, an existing benchmark for evaluating variable semantic representations. These results further substantiate the utility of more sophisticated models like CodeBERT, with larger model capacity, in place of the previous `word2vec`-based methods for learning variable representations, while showing that the contrastive learning pre-training step is critical to enabling the effectiveness of such models.
(4) Experimental results that demonstrate that both unsupervised pre-training and our proposed weakly-supervised contrastive pre-training are indispensable parts for advancing towards the state-of-the-art, for the former takes advantage of greater *data quantity* by leveraging a huge amount of unlabeled data, while the latter takes advantage of better *data quality* with our new GitHubRenames dataset.

Finally, we contribute a release of all data, code, and pre-trained models, aiming to provide a drop-in replacement for variable representations used in either existing or future program analyses that rely on variable names.[2]

2 PROBLEM DOMAIN

Variable names critically communicate developer intent and are thus increasingly used by a variety of automated techniques as a central source of information. Such techniques increasingly rely on

[1] word2vec [59] is an embedding algorithm based on the *distributional hypothesis*, which assumes words that occur in the same contexts tend to have similar meanings.

[2] Code, data, and pre-trained model available at https://github.com/squaresLab/VarCLR.

machine learning and embedding-based representation approaches to encode variable name meaning for these purposes. However, recent work [75] shows that while neural embeddings based on techniques like `word2vec` do a better job of capturing relationships between variables than syntactic edit distance does, they still struggle to capture actual variable similarity in terms of their interchangeability. In this paper, we show that this problem is amenable to a contrastive learning approach, enabling accurate general-purpose representations of variable name semantics.

We define the *variable semantic representation learning problem* as follows: given a collection of suitable variable data, learn a function f that maps a variable name string to a low-dimensional dense vector that can be used to benefit various downstream tasks (like variable similarity scoring in the simplest case, or arbitrarily complex name-based analyses). A good mapping function f for variable name representations should:

(1) *Capture similarity.* f should encode *similar* names such that they are close to one another. Two names are similar when they have similar or generally interchangeable meanings, like **`avg`** and **`mean`**. This is especially important for variables that are *related* but *not similar*, such as **`maximum`** and **`minimum`**. Indeed, antonyms are often closely related and can appear in similar contexts (**`max`** and **`min`** for example may be used together in loops finding extrema).

(2) *Capture component ordering and importance.* Variables often consist of component words or sub-words. We observe that the order of such components can affect meaning. For example, **`idx_to_word`** and **`word_to_idx`** contain the same sub-words, but have different meanings. Moreover, the importance of different component words in a variable can be different and the importance of the same word can vary between variables. For example, in variables **`onAdd`** and **`onRemove`**, **`on`** is less important, while **`add`** and **`remove`** are more important. In **`turnOn`** and **`turnOff`**, **`on`** and **`off`** are more important than **`turn`**. A good mapping function f should be able to capture these differences, instead of treating variables as an unordered bag of sub-words.

(3) *Transferability.* The representation should be general-purpose and usable for a wide range of tasks. Benefits of a transferable, shared representation include the ability to (1) improve accuracy on unsupervised or data-scarce tasks, where it can be hard to obtain high-quality variable representations from scratch, and (2) for complex tasks consisting of many sub-tasks, make better use of labeled data from multiple sub-tasks via multi-task learning.

This formulation of the problem motivates our use of *contrastive learning*, which is an effective way to learn similarity from labeled data. Conceptually, given an encoder network f_θ and a set of similar "positive pairs", contrastive learning returns a *new* encoder that attempts to locate similar "positive pair" instances closer together and dissimilar "negative pair" instances farther apart. In practice, this can be accomplished by re-training the original encoder on a new pre-training task: instance discrimination [82]. Instance discrimination casts the contrastive learning problem as a classification problem where only the "positive pair" instances are equivalent. Rather than explicitly adjusting the distances between points, the

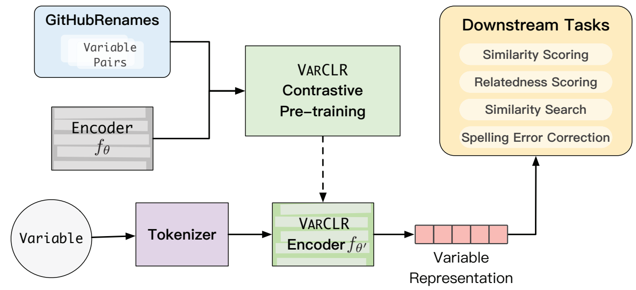


Figure 1: Conceptual overview of VarCLR.

encoder's parameters are trained to optimize its performance at discriminating similar instance from dissimilar instances. This naturally adjusts the parameters of the encoder such that similar instances are moved closer together (and vice-versa for dissimilar instances). The actual output of the contrastive learning process is a new encoder $f_{\theta'}$ that is identical to the original encoder in neural architecture, but has a different set of parameters θ' resulting from training on the instance discrimination task.

There are two central design choices in applying contrastive learning, however. First, *Which neural architectures should be used for f_θ?* This is usually decided by the problem domain in question. For example, in computer vision, ResNet [31] for learning image representations [13, 30, 63]; in natural language processing, Simple word embedding or BERT [20, 53] for learning sentence representations [24, 80]; and in data mining, Graph Neural Network [44] for learning graph representations [66]. Second, *How to construct similar (positive) and dissimilar (negative) training pairs?* Unsupervised data augmentation like cropping or clipping has been used to create different "views" of the same image as similar pairs in image processing [63, 72]; word dropout can augment text sentences for natural language processing [24]. For supervised contrastive learning, positive pairs can be created from labeled datasets directly [24], or via sampling instances from the same class [42]. Note that dissimilar pairs typically need not be explicitly defined. Instead, *in-batch negatives* [63] can be sampled from instance pairs that are not explicitly labeled as positive.

The choice of similar instances is very important, as it influences the learned similarity function and impacts downstream effectiveness [73]. For example, consider how training can lead to unintentional properties of a learned similarity function for `word2vec`. At a high level, `word2vec` [59] can be viewed as a form of unsupervised contrastive learning. It employs a word embedding layer as the encoder, and treats words co-occurring in the context window as similar pairs, while treating other words in the dictionary as dissimilar ones.[3] Due to its choice of "similar instances", it learns more of association (or relatedness) between words, instead of similarity in terms of how interchangeable two words are. For example, `word2vec` embeddings of cohyponym words such as red, blue, white, green are very close. While this might not be a problem in NLP applications, `word2vec` leads to unsatisfactory behavior

[3] We leave out the minor difference that `word2vec` produces two sets of embeddings, while contrastive learning usually uses a unified representation.

when applied to variable names [75], e.g., by identifying `minLength` and `maxLength` as similar.

3 METHOD

Figure 1 shows a high-level conceptual overview of VarCLR, our framework for learning effective semantic representations of variable names. VarCLR consists of a contrastive pre-training phase that takes two inputs: (1) a positive set of similar variable name pairs, and (2) an input encoder. The set of similar variables is crucial for VarCLR's performance. We thus produce GitHubRenames, a novel weakly-supervised dataset consisting of positive examples of similar variables by examining large amounts of source code history available from GitHub (Section 3.1). These variables must be suitably tokenized for encoding in a way that captures and retains relevant information (Section 3.2), both for pre-training and for downstream tasks. VarCLR also takes an input encoder f_θ with learnable parameters θ (Section 3.3). This encoder is then trained using contrastive learning (Section 3.4). The output of our framework is a contrastively-trained VarCLR encoder that converts tokenized variables into semantic representations suitable for a variety of tasks and name-based analyses, including similarity scoring or spelling error correction, among others.

3.1 Similar variables: GitHubRenames

A high-level definition of "similarity" [60, 75], is the degree to which two variables have the same meaning. Contrastive learning requires positive examples for training, and thus we need a set of appropriate positive pairs of similar variable names. As discussed in Section 2, these need not be manually constructed. Although IdBench [75] provides curated sets of human-judged "similar" variables, they are too small for training purposes (the largest set, has 291 variable pairs). This motivates an automated mechanism for constructing training data, with the added benefit that we need not be concerned about training and testing on the same dataset (as we use IdBench for evaluation).

Instead, we observe that one way to define variable similarity is to consider the degree to which two variables are explicitly *interchangeable* in code (close to IdBench's definition of "Contextual similarity"). We therefore collect a weakly supervised dataset of interchangeable variable names by mining source control version histories for commits where variable names change. These variable pairs are considered similar because they appear interchangeable in the same code context.

Concretely, we built upon existing open-source dataset collection code used to mine source control for the purpose of modeling changes [84].[4] Given a repository, this code mines all commits of less than six lines of code where a variable is renamed. The intuition is to look for commits that do not make large structural changes that might correspond to a major change in a variable's meaning. We applied dataset collection to an expanded version of the list of repositories used in ref [84], consisting of 568 C# projects.[5] The final GitHubRenames dataset contains 66,855 variable pairs, each consisting of a variable name before and after a renaming commit.

[4] https://github.com/microsoft/msrc-dpu-learning-to-represent-edits
[5] https://github.com/quozd/awesome-dotnet

The GitHubRenames dataset is only weakly supervised since developers were not asked to label variable pairs explicitly. The dataset may thus be noisy, and in particular we did not attempt to filter out renames corresponding to bug fixes. Indeed, we note that a number of pairs in GitHubRenames correspond to fixing spelling mistakes (Section 4.4). Overall, however, we note that our method transfers well to the IdBench validation set, and expect that more data will only improve VarCLR's effectiveness.

3.2 Input representation

A variable name as a text string must be preprocessed to be used as input to a neural network encoder. We observe two interesting aspects of variable names that inform our preprocessing. First, variable names are often composed of multiple words with interchangeable case styles, e.g., `max_iteration` vs `maxIteration`. Second, variable names are sometimes composed of short words or abbreviations, without an underscore or uppercase to separate them. e.g., `filelist`, `sendmsg`.

For the first problem, we apply a set of regex rules to canonicalize variable names into a list of *tokens*, e.g., `["max", "iteration"]`. The second problem is more challenging, and could cause Out-of-vocabulary (OOV) problems. To solve this, we use the pre-trained CodeBERT tokenizer [22], which is underlying a Byte Pair Encoding (BPE) model [70] trained on a large code corpus based on token frequencies. When encountering an unknown composite variable name such as `sendmsg`, it is able to split it into *subword tokens*, e.g., `["send", "##msg"]`, where `"##"` means this token is a suffix of the previous word.

3.3 Encoders

Generally, a neural encoder takes the input sequence, and encodes and aggregates information over the sequence to produce a hidden vector. That is, given a sequence of tokens $v = (v_1, v_2, \dots, v_n)$ corresponding to a tokenized variable name, an encoder outputs a hidden vector $\boldsymbol{h} \in \mathcal{R}^d$, where d is the dimension of the hidden representation:

$$\boldsymbol{h} = f_\theta(v), \tag{1}$$

f_θ denotes the encoder with learnable parameters θ.

Note that VarCLR is applicable to any encoder with this form. In this paper, we instantiate it specifically for Word Embedding Averaging (VarCLR-Avg), the LSTM Encoder (VarCLR-LSTM), and BERT (VarCLR-CodeBERT).

Word Embedding Averaging. Averaging the embeddings of input tokens is a simple but effective way to represent a whole input sequence, given sufficient data [80, 81]. Therefore, we consider this as a simple baseline encoder. Formally, given the tokenized variable name $v = (v_1, v_2, \dots, v_n), v_i \in \mathcal{V}$, and a word embedding lookup table $L_\theta : \mathcal{V} \to \mathcal{R}^d$:

$$\boldsymbol{h} = \frac{1}{n} \sum_{i=1}^{n} L_\theta(v_i), \tag{2}$$

where $\mathcal{V}$ is the vocabulary, i.e., the collection of all tokens the model can handle, $\theta \in \mathcal{R}^{|\mathcal{V} \times d|}$ is the learnable embedding matrix.

Although simple and efficient, this Word Embedding Averaging encoder suffers from two issues: 1) *Order*. The averaging operator

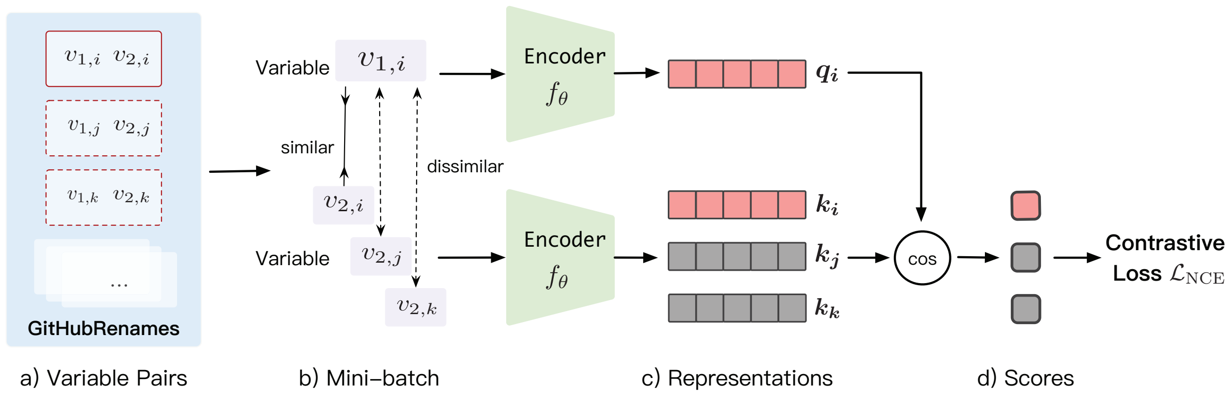


Figure 2: Overview of VarCLR's contrastive pre-training method. a) GitHubRenames contains interchangeable variable pairs. b) At each training step, sample a mini-batch of variable pairs, and aim to pull close the variables representations within a pair, e.g., $v_{1,i}$ and $v_{2,i}$, while pushing away the representations of other variables, e.g., $v_{1,i}$ and $v_{2,j}$. c) To achieve this, an encoder f_θ with learnable parameters θ, is adopted to encode the variable string to hidden vectors. d) contrastive loss is calculated based on the similarity scores as the cosine distance between encoded hidden vectors; the encoder f_θ is optimized with gradient descent.

discards word order information in the input sequence, and thus poorly represents variable names where this order is important, e.g., `idx_to_word` and `word_to_idx`. 2) *Token importance.* An unweighted average of word embeddings ignores the relative importance of words in a variable name, as well as the fact that the importance of a word can vary by context.

LSTM Encoder. Recurrent Neural Networks (RNNs) [69] generalize feed-forward neural networks to sequences. Given the tokenized variable name $v = (v_1, v_2, \ldots, v_n)$, a standard RNN computes a sequence of hidden vectors $(\boldsymbol{h_1}, \boldsymbol{h_2}, \ldots, \boldsymbol{h_n})$.

$$\boldsymbol{h_t} = \text{sigmoid}\left(W^{\text{hx}} L_{\theta_e}(v_t) + W^{\text{hh}} \boldsymbol{h_{t-1}}\right), \tag{3}$$

where $W^{\text{hx}}, W^{\text{hh}} \in \mathcal{R}^{d \times d}$ are weight matrices, and θ_e is the embedding matrix (as in Equation (2)). RNNs process the input sequence by reading in one token v_t at a time and combining it with the past context h_{t-1}. This captures sequential order information. After processing all input tokens, we can average the hidden states at each step to output a representation of the original variable:

$$\boldsymbol{h} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{h_i}. \tag{4}$$

We use bi-directional Long Short-Term Memory (LSTM) models [34], a variant of RNNs widely used in natural language processing. LSTMs introduce several new components, including the input and forget gates, controlling how much information flows from the current token, and how much to keep from past contexts, respectively. This better handles the token importance problem by dynamically controlling the weight of the input token at each step.

BERT. Transformer-based models [74] typically outperform LSTMs and are considered to be the better architecture for many NLP tasks. Pre-trained Language Models (PLMs), built upon Transformers, can leverage massive amounts of unlabeled data and computational resources to effectively tackle a wide range of natural language processing tasks. Useful PLMs for programming languages include CodeBERT [22] and Codex [12] PLMs not only capture component ordering and token importance that LSTMs do, but provide additional benefits: 1) BERT-based models are already pre-trained with self-supervised objectives such as Masked Language Modeling (MLM) [20] on a large amount of unlabeled data. It provides a good initialization to the model parameters and improves the model's generalization ability, requiring fewer data to achieve satisfactory performance [9]. 2) Transformer encoders are much more powerful than previous models thanks to the multi-head self-attention mechanism, allowing for the model to be much wider and deeper with more parameters. We therefore propose to use PLMs for programs as our most powerful choice of variable name encoder.

Effectiveness versus efficiency. Although BERT has the largest model capacity of these encoders, it also requires higher computation cost for both training and inference, and suffers from a longer inference latency. The trade-off posted between effectiveness and efficiency can vary according to different downstream applications. Therefore, we find it meaningful to compare all encoders in VarCLR. Different or better encoder models can be directly plugged into the VarCLR framework in the future. We omit further interior technical details of both LSTM and BERT models as they are beyond the scope of this paper.

3.4 Contrastive Learning Pre-training

VarCLR implements the design choices for input data, variable tokenization, and input encoder in a contrastive learning framework. Figure 2 provides an overview. Conceptually, contrastive learning uses encoder networks to encode instances (in this task, variables) into representations (i.e., hidden vectors), and aims to minimize the distance between similar instances while maximizing the distance between dissimilar instances.

Specifically, given a choice of encoder and set of labeled "positive pairs" of variable names, we use instance discrimination [82] as our pre-training task, and InfoNCE [63] as our learning objective. Given a mini-batch of encoded and L2-normalized representations of K similar variable pairs $\{(v_{1,i}, v_{2,i})|i = 1, \ldots, K\}$, we first encode them to hidden representations:

$$\boldsymbol{q}_{\boldsymbol{i}} = \frac{f_\theta(v_{1,i})}{\left\|f_\theta(v_{1,i})\right\|_2}, \tag{5}$$

$$\boldsymbol{k}_{\boldsymbol{i}} = \frac{f_\theta(v_{2,i})}{\left\|f_\theta(v_{2,i})\right\|_2}, \tag{6}$$

where $\|\cdot\|_2$ is ℓ_2-norm, f_θ denotes the encoder. Then, we define the InfoNCE loss as:

$$\mathcal{L}_{\text{NCE}}(\boldsymbol{q}, \boldsymbol{k}) = -\mathbb{E}\left(\log \frac{e^{\boldsymbol{q}_{\boldsymbol{i}}{}^\top \boldsymbol{k}_{\boldsymbol{i}}/\tau}}{\sum_{j=1}^{K} e^{\boldsymbol{q}_{\boldsymbol{i}}{}^\top \boldsymbol{k}_{\boldsymbol{j}}/\tau}}\right), \tag{7}$$

where τ is the temperature hyperparameter introduced by [82]. Intuitively, this objective encourages the model to discriminate the corresponding similar instance $v_{2,i}$ of an instance $v_{1,i}$ from other instances in the mini-batch $v_{2,j}$. This learning objective is very similar to the cross-entropy loss for classification tasks, while the difference is that instead of a fixed set of classes, it treats each instance as a distinct class. Following [26], we further make the loss symmetric and minimize the following objective function:

$$\mathcal{L} = \frac{1}{2}\mathcal{L}_{\text{NCE}}(\boldsymbol{q}, \boldsymbol{k}) + \frac{1}{2}\mathcal{L}_{\text{NCE}}(\boldsymbol{k}, \boldsymbol{q}). \tag{8}$$

In our task, this objective encourages the encoder to push the representations of a pair of similar variables to be close to each other, so that they can be discriminated from other variables.

We refer to this process as pre-training in the sense that the training is not intended for a specific task but is learning a general-purpose variable representation.

4 EXPERIMENTS

In this section, we evaluate VarCLR's ability to train models for variable representation along several axes. Section 4.1 addresses setup, datasets, and baselines common to the experiments. Then, we begin by addressing a central claim: How well do VarCLR models encode variable *similarity*, as distinct from *relatedness*? We answer this question by using pre-trained VarCLR models to compute similarity (and relatedness, resp) scores between pairs of variables, and evaluate the results on human-annotated gold standard ground truth benchmark (Section 4.2).

Next, we evaluate VarCLR-trained models on two other downstream tasks, demonstrating transferability: variable similarity search (Section 4.3), and variable spelling error correction (Section 4.4).

Finally, we conduct an ablation study (Section 4.5) looking at the influence of training data size, pre-trained language models, and pre-trained embeddings from unsupervised learning contribute to VarCLR's effectiveness.

4.1 Setup

Pre-training. For VarCLR-Avg and VarCLR-LSTM, we use the Adam optimizer [43] with $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1 \times 10^{-8}$, a learning rate of 0.001, and early stop according to the contrastive loss on the validation set. We use a mini-batch size of 1024. The input embedding and hidden representation dimensions are set to 768 and 150 respectively. we also initialize the embedding layer with the CodeBERT pre-trained embedding layer. For VarCLR-CodeBERT, we use the AdamW optimizer [54] with the same configuration and learning rate, and a mini-batch size of 32. [6] We use the BERT model architecture [20] and initialize the model with pre-trained weights from CodeBERT [22]. For all three methods, we apply gradient norm clipping in the range $[-1, 1]$, and a temperature τ of 0.05. A summary of the hyper-parameters can be found along with our data, code, and pre-trained models at https://bit.ly/2WIalaW.

Dataset. While we use the GitHubRenames for training VarCLR, we use the IdBench [75] dataset for evaluation.[7] IdBench is a benchmark specifically created for evaluating variable semantic representations. It contains pairs of variables assigned relatedness and similarity scores by real-world developers. IdBench consists of three sub-benchmarks— IdBench-small, IdBench-medium, and IdBench-large, containing 167, 247, 291 pairs of variables, respectively. Ground truth scores for each pair of variable are assessed by multiple annotators. Pairs with disagreement between annotators exceeding a particular threshold are considered dissimilar; the three benchmarks differ in the choice of threshold. The smaller benchmark provides samples with higher inter-annotator agreement, while the larger benchmark provides more samples with commensurately lower agreement. The medium benchmark strikes a balance. We describe customizations of the IdBench dataset to particular tasks in their respective sections.

Baselines. We compare VarCLR models to the previous state-of-the-art as presented in IdBench [75]. We reuse the baseline results provided by the IdBench framework. The IdBench paper evaluates a number of previous approaches as well as a new ensemble method that outperforms them; we include as baselines a subset of those previous techniques, and the ensemble method. Of the string distance/syntactic functions (still broadly used in various name-related applications [51, 68]), we include **Levenshtein Edit Distance (LV)** (the number of single-character edits required to transform one string into the other); it performs in the top half of techniques on scoring similarity, and is competitive with the other syntactic distance metric [62] on relatedness. Of the embedding-based single models, we include **FastText CBOW (FT-cbow) and SG (FT-sg)** [8], extensions of `word2vec` that incorporate subword information, to better handle infrequent words and new words. These were the best-performing embedding-based methods on both relatedness and similarity.

Finally, we include two **combined** models. IdBench [75] proposes an ensemble method that combines the scores of all models and variable features. For each pair in IdBench, the combined model trains a Support Vector Machine (SVM) classifier with all other pairs, then applies the trained model to predict the score of the left-out pair. Note that this approach is trained on the IdBench benchmark itself and is not directly comparable to other methods. For comparison, we add VarCLR-Avg, VarCLR-LSTM, VarCLR-CodeBERT scores as additional input features to the combined approach, and report the results for Combined-VarCLR.

[6]Larger mini-batch sizes make the contrastive learning task more challenging and improve the quality of learned representation, as shown in [13] and our preliminary experiments. We use batch size of 32 for VarCLR-CodeBERT due to GPU memory limitations.

[7]The IdBench evaluation scripts were updated after publication, leading to minor differences in evaluation scores. We use their latest code as of May 1st, 2021 to evaluate the baselines and our models.

4.2 Variable Similarity and Relatedness Scoring

Our central claim is that VarCLR is well-suited to capturing and predicting variable similarity. Formally, given two variables u and v, we obtain variable representations with pre-trained VarCLR encoder $f_{\theta'}$ and compute the variable similarity score as the cosine similarity between the two vectors:

$$\boldsymbol{h_u}, \boldsymbol{h_v} = f_{\theta'}(u), f_{\theta'}(v) \tag{9}$$

$$\hat{s}(u,v) = \frac{\boldsymbol{h_u} \cdot \boldsymbol{h_v}}{\|\boldsymbol{h_u}\|_2 \|\boldsymbol{h_v}\|_2}, \tag{10}$$

where $\hat{s}(u,v)$ denotes the VarCLR's predicted similarity score. Following IdBench [75], we then compare the similarity scores of pre-trained VarCLR representations with human ground-truth similarity scores by computing Spearman's rank correlation coefficient between them. This correlation coefficient falls in the range [-1, 1], where 1 indicates perfect agreement between the rankings; -1 indicates perfect disagreement; and 0 indicates no relationship.

Note that the VarCLR pre-training task is explicitly optimizing the distance between similar variable pairs. Thus, the variable similarity scoring task only really evaluates the performance of the pre-training itself. To more fully evaluate whether our method leads to better representations that can transfer, we also evaluate on the variable relatedness scoring task.

Results. Table 1 shows the models' performance on the similarity and relatedness tasks in terms of Spearman's rank correlation with ground truth. Table 1a shows that VarCLR-CodeBERT improves over the previous state-of-the-art on all three IdBench benchmarks, with an absolute improvement of 0.18 on IdBench-small and 0.13 on IdBench-large compared to the previous best approach, FT-cbow. This shows that VarCLR aligns much better with human developers' assessment of variable similarity than any of the previously proposed models. Interestingly, VarCLR-Avg also outperforms FT-cbow by a large margin (+0.12 on IdBench-small). This suggests that most of our gains do not come from the use of a more powerful encoder architecture such as BERT. Instead, we conclude that the GitHubRenames dataset is effective at providing supervision signals of variable similarity, and the contrastive learning objective is effective. Although their architectures are very similar, VarCLR-Avg outperforms FT-cbow.

That said, the improvements in VarCLR-CodeBERT (+0.06) and VarCLR (+0.03) over VarCLR-Avg verify our assumption that powerful models with larger representational capacity are necessary for learning better variable representations, since they are able to capture and encode more information (e.g., sequential order and token importance) than the embedding averaging methods.

Table 1b shows that VarCLR also achieves the state-of-the-art performance on IdBench in terms of relatedness prediction. It surpasses the previous best by 0.07 on IdBench-small and 0.07 on IdBench-large. This is noteworthy because VarCLR training does not explicitly optimize for relatedness. This suggests that the VarCLR pre-training task learns better generic representations, rather than overfitting to the target task (i.e., variable similarity). This is very important, and supports our major contribution: By pre-training for the similarity learning task on GitHubRenames with a contrastive objective, VarCLR achieves better representations which can be applied to general tasks.

Table 1: Spearman's rank correlation with IdBench-small, IdBench-medium, IdBench-large of single models (top) and ensemble models (bottom), by increasing performance.

(a) Similarity scores

Method	Small	Medium	Large
FT-SG	0.30	0.29	0.28
LV	0.32	0.30	0.30
FT-cbow	0.35	0.38	0.38
VarCLR-Avg	0.47	0.45	0.44
VarCLR-LSTM	0.50	0.49	0.49
VarCLR-CodeBERT	**0.53**	**0.53**	**0.51**
Combined-IdBench	0.48	0.59	0.57
Combined-VarCLR	**0.66**	**0.65**	**0.62**

(b) Relatedness scores

Method	Small	Medium	Large
LV	0.48	0.47	0.48
FT-SG	0.70	0.71	0.68
FT-cbow	0.72	0.74	0.73
VarCLR-Avg	0.67	0.66	0.66
VarCLR-LSTM	0.71	0.70	0.69
VarCLR-CodeBERT	**0.79**	**0.79**	**0.80**
Combined-IdBench	0.71	0.78	0.79
Combined-VarCLR	**0.79**	**0.81**	**0.85**

4.3 Variable Similarity Search

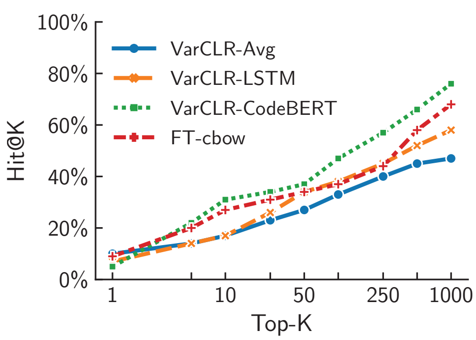
We next evaluate our learned representations in the context of a more applied downstream application: similar variable search. Similar variable search identifies similar variable names in a set of names given an input query. This can be useful for refactoring code, or for assigning variables more readable names (e.g., replacing `fd` with `file_descriptor`). For a given set of variables $\mathcal{V}$ and a pre-trained VarCLR encoder $f_{\theta'}$, we compute representation vectors $\mathcal{K} = \{f_{\theta'}(v) | v \in \mathcal{V}\}$. For a query variable u, we find top-k similar variables in $\mathcal{V}$ with the highest cosine similarity to $f_{\theta'}(u)$.

To quantitatively evaluate effectiveness in finding similar variables, we created a new mini-benchmark VarSim from the original IdBench benchmark. We select variable pairs which have human-assessed similarity scores greater than 0.4 in IdBench. This leaves us with 100 'similar' variable pairs from all 291 variable pairs in the IdBench-large benchmark. We use the variable collection provided in IdBench containing 208,434 variables as the overall candidate pool. We use Hit@K as our evaluation metric, computing the cosine similarity of the representations of a query variable u and all the variables in the candidate pool. We select the top-K variables with the highest similarity scores and check whether the corresponding similar variable v is in the top-K list. We choose K to be 1, 5, 10, 25, 50, 100, 250, 500, 1000.

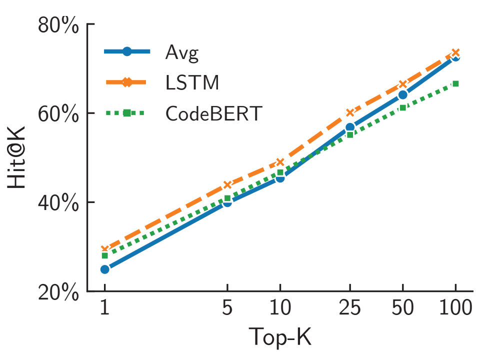
Results. As shown in Figure 3a, VarCLR-CodeBERT achieves the best similarity search performance, with 47% at K=100 and 76% at K=1000, compared to FT-cbow (37% at K=100, 68% at K=1000). This

Table 2: Variable Similarity Search. Top-5 most similar variables found by the IdBench method and VarCLR-CodeBERT.

Variable	Method	Top 5 Similar Variables				
`substr`	FT-cbow	`substring`	`substrs`	`subst`	`substring1`	`substrCount`
	VarCLR-CodeBERT	`subStr`	`substring`	`substrs`	`stringSubstr`	`substrCount`
`item`	FT-cbow	`itemNr`	`itemJ`	`itemL`	`itemI`	`itemAt`
	VarCLR-CodeBERT	`pItem`	`itemEl`	`mItem`	`itemEls`	`itemValue`
`count`	FT-cbow	`countTbl`	`countInt`	`countRTO`	`countsAsNum`	`countOne`
	VarCLR-CodeBERT	`sCount`	`countOf`	`counts`	`countInt`	`countTh`
`rows`	FT-cbow	`rowOrRows`	`rowXs`	`rows_1`	`rowsAr`	`rowIDs`
	VarCLR-CodeBERT	`drows`	`allrows`	`rowsArray`	`ows`	`nRows`
`setInterval`	FT-cbow	`resetInterval`	`setTimeoutInterval`	`clearInterval`	`getInterval`	`retInterval`
	VarCLR-CodeBERT	`pInterval`	`mfpSetInterval`	`setTickInterval`	`clockSetInterval`	`iInterval`
`minText`	FT-cbow	`maxText`	`minLengthText`	`microsecText`	`maxLengthText`	`minuteText`
	VarCLR-CodeBERT	`minLengthText`	`minContent`	`maxText`	`minEl`	`min`
`files`	FT-cbow	`filesObjs`	`filesGen`	`fileSets`	`extFiles`	`libFiles`
	VarCLR-CodeBERT	`filesArray`	`aFiles`	`allFiles`	`fileslist`	`filelist`
`miny`	FT-cbow	`min_y`	`minBy`	`minx`	`minPt`	`min_z`
	VarCLR-CodeBERT	`ymin`	`yMin`	`minY`	`minYs`	`minXy`



(a) Similarity Search



(b) Spelling Error Correction

Figure 3: Hit@K score comparison on VarSim and VarTypo.

indicates that our method is effective at finding similar variables, able to distinguish the most similar variable to the query variable out of 200 distractors around 76% of the time.[8] Interestingly, VarCLR-Avg and VarCLR-LSTM are less effective at similarity search than FT-cbow, even though they outperform FT-cbow by a large margin in the similarity scoring task. Embedding-based methods are still a strong baseline for variable similarity search. However, contrastive methods still amplify the effectiveness of unsupervised embedding methods.

Similarity scoring and similarity search are distinct tasks, and so it is not unexpected that techniques will be equally effective on both. For example, `word2vec` tends to put the embeddings of similar rare words close to some common frequent word. This behavior does not affect the similarity search effectiveness because the rare words are able to find each other, and the frequent word is close enough to *its* similar word than to these rare words. However, this will hurt similarity scoring between the rare words and the frequent variable, since they are actually not similar. In comparison, VarCLR is able to avoid these kinds of scoring mistakes.

Case Study. We demonstrate our results qualitatively by choosing the same set of variables used to demonstrate this task in the IdBench paper, and displaying the comparative results in Figure 3a. For space, we omit two of the variables (**`rows`** and **`count`**) in the set; the two methods perform comparably (such as on **`substr`**). We observe that the overall qualities of the two methods' results are similar. This is understandable since the gap between the two methods on variable similarity search is relatively small as shown in Table 2.

Meanwhile, it is worth noting that VarCLR-CodeBERT is better at penalizing distractive candidates that are only related but not similar. For example, for **`minText`**, VarCLR-CodeBERT ranks **`minLengthText`**, **`minContent`** before **`maxText`**, while FT-cbow suggests the opposite. For **`miny`**, VarCLR-CodeBERT ranks **`ymin`**, **`yMin`**, **`minY`** as top-3, while FT-cbow suggests related but dissimilar variables such as **`minBy`** and **`minx`**. This provides additional evidence that

[8] Since we evaluate the Hit@1000 score in a candidate pool of size ~200,000, the "resolution" of this retrieval task is $\frac{1000}{200000} = \frac{1}{200}$. Although inspecting the top 1000 may not be practical as an real-world application itself, it is still an informative metric of the representation quality, and may indicate effectiveness in other settings, e.g., a developer looking at the top 5 similar variables from a limited 1,000 candidates, which has the same requirement on resolution. Another possible application is to use VarCLR to retrieve a large candidate pool as the first stage to other methods, e.g., natural variable name suggestion.

our method is able to better represent semantic similarity rather than pure relatedness.

4.4 Variable Spelling Error Correction

Spelling Error Correction is a fundamental yet challenging task in natural language processing [37]. We explore the possibility of applying VarCLR models to perform spelling error correction on variable names. If the representations of misspelled variable names are close to their correct versions, corrections may be found via nearest neighbor search. Fortunately, the GitHubRenames dataset enables this goal, because a portion of renaming edits in GitHubRenames are actually correcting spelling errors in previous commits. We can therefore reformulate this problem as a variable similarity search task, since our method treats these misspelled names as similar to their corrected versions.

We create a new synthetic variable spelling error correction dataset, VarTypo, with 1023 misspelled variables and their corrections. Specifically, we create this dataset by sampling variables from the 208,434 variable pool from IdBench, and use the `nlpaug`[9] package [56] to create misspelled variables from the correct ones. We use `KeyboardAug` which simulates typo error according to characters' distance on the keyboard. This task is challenging because our method does not leverage any external dictionary or hand-crafted spelling rules. Meanwhile, although string distance functions such as Levenshtein distance can potentially perform better, these functions require expensive one-by-one comparisons between the query variable and every variable in the pool, which is very time consuming, while our method uses GPU-accelerated matrix multiplication to compute all cosine distances at once and can potentially adopt an even more efficient vector similarity search library such as `faiss`. Therefore, we believe it is still an informative benchmark for evaluating variable representations.

Results. Similar to variable similarity search, we evaluate the effectiveness as the Hit@K score of using the representation of misspelled variables to retrieve the corresponding correct variable. As shown in Figure 3b, VarCLR can successfully correct the 29.4% of the time at Top-1, and 73.6% of the time at Top-100. One interesting observation we find is that in this task, the gap (-4.5% at Top-1 and -1.0% at Top-100) between VarCLR-Avg and the other two powerful encoders is relatively small. It even outperforms VarCLR-CodeBERT after K=25. One possible explanation is that fixing a typo requires neither word sequential order or word importance information, i.e., being able to model the variable as a sequence instead of a bag of words does not benefit this task.

Case Study. For illustration, we randomly select misspelled variable names and use our VarCLR to find the most similar correct variable names. As shown in Table 3., our model is able to correct some of the misspelled variables, including insertions, deletions, and modifications, while failing to recover others. Notably, variable names consisting of multiple words such as **`minSimilarity`** can be corrected successfully.

[9] https://github.com/makcedward/nlpaug

Table 3: The top-3 most similar variables to misspelled variables, found by VarCLR.

Variable	Top 3 Similar Variables
`temepratures`	`temperatures, temps, temlp`
`similarlity`	`similarity, similarities, similar`
`minSimilarlity`	`minSimilarity, similarity, minRatio`
`program_able`	`programmable, program, program6`
`supervisior`	`superior, superview, superc`
`producitons`	`obligations, proportions, omegastructors`
`transaltion`	`transac, trans, transit`

4.5 Ablation Studies

So far we have demonstrated the importance of both contrastive learning and sophisticated models like CodeBERT for VarCLR performance. Here, perform ablation studies to measure the effect of additional design decisions in VarCLR: of training data size, of using pre-trained language models, and of using pre-trained embeddings from unsupervised learning.

4.5.1 Effect of Data Size on Contrastive Pre-training. Pre-training VarCLR requires weakly-supervised data scraped from public repositories. Thus, we evaluate how much data is required to train an effective model, to elucidate data collection costs. To evaluate this, we train VarCLR-Avg, VarCLR-LSTM, VarCLR-CodeBERT on 0%, 0.1%, 1%, 3.16%, 10%, 21.5%, 46.4%, 100% percent of the full dataset, measuring the similarity score on IdBench-medium.

Figure 4 shows the results. For all three VarCLR variants, training data size has a significant positive effect on effectiveness. This is especially true for VarCLR-CodeBERT, but performance flattens and converges as training data size approaches 100%. This suggests that GitHubRenames is of an appropriate size for this task.

Another interesting observation is that VarCLR-Avg outperforms VarCLR-LSTM with smaller amounts of training data. This indicates the more powerful LSTM model does not surpass a simple one until the data size reaches a critical threshold. This is likely because a more complex model has more parameters to train and requires more data to reach convergence. With sufficient data, larger models win, thanks to their representational capacity. This suggests a caveat in applying representation learning models: it is important to choose a model with an appropriate complexity given the amount of available data, rather than defaulting to the best-performing model overall.

4.5.2 Using a Pre-trained Language Model. Before contrastive pre-training on GitHubRenames, VarCLR-CodeBERT is initialized with a model (pre-)pre-trained on a large code corpus. The effect of this pre-training is also illustrated in Figure 4. Although VarCLR-CodeBERT has a much larger number of parameters, it outperforms VarCLR-Avg and VarCLR-LSTM after contrastive pre-training on only 1% of GitHubRenames. While this seems to contradict the conclusion reached in the comparison between VarCLR-LSTM and VarCLR-Avg, it displays the benefit of initialization with a pre-trained model. Compared to VarCLR-LSTM, which contains randomly initialized parameters that have to be trained from scratch, VarCLR-CodeBERT parameters produce reasonable representations from the start. Therefore, it requires less data to converge,

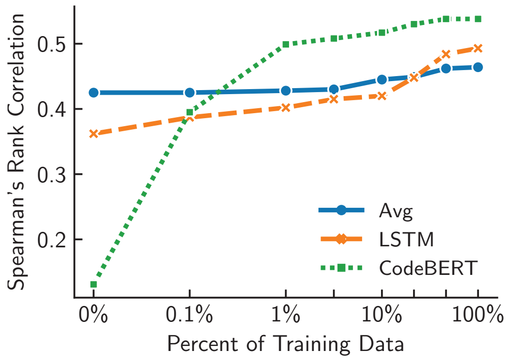


Figure 4: Effect of contrastive pre-training data size on learned VarCLR representations, evaluated on IdBench-medium.

Table 4: Effect of pre-trained CodeBERT embeddings on similarity score effectiveness (Spearman's). Models are either randomly initialized and contrastively pre-trained (Contrastive), initialized with CodeBERT embeddings (CodeBERT), or both (VarCLR).

Method	Small	Medium	Large
Contrastive-Avg	0.34	0.33	0.30
CodeBERT-Avg	0.44	0.43	0.40
VarCLR-Avg	**0.47**	**0.45**	**0.44**
Contrastive-LSTM	0.35	0.33	0.30
CodeBERT-LSTM	0.36	0.36	0.36
VarCLR-LSTM	**0.50**	**0.49**	**0.49**

and thanks to its large model capacity, ultimately outperforms the other two variants by a large margin.

Despite the fast convergence, directly applying CodeBERT without contrastive pre-training leads to poor performance (0.13 at 0% data). One possible reason is that CodeBERT was originally trained for whole-program representations, and using it with variable names as inputs leads to a problematic divergence from its training data distribution.

4.5.3 Effect of Pre-trained CodeBERT Embeddings. Both VarCLR-Avg and VarCLR-LSTM are initialized with the word embeddings from CodeBERT before contrastive pre-training. To study the effect of these pre-trained embeddings, we measure the Spearman's correlation coefficient of the similarity scores of the models modified in two ways: one with randomly-initialized embeddings that is then contrastively pre-trained ("Contrastive" in Table 4), and one that is initialized with CodeBERT embeddings but *not* contrastively pre-trained ("CodeBERT" in Table 4).

The results show that pre-trained CodeBERT embeddings are essential to the performance of VarCLR-Avg and VarCLR-LSTM. However, directly adopting the pre-trained embeddings alone is still insufficient, especially for LSTMs. This implies that both unsupervised pre-training and weakly supervised pre-training are indispensable for useful variable representations: the former takes advantage of *data quantity* by leveraging a huge amount of unlabeled data, while the latter takes advantage of *data quality* using the weakly supervised GitHubRenames dataset.

5 RELATED WORK

Variable Names and Representations. Variable names are important for source code readability and comprehension [25, 46]. Because of this, there has been recent work focusing on automatically suggesting clear, meaningful variable names for tasks such as code refactoring [3, 5, 52] and reverse engineering [35, 45].

A common approach involves building prediction engines on top of learned variable representations. Representation learning is a common task in Natural Language Processing (NLP), and these techniques are often adapted to source code. Simpler approaches model variable representations by applying `word2vec` [58] to code tokens [8, 16, 59, 75], while more advanced techniques have adapted neural network architectures [41] or pre-trained language models [12]. Source code representation is a common enough task that researchers have developed benchmarks specifically for variable [75] and program representations [76].

Similarity and Relatedness. A fundamental concern with existing variable representations and suggestion engines is the difference between "related" and "similar" variables [60, 75]. "Related" variables reference similar core concepts without concern for their precise meaning, while "similar" variables are directly interchangeable. For example, **`minWeight`** and **`maxWeight`** are related but not similar, while **`avg`** and **`mean`** are both. Unlike state-of-the-art techniques, which only model relatedness, VarCLR explicitly optimizes for similarity by adapting contrastive learning techniques from NLP and computer vision research. In NLP, systems are often designed to focus on text relatedness [10, 23, 85], similarity [32], or both [2]. While document search might only be concerned with relatedness [23] similarity is particularly important in systems designed for paraphrasing documents [79, 81].

VarCLR relies on *contrastive learning* to optimize for similarity. Contrastive learning is particularly useful for learning visual representations without any supervision data [11, 13, 14, 26, 30, 72, 82], but has also been used for NLP [61]. Recent work has applied contrastive learning to the pre-training of language models to learn text representations [17] and, similar to our task, learn sentence embeddings for textual similarity tasks [24]. Contrastive learning has also been used for code representation learning [36] where source-to-source compiler transformation is applied for generating different views of a same program. Different from this work, we focus on learning representations for variable names, and leverage additional data from GitHub for better supervision.

String similarity and spelling errors. Efficient string similarity search remains an active research area [6, 19, 49, 87]. Most of these methods can be categorized as *sparse retrieval* methods, focusing on distance functions on the original string or n-grams. These algorithms depend on the lexical overlap between strings and thus cannot capture the similarity between variables pairs such as **`avg`** and **`mean`**. More recently, *dense retrieval* methods have been shown effective in NLP tasks [39, 47]. These methods perform similarity search in the space of learned representations, so that sequences with similar meanings but low lexical overlap can be found. Meanwhile, extremely efficient similarity search frameworks for dense vectors such as `faiss` [38] can be applied. VarCLR introduces the concept of dense retrieval into the variable names domain, enabling more effective and efficient finding of a list of candidates that are similar to a given variable name.

Neural models for spelling error correction usually require parallel training data which are hard to acquire in practice [28, 86]. Recent work adopts different mechanisms to create synthetic parallel data, including noise injection [37], and back-translation models [27]. We leave a detailed comparison to future work, but note that VarCLR shows promise without expensive training data.

Name- and Machine Learning-based Program Analyses. Our downstream tasks are examples of program analyses based on information gathered with machine learning (ML). Name-based based program analyses predicated on machine learning have been used in many contexts. In the context of code modification, they have been used for variable name suggestion from code contexts [5], method and class name rewriting [52] and generation [4], code generation directly from docstrings [12], and automated program repair [15, 77]. They have also been used for type inference from natural language information [57, 83], detecting bugs [40, 64, 65, 68], and detecting vulnerabilities [29]. VarCLR can serve as a drop-in pre-training step for such techniques, enabling more effective use of the semantic information contained in variable names for a wide range of such analyses.

6 DISCUSSION

In this paper, we study variable representation learning, a problem with significant implications for machine learning and name-based program analyses. We present a novel method based on contrastive learning for pre-training variable representations. With our new weakly-supervised GitHubRenames dataset, our method enables the use of stronger encoder architectures in place of `word2vec`-based methods for this task, leading to better generalized representations. Our experiments show that VarCLR greatly improves representation quality not only in terms of variable similarity, but also for other downstream tasks. While these downstream tasks may not be immediately practical themselves, our approach is promising as a drop-in pre-training solution for other variable name-based analysis tasks, which we hope others will attempt in future work. For example, VarCLR can replace the the word2vec-CBOW embeddings used in a name-based bug detector [64], or the n-gram based language model used as a similarity scoring function for name suggestion [3]. Existing dictionary-based IDE spell-checkers may also benefit from using VarCLR to rank suggestions based on the pretrained semantic similarity.

We note limitations and possible threats in our study. Our dataset is automatically constructed from git commits from GitHub, and likely contains noise that can harm contrastive learning performance [50]. However, our results show that despite this noise, our models transfer well, and our evaluation is based on an entirely distinct test set. Knowledge distillation and self-training methods [21, 33] such as momentum distillation [50] can be applied to deal with the noise in weak supervision data [50, 71].

In this work, we applied VarCLR exclusively to unsupervised downstream tasks. Fine-tuning VarCLR models with labeled data might further enable significant performance improvements for more complicated tasks, like natural variable name suggestion [3]. Beyond constructing similar variable names, it is also conceptually possible to construct similar pairs of larger code snippets from git diffs describing patches. Applying contrastive learning on these pairs can potentially improve CodeBERT code representation and understanding, which could benefit tasks well beyond variable similarity, such as code search. Finally, we used instance discrimination [82] to guide our contrastive learning approach, with promising results. This suggests that more advanced contrastive learning methods such as MoCo [30], BYOL [26], SwAV [11] be adapted to this task for better representation learning in general.

ACKNOWLEDGMENTS

The authors would like to thank Michael Pradel and the authors of IdBench for providing us with data for our experiments. This material is based upon work supported in part by the National Science Foundation (awards 1815287 and 1910067).

REFERENCES

[1] Daniel Adiwardana, Minh-Thang Luong, David R So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, et al. 2020. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977* (2020).
[2] Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Kravalova, Marius Pasca, and Aitor Soroa. 2009. A study on similarity and relatedness using distributional and wordnet-based approaches. (2009).
[3] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Symposium on the Foundations of Software Engineering (FSE)*. 281–293.
[4] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 38–49.
[5] R. Bavishi, M. Pradel, and K. Sen. 2017. *Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts.* Technical Report. TU Darmstadt, Department of Computer Science.
[6] Roberto J Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*. 131–140.
[7] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering* 18, 2 (2013), 219–276.
[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.
[9] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
[10] Elia Bruni, Nam-Khanh Tran, and Marco Baroni. 2014. Multimodal distributional semantics. *Journal of artificial intelligence research* 49 (2014), 1–47.
[11] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. 2020. Unsupervised learning of visual features by contrasting cluster assignments. *arXiv preprint arXiv:2006.09882* (2020).
[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[13] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.
[14] Xinlei Chen and Kaiming He. 2021. Exploring simple siamese representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 15750–15758.
[15] Zimin Chen and Martin Monperrus. 2018. The Remarkable Role of Similarity in Redundancy-based Program Repair. *arXiv prepring arXiv:1811.05703* (2018).

[16] Zimin Chen and Martin Monperrus. 2019. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061* (2019).

[17] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2019. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations*.

[18] Fred J. Damerau. 1964. A Technique for Computer Detection and Correction of Spelling Errors. *Commun. ACM* (1964), 171–176.

[19] Dong Deng, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2013. Top-k string similarity search with edit-distance constraints. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 925–936.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[21] Jingfei Du, Édouard Grave, Beliz Gunel, Vishrav Chaudhary, Onur Celebi, Michael Auli, Veselin Stoyanov, and Alexis Conneau. 2021. Self-training Improves Pre-training for Natural Language Understanding. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 5408–5418.

[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*. 1536–1547.

[23] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. 2001. Placing search in context: The concept revisited. In *Proceedings of the 10th international conference on World Wide Web*. 406–414.

[24] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. *arXiv preprint arXiv:2104.08821* (2021).

[25] Edward M. Gellenbeck and Curtis R. Cook. 1991. *An Investigation of Procedure and Variable Names as Beacons During Program Comprehension*. Technical Report. Oregon State University.

[26] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre H Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, et al. 2020. Bootstrap your own latent: A new approach to self-supervised learning. *arXiv preprint arXiv:2006.07733* (2020).

[27] Jinxi Guo, Tara N Sainath, and Ron J Weiss. 2019. A spelling correction model for end-to-end speech recognition. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 5651–5655.

[28] Masato Hagiwara and Masato Mita. 2020. GitHub Typo Corpus: A Large-Scale Multilingual Dataset of Misspellings and Grammatical Errors. In *Proceedings of the 12th Language Resources and Evaluation Conference*. 6761–6768.

[29] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. 2018. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497* (2018).

[30] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9729–9738.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[32] Felix Hill, Roi Reichart, and Anna Korhonen. 2015. Simlex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics* 41, 4 (2015), 665–695.

[33] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[34] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[35] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. 2018. Meaningful Variable Names for Decompiled Code: A Machine Translation Approach. In *International Conference on Program Comprehension (ICPC '18)*. 20–30.

[36] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973* (2020).

[37] Sai Muralidhar Jayanthi, Danish Pruthi, and Graham Neubig. 2020. NeuSpell: A Neural Spelling Correction Toolkit. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 158–164.

[38] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* (2019).

[39] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 6769–6781.

[40] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian Elbaum, and Zhaogui Xu. 2018. Phys: probabilistic physical unit assignment and inconsistency detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 563–573.

[41] Deborah S. Katz, Jason Ruchti, and Eric Schulte. 2018. Using Recurrent Neural Networks for Decompilation. In *International Conference on Software Analysis, Evolution and Reegnineering (SANER '18)*. 346–356.

[42] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *arXiv preprint arXiv:2004.11362* (2020).

[43] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[44] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR '17*.

[45] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 628–639.

[46] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *International Conference on Program Comprehension (ICPC '06)*. 3–12.

[47] Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. 2019. Latent retrieval for weakly supervised open domain question answering. *arXiv preprint arXiv:1906.00300* (2019).

[48] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871–7880.

[49] Chen Li, Jiaheng Lu, and Yiming Lu. 2008. Efficient merging and filtering algorithms for approximate string searches. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 257–266.

[50] Junnan Li, Ramprasaath R Selvaraju, Akhilesh Deepak Gotmare, Shafiq Joty, Caiming Xiong, and Steven Hoi. 2021. Align before Fuse: Vision and Language Representation Learning with Momentum Distillation. *arXiv preprint arXiv:2107.07651* (2021).

[51] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. 2016. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *Proceedings of the 38th International Conference on Software Engineering*. 1063–1073.

[52] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1–12.

[53] 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. *arXiv preprint arXiv:1907.11692* (2019).

[54] Ilya Loshchilov and Frank Hutter. 2018. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

[55] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. https://openreview.net/forum?id=6lE4dQXaUcb

[56] Edward Ma. 2019. NLP Augmentation. https://github.com/makcedward/nlpaug.

[57] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 304–315.

[58] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffery Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *Computing Research Repository (CoRR)* abs/1310.4546 (2013).

[59] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[60] George A Miller and Walter G Charles. 1991. Contextual correlates of semantic similarity. *Language and cognitive processes* 6, 1 (1991), 1–28.

[61] Andriy Mnih and Koray Kavukcuoglu. 2013. Learning word embeddings efficiently with noise-contrastive estimation. In *Advances in neural information processing systems*. 2265–2273.

[62] Saul B Needleman and Christian D Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal*

of molecular biology 48, 3 (1970), 443–453.

[63] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).

[64] Michael Pradel and Thomas R Gross. 2011. Detecting anomalies in the order of equally-typed method arguments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 232–242.

[65] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[66] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. 2020. Gcc: Graph contrastive coding for graph neural network pre-training. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1150–1160.

[67] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21 (2020), 1–67.

[68] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting argument selection defects. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–22.

[69] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.

[70] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1715–1725.

[71] Antti Tarvainen and Harri Valpola. 2017. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. *Advances in Neural Information Processing Systems* 30 (2017).

[72] Yonglong Tian, Dilip Krishnan, and Phillip Isola. 2020. Contrastive multiview coding. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XI 16*. Springer, 776–794.

[73] Yonglong Tian, Chen Sun, Ben Poole, Dilip Krishnan, Cordelia Schmid, and Phillip Isola. 2020. What makes for good views for contrastive learning? *arXiv preprint arXiv:2005.10243* (2020).

[74] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*. 6000–6010.

[75] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. IdBench: Evaluating Semantic Representations of Identifier Names in Source Code. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 562–573.

[76] Ke Wang and Mihai Christodorescu. 2019. Coset: A benchmark for evaluating neural program embeddings. *arXiv preprint arXiv:1905.11445* (2019).

[77] Martin White, Michele Tufano, Matías Martínez, Martin Monperrus, and Denys Poshyvanyk. 2018. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. *arXiv preprint arXiv:1707.04742* (2018).

[78] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 479–490.

[79] John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. 2015. Towards universal paraphrastic sentence embeddings. *arXiv preprint arXiv:1511.08198* (2015).

[80] John Wieting, Kevin Gimpel, Graham Neubig, and Taylor Berg-Kirkpatrick. 2019. Simple and Effective Paraphrastic Similarity from Parallel Translations. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4602–4608.

[81] John Wieting, Kevin Gimpel, Graham Neubig, and Taylor Berg-Kirkpatrick. 2021. Paraphrastic Representations at Scale. *arXiv preprint arXiv:2104.15114* (2021).

[82] Zhirong Wu, Yuanjun Xiong, Stella X Yu, and Dahua Lin. 2018. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3733–3742.

[83] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 607–618.

[84] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. 2018. Learning to Represent Edits. In *International Conference on Learning Representations*.

[85] Torsten Zesch, Christof Müller, and Iryna Gurevych. 2008. Using Wiktionary for Computing Semantic Relatedness.. In *AAAI*, Vol. 8. 861–866.

[86] Shaohua Zhang, Haoran Huang, Jicong Liu, and Hang Li. 2020. Spelling Error Correction with Soft-Masked BERT. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 882–890.

[87] Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, and Divesh Srivastava. 2010. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 915–926.

[88] Hao Zhong, Tao Xie, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *European Conference on Object-Oriented Programming (ECOOP)*. 318–343.