Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification

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

Yijun Yu
Center for Research of Computing
The Open University, UK
yijun.yu@open.ac.uk

Lingxiao Jiang
School of Information Systems
Singapore Management University
lxjiang@smu.edu.sg

***Abstract*—Algorithm classification is to automatically identify the classes of a program based on the algorithm(s) and/or data structure(s) implemented in the program. It can be useful for various tasks, such as code reuse, code theft detection, and malware detection. Code similarity metrics, on the basis of features extracted from syntax and semantics, have been used to classify programs. Such features, however, often need manual selection effort and are specific to individual programming languages, limiting the classifiers to programs in the same language.**

To recognize the similarities and differences among algorithms implemented in different languages, this paper describes a framework of Bilateral Neural Networks (Bi-NN) that builds a neural network on top of two underlying sub-networks, each of which encodes syntax and semantics of code in one language. A whole Bi-NN can be trained with bilateral programs that implement the same algorithms and/or data structures in different languages and then be applied to recognize algorithm classes across languages.

We have instantiated the framework with several kinds of token-, tree- and graph-based neural networks that encode and learn various kinds of information in code. We have applied the instances of the framework to a code corpus collected from GitHub containing thousands of Java and C++ programs implementing 50 different algorithms and data structures. Our evaluation results show that the use of Bi-NN indeed produces promising algorithm classification results both within one language and across languages, and the encoding of dependencies from code into the underlying neural networks helps improve algorithm classification accuracy further. In particular, our custom-built dependency trees with tree-based convolutional neural networks achieve the highest classification accuracy among the different instances of the framework that we have evaluated. Our study points to a possible future research direction to tailor bilateral and multilateral neural networks that encode more relevant semantics for code learning, mining and analysis tasks.

***Index Terms*—cross-language mapping, program classification, algorithm classification, code embedding, code dependency, neural network, bilateral neural network**

I. Introduction

Algorithm classification is a long-standing problem related to program reuse and synthesis [8], [11], [47]. It aims to assign class labels or concepts to programs based on code structures and semantics [28]. Automated classification of a piece of code could ease a number of software engineering tasks, such as program comprehension [28], concept location [44], algorithm plagiarism detection [55], bug fix classification [27], [41] and malware detection [10]. The algorithm labels for the code can serve to some extent as the summary of the code [39], which help to modularize, abstract, analyze, and reuse the code.

Even though it is different from the problem of program equivalence checking [7], this problem remains challenging because what is considered to be the "same algorithm" can look different under different situations. An "appropriate" classification should not only take sufficiently detailed information about the code into consideration, but also ignore irrelevant details depending on the abstraction level of an algorithm class. For example, a program A implementing `bubblesort` for an integer array may not be the "same" as a second program B implementing `bubblesort` for integers stored in a linked list, and both of them may not be the "same" as a third program C implementing `mergesort`, while the three programs may all be considered the "same" as variant of sorting algorithms.

It can be even a greater challenge to bring the benefit of algorithm classification across different programming languages, so as to facilitate program reuse and synthesis across languages, reducing the need of reimplementing the "same" algorithms in different languages repeatedly.

Past studies on algorithm classification can neglect differences in different programming languages by simply processing the programs as a bag or a sequence of tokens or simple call graphs [7], [44], which do not utilize the rich code syntactic structures and semantics, or by taking advantages of system-level APIs used and/or higher-level descriptions available in human languages [52], [54]. On the other hand, program classification and functional cloning studies utilizing code syntax structures are mostly limited to individual languages [22], [23], [31], [51], which have not been adapted to the classification problem across languages.

Our research goal here is to find a suitable representation for given pieces of code in different languages that can be used to identify the algorithm classes of the code. Specifically, we present a framework of *bilateral* neural networks (Bi-NN), an idea adapted from the area of neural machine translation [9], [16], [50] and Siamese neural networks [9], [32], aiming to generalize a previous study on cross-language program classification [6], to encode code syntactic and semantic information for programs written in two different languages, and train the bilateral neural networks to recognize code implementing the same algorithms across languages. Two

technical aspects of the framework are important for the effectiveness of cross-language algorithm classification:

(1) One is to build a bilateral structure of neural networks that consists of *two* (thus the name *bilateral*) underlying neural networks, each of which encodes code in one language, and another classification model on top of the two to link them together.
(2) The other is to explicitly embed code dependencies (e.g., variable def-use relations) into the intermediate representations (IR) of code for the neural networks to learn code representations.

Such a framework enables us to explore different ways to use different kinds of code intermediate representations with different kinds of neural networks to search for optimal algorithm classification solutions. Every instance of Bi-NN can be trained with bilateral programs that implement the same algorithms and/or data structures in two different languages. The trained Bi-NN models can then be applied to recognize code implementing the algorithms and/or data structures in different languages.

In this paper, we instantiate the Bi-NN framework with token-, sequence-, tree-, and graph-based machine learning techniques to train different Bi-NN models on large code bases to classify programs into different algorithms. Empirical evaluations on two code bases, (1) 52000 C++ files from previous studies written by computer science students implementing 104 different algorithms and (2) 4932 unique Java and 4732 unique C++ single-file programs from GitHub implementing 50 different algorithms, show that the Bi-NN model trained by tree-based convolutional neural networks (TBCNN) using our custom-built dependency trees (DTs) representing code syntax and semantics achieves a reasonable accuracy of 86% in classifying cross-language programs according to the ground-truth algorithm class labels. The accuracy of this model (referred to as a Bilateral Dependency Tree Based CNN model, or Bi-DTBCNN in short) is the highest among several other Bi-NN models based on bags-of-words, n-gram, tf-idf, long-short term memeory (LSTM), gated graph neural network (GGNN), etc. Even for the simpler problem of algorithm classification in a single language, our evaluations show that DTBCNN models (without using the bilateral structure) achieve the highest classification accuracy of 93% among different models we have evaluated.

The main conceptual and empirical contributions of the paper are as follows:

- We generalize a bilateral neural network (Bi-NN) framework for the cross-language algorithm classification task;
- We adapt various learning techniques, including n-grams, bags-of-words, tf-idf, tree-based convolution neural networks (TBCNN), long short-term memory (LSTM), and gated graph neural networks (GGNN) to instantiate the bilateral code representation framework to represent both syntax and semantics for algorithm classification;
- We custom-build a *dependency* tree-based convolutional neural network (DTBCNN) as an extension to TBCNN to encode semantics for more accurate classification;
- We collect a benchmark of 9664 unique programs in Java and C++ implementing 50 algorithms, and evaluate the performance of various Bi-NN models. The results demonstrate the effectiveness of Bi-NN models for cross-language algorithm classification. In particular, Bi-DTBCNN achieves the highest classification accuracy in our evaluation.

The rest of the paper is organized as follows. Section II overviews the Bi-NN framework for the task of algorithm classification across languages. Section III describes various instantiations of the Bi-NN framework such as dependency trees, gated graphs, LSTM, and n-gram. Section IV evaluates these instantiations, providing empirical results to identify best performing instances, and discusses possible threats to validity. Section V discusses related work on algorithm classification and machine learning techniques, and finally Section VI concludes.

II. Framework of Bilateral Neural Networks

A. Work Flow

For the purpose of cross-language algorithm classification, we want to align code representations for different programming languages so that a model learned from algorithms in one programming language can be used to recognize code implementing the algorithms in another language. To achieve this purpose, a work flow based on the Bilateral Neural Networks (Bi-NN) model can be used; it consists of two stages (see Figure 1): (1) training of an instance of Bi-NN with a set of input program pairs in two languages, and (2) classification by applying the trained Bi-NN models to pairs of test programs.

The training stage takes in pairs of programs in two programming languages as input. Each program in a pair has an algorithm class label, indicating whether the pair implements the same algorithm or not. Parsers are used to convert the input programs to certain intermediate representations (IR) that expose code syntax and semantics, which can be based on tokens, sequences, trees, or graphs. Then, the IR of all the input pairs, either implementing the same algorithm or not, is used to train a Bi-NN model that minimizes the classification errors for the inputs.

The classification stage takes in a pair of test programs in two programming languages without knowning their algorithm class labels, converts the inputs into the same kind of IR as those used in the training stage, and uses the trained Bi-NN model to predict the likelihood for the two test programs belonging to the same algorithm class. We call this classification a binary classification as its output only tells whether or not two programs belong to the same algorithm class.

One can also utilize such binary classifications to determine the algorithm class for one given test program. For each known algorithm class, one can pick an arbitrary program from the class to form a pair of input programs with the given test program, and feed them into the trained Bi-NN model to predict the likelihood for the pair to be in the same algorithm class. Repeating this step for every known algorithm class produces a set of likelihoods; each indicates the likelihood of the given test

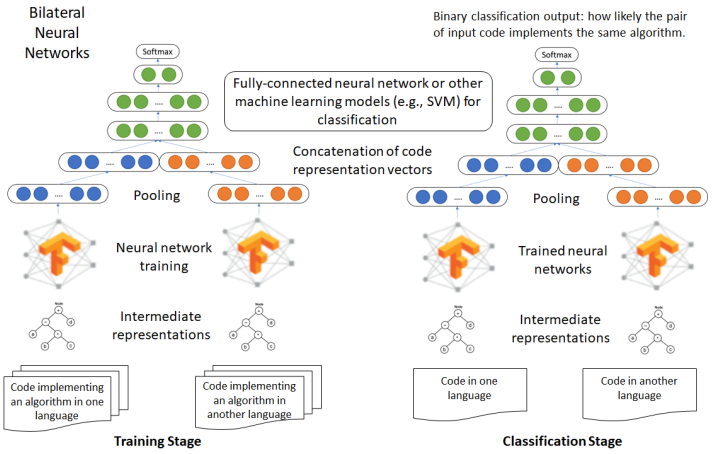

Fig. 1. Overview of the cross-language algorithm classification work flow

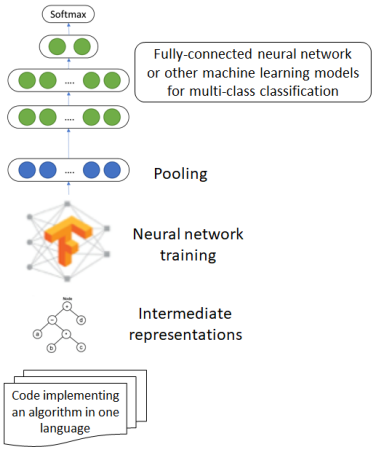

Fig. 2. Setting of single-language algorithm classification

program belonging to the corresponding algorithm class. From these predictions, the algorithm class label with the highest likelihood is assigned to the given test program. If none of the likelihood is high enough (e.g., above 0.5), one may choose to leave the test program as unknown.

The capability of the classification is naturally limited by the known algorithm classes used for training. In our evaluation later (Section IV), we show the effectiveness of such classifications for the numbers of algorithm classes ranging from 10 to 104.

B. The Design of Bi-NN

The key component of the proposed work flow is the Bi-NN. It is constructed as two underlying subnetworks and another classification model (which can be a neural network or other kinds of classifiers) on top of the two.

Each of the two underlying subnetworks can be any neural network, such as a LSTM, GGNN, or others, as we show later in this paper. During the training, each subnetwork takes in code representations of one-specific language to recognize the code in that language. For our cross-language algorithm classification task, the two subnetworks are designed to take in code representations of different languages. If both subnetworks took in code representations of the same language, the Bi-NN could be suitable for single-language classification too (see the next subsection).

The classification model on top connects the two underlying subnetworks. It can be another neural network. As illustrated in Figure 1, It consists of (1) two pooling layers, each of which aggregates the code learning output from one of the two subnetworks, (2) a "joint feature representation layer" that concatenates the pooling outputs of the two subnetworks, (3) two or more fully connected hidden layers above the joint feature representation layer, and (4) a Softmax layer on the top to determine how likely the input code pair belongs to the same algorithm class. The layers of (3) and (4) essentially form a classifier that can be trained for the inputs from the layers below. They do not have to be neural networks, and they may be substituted by any classifier, such as Support Vector Machines (SVM [21]) and Random Forests [19].

Note that, when we instantiate the Bi-NN framework with different machine learning models, it becomes essentially the same as Siamese networks in the literature [9]. Siamese networks are a class of neural network architectures that contain two (or more) subnetworks, which are merged via an energy cost function over a joint layer on top of the subnetworks. Its high-level architecture is similar to the Bi-NN framework illustrated in Figure 1. Many choices for the subnetworks and the loss function over the joint layer can be adopted for different tasks. Section III provides ways to instantiate the Bi-NN framework with different concrete code representations and machine learning models for the task of cross-language algorithm classification. For cross-language training specifically when node types are used, we implement an additional alignment step to ensure the shared node types multiple language are mapped to the same node.

C. Single-Language Classification

Besides the task of cross-language algorithm classification, we can also use the Bi-NN framework to classify programs in a single language. That is, the input programs are all in one language only. Although we may use both sides of the Bi-NN framework as mentioned in the previous subsection to train the classification model, it is more straightforward to use "half" of the Bi-NN framework. As illustrated in Figure 2, by disabling the "joint feature representation layer" and the right side of the Bi-NN framework, we can obtain a classification model that can be trained with programs in one language to classify algorithm classes in the same language too.

III. Instantiations of Bi-NN

The Bi-NN may be instantiated with different kinds of structures to represent various syntax and semantic information of given pieces of code and different kinds of neural networks to learn the code representations. This section presents several variants that we consider to be promising for learning and classifying algorithms.

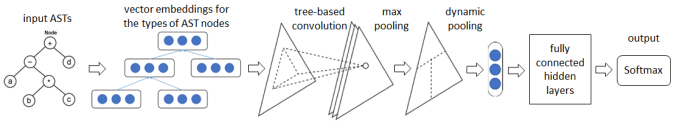

Fig. 3. Structure of a TBCNN, adapted from [31]

A. AST and Tree-Based Convolutional Neural Networks

Abstract syntax trees (AST) are a very commonly used code representation that faithfully encodes the syntax of a program, and given a well-formed AST, the responding program can be regenerated. Thus, it is a natural way to use AST as the intermediate representation and a tree-based neural network to learn the representation in the Bi-NN framework.

Mou et al. [31] have proposed to use tree-based convolutional neural networks (TBCNN) to learn AST and classify C++ programs. Figure 3 illustrates the structure of a TBCNN. Each AST node is represented as a vector by using an encoding layer that basically embeds AST node types into a continuous vector space where contextually similar node types are mapped to nearby high-dimension points in the vector space. For example, the node types 'while' and 'for' are similar because they are both loops and thus their vectors will be close to each other. Given an AST where every node is turned into a vector representation, Mou et al. [31] uses a CNN and a set of fixed-depth subtree filters sliding over the AST to "convolute" structural information of the entire tree. A dynamic pooling layer [46] is applied to deal with varying numbers of children of AST nodes to generate one high-dimension vector to represent the whole tree. Finally, they use a hidden layer and an output layer, similar to the common neural network on top in our Bi-NN framework, to classify the programs.

The TBCNN was used to classify algorithms written in one language only and it only encodes code syntax without explicit semantics, but it inspires us to improve it for cross-language classification that encodes more code semantics. We can instantiate the two subnetworks in the Bi-NN framework with two TBCNN for different languages, say, one for Java, and the other for C++, and then we can train a bilateral tree-based convolutional neural network model (Bi-TBCNN) for cross-language algorithm classification.

B. Dependency Trees and TBCNN

Although abstract syntax trees can faithfully encode the syntax of a program, it may not be obvious or explicit about various kinds of semantic information in the program, such as def-use relations, call relations, class inheritances, etc. Therefore, we consider encoding such semantic dependency information directly into abstract syntax trees so that the *dependency tree* based code representation can help tree-based neural networks to learn code semantics more accurately to recognize code of different algorithms.

Our basic idea is to insert additional nodes that represent some semantic information relations into AST to form *dependency trees*: given a program or a code snippet, it is first parsed into AST represented in the Pickle format using our tool; then, dependency relations, especially def-use relations, are extracted from the code using srcML and srcSlice [5], [12], and the nodes related to a Def are appended to the nodes representing the uses of the Def, and vice versa.

For example, Figure 4 (ignoring the boxes with dashed lines first) represents the AST of the following piece of code:

```
int a = 1; int b = 2; int x = a * b;
y = x + 1; z = x + 2;
```

In this example, the defs "`int a = 1`" and "`int b = 2`" affect the definition of the variable `x` in "`int x = a * b`", and `x` is used to compute both `y` and `z`. Therefore, the expanded *dependency tree* is a tree shown in Figure 4 (including the boxes with dashed lines), where (1) the subtrees in the original AST representing "`int x = a * b`" are duplicated and inserted as children of the nodes representing the definitions of `a` and `b` respectively; (2) the subtrees representing the definitions of `y` and `z` are duplicated and inserted as children of the node representing the definition of `x`; and symmetrically, (3) the subtrees representing the definitions of `a` and `b` are also duplicated and inserted as children of the nodes representing the uses of `a` and `b` respectively; and (4) the subtrees representing the definition of `x` are duplicated and inserted as children of the nodes representing the uses of `x`.

Notice that many subtrees are duplicated multiple times in our dependency trees. In the literature, program dependency *graphs* are typically used to avoid duplication, by referring to the dependencies as edges between depenters and dependees. However, our intuition is that representing the relations as trees and allowing such node duplication can have an advantage of separating the contexts of each use-def from each other to faciliatate context-sensitive learning while making it more efficient to train tree-based neural networks than graph-based ones. Therefore, we instantiate Bi-NN with dependency tree-extended tree-based convolutional neural networks to train cross-language algorithm classification models, which we simply call Bi-DTBCNN.

Apart from tree structure differences between DTBCNN and TBCNN, we also employ a different vector embedding strategy for the tree nodes to bootstrap the training of the tree-based neural networks with more code semantics.

Tree-node embedding for boostrapping the training: As illustrated in Figure 3, the training of TBCNN needs a vector representation for each tree node. Mou et al. [31] use the "coding criterion" from Peng et al. [40] to learn the vector for each AST node type. We adapt the skip-gram neural network model used for *word2vec* [29] to the context of AST nodes. The skip-gram model, given an input word in a sentence during training, looks at the words spatially nearby and picks one at random, and produces the probability for each word in the whole vocabulary to be a "nearby word" of the input word; i.e., it can be used to "predict the contextual words for an input word". When such a model is trained to produce the probabilities of nearby words, its hidden layers can produce numerical vectors representing the words, i.e., the word embeddings.

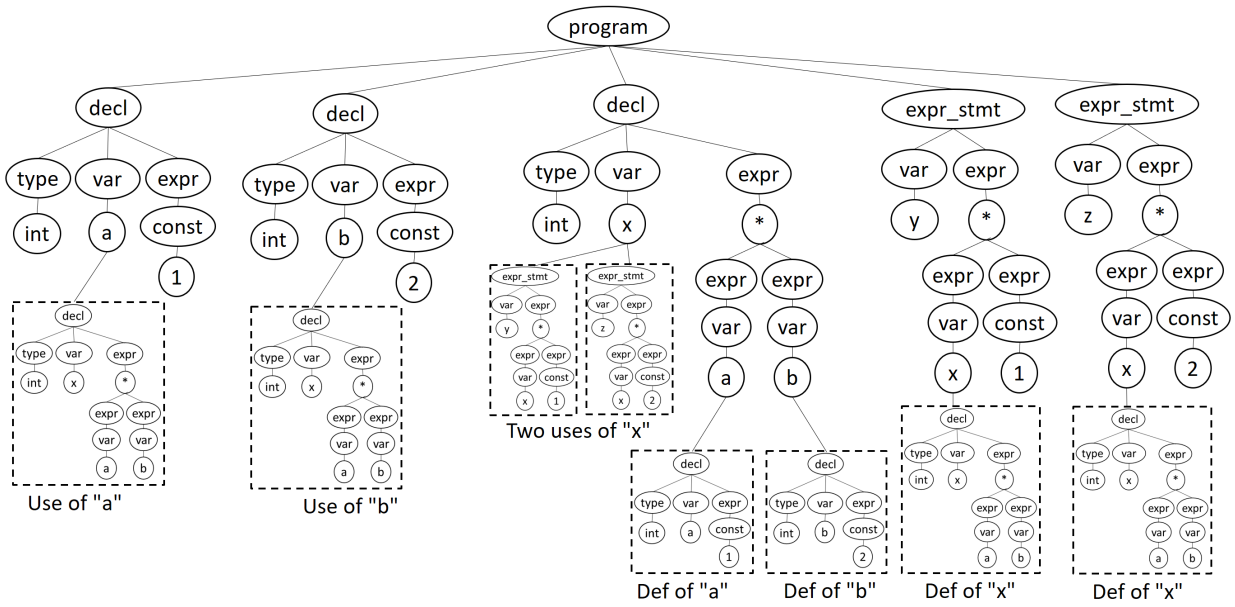

Fig. 4. A Sample AST Expanded into a Dependency Tree.

We apply this idea for the so-called *AST2vec* task. That is, we view AST node types as the vocabulary words and consider nodes to be "nearby" in the AST if they have parent-child or sibling relations, and train a skip-gram neural network model using all the AST generated from our code base to produce the probability for each node type in the whole vocabulary to be a child of any given node type. The size of the vocabulary of node types for a programming language is rather small. Based on the unified grammar in SrcML, we estimate that the size is below 450, even when all AST node types from C/C++, C#, Objective-C, and Java are combined. After training the *AST2vec* for all node types, we obtain a numerical vector, i.e., an embedding, for each node type, and use the vectors to start the training of DTBCNN and Bi-DTBCNN models.

C. Gated Graph Neural Networks (GGNN)

GGNN and Gated Graph Sequence Neural Networks have been proposed as a general way to learn graphs [26]. Allamanis et al. [4] custom-build program graphs to encode both syntax and semantics of C# source code and extend GGNN to learn the graphs for several software maintenance tasks such as predicting misused variable names. Although our algorithm classification tasks are different from those published in previous GGNN-based tasks, we adopt the same schema for encoding the code as graphs, as shown in [4] that the graphs may encode more code semantics and graph-based neural networks may produce better code representations .

Each sample (i.e., program compilation unit) is represented by a graph as follows. The AST are encoded as a set of edges representing `Child` relations, whilst the ordering of children are kept by the `NextSibling` relations. Since our classification task does not care about the exact names of the identifiers compared to the variable misuse prediction task in [4], we can further reduce the number of nodes in the graph.

On the semantic side, the 'def-use' relations we obtained from program slicing are encoded directly into edge types `LastWrite` and `LastUse`. Following the schema used in [4], the 'returns' statements are recorded as a special relation `ReturnsTo`. Similarly, the `LastUse` relations are inferred from the otherwise discarded variable names, and the `ComputeFrom` relations are derived from the variables used in right-hand side and left-hand side of assignment expression AST. In this encoding, these semantic relations are rather agnostic to the concrete language syntax. It is therefore our hope that the graph representation can capture more commonalities between the structures. Compared to TBCNN encodings, such semantic edges are explicitly identified by static analysis tools, instead of learnt by the NN from the extracted features.

In this work, we have faithfully used the suggested approach in the original GGNN work [26] to aggregate the node-level embeddings learnt from graph propagation to the graph-level. Even though the schema from early work is adopted [4], there are still many configuration in the encoding to adjust, e.g., whether or not to encode the backward edges for semantic relations, and for our algorithm classfication tasks the best configuration needs to be found empirically. [1]

D. Token and Sequence based Neural Networks

There are also many other techniques based on tokens or token sequences or others to learn code representations [3], many of which have originated from natural language processing (NLP). Most of the techniques have the common underlying idea that whatever code representations can be turned into feature vectors which can then be classified through various machine learning models. This common underlying idea also aligns well with the Bi-NN framework, and we can instantiate the framework with those techniques too.

[1]The implementation of GGNN: https://github.com/bdqnghi/ggnn_graph_classification

Particularly, we instantiate the Bi-NN framework with the following commonly used models for our evaluation later.

BoW: The bag-of-words (BoW [25]), a.k.a. vector space model, counts the occurrences of each token as a feature to generate feature vectors for input. We adapt the model to generate vectors for source code.

N-gram: The BoW model does not consider the ordering among words, while n-gram models partially consider the ordering by using the count of n consecutive tokens in the input as a feature to generate feature vectors. Hellendoorn and Devanbu [17] found that a n-gram model (where n can be 3 or 5) can be a strong baseline for modeling large amount of source code. Notice that BoW can be seen as a special case of n-gram (i.e., unigram) models.

Tf-idf: The term frequency–inverse document frequency weighting scheme gives different weights to tokens of different occurrence frequencies: tokens appearing more frequently in one input are given higher weights while tokens appearing more frequently cross different inputs are given lower weights. As indicated in [48], such feature vectors may be better in identifying features that are more discriminative across programs, e.g., the token "bfs" for programs involving breadth-first searches.

LSTM-based: The above language models generate feature vectors by simply counting consecutive tokens, which may miss relations among separated tokens. Some neural network (NN) based models, such as long short-term memory (LSTM) [20], can take as input word embeddings, instead of token counts, to learn sequences better. In the literature, Siamese-LSTM [32] has been shown to achieve good performance for matching sentences in different natural languages. Therefore, we instantiate the Bi-NN framework with LSTMs to construct a Siamese-LSTM for comparison too. In our case, we use word2vec [30] to train the embedding vectors for tokens in C++ and Java corpus respectively. Then, the word2vec vectors for C++ tokens and Java tokens are fed as input into each of the two sub-LSTMs to train the whole Siamese-LSTM.

We have implemented various preprocessing steps to normalize tokens in source code as we know that not all tokens in programs are useful for determining code semantics. Sample preprocessing steps are removal of punctuation characters, identifier splitting based on CamelCase and underscores, converting all tokens into lower cases, and replacing single-letter variable names with a unified "id", etc.

For the classification model on top, we use fully-connected neuron layers with a Softmax output for the Siamese-LSTM. The Softmax classifier, which can be seen as the Multinominal Logistic Regression, is a classifier for multi-class classification. The Softmax layers can also be used as the classifier for the BoW, n-gram, tf-idf models where the feature vectors are based on token counting. Although we can also use any other classifier, such as SVM and Random Forests, instead of the Softmax layers, we use Softmax consistently through this paper for easier comparison and leave evaluation on the effectiveness of different classifiers to future work.

IV. Empirical Evaluation

A. Datasets

We use two datasets for evaluation. The first dataset inherits from the TBCNN work by Mou et al. [31], let's call this dataset as Dataset A. This dataset includes samples of 104 programming problems used in university programming lectures, and each class comprises of 500 different C++ programs. The total of 52,000 samples in this dataset is used to evaluate whether our implementation of TBCNN can correctly reproduce the same level of performance compared to [31].

For cross-language algorithm classification, however, the first dataset is insufficient because it only has C++ samples. Therefore, the second dataset is crawled from the GitHub, which contains 50 distinct algorithms. We use GitHub Developer APIs to retrieve the algorithm instances for each given algorithm class in each programming language. We use the name of the algorithms as the keywords to search for the files whose names or contents contain such keywords.[2] For the evaluation purpose, we collect the same algorithms in both C++ and Java. To prevent from getting toy examples, we only retrieve the files of a size larger than 500 bytes.

The raw samples from GitHub, however, contain clones which may affect the training performance. To reduce the impact of duplicated data on classification results, we use NiCad [13] to detect clones among the data, and remove all of the Type 1, Type 2 and Type 3 clones with a dissimilarity lower than 10%. After the clone removal, we obtained 4,932 algorithm files in Java and 4,732 for C++. On average, each class contains from 90 to 100 programs.

For the training and testing purpose, we divide either the C++ or Java dataset into the training set and testing set with a split ratio of 80/20 for the programs per class, and use 80% of the data for training and 20% of the data for testing. To form the pairwise data for the cross-language settings, we take each program in one language and pair it up with another program in the other language. Given all the combinations of the C++ and Java programs in the training set, we have about 312K pairs of bilateral programs implementing the same algorithm in different languages, and more than 15 millions of pairs of programs that implement different algorithms. Because of the imbalance, we only randomly select 312K pairs that implement different algorithms to balance the training and testing data for binary classification.

B. Implementation and Research Questions

We have implemented multiple instantiations of the Bi-NN framework (cf. Section III) for evaluation and comparison: 1) BoW model, 2) 3-gram model, 3) 5-gram model, 4) Tf-idf model, 5) Siamese-LSTM model, 6) Bi-TBCNN model, 7) Bi-DTBCNN model, and 8) Bi-GGNN model.

[2]A detailed list of the algorithms can be found here: https://github.com/bdqnghi/bi-tbcnn/blob/master/algorithms.txt

We adapt srcML to get Java and C++ programs into AST, and we annotate the AST using def-use relations extracted by `srcSlice` [5]. From these AST and the def-use relations we build dependency trees and derive GGNN-compatible graphs. We use `Tensorflow`[3] to build our Bi-NN. For the hidden layers, we add dropout with the probability of 0.7 to prevent the models from over-fitting. We use leaky `ReLU` as the activation function of the hidden layers. The GGNN encoding is implemented on the basis of the schema provided by Miltiadis et al. [4] and adapted by a preprocessing step to convert node tokens into node types, then the GGNN implementation is used to train/test the converted code graphs. Our techniques are implemented in a mix of Python and Bash scripts [4].

1) Research Questions: For our study on algorithm classification, we measure the effectiveness of each model by using the usual **accuracy** metric for the classification results. I.e., for a given set of test inputs, the accuracy of a model is the percentage of the tests for which the model produces a correct classification according to the ground-truth labels of the tests. Two kinds of classifications are considered in this paper: binary classification for determining whether or not two programs in two different languages implement the same algorithm (cf. Section II-A), and multi-class classification for determining which algorithm class a given program implements (cf. Section II-C).

We aim to compare the models in various settings against each other by answering the following research questions:

RQ1 Which instantiation of the Bi-NN framework achieves the best classification accuracy?

RQ2 Does this best instantiation for cross-language classification achieve better classification accuracy than others in single-language settings too?

RQ3 How sensitive is this best instantiation when the number of classes varies?

RQ4 Does adding dependencies in the code representations achieve better classification accuracy?

We run our evaluations on a server machine with an Intel Xeon CPU E5-2640 v4 and an Nvidia P100 GPU with 12GB of memory and 4.7 TeraFLOPS double-precision performance. The server is shared among multiple users and its workload may affect the time measurements of the evaluations.

C. Summary of Classification Results

We provide summarized answers to the research questions here and present more details in later subsections.

RQ1 To classify programs across languages, we found that all the statistical language models, such as tf-idf, bag-of-words, n-gram, and LSTM can be employed, but with different effectiveness. Amongst various instances of Bi-NN models, our Bi-DTBCNN model achieves a better cross-language classification accuracy of 86% than others ranging from 46% to 77%, but at the price of being the slowest in training (see Table I).

[3]https://github.com/tensorflow/tensorflow

[4]The code and evaluation results are available at https://github.com/bdqnghi/bi-tbcnn

TABLE I
RESULTS FOR CROSS-LANGUAGE BINARY CLASSIFICATION BY DIFFERENT CODE LEARNING TECHNIQUES.

Model	Accuracy	Training Time(hm)
Bag of words	0.46	5m
3-grams	0.51	5m
5-grams	0.52	5m
Tf-idf	0.49	7m
Siamese-LSTM	0.73	1h50m
Bi-TBCNN	0.77	3h10m
Bi-GGNN	0.76	5h50m
Bi-DTBCNN	0.86	8h25m

RQ2 All the models can be adjusted to recognize algorithm classes in a single language (SL) too (cf. Section II-C). Our DTBCNN models achieves an accuracy of 91% and 93% for Java and C++ respectively, the highest among other models and much better than the cross-language (CL) settings (see Table II).

RQ3 The number of algorithm classes has varying effects on different learning models. In both SL and CL settings, DTBCNN models maintain relatively good accuracy above 93% when the number of classes increases, whilst GGNN performance is much more sensitive and its accuracy decreases from 94% to 66% when the number of classes increases from 10 to 50 (see Table III).

RQ4 DTBCNN with dependencies achieves significantly better accuracy than TBCNN for the binary cross-language classification task, improving it from 77% to 86% for the Github dataset (see Table IV). For the single language classification task, DTBCNN achieves comparable accuracy to TBCNN (see Table II), which may imply that making implicit dependencies explicit in a single-language may not be necessary.

D. Details of Classification Results

RQ1: Results on Different Code Learning Techniques: Table I shows the results of various models for the binary cross-language algorithm classification task. The training time in Table I is measured as wall clock time by taking the average of 3 separate runs per configuration. The termination condition for each training depends on whether the loss function of a model reaches a certain threshold or the number of iterations/epochs in training exceeds a certain limit. The time needed to classify a test program is typically very short within a second.

The results show that NN-based models perform significantly better than the other token-counting based models (BoW, n-gram, and tf-idf). Our Bi-DTBCNN achieves the highest accuracy of 86%, but it takes the longest training time. GGNN training is faster than DTBCNN because both gated graphs and dependency trees are extensions of ASTs and our custom-built dependency trees in fact have more nodes than gated graphs although gated graphs may have more edges. Optimizing tree or graph representations of code and the training algorithms can be useful future research for better code learning.

RQ2: Results on Single-Language Classification: For each of TBCNN, DTBCNN and GGNN, we train a single-language model using the Java corpus and another single-language

TABLE II
SINGLE-LANGUAGE ALGORITHM CLASSIFICATION RESULTS

Model	Dataset		
	Github		A
	C++	Java	C++
TBCNN, Mou et al. [31]	0.93	0.89	0.93
GGNN, Allamanis et al. [4]	0.66	0.56	0.49
DTBCNN	0.93	0.91	0.93

model using the C++ corpus, and compare their classification accuracies. Table II shows the results. DTBCNN models slightly improve the classification accuracies of TBCNN models [31], which are also higher than GGNN models.

RQ3: Results on Sensitivity Analysis wrt Numbers of Classes: We hypothesize that the number of algorithm classes would affect the performance of all the code learning techniques. In particular, it would be interesting to find out whether the numbers of classes affect GGNN more than tree-based models.

Thus we perform the sensitivity analysis by reducing the number of classes to see how it affects the performance of the models in both single- and cross-language settings. As shown by the results in both SL and CL settings in Table III, DTBCNN models maintain relatively good performance when the number of classes increases, whilst GGNN performance is more sensitive to the number of classes. For SL settings, both GGNN and TBCNN perform well with 10 classes, reaching around 90% accuracy or better for both Github Dataset and Dataset A. However, when the number of classes increases, the GGNN cannot keep up the good performance but reduces the accuracy drastically, e.g 66% for Github C++ Dataset with 50 classes and 45% for Dataset A with 50 classes. On the contrary, DTBCNN can still maintain superior performance around 90%+ accuracy. The same situation occurs for cross-language (CL) settings: both Bi-GGNN and Bi-DTBCNN perform well when there are 10 classes in sub-components, but when the number of classes increases, the performance of Bi-GGNN drops and is more volatile than that of Bi-DTBCNN.

For SL settings, both GGNN and TBCNN perform well with 10 classes, reaching above 90% accuracy for both Github Dataset and Dataset A. When the number of classes increases to 50 for Github Dataset and 50 or 104 for Dataset A, the accuracy of GGNN reduces drastically, e.g., 66% for Github C++ Dataset with 50 classes and 45% for Dataset A with 104 classes. On the contrary, DTBCNN can maintain its performance above 93%. The same situation occurs for cross-language (CL) settings: both Bi-GGNN and Bi-DTBCNN perform well for 10 algorithm classes; when the number of classes increases, the performance of Bi-GGNN drops and is more volatile for different runs than that of Bi-DTBCNN.

RQ4: Results on Using Dependencies in the Models: A major intuition for adding dependencies into tree or graph based code representations is to expose more code semantics to help machine learning techniques to learn better. GGNN [4] is built on such an intuition to add many different kinds of edges into ASTs to form gated graphs, and it is shown to be useful for predicting variable names. For DTBCNN, we also want to find out whether adding dependencies into ASTs contaminates the code representations to make it harder to learn and how much effect adding dependencies has on the classification accuracy. As shown in Table IV, a neural network learning model trained on plain AST (i.e., TBCNN) produces worse cross-language classification accuracy than the same model trained on ASTs embedded with def or def-use relations (77% vs. 83% vs. 86%). On the other hand, for the single-language setting, TBCNN produces closely comparable accuracies to DTBCNN (see Table II). This phenomena may indicate that in a cross-language setting, the code syntax can differ a lot between different languages and require additional dependencies to help learn better representations for cross-language classification; while in a single-language setting, most code semantics are expressed in code syntax already, and the usefulness of extra dependencies may be reduced. It would be interesting future study to investigate further what dependencies may be useful for what kinds of tasks.

TABLE III
SENSITIVITY ANALYSIS

Model	Dataset	Num Classes	Setting		
			SL		CL
			C++	Java	
GGNN	Github	50	0.66	0.56	0.76
		30	0.93	0.88	0.73
		10	0.94	0.94	0.88
	A	104	0.45	–	–
		50	0.48	–	–
		25	0.68	–	–
		10	0.89	–	–
DTBCNN	Github	50	0.93	0.91	0.86
		30	0.95	0.93	0.88
		10	0.96	0.95	0.88
	A	104	0.94	–	–
		50	0.95	–	–
		25	0.95	–	–
		10	0.98	–	–

TABLE IV
RESULTS OF CROSS-LANGUAGE ALGORITHM CLASSIFICATION WITH DIFFERENT DEPENDENCY TREES.

	Plain AST	AST+Def	AST+Def+Use
Accuracy	0.77	0.83	0.86

E. Threats to Validity and Discussions

We discuss several threats to the validity of our study, and discuss possible alternatives that can be done in future studies.

1) Threats to Validity: **Data collections**. Using Github Search API, we collected the code implementation of algorithms based on some specific keywords to identify the name of algorithms, such as “bfs", “bubblesort", “linkedlist", etc. This approach may find some source code that are not actually related to the algorithm (i.e., false positives). To reduce the impact of such cases, we have to restrict the size of crawled code file to e.g. 500 bytes in order to exclude code files associated with auxiliary library code or details irrelevant to the algorithms. However, it is also possible that we have excluded many useful implementation of algorithms (i.e., false negatives). However, the quality of the Github search is an uncontrolled variable for the experiments, even though the authors have randomly inspected 200 returned results to find

the false positive acceptable. Moreover, these samples from Github do not nessessarily compile, hence we choose srcML parser to generate AST and slicing information, instead of the production compilers from JDK (for Java) and clang (for C++).

Merging Layers. We used a subnetwork merging strategy and a softmax layer to classify programs, in either SL or CL settings. The merge can also be done using energy functions such as Manhattan euclidean distances in Siamese-LSTM [32] or using multi-layered perceptrons to fuse the vectors. We leave it for future work to explore the effectiveness of different alternatives for merging the subnetworks.

Node granularity: In our implementation of GGNN, TBCNN, and DTBCNN, we model source code using the AST node types mostly (ignoring identifier names), which are less fine-grained than other models that consider concrete tokens. Despite being 'coarser', such node-type level encodings outperforms token-based LSTM. A possible explanation is that node types can already keep structural and semantic features of source code relevant to an algorithm, without losing critical information. On the other hand, concrete token information can be useful for certain learning tasks, such as predicting wrong variable names [4]. A future improvement is to combine both the node type level and token level information and see how it will improve the performance of code learning models.

2) Justification for baseline results: As described by Hellendoorn and Devanbu [17], the n-gram model, if configured carefully, can achieve a comparable result to a neural network-based model. However, our results in Table I show that all of the token feature-based models, include the n-gram, underperform neural network-based models.

We would like to find the reason for the worse performance in feature-based models. As described in [17], the code model is built *per project*, that is, an n-gram sequence can be reused *across files in a project*, while in our work, each file in the corpus is an *isolated program* and all the files have no explicit connection to each other. This makes the token features extracted by n-gram, bag of words and tf-idf sparse, except for some common language keywords (if, else, include, etc) or common variable declarations (i, j, str, etc). In short, the feature-based models do not capture well the relations among the tokens well in our dataset.

In contrast, NN-based models (LSTM, CNN) take the input as the pretrained embedding of words. These pretrained embeddings capture the relations among tokens into a low dimensional continuous vector space. Thus, the NN-based models can produce better vector representations for the token features, which leads to better classification results.

3) Comparison between neural-network based models: Among the NN-based models, our custom-built DTBCNN achieves higher accuracy than the others. Here we want to provide some justifications for such results. The NN-based models considered in this paper can be categorized into two types: sequence-based (LSTM and CNN) and structure-based (GGNN, TBCNN and DTBCNN). The sequence-based models consider the source code at the token level, while the structure-based model considers the source code at the structure level (e.g., AST node types and dependencies). The advantage of treating source code as token sequences is that it is simpler to adapt well-known NLP techniques. One disadvantage of such techniques is that they cannot make use of inherent features that hide inside the implicit structure of source code, such as data and control dependency, class inheritance, call relations, etc. Another disadvantage is that the number of tokens can be arbitrary as developers can introduce new tokens when writing code [48], making it harder to capture the relations among essentially the same tokens that appear differently.

On the other hand, AST or graph representation of source code are closer to the structural nature of algorithms. Since our goal is to classify algorithms *across different languages*, the inherent features of code structures can be more important than those specific tokens to distinguish the program.

For example, if we consider control dependency, the bubblesort makes multiple passes through a list of items. The code structure usually contains 2 nested `for` loop because we want to compare each item with every other item at least once, we also need an `if` inside the second `for` loop to check if the previous item is bigger than the current item for swapping. In addition, if we consider data dependency, the bubblesort involves comparison and swapping of items in a list and does not need to introduce many variable declarations since it is an in-place algorithm. All of these, unlike token sequences, can be features of bubblesort to distinguish itself from others, no matter in which language the code is written. In addition, we consider only the node types of AST instead of actual tokens, making the embedding vocabulary smaller to generate more compact vector representations.

In addition, as mentioned in Section III-C, encoding a program as the graph intuitively adds richer semantic information of the program to the AST, thus is expected to yield a better result. However, our results shown in Table I and Table IV are against this intuition. A possible reason is that graphs encode both control and data dependencies as edges between the nodes in the AST, thus complicating the structure of the AST, while our approach adds richer semantic information but remains to be tree-based. Graph-based similarity comparison boils down to graph isomorphism, which is a much harder problem than tree-based comparison.

In the future, we will conduct more experiments to observe the internal representations of the neural networks (NNs) with more datasets, in order to really understand what the NNs have learned to represent the programs.

The approach proposed is general to any pair of programming languages, although we only used Java and C++ to evaluate in this work. When the programming languages are cross paradigms, such as object-oriented versus functional, it would be interesting to see whether the framework needs any further customizations, we leave this task for the future. More algorithms e.g., those listed in Rosatta Code can be added to the dataset in future, however, the 50 algorithms are already sufficient for us to assess multiple baselines and challenge the scalability of their training.

V. Related Work

There have been a number of studies on program comprehension to understand code artifacts (see Table V).

Learning Code as Natural Languages Tokens: Mining a large corpus of open-source Java projects on the repositories, Hindle et al. [18] showed that programming languages are largely in common to natural languages in terms of probabilistic predictability (i.e., low entropy) of next tokens, hence statistical methods apply well to model repetitiveness in source code, and that language models can be used for the code suggestion task in IDEs. Furthermore, it has been established that programmers tend to focus on code local to their context, hence the locality can be exploited by 'caches' while keeping both the long and short-term memory of the sequences processed. For example, Hellendoorn et al. [17] employed both n-grams (with and without cache) and LSTM NN to train statistical models and demonstrated that such straight application of NN to sequences do not necessarily enhance the accuracy greatly compared to the cached n-gram models. In a survey, Allamanis et al. [3] further categorized the related research directions in this area.

Learning Programs with Structures: It is known that code is structured (e.g., nested in syntax) and programmers do not read the code from the beginning to the end. Static program analysis tools relies on AST and program dependence graphs to capture such information. However, exist tools for translating code among specific languages (e.g., Java2CSharp [1]) are mostly rule-based, rather than statistics-based [24]. Boccardo et al. [7] proposed to use neural networks for learning program equivalence based on callgraphs, the accuracy was not too good because it does not take into account code syntax structures. Alexandru et al. [2] proposed to analyse only the revised source files from a Git repository, instead of parsing or analysing those unchanged revisions, reducing redundancies in analysing the evolution of source code of different programming languages. On learning from code syntax structures, TBCNN were first proposed by Mou et al. [31], which designed a set of fixed-depth subtree filters sliding over an entire AST to extract structural information of the tree. They also proposed "continuous binary trees" and applied dynamic pooling [46] to deal with varying numbers of children of AST nodes.

Combining syntactical and semantic information: For semantic information such as program dependence graphs, Allamanis et al. [4] used GGNN to unify the information with AST. Using open-source C# projects they have demonstrated significant improvement on predicting the correct or misused names. The evaluation shows our Bi-DTBCNN approach performs better than GGNN for our algorithm classification tasks. A possible reason is we embed *all* contextual information on the AST whilst GGNN relies on how certain semantic relations that can be derived from AST are explicitly encoded.

Bi-lateral representations for cross-language learning: Cross-language learning representation structures are typically bilateral. Studies on sentence comparisons and translations in NLP involve variants of bilateral structures as shown by Wang et al. [50]. Among them, Bromley et al. [9] pioneered "Siamese" structures to join two subnetworks for written signature comparison. He et al. [16] also use such structures to compute sentence features at multiple levels of granularity. Yin and Neubig [53] and Oda et al. [39] used Seq2Seq NN to perform code generation from one programming language to another. One may see pseudo code as a programming language for algorithms, such bilateral NN may be considered useful for their tasks as well. However, in this work, we aim at algorithm classifications, which would not require one-to-one transplanting the code. However, these studies have not considered more accurate representation of code.

Applicable Software Engineering Tasks: Once the statistical models can be established from the code corpus, they can be used for many useful software engineering tasks. Peters et al [42] find that text filtering and ranking can significantly improve security bug prediction in the presence of class imbalance and data scarcity. Specific to code artefacts, sequential representations as tokens [35], phrases [37], [38], or API's [33], [34], [43], [56], [57] are amongst the first applications of machine learning to SE tasks.

In this work, we choose the algorithm classification tasks to demonstrate that cross-language learning models can be used to capture the "essence" of programs that is not always expressed in the same language. Specifically, cross-language language models can benefit some of the translation tasks, e.g., Gu et al. [14], [15] applied NN at the API level for language recognition and translation; Nguyen et al. [36] show how statistical machine translation could be used to port applications from Java to C#. Vasilescu et al [49] also use machine translation as a way to recover helpful variable names in "minimized" Javascript. Ray et al. [45] use statistical language models to localize defective code, and also to improve the performance of static bug detectors.

VI. Conclusions and Future Work

In this work, we generalize a Bilateral Neural Network (Bi-NN) framework for cross-language algorithm classification problems. We instantiate this framework with different intermediate representations of 'Big Code' learning, including our own dependency tree-based convolutional neural networks (DTBCNN), and evaluate them on the tasks of classifying thousands of programs files as 50 algorithms, across different programming languages such as Java and C++.

We introduce DTBCNN to encode def-use relations (aka program dependencies) as part of abstract syntax trees, which can achieve the highest classification accuracy compared to other commonly used models (e.g., bags-of-words, n-gram, tf-idf, long short-term memory, gated graph neural networks).

We plan to do a more concise evaluation by introducing the validation set, along with the k-fold validation setting, to fully evaluate the effectiveness of the proposed models. In addition, we also plan future work to evaluate Bi-NN on more programming languages and more algorithm classes, and to extend Bi-NN to encode more relevant code semantics for tasks beyond algorithm classification.

TABLE V
COMPARISON OF RELATED WORK IN CODE LEARNING

Approaches	Program Representations	Model	SE Task
Boccardo et al. [7]	Callgraphs	ANN	program equivalence
Hindle et al. [18]	tokens	n-grams	code recommendation
Ray et al. [45]	tokens	cached n-grams	bug localization
Hellendoorn and Devanbu [17]	tokens	cached n-grams, LSTM	code recommendation
Nguyen et al. [36]	AST	NN	cross-language translation
Mou et al. [31]	AST	TBCNN	program classification
Yin and Neubig [53], Oda et al. [39]	transformed AST	Seq2Seq	code generation
Allamanis et al. [4]	AST and Dependence Graphs	Gated Graph NN	variable name recovery
This work	Dependence Trees	Bi-DTBNN	algorithm taxonomy classification

REFERENCES

[1] "Java2csharp." [Online]. Available: https://github.com/codejuicer/java2csharp

[2] C. V. Alexandru, S. Panichella, and H. C. Gall, "Reducing redundancies in multi-revision code analysis," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 148–159.

[3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[4] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations (ICLR)*, 2018.

[5] H. W. Alomari, M. L. Collard, J. I. Maletic, N. Alhindawi, and O. Meqdadi, "srcSlice: Very efficient and scalable forward static slicing," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 931–961, 2014.

[6] Anonymous Authors, "Anonymous paper," in *Anonymous Publication Venue*.

[7] D. Boccardo, T. Monteiro Nascimento, C. Prado, L. Fernando Rust da Costa Carmo, and R. Machado, "Program equivalence using neural networks," in *5th International ICST Conference on Bio-Inspired Models of Network, Information, and Computing Systems, At Boston, United States*, 10 2010.

[8] J. Börstler, "Feature-oriented classification for software reuse," in *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 1995, pp. 204–211.

[9] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," in *Proceedings of the 6th International Conference on Neural Information Processing Systems (NIPS)*, 1993, pp. 737–744.

[10] S. Cesare and Y. Xiang, "A fast flowgraph based classification system for packed and polymorphic malware on the endhost," in *24th IEEE International Conference on Advanced Information Networking and Applications, AINA 2010, Perth, Australia, 20-13 April 2010*, 2010, pp. 721–728.

[11] K. L. Clark and J. Darlington, "Algorithm classification through synthesis," *The Computer Journal*, vol. 23, no. 1, pp. 61–65, 1980. [Online]. Available: http://dx.doi.org/10.1093/comjnl/23.1.61

[12] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight transformation and fact extraction with the srcML toolkit," in *11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2011, pp. 173–184.

[13] J. R. Cordy and C. K. Roy, "Tuning research tools for scalability and performance: The nicad experience," *Sci. Comput. Program.*, vol. 79, pp. 158–171, 2014.

[14] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, November 13-18 2016, pp. 631–642.

[15] ——, "DeepAM: Migrate APIs with multi-modal sequence to sequence learning," in *26th International Joint Conference on Artificial Intelligence (IJCAI)*, August 19-25 2017, pp. 3675–3681.

[16] H. He, K. Gimpel, and J. J. Lin, "Multi-perspective sentence similarity modeling with convolutional neural networks," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

[17] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE*, 2017, pp. 763–773.

[18] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[19] T. K. Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, Aug 1995, pp. 278–282 vol.1.

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[21] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE Transactions on Neural Networks*, vol. 13, no. 2, pp. 415–425, Mar 2002.

[22] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects," in *2012 19th Working Conference on Reverse Engineering*, Oct 2012, pp. 387–391.

[23] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, 2009, pp. 81–92.

[24] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2014. New York, NY, USA: ACM, 2014, pp. 173–184.

[25] Y. Ko, "A study of term weighting schemes using class information for text classification," in *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '12. New York, NY, USA: ACM, 2012, pp. 1029–1030.

[26] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," in *International Conference on Learning Representations (ICLR)*, Nov. 2016, arXiv: 1511.05493.

[27] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. L. Traon, "Mining fix patterns for findbugs violations," *CoRR*, vol. abs/1712.03201, 2017. [Online]. Available: http://arxiv.org/abs/1712.03201

[28] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proceedings of the 23rd International Conference on Software Engineering*, ser. ICSE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 103–112. [Online]. Available: http://dl.acm.org/citation.cfm?id=381473.381484

[29] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems (NIPS)*, 2013, pp. 3111–3119.

[31] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, February 12-17 2016, pp. 1287–1293.

[32] J. Mueller and A. Thyagarajan, "Siamese recurrent architectures for learning sentence similarity," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016, pp. 2786–2792. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12195

[33] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining API usage mappings for code migration," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014, pp. 457–468.

[34] ——, "Statistical learning of API mappings for language migration," in *36th International Conference on Software Engineering - Companion (ICSE)*, May 31 - June 07 2014, pp. 618–619.

[35] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, August 18-26 2013, pp. 651–654.

[36] ——, "Migrating code with statistical machine translation," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 544–547.

[37] ——, "Divide-and-conquer approach for multi-phase statistical migration for source code (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, November 9-13 2015, pp. 585–596.

[38] A. T. Nguyen, Z. Tu, and T. N. Nguyen, "Do contexts help in phrase-based, statistical source code migration?" in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, October 2-7 2016, pp. 155–165.

[39] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 574–584.

[40] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management (KSEM)*, October 28-30 2015, pp. 547–553.

[41] F. Peters, T. Tun, Y. Yu, and B. Nuseibeh, "Text filtering and ranking for security bug report prediction," *IEEE Transactions on Software Engineering*, vol. 1, no. 1, pp. 1–1, 2018.

[42] ——, "Text filtering and ranking for security bug report prediction," *IEEE Transactions on Software Engineering*, p. (Early Access), 2018. [Online]. Available: http://oro.open.ac.uk/53059/

[43] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen, "Statistical migration of API usages," in *39th International Conference on Software Engineering - Companion Volume (ICSE)*, May 20-28 2017, pp. 47–50.

[44] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings of the 10th International Workshop on Program Comprehension*, ser. IWPC '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 271–. [Online]. Available: http://dl.acm.org/citation.cfm?id=580131.857012

[45] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 428–439.

[46] R. Socher, E. H. Huang, J. Pennington, A. Y. Ng, and C. D. Manning, "Dynamic pooling and unfolding recursive autoencoders for paraphrase detection," in *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS)*, December 12-14 2011, pp. 801–809.

[47] A. Taherkhani, A. Korhonen, and L. Malmi, "Recognizing algorithms using language constructs, software metrics and roles of variables: An experiment with sorting algorithms," *Comput. J.*, vol. 54, no. 7, pp. 1049–1066, 2011.

[48] Z. Tu, Z. Su, and P. T. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 269–280.

[49] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated js names," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 683–693.

[50] Z. Wang, W. Hamza, and R. Florian, "Bilateral multi-perspective matching for natural language sentences," in *26th International Joint Conference on Artificial Intelligence (IJCAI)*, August 19-25 2017, pp. 4144–4150.

[51] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI)*, 2017, pp. 3034–3040.

[52] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "A new android malware detection approach using bayesian classification," in *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, March 2013, pp. 121–128.

[53] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2017, pp. 440–450.

[54] C. Yuan, S. Wei, Y. Wang, Y. You, and S. ZiLiang, "Android applications categorization using bayesian classification," in *2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, Oct 2016, pp. 173–176.

[55] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 111–121.

[56] H. Zhong, S. Thummalapenta, and T. Xie, "Exposing behavioral differences in cross-language API mapping relations," in *Proceedings of 16th International Conference on Fundamental Approaches to Software Engineering (FASE), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, March 16-24 2013, pp. 130–145.

[57] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE)*, May 1-8 2010, pp. 195–204.