2-2018

Cross-language learning for program classification using bilateral tree-based convolutional neural networks

Duy Quoc Nghi BUI
Singapore Management University, dqnbui.2016@phdis.smu.edu.sg

Lingxiao JIANG
Singapore Management University, lxjiang@smu.edu.sg

Yijun YU

Cross-Language Learning for Program Classification
Using Bilateral Tree-Based Convolutional Neural Networks

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

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

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

Abstract

Towards the vision of translating code that implements an algorithm from one programming language into another, this paper proposes an approach for automated program classification using *bilateral tree-based convolutional neural networks* (BiTBCNNs). It is layered on top of two tree-based convolutional neural networks (TBCNNs), each of which recognizes the algorithm of code written in an individual programming language. The combination layer of the networks recognizes the similarities and differences among code in different programming languages. The BiTBCNNs are trained using the source code in different languages but known to implement the same algorithms and/or functionalities. For a preliminary evaluation, we use 3591 Java and 3534 C++ code snippets from 6 algorithms we crawled systematically from GitHub. We obtained over 90% accuracy in the cross-language binary classification task to tell whether any given two code snippets implement a same algorithm. Also, for the algorithm classification task, i.e., to predict which one of the six algorithm labels is implemented by an arbitrary C++ code snippet, we achieved over 80% precision.

1. Introduction

Software engineers need to classify a code snippet against known algorithms, such as Quick Sort, in order to understand it. All algorithms, however, can be implemented in different programming languages, making it hard to recognise an algorithm from the knowledge of its implementation in other languages. It is, therefore, useful to recognise certain algorithms from programs in different languages, i.e., performing *cross-language program classification*.

For a similar problem of language migration, statistical language models have been studied for tokens (**?**), phrases (**?**; **?**), or APIs appeared in the code(**?**; **?**; **?**; **?**; **?**). Some of these (i.e., for language recognition and API migration) have been helped by deep neural networks (**?**; **?**), however, little has been done on cross-language program classification.

This paper proposes to use bilateral neural networks (BiNNs), a technique originally developed for comparing natural language sentences, to recognise code snippets in languages that have similar syntax and potentially semantics. Our basic idea is to construct individual subnetworks to

encode abstract syntax trees (ASTs) of individual languages, and then construct a combination layer of subnetworks to encode similarities and differences among code structures in different languages.

Our proposed BiTBCNNs are a combination of three major constructs: (i) BiNNs using *softmax* operation for structured data to be compared for classification; (ii) a variant of tree-based convolutional neural networks (TBCNNs) on each side of the BiNNs to encode AST structures, independent of the programming language of choice; and (iii) a unified encoding of AST in multiple programming languages that enables cross-language program classification. With collected programs in different languages, evaluation has shown that our BiTBCNNs have over 80% accuracy in classifying them according to their underlying algorithms.

The remainder of the paper is organised as follows: Section 2 presents an overview of the proposed process of classifying code in different programming languages; Section 3 details how we make TBCNNs bilateral for the cross-language program classification tasks; Section 4 evaluates the effectiveness of cross-language benchmarks we collected and how they behave when transferring the models across languages or across algorithms; Section 5 presents related work; and finally, Section 6 concludes the findings and suggests some further directions.

2. Overview of Our Approach

An overview of our approach is illustrated in Figure 1. As shown by our open-source repository[1], each step in the process is supported by a simplified `docker` command.

First, a programmer may define a list of algorithms (e.g., mergesort, quicksort, breath-first-search, linkedlist, bublesort, knapsack), and a list of programming languages (e.g., Java, C++, Python). Given the two lists, automated calls to GitHub RESTful APIs *crawl* GitHub repositories to retrieve about 600 instances of program code that implement each algorithm in each programming language.

These code snippets are then *parsed*[2] into abstract syntax trees (ASTs) represented in Pickle format. These ASTs are loaded into memory and converted into a vector form that preserve the distances of similar features (*AST2vec*) by

---

[1]https://github.com/bdqnghi/bi-tbcnn
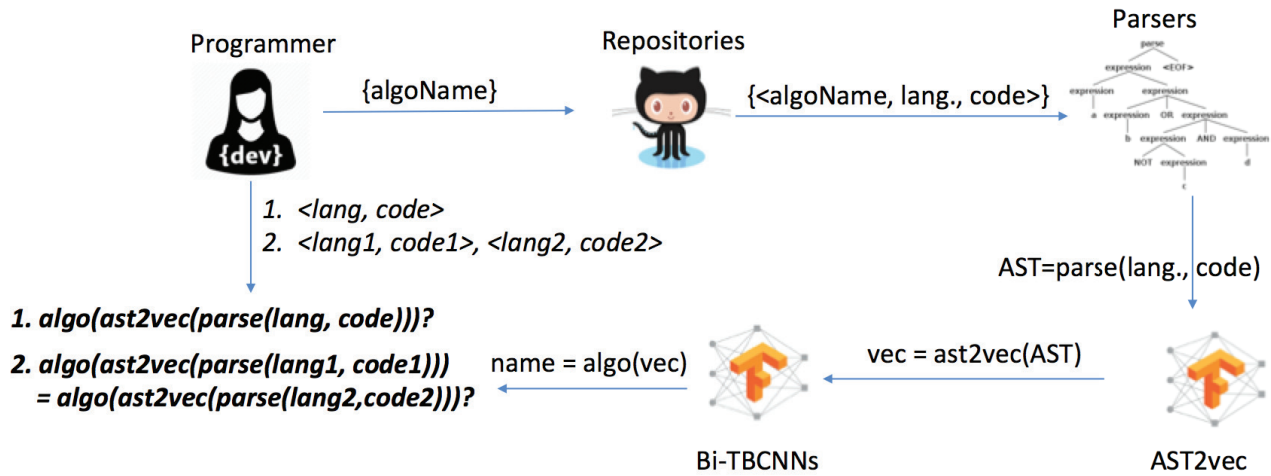[2]https://bitbucket.org/yijunyu/fast

Figure 1: An overview of our proposed program classification process

training the embeddings of leaf-level tokens, using Tensorflow, according to the programming language (e.g., 385 token types for Java/C/C++/C#/Objective C in SrcML grammar and 82 token types for Python).

The vectors of both programs of cross-language nature are then combined into *BiTBCNNs*, using Tensorflow again, approximating a function that classifies these vectors to algorithm names.

The trained model can be used to answer many types of queries, e.g., (i) what is the algorithm of a piece of code and (ii) whether two programs in different languages implement the same algorithm.

3. Construction of BiTBCNNs

Our BiTBCNNs construct extends TBCNNs with pretrained vectors as bilateral NNs for cross-language classification.

3.1 Tree-based convolutional neural networks

TBCNNs were proposed by **?** (**?**). Figure 2 shows its architecture. Each AST node is represented as a vector by using an encoding layer, whose task is to embed AST node types in a continuous vector space where semantically similar types are mapped to nearby points. For examples, the types 'while' and 'for' are similar because they are both loop statements. We use a different strategy to embed AST node types, as described in Section 4.2.

? (**?**) designed a set of fixed-depth subtree filters sliding over an entire AST to extract structural information of the tree. The pooling layer was added to gather the extracted information over various parts of the tree. They also proposed "continuous binary trees" and applied dynamic pooling (**?**) to deal with varying numbers of children of AST nodes. Finally, they added a hidden layer and an output layer to classify programs.

3.2 The pre-trained vector

To train the TBCNNs, one needs an initial vector representation for each tree node. **?** (**?**) use the "coding criterion" from **?** (**?**) to learn the vector representation for each AST node. We use a different strategy, similar to the skip-gram model of *word2vec* (**?**), but applied to the context of ASTs. The skip-gram model, given an input word in a sentence, looks at the words nearby and picks one at random. The model predicts the probability for each word in the whole vocabulary to be a "nearby word" of the input word. So the task here is to "predict the contextual words given an input word".

With this idea in mind, we apply it for the so-called *AST2vec* task. That is, we pick random children of a given AST node. The networks, in this case, will tell us the probability for each node in the whole AST vocabulary to be the "chosen children". The vocabulary words, in this case, are the AST node types, which are of rather small sizes (around 450 combining C/C++, C#, Objective C, and Java).

3.3 The neural network classification model

Our model in Figure 3 has (1) a bilateral structure with two subnetworks, each of which processes a tree representation in parallel, and (2) the classification networks, which are simply fully connected layers connecting the two trees to the final Softmax layer, classifying if the two code snippets represented by the trees implement a same algorithm.

The subnetworks are adapted from TBCNNs (**?**). Each subnetwork receives the AST representation of a program as the input. The TBCNNs will perform a convolutional step to extract features from the trees. In our case, after the pooling layer of each TBCNN, we get the feature representation vector of each program, and concatenate the two vectors to a merged vector, so called the "joint feature representation layers". Then, two more fully connected hidden layers above the joint feature representation layers are added
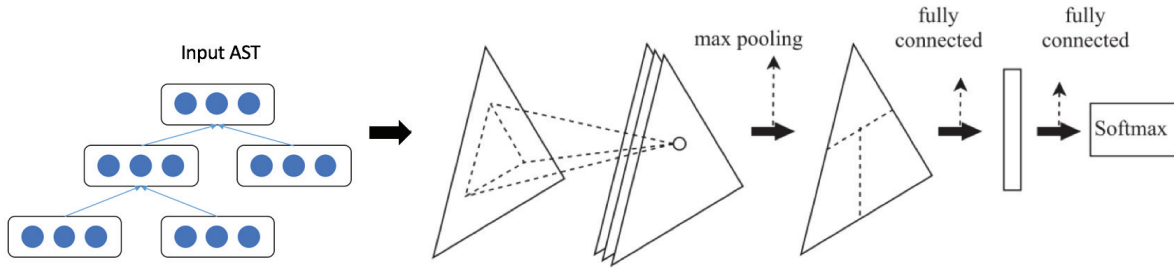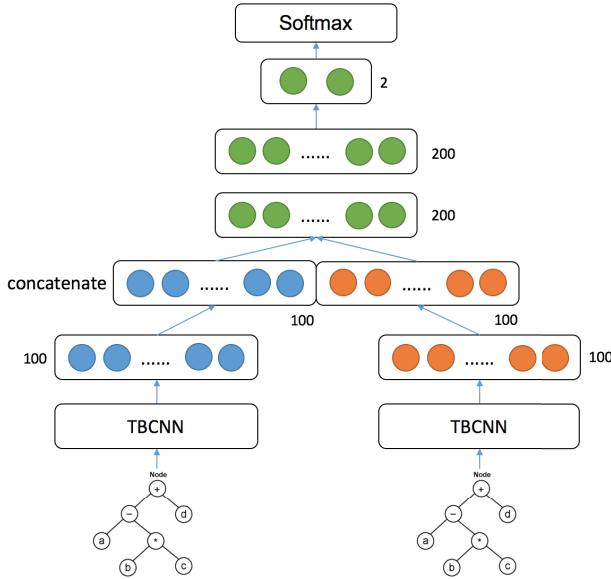
Figure 2: TBCNNs, excerpt from (**?**)

Figure 3: BiTBCNN architecture for program classification

Table 1: C++ and Java code sampled from GitHub

Lang.	ms	bs	qs	ll	bfs	kns	total
C++	588	531	567	609	609	630	3534
Java	588	609	567	588	609	630	3591

and connected to a Softmax layer to classify if the two input programs implement a same algorithm or not.

4. Evaluation

4.1 Datasets

We have collected data from GitHub for six algorithms: mergesort (ms), bubblesort (bs), quicksort (qs), linkedlist (ll), breadth first search (bfs) and knapsack (kns), both in C++ and Java. For each language, we get approximately 3500 programs. The details of our dataset are depicted in Table 1, where the number of instances of programs crawled from the GitHub for specific algorithms are shown.

We use Tensorflow[3] to build our model. For the hidden layers, we add dropout with the probability of 0.7 to prevent the model from over-fitting. We use leaky `ReLU` as the activation function of the hidden layers to avoid gradient vanishing/exploding problem.

4.2 Experiments

Our experiments include two settings. The first checks whether BiTBCNNs perform well in classifying whether a random pair of cross-language programs implements a same algorithm. The second is to check whether the classifier still works well when it is applied to classify which algorithm an unknown program implements, based on a set of known programs in another language, simulating a cross-language learning situation that motivates this work in Section 1.

4.2.1 Binary classification. This task is designed to verify if our model can successfully detect whether two programs from two different languages are the same or not. For programs in the original dataset of each language and each algorithm, we split 70 percent for training and 30 percent for testing. Thus we have approximately 2,500 programs on each language for training and 1,000 programs for testing. With 2,500 programs on each side, we get 6,250,000 pairs of programs (about 1,100,000 similar pairs and 5,100,000 dissimilar pairs). At this moment, we feed into the left subnetwork C++ programs and the right subnetwork Java programs. For each training epoch, we randomly select 1,000 similar pair and 1,000 dissimilar pairs to get balanced inputs for the epoch. We train the model for 100 epochs.

For the testing, as we have approximately 1,000 C++ programs and 1,000 Java programs, we could have approximately 1,000,000 pairs in total. To save time, we randomly select 2,000 similar pairs and 2,000 dissimilar pairs, which amount to around 0.4% of all the testing pairs. We use precision, recall and f1 score as the metrics to evaluate this task. The result is shown in Table 2.

4.2.2 Algorithm Detection. This task evaluates how well our model performs in classifying the actual algorithm implemented by a given input program. Taking a random program A for testing, we use it as the input for the left subnetwork, and pick a known program B implementing a known

---

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

Table 2: Results of cross-language program classifications. The first column shows the labels of pairs, 1 means similar, 0 means dissimilar. The rest are metrics to evaluate this task.

label	precision	recall	f1
1	0.98	0.91	0.95
0	0.92	0.94	0.93

algorithm and use it for the right. In this way, one can infer the algorithm label of the program A based on outputs from the above binary classifier. Note that in our experiment, we always assume that the left input is a C++ program and the right input is a Java program.

We thus take 1,000 random C++ programs from the testing data. Then for each of the C++ programs, we randomly pick one known/training Java programs from each of the six algorithm labels. We compare each C++ program with each of six Java programs using BiTBCNNs, in order to tell which one yields the highest probability in the softmax layer, and we use the algorithm label of the Java program that yields the highest probability as the predicted label of the C++ program. Finally, we compare the true label of the C++ program with the predicted one, and get a precision of 80.5%.

4.3 Threats to Validity & Discussion

We have not looked at all available programming languages or algorithms. We will need to verify whether the current code collected via GitHub search APIs may have biases, and evaluate our approach with more languages and more algorithms, e.g., using Rosetta Code (http://rosettacode.org/wiki/Rosetta_Code).

The programs we used are relatively small with relatively clearly defined algorithms. If a program becomes larger or contains mixed set of algorithms, our approach may not be applicable directly. Training speed may become a concern too when more data is used, although each round of training for our limited dataset only took tens of minutes on a commodity desktop machine. We think traditional program analysis (e.g., dependence-based slicing) may be useful for alleviating such problems by partitioning a large program into smaller ones first before applying our approach.

The architecture of our BiTBCNNs may be varied in many ways as studies in the area of natural language processing have shown. And we have only used simple data dropout rate of 0.7 to reduce over-fitting. There is still much work to explore various neural network structures. Also, our encoding of the trees removed identifier names. In future we will consider leveraging on the similarity in names and more code semantics (e.g., dependencies among code elements) for more accurate code encoding.

5. Related Work

For the problem of cross-language program translation, much work has utilized various statistical language models for tokens (**?**), phrases (**?**; **?**), or APIs (**?**; **?**; **?**; **?**;

?). Only a few studies have used deep learning for language recognition and translation, at the API level (**?**; **?**), which is still far from classifying functionally similar code fragment or performing translation for any code fragment. Although some practical tools exist for translating code among specific languages (e.g., Java2CSharp: https://github.com/codejuicer/java2csharp), they are mostly rule-based, rather than statistics-based (**?**) depending on clearly defined grammars of individual languages, and not easily extensible for different languages.

For natural languages, many studies on sentence comparisons and translations involve variants of bilateral structures as shown by **?** (**?**). Among them, **?** (**?**) pioneered "Siamese" structures to join two subnetworks for written signature comparison. **?** (**?**) also use such structures to compute sentence features at multiple levels of granularity. However, these studies have not considered tree-based structures that are more accurate representations of code.

In code learning, **?** (**?**) point out that simpler models (e.g., n-gram) improved with cached information about code locality and hierarchy may even outperform complex models (e.g., deep neural networks). But this also indicates to us that incorporating code locality and structural information with deep learning by using tree-based convolutional neural networks (TBCNNs) may improve code learning accuracy. Using TreeNNs, **?** (**?**) propose to represent symbolic expressions; however, it has not been applied to other type of code structures. Although **?** (**?**) introduce TBCNNs to classify C++ programs based on functionality and to detect code of certain patterns and others use tree-based encodings too (e.g., **?** (**?**) for code clone detection), it has not been applied to cross-language program classification.

6. Conclusions & Future Work

In this paper, we have presented the BiTBCNNs approach to the cross-language program classification problem, where algorithms are identified from source AST structures automatically. Using benchmarks of algorithms crawled from GitHub, we have shown that it is possible to train a model on multiple languages, with an accuracy of above 80%. The number and representativeness of training datasets may affect the ultimate performance, while cross-language deep learning makes it likely possible to reuse the implementation of algorithms from different languages.

Our future work include tuning BiTBCNNs structures and parameters, supporting more programming languages and more algorithms with more training data, learning from more code semantic information such as dependence data, and applying to more tasks such as cross-language code clone detection, algorithm patent protection, and bug fixing.