Code Author Classification Using Tree-Based Convolution Neural Network

Sobol Valentine

Abstract—**Programming language processing based on machine learning approaches is a hot research topic in the field of software engineering. In this paper we use Tree-Based Convolution Neural Network to solve task of author identification. TBCNN performs feature extraction over tree structure like abstract syntax tree and well suited for tasks connected with program analysis.**

I. Introduction

Many researches from different communities are demonstrating growing interest in applying machine learning techniques to solve software engineering and software analysis problems. In software engineering area, program source code analysing called *programming language processing* and take a great role in this paper.

Even though programmers can write a source code we can't truly identify a real code author. Analysing source code provides a way of estimating code complexity, structure, style, etc. For instance automatically detecting source code of specific pattern help to identify author or discover bad styled source code so as to improve code quality. Another example is managing large repositories with big amount of contributors, where automatic code author identification is crucial.

There is a similarity between programming and nature languages as Hindle et al. (2012) (2012) demonstrate, so code contain statistical information which is important for author identification. This information is very hard to capture by human, and this fact explain learning-based techniques for programming language processing. Existing machine learning approaches depends on feature extraction, which is hardly connected to a specific task like code clone detection Chilowicz, Duris, and Roussel (2009), and bug detection Steidl and Gode (2013). Also, information in the machine learning literature proof that features created by human may fail to capture the pattern of data, so they may be worse then features learned automatically.

The deep neural network, also known as *deep learning*, is a highly automated learning machine. By exploring multiple layers of non-linear transformation, the deep architecture can automatically learn complicated underlying features, which are crucial to the solving task. Over the past few years, deep learning has made significant breakthroughs in various fields, such as speech recognition Dahl, Mohamed, and Hinton (2010), computer vision Krizhevsky, Sutskever, and Hinton (2012), and natural language processing Collobert and Weston (2008).

There are some similarities between natural languages and programming languages, but there are also differences Pane, Ratanamahatana, and Myers (2001). Because programs are based on a formal languages, its contain a lot of structural information. Structure of natural language is not as stringent as in program. Pinker (1994) illustrates an interesting example, "The dog the stick the fire burned beat bit the cat." This sentence matches to grammar rules, but contains a lot of nested clauses. It's similar to common nested structures like loops or if statements in programs. In fact, the syntax (or parse) tree of a program is typically much larger then syntax tree of a natural language sentence. There are about 200 nodes on average for program syntax tree, when a sentence contains about 20 words (information from sentiment analysis dataset Socher et al. (2013)). Another thing is relation between program components. For example statements inside a loop block form a semantic group, and statements outside a loop form another semantic group. Statements inside one group are "neighbours", when statements in different groups are not. We need effective neural model to capture structure information and relations in source code and then translate it to the form, that we can use in author classification task.

In this paper, we use *Tree-Based Convolution Neural Network* based on programs abstract syntax trees (AST). The TBCNN model is a generic architecture, and can be applied to many software engineering tasks, but in our experiments we apply it to the task of source code author identification. It outperforms baseline methods like recursive neural network Socher et al. (2011b) proposed for NLP. [1]

II. Related Work

Deep neural networks have made significant breakthroughs in many fields. Stacked restricted Boltzmann machines and autoencoders are successful pretraining methods Hinton, Osindero, and Teh (2006); Bengio et al. (2006). They explore the underlying features of data in an unsupervised manner, and give a better initialization of weights for later supervised learning with deep neural networks. These approaches work well with generic data (e.g. data located in a certain dimensional space), but they may not be suitable for programming language processing, because programs contain rich structural

[1]We make our source code and the collected dataset available through the website(https://github.com/Saloed/VectorBuilder).

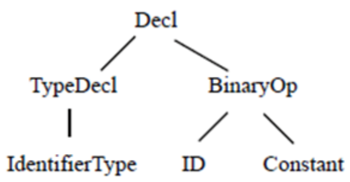


Fig. 1. An example of program AST

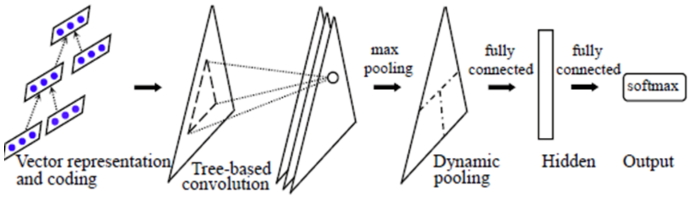


Fig. 2. A general structure of tree-based convolution neural network

information. Also, AST structures are very different from data sample (program) to another sample, and they can't be fed to a fixed-size network.

To extract various features from data, it's important to use human priors in networks Bengio, Courville, and Vincent (2013). For example convolution neural networks (CNNs, LeCun et al. 1995; Krizhevsky, Sutskever, and Hinton 2012), which specify information about relations in data. CNNs work with information in specified dimension and also fail to capture tree-structural information as in programs.

Some researches use recursive neural network (RNN) Socher et al. (2013, 2011b) as for NLP. We can code structural information is RNN, but the major problem is that only the root features are used for supervised learning. RNNs also suffer from the difficulty of training due to the long dependency path during back-propagation Bengio, Simard, and Frasconi (1994).

III. Tree-Based Convolution Neural Network

For programming languages we can build a tree representation (AST). The AST for a small code snippet shown on a figure 1. Each node in the AST is an abstract component in program source code. In the AST information about code structure store as parent/children relations. A node represent as p with children $c_1, \cdots, c_n$. To work with AST nodes we need to represent them as vector with real values. Such vector representation named *embedding*.

A. Embeddings

Every embedding capture information about type and relation information (position in tree) for it AST node.

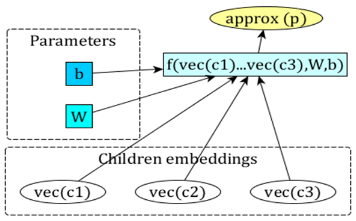


Fig. 3. A visualization of children approximation

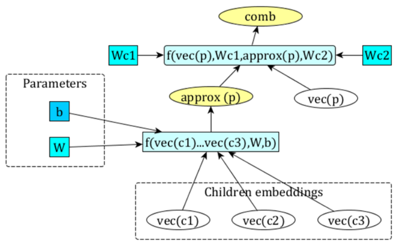


Fig. 4. A visualization of embeddings and children approximation combination

During the embedding learning process we want similar vector representation (with a small distance) for nodes, that are often found in a similar context (i.e FOR and WHILE). To capture relation information we use children approximation (figure 3).

Formally, let $\mathrm{vec}(\cdot) \in \mathbb{R}^{N_f}$ be the feature representation of a node, where N_f is the feature dimension. For each non-leaf node p and its direct children $c_1, \cdots, c_n$, we would like

$$\mathrm{vec}(p) \approx \tanh\left(\sum\nolimits_i l_i W_{\text{code},i} \cdot \mathrm{vec}(c_i) + \boldsymbol{b}_{\text{code}}\right) \quad (1)$$

where $W_{\text{code},i} \in \mathbb{R}^{N_f \times N_f}$ is the weight matrix corresponding to the node c_i; $\boldsymbol{b}_{\text{code}} \in \mathbb{R}^{N_f}$ is the bias. $l_i = \frac{\#\text{leaves under } c_i}{\#\text{leaves under } p}$ is the coefficient of the weight. (Weights $W_{\text{code},i}$ are weighted by leaf numbers.)

B. Combination layer

After we trained embeddings for every AST node, we would like to feed them to a convolution layer. For leaf node we use learned representation, but for non-leaf nodes we want to use a combination of trained embedding and children approximation (figure 4). Formally let $c_1, \cdots, c_n$ be the children of node p and the combined vector for this node is $\boldsymbol{p}$. We have

$$\begin{aligned}\boldsymbol{p} &= W_{\text{comb1}} \cdot \mathrm{vec}(p) \\ &\quad + W_{\text{comb2}} \cdot \tanh\left(\sum\nolimits_i l_i W_{\text{code},i} \cdot \mathrm{vec}(x_i) + \boldsymbol{b}_{\text{code}}\right)\end{aligned}$$

where $W_{\text{comb1}}, W_{\text{comb2}} \in \mathbb{R}^{N_f \times N_f}$ are the parameters for combination. They are initialized as diagonal matrices and then fine-tuned during training.

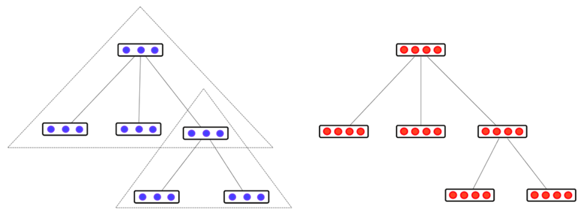


Fig. 5. A visualization of convolution over tree structure

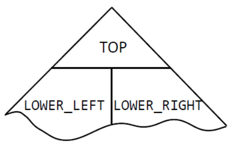


Fig. 6. Subtrees (parts) of original tree

C. Convolution layer

Now we have vector representation for each node in tree and now we apply convolution (figure 5). We applying convolution for each depth-1 subtree to extract features from every node relation group. During convolution process we translate vectors for each node from embedding space to convolution space.

Formally the convolution process is:

$$\boldsymbol{y} = \text{reLu}\left(\sum\nolimits_{i=1}^{n} W_{\text{conv},i} \cdot \boldsymbol{x}_i + \boldsymbol{b}_{\text{conv}}\right)$$

where $\boldsymbol{y}, \boldsymbol{b}_{\text{conv}} \in \mathbb{R}^{N_c}$, $W_{\text{conv},i} \in \mathbb{R}^{N_c \times N_f}$. ($N_c$ is the number of feature detectors.)

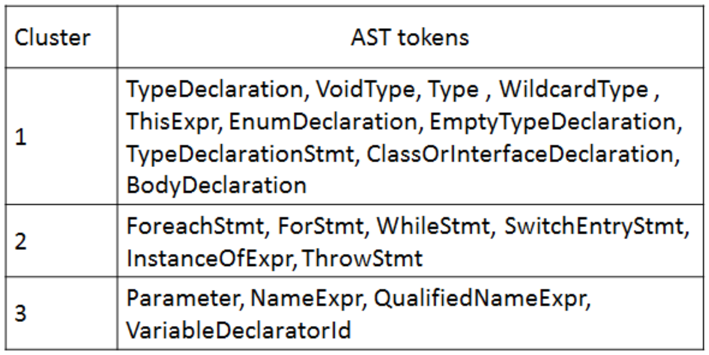
D. Pooling

After convolution we have a new tree in a convolution space with extracted features. The new tree has the same shape and size as the original one, which is varying among different programs. So, the extracted features cannot be fed to a fixed-size neural layer. Dynamic pooling Socher et al. (2011a) is applied to solve this problem.

The simplest approach is to pool all features to one vector. We call this *one-way pooling*. Concretely, the maximum value in each dimension is taken from the features that are detected by tree-based convolution. In other words we pick a strongest detected feature. An alternative, is *three-way pooling*, where features are pooled to 3 parts, TOP, LOWER_LEFT, and LOWER_RIGHT, according to the their positions in the AST (figure 6).

E. Classifier

We use vectors after pooling as classifier inputs to solve the problem of author identification. Each one of three vectors contains information about author code style, structure and other things which neural network understood during training process. In out of classifier we have a one-hot encoded vector, represented an author from an original subset of authors.



Cluster	AST tokens
1	TypeDeclaration, VoidType, Type , WildcardType , ThisExpr, EnumDeclaration, EmptyTypeDeclaration, TypeDeclarationStmt, ClassOrInterfaceDeclaration, BodyDeclaration
2	ForeachStmt, ForStmt, WhileStmt, SwitchEntryStmt, InstanceOfExpr, ThrowStmt
3	Parameter, NameExpr, QualifiedNameExpr, VariableDeclaratorId

Fig. 7. K-means clustering of embeddings

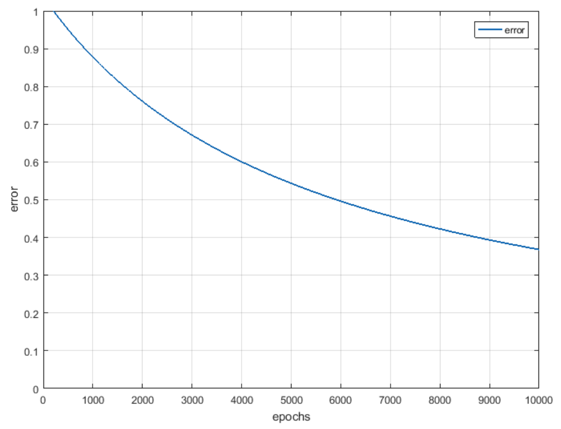


Fig. 8. Error of author classification during training process

IV. Evaluation

At first we trained embeddings on a large dataset with Java 8 code. We confirm that learned vectors are respond to their definition (small distance for similar AST tokens).
Secondly we evaluate classifier on an open source repository from GitHub with 20 authors.

A. Embeddings evaluation

To evaluate quality of trained embeddings we apply K-means clustering to a subset of embeddings. As shown at a result table (figure 7) tokens which are corresponds to similar context have a quite similar vector representation.

B. Classifier evaluation

To evaluate quality of author classifier based on tree-based convolution we train TBCNN and classifier for

10000 epoch and measure an error of author identification (figure 8). We use methods from target repository as dataset for our network. After many attempts we achieve about 60 percent accuracy in author classification task.

References

Bengio, Y.; Lamblin, P.; Popovici, D.; and Larochelle, H. 2006. Greedy layer-wise training of deep networks. In *NIPS*.

Bengio, Y.; Courville, A.; and Vincent, P. 2013. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* 35(8):1798–1828.

Bengio, Y.; Simard, P.; and Frasconi, P. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks* 5(2):157–166.

Chilowicz, M.; Duris, E.; and Roussel, G. 2009. Syntax tree fingerprinting for source code similarity detection. In *Proc. IEEE Int. Conf. Program Comprehension*, 243–247.

Collobert, R., and Weston, J. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*.

Dahl, G.; Mohamed, A.; and Hinton, G. 2010. Phone recognition with the mean-covariance restricted Boltzmann machine. In *NIPS*.

Dietz, L.; Dallmeier, V.; Zeller, A.; and Scheffer, T. 2009. Localizing bugs in program executions with graphical models. In *NIPS*.

Duvenaud, D.; Maclaurin, D.; Aguilera-Iparraguirre, J.; Gómez-Bombarelli, R.; Hirzel, T.; Aspuru-Guzik, A.; and Adams, R. 2015. Convolutional networks on graphs for learning molecular fingerprints. *arXiv preprint arXiv:1509.09292*.

Ghabi, A., and Egyed, A. 2012. Code patterns for automatically validating requirements-to-code traces. In *ASE*, 200–209.

Hao, D.; Lan, T.; Zhang, H.; Guo, C.; and Zhang, L. 2013. Is this a bug or an obsolete test? In *Proc. ECOOP*, 602–628.

Hermann, K., and Blunsom, P. 2014. Multilingual models for compositional distributed semantics. In *ACL*, 58–68.

Hindle, A.; Barr, E.; Su, Z.; Gabel, M.; and Devanbu, P. 2012. On the naturalness of software. In *ICSE*, 837–847.

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

Kalchbrenner, N.; Grefenstette, E.; and Blunsom, P. 2014. A convolutional neural network for modelling sentences. In *ACL*, 655–665.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. 2012. ImageNet classification with deep convolutional neural networks. In *NIPS*.

LeCun, Y.; Jackel, L.; Bottou, L.; Brunot, A.; Cortes, C.; Denker, J.; Drucker, H.; Guyon, I.; Muller, U.; and Sackinger, E. 1995. Comparison of learning algorithms for handwritten digit recognition. In *Proc. Int. Conf. Artificial Neural Networks*.

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

Mou, L.; Peng, H.; Li, G.; Xu, Y.; Zhang, L.; and Jin, Z. 2015. Discriminating neural sentence modeling by tree-based convolution. In *EMNLP*, 2315–2325.

Pane, J.; Ratanamahatana, C.; and Myers, B. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Human-Computer Studies* 54(2):237–264.

Peng, H.; Mou, L.; Li, G.; Liu, Y.; Zhang, L.; and Jin, Z. 2015. Building program vector representations for deep learning. In *Proc. 8th Int. Conf. Knowledge Science, Engineering and Management*, 547–553.

Piech, C.; Huang, J.; Nguyen, A.; Phulsuksombati, M.; Sahami, M.; and Guibas, L. 2015. Learning program embeddings to propagate feedback on student code. In *ICML*.

Pinker, S. 1994. *The Language Instinct: The New Science of Language and Mind*. Pengiun Press.

Socher, R.; Huang, E.; Pennin, J.; Manning, C.; and Ng, A. 2011a. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *NIPS*.

Socher, R.; Pennington, J.; Huang, E.; Ng, A.; and Manning, C. 2011b. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *EMNLP*, 151–161.

Socher, R.; Perelygin, A.; Wu, J.; Chuang, J.; Manning, C.; Ng, A.; and Potts, C. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 1631–1642.

Socher, R.; Karpathy, A.; Le, Q.; Manning, C.; and Ng, A. Y. 2014. Grounded compositional semantics for finding and describing images with sentences. *TACL* 2:207–218.

Steidl, D., and Gode, N. 2013. Feature-based detection of bugs in clones. In *7th Int. Workshop on Software Clones*, 76–82.

Zaremba, W., and Sutskever, I. 2014. Learning to execute. *arXiv preprint arXiv:1410.4615*.