Efficient Framework for Learning Code Representations through Semantic-Preserving Program Transformations

Nghi D. Q. Bui
School of Information Systems
Singapore Management University
dqnbui.2016@phdcs.smu.edu.sg

Abstract

Recent learning techniques for the representation of code depend mostly on human-annotated (labeled) data. In this work, we are proposing Corder, a self-supervised learning system that can learn to represent code on unlabeled data. The key innovation is that we train the source code model by asking it to recognize similar and dissimilar code snippets through a *contrastive learning paradigm*. We use a set of semantic-preserving transformation operators to generate snippets that are syntactically diverse but semantically equivalent. The contrastive learning objective, at the same time, maximizes agreement between different views of the same snippets and minimizes agreement between transformed views of different snippets.

We train different instances of Corder on 3 neural network encoders, which are Tree-based CNN, ASTNN, and Code2vec over 2.5 million unannotated Java methods mined from GitHub. Our result shows that the Corder pre-training improves code classification and method name prediction with large margins. Furthermore, the code vectors generated by Corder are adapted to code clustering which has been shown to significantly beat the other baselines .

Introduction

Building deep learning model for code have been found useful in many software engineering tasks, such as predicting bugs (Yang et al. 2015; Li et al. 2017; Zhou et al. 2019), translating programs (Chen, Liu, and Song 2018; Gu et al. 2017), classifying program functionality (Nix and Zhang 2017; Dahl et al. 2013), code search (Gu, Zhang, and Kim 2018; Kim et al. 2018; Sachdev et al. 2018), code comment generation (Hu et al. 2018; Wan et al. 2018; Alon et al. 2019b), etc. While offering promising performance for the tasks, these techniques mostly rely on human-annotated data or based on some heuristic to generate the label to train suitable models. For example, Code2vec (Alon et al. 2019b) uses a heuristic to automatically extract method names from function as labels and train the supervised deep learning model for the method name prediction task. Jiang, Liu, and Jiang (2019) shows that a large amount of method name does not reflect the implementation of the method body correctly, which can make the model becomes bias.

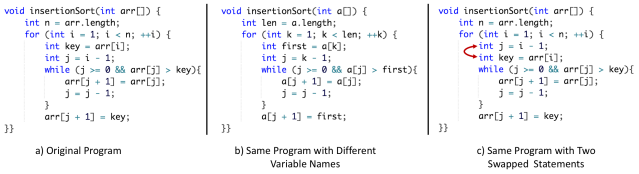

Figure 1: Example of Semantical Equivalent Programs

On the other hand, self-supervised pre-training has been proved to be useful in visual learning and natural language processing. The pre-trained model from self-supervised learning can be used for the fine-tuning process to increase the performance of certain code learning tasks significantly. In this work, we are developing Corder, a self-supervised representation learning framework for source code that uses semantic-preserving transformation techniques (e.g., dead code insertion, variable renaming, permuting statement, etc.,) to generate syntactically diverse yet semantically equivalent programs. Our work is inspired by recent work that analyzes the robustness of the source code model with adversarial examples (Bielik and Vechev 2020; Rabin, Islam, and Alipour 2020), most of them rely on program transformation techniques to generated such adversarial examples. An example of semantically equivalent programs is shown in Figure 1. These 3 code snippets implement the same insertion sort algorithm, but they are different in syntax and textual detail, e.g. different variable names, permute statements in the basic block that do not depend on each other.

We use these equivalent programs to create a pretext task that allows the model to recognize equivalent programs from a large dataset. In doing so, the encoder trained for this task will learn about the similarity of programs in such a way that semantically related programs should be close in vector space. To achieve this goal, Corder uses the contrastive learning methods that have been commonly used to learn representation for image (Chen, Liu, and Song 2018) without the need for labeled data. The **neural network encoder** in Corder first learns generic representations of code snippets on an unannotated dataset, and then can be fine-tuned with a small amount of labeled data to achieve good performance for a given code prediction task, such as code clas-

Preprint

sification or method name prediction. Generic representations are learned by simultaneously maximizing the agreement between the differently transformed views of the same programs and minimizing the agreement between the transformed views of different programs, using a technique called contrastive learning. Updating the parameters of a neural network using this contrastive learning objective causes representations of corresponding views to '**attract**' each other, while representations of non-corresponding views to '**repel**' each other.

We trained 3 Corder instances with 3 encoders, which are Tree-based CNN (Mou et al. 2016), ASTNN (Zhang et al. 2019), Code2vec (Alon et al. 2019b) based on a larged set of Java code named Corder-TBCNN, Corder-ASTNN and Corder-Code2vec. In 3 downstream tasks, we evaluated the utility of the pre-trained code representations, the first task being code clustering to determine whether Corder would generate identical vectors for semantic equivalent code snippets. The other 2 are code classification and method name prediction to evaluate whether the fine-tuning process can improve the performance of such tasks over the pre-trained Corder. For *code clustering*, our results using Corder outperform the best baseline (Code2vec) by 12% in term of Adjusted Rand Index; For the two supervised tasks, we utilize the weights of the pre-trained model from Corder to fine-tune the specific prediction model for each task: our results using the fine-tuning process increases the performance of TBCNN for *code classification* by 4.2% in term of accuracy, and increase the performance Code2vec and Code2seq for *method name prediction* by an average of 6%.

Related Work

Self-Supervised Learning has made tremendous strides in the field of visual learning (Mahendran, Thewlis, and Vedaldi 2018; Gidaris, Singh, and Komodakis 2018; Zhang, Isola, and Efros 2016; Korbar, Tran, and Torresani 2018; Kim, Cho, and Kweon 2019; Fernando et al. 2017), and for quite some time in the field of natural language processing (Mikolov et al. 2013; Le and Mikolov 2014; Kiros et al. 2015; Devlin et al. 2018). Such techniques allow for neural network training without the need for human labels. Typically a self-supervised learning technique reformulates an unsupervised learning problem as one that is supervised by *generating virtual labels automatically from existing (unlabeled) data*. Towards this goal, *contrastive learning* is a new paradigm unifying many past approaches to self-supervised learning by formulating the supervised learning problem as the task of comparing similar and dissimilar items, such as Siamese Neural Networks (Bromley et al. 1994), triple loss (Schroff, Kalenichenko, and Philbin 2015). In a very high level, contrastive learning methods works by minimize a distance between similar data (positives) representations and maximize the distance between dissimilar data (negatives).

Deep Learning Models of Code: There has been a huge interest in applying deep learning techniques for software engineering tasks such as program functionality classification (Mou et al. 2016; Zhang et al. 2019), bug localization (Pradel and Sen 2018; Gupta, Kanade, and Shevade 2019), function name prediction (Fernandes, Allamanis, and Brockschmidt 2019), code clone detection (Zhang et al. 2019), program refactoring (Hu et al. 2018), program translation (Chen, Liu, and Song 2018), and code synthesis (Brockschmidt et al. 2019). Allamanis, Brockschmidt, and Khademi (2018) extend ASTs to graphs by adding a variety of code dependencies as edges among tree nodes, intended to represent code semantics, and apply Gated Graph Neural Networks (GGNN) (Li et al. 2016) to learn the graphs; Code2vec (Alon et al. 2019b), Code2seq (Alon et al. 2019a), and ASTNN (Zhang et al. 2019) are designed based on splitting ASTs into smaller ones, either as a bag of path-contexts or as flattened subtrees representing individual statements. They use various kinds of Recurrent Neural Network (RNN) to learn such code representations. Survey on code embeddings (Ingram 2018; Chen and Monperrus 2019) presents evidence to show that there is a strong need to alleviate the requirement of labeled data for code modeling and encourage the community to invest more effort in the methods on learning source code with unlabeled data. Unfortunately, there is little effort that invests to design the source code model with unlabeled data: Yasunaga and Liang (2020) presents a self-supervised learning paradigm for program repair, but it is designed specifically for program repair only. There are methods, such as (Hussain et al. 2020; Feng et al. 2020) that perform pretraining source code data on natural language model (BERT, RNN, LSTM), but they simply train the code tokens similar to the way pretrained language models on text do, so they miss a lot of information about syntactical and semantical features of code that can be extracted from program analysis.

Approach

Approach Overview

The concept of Corder can be represented like this at a very high level. A code snippet P is taken and random transformations are applied to it to get two transformed snippets Pi and Pj. – snippet of that pair is passed through the encoder to get the code vectors vi and vj, respectively. The goal is to maximize the similarities between these two vi and vj representations for the same snippet. In addition, we also need a way to minimize the similarity between a random pair of snippets. Inspired by recent contrastive learning algorithms, Corder learns representations by maximizing agreement between differently transformed views of the same snippet example via a contrastive loss in the vector space.

Approach Details

As illustrated in Figure 2, this framework comprises the following three major components.

- **A program transformation module** that transforms the AST representation of any given code snippet example randomly resulting in two correlated views of the same example, denoted P_i and P_j. which we consider as a *positive pair*. We also need to generate negative pairs for the Contrastive Loss function. In this work, we sequentially

apply three transformations: variable renaming, dead code insertion, permute statement.

- **A neural network encoder** that can receive the representation of a code snippet (such as AST) and map it into a vector representation. In this case, it should map the P_i and P_j into two code vectors h_i and h_j, respectively.
- **A contrastive loss function** is defined for the contrastive learning task. Given a set P_k including a positive pair of examples P_i and P_j, the *contrastive prediction task* aims to identify P_j in $\{P_k\}_{k \neq i}$ for a given P_i.

When training the model, we randomly sample a minibatch of N samples from a large set of code snippets and define the contrastive prediction task on pairs of transformed examples derived from the minibatch, resulting in $2N$ data points. We do not sample negative examples explicitly. Instead, we follow (Chen et al. 2020) to treat $2(N-1)$ transformed examples within a minibatch (excluded the positive pair) as negative examples For example, let's take $N = 2$ to illustrate the steps. For each sample in this batch, a random transformation function is applied to get a pair of 2 images. Thus, for a batch size of 2, we get $2 * N = 2 * 2 = 4$ total transformed code snippets. In summary, the steps for $N = 2$ can be described as:

- Given two code snippets P_1 and P_2 in the batch sampled from a large set of unlabeled data, each snippets will be applied two random program transformation operators, resulting into P_{1_i}, P_{1_j} and P_{2_i}, P_{2_j}.
- Each of the above transformed snippets will be fed into the same encoder to get v_{1_i}, v_{1_j} and v_{2_i}, v_{2_j}.
- Each of the above transformed snippets will be fed into the same encoder to get the representations v_{1_i}, v_{1_j}, v_{2_i}, v_{2_j}.
- We use the Noise Contrastive Estimate loss (NCE) (Chen, Liu, and Song 2018) function to compare the similarities of these representations, our goal is to make each of the pair v_{1_i} and v_{1_j}, v_{2_i} and v_{2_j} to be as close in the vector space as possible. On the other hand, we want to make these 4 pairs as dissimilar as possible: v_{1_i} and v_{2_i}, v_{1_i} and v_{2_j}, v_{1_j} and v_{2_i}, v_{1_j} and v_{2_j}.

Program Transformation Operators The key idea to enable the neural network encoder to learn a set of diverse code features without the need for labeled data is that we can generate multiple versions of a program without changing the semantic of it. To do so, we are relying on program analysis to apply a set of program transformation operators to generate such different versions. Although there are many methods for transforming the code (Rabin, Islam, and Alipour 2020), we mainly apply three transformations in this work, which are variable renaming, adding dead code (unused statements), and permute statement to reflect different ways to change the structure of the AST, the more sophisticated the change is, the better the neural network encoder can learn. Figure 3 illustrates how the AST structure changes with the corresponding transformation operator. Further transformation operators will be considered in the future.

- **Variable Renaming (VN)**: Variable renaming is a refactoring method that renames a variable in code, where the new name of the variable is taken randomly from a set of variable vocabulary in the training set. Noted that each time this operator is applied to the same program, the variable names are renamed differently. This operator does not change the structure of the AST representation of the code, it only changes the textual information, which is a feature of a node in the AST. With this operator, we want the NN to understand that even the change in textual details does not affect the semantic meaning of the source code. This is inspired by a recent finding of Zhang et al. (2020), it is proposed that the source code model should be equipped with adversarial examples of token changes.
- **Unused Statement (US)** is the operator to insert dead code fragments, such as unused statement(s) to a randomly selected basic block in the code. Each time the operator is applied, we traverse the AST to identify the blocks and randomly select one block to insert predefined dead code fragments into it. This operator will add more nodes to the AST. For this, we want the NN still to learn how to catch the similarity between two similar similar programs even though the number of nodes in the tree structure has increased.
- **Permute Statement (PS** is to swap two statements that have no dependency on each other in a basic block in the code. Each time the operator is applied, we traverse the AST and analyze the data dependency to extract all of the possible pairs swap-able statements. If a program only contains one such pair, it will generate the same output every time we apply the operator, otherwise the output will be different. This operator does not add nodes into the AST but it will change the position of the subtrees in the AST. We want the NN to be able to detect the two similar trees even if the locations of the subtrees have changed.

.

Neural Network Encoder for Source Code The neural network can also be called as an *encoder*. The encoder receives the intermediate representation (IR) of code and maps it into a code vector embedding $\vec{v}$ (usually a combination of various kinds of code elements), then $\vec{v}$ can be fed into the next layer(s) of a learning system and trained for an objective function of the specific task of the learning system. For example, in Code2vec (Alon et al. 2019b), $\vec{v}$ is a combination of different AST paths. In GGNN (Allamanis, Brockschmidt, and Khademi 2018) or TBCNN (Mou et al. 2016), $\vec{v}$ is a combination of AST nodes.

In this work, we choose Tree-based Convolutional Neural Network (Mou et al. 2016) (TBCNN), Abstract Syntax Tree Neural Network (Zhang et al. 2019) (ASTNN), and Code2vec (Alon et al. 2019b) as the encoders. We choose these 3 because all of them work on the AST representation of code. There are also other techniques, such as GGNN (Allamanis, Brockschmidt, and Khademi 2018), Graph2vec (Narayanan et al. 2017) that represent the programs as graphs and use graph learning techniques to learn the graph. However, most of them, especially the graph-

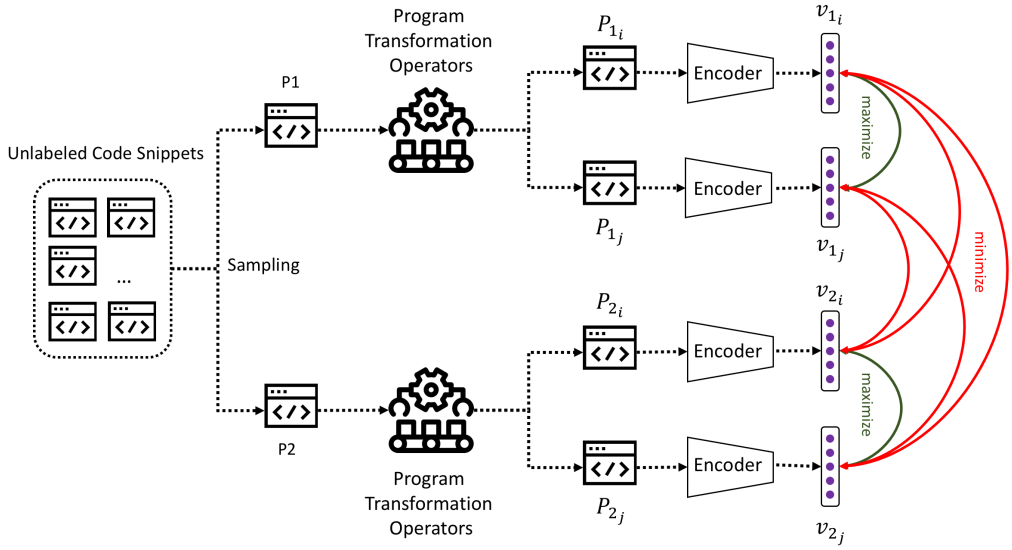

Figure 2: Overview of our approach

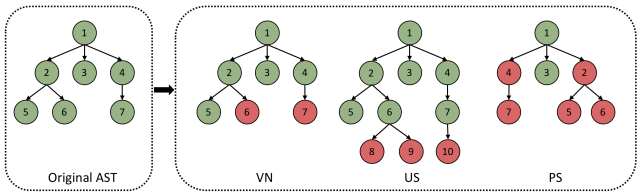

Figure 3: Example of how the AST structure is changed with different transformation operators

based models, are unable to scale and generalize for different programming languages. Although the graph representation proposed by Narayanan et al. (2017); Allamanis, Brockschmidt, and Khademi (2018) has been proved to work well on tasks, such as supervised clone detection, code summarization, variable name prediction, etc., choosing the suitable edges to be included in the graph representations for such tasks can be time-consuming and not generalizable. LambdaNet (Wei et al. 2020) is another graph-based model that also contains semantic edges designed specifically for the type prediction task. As such, it is not straightforward to transfer a pre-trained graph learning model through different code learning tasks and it is not easy to scale the graph representation of code into multiple languages.

Contrastive Learning Loss Let $\text{sim}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}^\top \boldsymbol{v}/\|\boldsymbol{u}\|\|\boldsymbol{v}\|$ denote the dot product between ℓ_2 normalized $\boldsymbol{u}$ and $\boldsymbol{v}$ (i.e. cosine similarity). Then the loss function for a positive pair of examples (i, j) is defined as

$$\ell_{i,j} = -log \frac{exp(sim(v_i, v_j))}{\sum_{k=1}^{2N} \mathbf{1}_{k \neq i} \exp(\text{sim}(v_i, v_k))} \qquad (1)$$

where $\mathbf{1}_{k \neq i} \in \{0, 1\}$ is an indicator function evaluating to 1 iff $k \neq i$. The final loss is computed across all positive pairs, both (i, j) and (j, i), in a mini-batch, which can be written as:

$$L = \frac{1}{2N} \sum_{k=1}^{N} [\ell(2k-1, 2k) + \ell(2k, 2k-1)] \qquad (2)$$

This loss is to pre-train the encoder to classify the positive P among all P_k using the normalizing denominator to define possible labels.

Use Cases

Code Embedding Vectors for Unsupervised Tasks

Code Clustering We use the code clustering task to demonstrate that the encoder is able to produce vectors such that the vectors that represent for semantically similar programs should be close in the vector space. Code clustering task is to put similar code snippets automatically into the same cluster without any supervision. Given the code vectors $\vec{v}$ produced by the pre-trained InferCode for any code snippets, we can realize the task by defining a similarity metric based on Euclidean distance and applying a clustering algorithm such as K-means(Kanungo et al. 2002).

Fine-Tuning for Supervised Learning Tasks

The paradigm to make a good use of large amount of unlabelled data is *self-supervised pretraining followed by a supervised fine-tuning* (Hinton, Osindero, and Teh 2006; Chen et al. 2020), which reuses parts (or all) of a trained neural network on a certain task and continue to train it or simply using the embedding output for other tasks. Such fine-tuning processes usually have the benefits of (1) speeding up the training as one does not need to train the model from randomly initialized weights and (2) improving the generalizability of the downstream model even when there are only small datasets with labels. The encoder of Corder serves as a pretrained model, in which the weights resulted from

the self-supervised learning are transferred to initialize the model of the downstream supervised learning task.

Code classification We use *code classification* (Mou et al. 2016) as a downstream task to demonstrate the usefulness of the fine-tuning process. This task is to, given a piece of code, classify the functionality class it belongs to.

Method name prediction We use *method name prediction* (Alon et al. 2019b) as the second downstream task. This task is to, given a piece of code (without its function header), predict a meaningful name that reflects the functionality of the code. .

Empirical Evaluation

General Settings To train our model, we collect a large number of Java repositories from Github, to ensure the quality, we only choose the repository that has more than 5 stars, which results in 5000 repositories (around 4 million files). Since we perform the transformation at the function level, we extract the functions from all of the files as the training code snippets. To avoid the same code snippet duplicated in multiple locations, we removed duplicates at the project level, file level, and method level. We do this by taking hashes of these entities and by comparing these hashes. Noted that the size of an AST representation of a code snippet can be large (up to 7000 nodes), which makes it difficult to train in large batch on GPU size so that we remove the ASTs that have a size larger than 1000. After removing duplicates and large ASTs, the corpus contains 2.5 million code snippets. Then, we parse all the snippets into ASTs using SrcML (Collard, Decker, and Maletic 2013). We also perform the transformation on all of the ASTs to get the transformed ASTs based on the transformation operators described in Section . Having the ASTs as well as the transformed ASTs, we train 3 instances of Corder for TBCNN (Corder-TBCNN), ASTNN (Corder-ASTNN), and Code2vec (Corder-Code2vec) by using the NCE as the objective loss function and choose Adam (Kingma and Ba 2014) as the optimizer with an initial learning rate of 0.001 on an Nvidia Tesla P100 GPU. We train each of the models with 100 epochs and choose 100 as the batch size.

Code Clustering

Datasets, Metrics, and Baselines We use two datasets for this task. The first is the OJ dataset that contains 52,000 C code snippets known to belong to 104 classes (Mou et al. 2016). The second is the Sorting Algorithm (SA) dataset used in (Nghi, Yu, and Jiang 2019), which consists of 10 classes of sorting algorithm written in Java, each algorithm has approximately 1000 code snippets. These two datasets have been used for code classification task (Mou et al. 2016; Nghi, Yu, and Jiang 2019), where a fraction of the data is split for training and the remaining fraction is for testing. In this work, we use these datasets for the code clustering task. Our clustering task here is to cluster all the code snippets (without class labels) according to the similarity among the code vectors: For the OJ dataset, we use K-means (K=104) to cluster the code into 104 clusters; For the SA dataset, we use K-means (K=10) to cluster the code. Then we use the class labels in the datasets to check if the clusters are formed appropriately. We use the Adjusted Rand Index (Santos and Embrechts 2009) as the metric to evaluate the clustering results. For the baselines, if we treat source code as text, the self-supervised learning techniques in NLP can also be applied for code. As such, we include two well-known baselines from NLP, Word2vec (Mikolov et al. 2013), and Doc2vec (Le and Mikolov 2014). We also include another baseline from (Hill, Cho, and Korhonen 2016), a state-of-the-art method to learn sentence representation. This method uses a Sequential Denoising Auto Encoder (SAE) method to encode the text into an embedding, and reconstruct the text from such embedding. We also compare with two baselines for code modeling, Code2vec (Alon et al. 2019b) and Code2seq (Alon et al. 2019a). Code2vec works by training a path encoder on bag-of-paths extracted from the AST. The path encoder will encode the paths into an embedding $\vec{v}$, then use $\vec{v}$ to predict the method name. Code2seq shares a similar principle, but the $\vec{v}$ is used to generate a text summary of code. In either case, we use the path encoders of Code2vec and Code2seq to produce the code vectors and also perform the same clustering process as InferCode.

Results Table 1 shows the results of code clustering using different models. The variants of Corder that are trained on different encoders (TBCNN, ASTNN, Code2vec) performs the best for both datasets. The pre-trained Code2vec from (Alon et al. 2019b) and the pre-trained Code2seq from (Alon et al. 2019a) performs worse than our Corder instances significantly. This is reasonable because Code2vec and Code2seq are trained specifically for the method name prediction and code summarization tasks so that the representations provided by them are not as good as ours. The NLP methods underperform other code learning methods. We will provide a deeper analysis of the clusters by providing visualizations of the vectors produced by different methods.

Table 1: Results of Code Clustering in Adjusted Rand Index (ARI)

Model	Performance (ARI)	
	OJ Dataset (C)	SA Dataset (Java)
Word2vec	0.2853	0.2451
Doc2vec	0.4235	0.2986
SAE	0.4178	0.3161
Code2vec	0.5879	0.5129
Code2seq	0.5345	04934
Corder -TBCNN	**0.7146**	**0.6892**
Corder - ASTNN	**0.7558**	**0.6901**
Corder - Code2vec	**0.7245**	**0.6521**

Fine-Tuning for Supervised Learning Tasks

Datasets, Metrics, and Baselines

Code Classification We again use the OJ Dataset for this task. We split this dataset into three parts for training, test-

Table 2: Results of Code Classification in Accuracy with Fine-Tuning (FT) vs Supervised Training from Scratch (Sup) on the OJ dataset

Approach	FT (1%)	FT (10%)	FT (100%)	Sup
TextCNN	-	-	-	88.7%
2-layer BiLSTM	-	-	-	88.0%
Corder-TBCNN	70.4%	86.1%	**98.0%**	94%
Corder-ASTNN	73.12%	88.9%	**98.50%**	97.8%

ing, and validation by the ratio of 70:20:10. Out of the training data, we feed X% to the neural model, where X = 1, 10, 100. We then initialize the neural model either randomly or with the weights from the pre-trained InferCode. Therefore, we have four settings for training the supervised model for comparison: fine-tuning the TBCNN encoder with 1%, 10%, or 100% of the labeled training data respectively, and the randomly initialized model. Using only 1% or 10% is to demonstrate that given a pre-trained model, one only needs a small amount of labeled data to achieve reasonably good performance for the downstream task.

We use the accuracy metric widely used for classification tasks. As the baselines, we include the ASTNN (Zhang et al. 2019) trained from scratch, which is a state-of-the-art model for code classification on the OJ dataset, and TextCNN (Kim 2014) and Bi-LSTM (Schuster and Paliwal 1997) trained with 100% of the training data, which are widely used for text classification.

Method Name Prediction We use the Java-Small dataset widely used as a benchmark for method name prediction and has been used in Code2vec (Alon et al. 2019b) and Code2seq (Alon et al. 2019a). This dataset has already been split into three parts, namely training, testing, and validation. We perform the same evaluation protocol as the code classification task by fine-tuning the model with 1%, 10%, and 100% of the labeled training data, in contrast to random initialization of the model without fine-tuning. To predict the method name, we follow Code2vec to use the code vector $\vec{v}$ to predict the embedding of a method name from a lookup table (see Section 4.2 in Code2vec (Alon et al. 2019b)). We measure prediction performance using precision (P), recall (R), and F1 scores over the sub-words in generated names, following the metrics used by Alon et al. (2019b). For example, a predicted name `result_compute` is considered as an exact match of the ground-truth name `computeResult`; predicted `compute` has full precision but only 50% recall; and predicted `compute_model_result` has full recall but only 67% precision.

Results Table 2 shows the results for code classification. Fine-tuning on 10% of the training data gets comparable results with the NLP baselines. Fine-tuning on 100% of the training data gets a 4% improvement over training from scratch for TBCNN, and 0.7% improvement over training from scratch for ASTNN.

Table 3: Result of Method Name Prediction in F1 with Fine-Tuning (FT) vs Supervised Training from Scratch (Sup) on the Java-Small Dataset

Approach	FT (1%)	FT (10%)	FT (100%)	Sup
2-layer BiLSTM	-	-	-	31.56%
Transformer	-	-	-	32.33%
TextCNN	-	-	-	25.67%
Corder-Code2vec	17.44%	25.22%	**28.41%**	19.59%
Corder-Code2seq	20.31%	38.54%	**47.33%**	43.02%

Table 3 shows the results for method name prediction. Fine-tuning on 100% of the training data gets 9% improvement over training from scratch for Code2vec, and 4% improvement over training from scratch for Code2seq.

Summary

Through the evaluation of code clustering, we have shown that vectors generated by different Corder instances are useful. The vectors of the semantic equivalent programs are close in vector space, which explains the good performance of the K-means clustering similar snippets into the same category. The results of the fine-tuning process on code classification and method name prediction also show that the pre-trained Corder helps significantly improve the efficiency of these tasks.

Analysis and Ablation Study

In this section, we perform some analysis and ablation studies to measure how different design choices can affect the performance of Corder.

Impact of Different Transformation Operators We perform an ablation study to measure how each transformation operator affects the performance of particular code learning tasks. We train Corder with three encoders: TBCNN, ASTNN, and Code2seq with similar settings Evaluation Section. but we only use one operator at a time, resulting in six pre-trained models. Then we perform the code clustering task, the fine-tuning process on the two classification task, also similar to the Evaluation Section. Table 4 shows that the Unused Statement operator consistently among the operator that perform the best for most of the tasks.

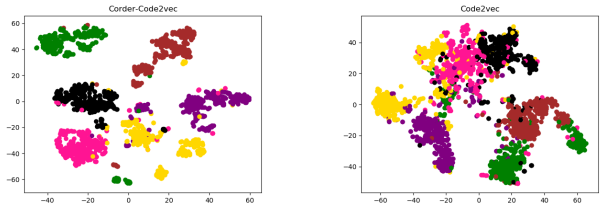

Figure 4: Visualization of the Code Vectors of the Programs from 6 classes in the OJ Dataset produced by Corder-Code2vec and Code2vec

Cluster Visualization To help understand why the vectors produced by Corder pre-training are better than the vectors produced by others, we visualize the vectors of the

Table 4: Ablation Study Result on the Impact of Different Transformation Operators on Tasks - Code Clustering (CC), Fine-tuning over Code Classification (FT-CC), Fine-tuning over Method Name Prediction (FT-MNP)

Methods	Ops	Tasks		
		CC (ARI)	FT-CC (Acc%)	FT-MNP (F1%)
Corder-TBCNN	VR	0.5823	94.98%	-
	US	**0.6821**	**97.21%**	**-**
	PS	0.6596	96.01%	-
	All	**0.7146**	98.24%	-
Corder-ASTNN	VR	0.5651	97.8%	-
	US	**0.6981**	98.23%	**-**
	PS	0.6981	**98.35%**	-
	All	0.7558	98.50%	-
Corder-Code2seq	VR	0.5452	-	44.68%
	US	**0.6981**	-	**47.34%**
	PS	0.6431	-	45.91%
	All	0.7245	-	50.87%

programs from the OJ dataset that have been used for the code clustering. We choose the embeddings produced by Code2vec (Alon et al. 2019b) and the Corder pretraining of Code2vec for the first 6 classes of the OJ dataset, then we use T-SNE (Maaten and Hinton 2008) to reduce the dimension of the vectors into two-dimensional space and visualize. As shown in Figure 4, (1) the vectors produced by Corder-Code2vec group similar code snippets into the same cluster with clearer boundaries, and (2) The boundaries among clusters produced by Code2vec are less clear, which makes it more difficult for the K-means algorithm to cluster the snippets correctly. This is aligned with the performance of the code clustering task (Table 1).

Conclusion

We have proposed Corder, a self-supervised learning approach that can leverage large scale unlabeled data of source code. Corder works by training the network over a contrastive learning objective to compare similar and dissimilar programs that are generated from the concept of semantic-preserving program transformation. These programs are syntactical diverse but semantical equivalent. The goal of the contrastive learning methods is to minimize a distance between the representations of similar programs (positives) and maximize the distance between dissimilar programs (negatives). We adapted Corder into 3 tasks: code clustering, fine-tuning for code classification, fine-tuning for method name prediction and find that Corder pre-training significantly improves accuracy on these tasks.

References

Allamanis, M.; Brockschmidt, M.; and Khademi, M. 2018. Learning to Represent Programs with Graphs. In *ICLR*.

Alon, U.; Brody, S.; Levy, O.; and Yahav, E. 2019a. code2seq: Generating Sequences from Structured Representations of Code. In *ICLR*.

Alon, U.; Zilberstein, M.; Levy, O.; and Yahav, E. 2019b. Code2Vec: Learning Distributed Representations of Code. In *POPL*, 40:1–40:29.

Bielik, P.; and Vechev, M. 2020. Adversarial Robustness for Code. *arXiv preprint arXiv:2002.04694* .

Brockschmidt, M.; Allamanis, M.; Gaunt, A. L.; and Polozov, O. 2019. Generative Code Modeling with Graphs. In *7th ICLR*.

Bromley, J.; Guyon, I.; LeCun, Y.; Säckinger, E.; and Shah, R. 1994. Signature verification using a" siamese" time delay neural network. In *Advances in neural information processing systems*, 737–744.

Chen, T.; Kornblith, S.; Norouzi, M.; and Hinton, G. 2020. A simple framework for contrastive learning of visual representations. *arXiv preprint arXiv:2002.05709* .

Chen, X.; Liu, C.; and Song, D. 2018. Tree-to-tree neural networks for program translation. In *NeurIPS*, 2547–2557.

Chen, Z.; and Monperrus, M. 2019. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061* .

Collard, M. L.; Decker, M. J.; and Maletic, J. I. 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *ICSM*, 516–519.

Dahl, G. E.; Stokes, J. W.; Deng, L.; and Yu, D. 2013. Large-scale malware classification using random projections and neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 3422–3426.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* .

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* .

Fernandes, P.; Allamanis, M.; and Brockschmidt, M. 2019. Structured Neural Summarization. In *7th ICLR*.

Fernando, B.; Bilen, H.; Gavves, E.; and Gould, S. 2017. Self-supervised video representation learning with odd-one-out networks. 3636–3645.

Gidaris, S.; Singh, P.; and Komodakis, N. 2018. Unsupervised representation learning by predicting image rotations. *arXiv preprint arXiv:1803.07728* .

Gu, X.; Zhang, H.; and Kim, S. 2018. Deep code search. In *40th ICSE*, 933–944.

Gu, X.; Zhang, H.; Zhang, D.; and Kim, S. 2017. DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning. In *IJCAI*, 3675–3681.

Gupta, R.; Kanade, A.; and Shevade, S. 2019. Neural Attribution for Semantic Bug-Localization in Student Programs. In *NeurIPS*, 11861–11871.

Hill, F.; Cho, K.; and Korhonen, A. 2016. Learning distributed representations of sentences from unlabelled data. *arXiv preprint arXiv:1602.03483* .

Hinton, G. E.; Osindero, S.; and Teh, Y.-W. 2006. A fast learning algorithm for deep belief nets. *Neural computation* 18(7): 1527–1554.

Hu, X.; Li, G.; Xia, X.; Lo, D.; and Jin, Z. 2018. Deep code comment generation. 200–210.

Hussain, Y.; Huang, Z.; Zhou, Y.; and Wang, S. 2020. Deep transfer learning for source code modeling. *International Journal of Software Engineering and Knowledge Engineering* 30(05): 649–668.

Ingram, B. 2018. A Comparative Study of Various Code Embeddings in Software Semantic Matching. https://github.com/waingram/code-embeddings.

Jiang, L.; Liu, H.; and Jiang, H. 2019. Machine learning based recommendation of method names: how far are we. In *34th ASE*, 602–614.

Kanungo, T.; Mount, D. M.; Netanyahu, N. S.; Piatko, C. D.; Silverman, R.; and Wu, A. Y. 2002. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.* 24(7): 881–892.

Kim, D.; Cho, D.; and Kweon, I. S. 2019. Self-supervised video representation learning with space-time cubic puzzles. In *AAAI*, volume 33, 8545–8552.

Kim, K.; Kim, D.; Bissyandé, T. F.; Choi, E.; Li, L.; Klein, J.; and Traon, Y. L. 2018. FaCoY: a code-to-code search engine. 946–957.

Kim, Y. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* .

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

Kiros, R.; Zhu, Y.; Salakhutdinov, R. R.; Zemel, R.; Urtasun, R.; Torralba, A.; and Fidler, S. 2015. Skip-thought vectors. In *NeurIPS*, 3294–3302.

Korbar, B.; Tran, D.; and Torresani, L. 2018. Cooperative learning of audio and video models from self-supervised synchronization. In *NeurIPS*, 7763–7774.

Le, Q.; and Mikolov, T. 2014. Distributed representations of sentences and documents. 1188–1196.

Li, J.; He, P.; Zhu, J.; and Lyu, M. R. 2017. Software defect prediction via convolutional neural network. In *IEEE QRS*, 318–328.

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. 2016. Gated Graph Sequence Neural Networks. In *ICLR*.

Maaten, L. v. d.; and Hinton, G. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research* 9(Nov): 2579–2605.

Mahendran, A.; Thewlis, J.; and Vedaldi, A. 2018. Cross pixel optical-flow similarity for self-supervised learning. In *Asian Conference on Computer Vision*, 99–116.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013. Distributed representations of words and phrases and their compositionality. In *NeurIPS*, 3111–3119.

Mou, L.; Li, G.; Zhang, L.; Wang, T.; and Jin, Z. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI*.

Narayanan, A.; Chandramohan, M.; Venkatesan, R.; Chen, L.; Liu, Y.; and Jaiswal, S. 2017. graph2vec: Learning Distributed Representations of Graphs. *CoRR* abs/1707.05005.

Nghi, B. D. Q.; Yu, Y.; and Jiang, L. 2019. Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification. In Wang, X.; Lo, D.; and Shihab, E., eds., *26th SANER*, 422–433. doi:10.1109/SANER.2019.8667995.

Nix, R.; and Zhang, J. 2017. Classification of Android apps and malware using deep neural networks. In *International Joint Conference on Neural Networks*, 1871–1878.

Pradel, M.; and Sen, K. 2018. DeepBugs: A learning approach to name-based bug detection. *ACM on Programming Languages* 2(OOPSLA): 147.

Rabin, M.; Islam, R.; and Alipour, M. A. 2020. Evaluation of Generalizability of Neural Program Analyzers under Semantic-Preserving Transformations. *arXiv preprint arXiv:2004.07313* .

Sachdev, S.; Li, H.; Luan, S.; Kim, S.; Sen, K.; and Chandra, S. 2018. Retrieval on Source Code: A Neural Code Search. In *2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 31–41. ISBN 9781450358347. doi:10.1145/3211346.3211353.

Santos, J. M.; and Embrechts, M. 2009. On the use of the adjusted rand index as a metric for evaluating supervised classification. In *International conference on artificial neural networks*, 175–184.

Schroff, F.; Kalenichenko, D.; and Philbin, J. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 815–823.

Schuster, M.; and Paliwal, K. K. 1997. Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.* 45(11): 2673–2681.

Wan, Y.; Zhao, Z.; Yang, M.; Xu, G.; Ying, H.; Wu, J.; and Yu, P. S. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *33rd ASE*, 397–407. New York, NY, USA. ISBN 9781450359375. doi: 10.1145/3238147.3238206.

Wei, J.; Goyal, M.; Durrett, G.; and Dillig, I. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. *arXiv preprint arXiv:2005.02161* .

Yang, X.; Lo, D.; Xia, X.; Zhang, Y.; and Sun, J. 2015. Deep Learning for Just-in-Time Defect Prediction. In *IEEE QRS*, 17–26.

Yasunaga, M.; and Liang, P. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. *arXiv preprint arXiv:2005.10636* .

Zhang, H.; Li, Z.; Li, G.; Ma, L.; Liu, Y.; and Jin, Z. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *34th AAAI*.

Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; and Liu, X. 2019. A novel neural source code representation based on abstract syntax tree. In *41st ICSE*, 783–794.

Zhang, R.; Isola, P.; and Efros, A. A. 2016. Colorful image colorization. In *European conference on computer vision*, 649–666.

Zhou, Y.; Liu, S.; Siow, J. K.; Du, X.; and Liu, Y. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *NeurIPS*, 10197–10207.