Plagiarism Detection in Programming Assignments Using Deep Features

Jitendra Yasaswi, Suresh Purini, C. V. Jawahar
IIIT Hyderabad, India
jitendra.katta@research.iiit.ac.in, suresh.purini@iiit.ac.in, jawahar@iiit.ac.in

***Abstract*—This paper proposes a method for detecting plagiarism in source-codes using deep features. The embeddings for programs are obtained using a character-level Recurrent Neural Network (char-RNN), which is pre-trained on Linux Kernel source-code. Many popular plagiarism detection tools are based on n-gram techniques at syntactic level. However, these approaches to plagiarism detection fail to capture long term dependencies (non-contiguous interaction) present in the source-code. Contrarily, the proposed deep features capture non-contiguous interaction within n-grams. These are generic in nature and there is no need to fine-tune the char-RNN model again to program submissions from each individual problem-set. Our experiments show the effectiveness of deep features in the task of classifying assignment program submissions as copy, partial-copy and non-copy. Comparing our proposed features with handcrafted features (source-code metrics and textual features), we report *f1-score* improvement of 9.5% for binary classification and 5% for three-way classification tasks respectively.**

***Keywords*-deep features; recurrent neural networks; plagiarism detection; source-code;**

I. Introduction

The task of plagiarism detection can be treated as assessing the amount of similarity presented within given entities. These entities can be anything like documents containing text, source-code etc. Plagiarism detection can be formulated as a fine-grained pattern classification problem. The detection process begins by transforming the entity into feature representations. These features are representatives of their corresponding entities in a discriminative high-dimensional space, where we can measure for similarity. Here, by entity we mean solution to programming assignments in typical computer science courses. The quality of the features determine the quality of detection.

Popular source-code plagiarism detection tools such as MOSS [1] or JPlag [2] relay on analysis at textual level, and compare two student submitted assignment solutions for signs of plagiarism. These tools are based on n-gram techniques and create fingerprints to measure for similarity between submissions. Moreover, these tools ignore some hints like similar comments and white-spaces which are important cues for plagiarism [3]. Moreover, these approaches to plagiarism detection can not capture non-contiguous interaction present in the code and they can only tolerate small local changes. Simple obfuscation, such as noise injection, can evade from being detected as copy cases [4].

We aim at automatically learning feature representations for solutions submitted by students to programming assignments that can capture non-contiguous interactions.

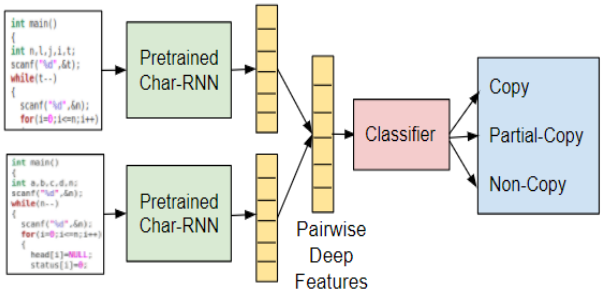

Figure 1. Our model, consists of a char-RNN followed by a SVM classifier. From two individual program submissions, pairwise deep features are obtained and they are classified as copy/partial-copy/non-copy.

We believe that features which capture such interactions can detect plagiarism even if the source-code is obfuscated. However, programs are highly structured in nature and applying machine learning directly to data in the form of programs is difficult. We overcome this difficulty by exploiting the fact that Programming Language Processing (PLP) can be treated similar to Natural Language Processing (NLP). Utilizing the "naturalness" in source-code [5] and treating programs as natural language, the traditional n-gram models may not be sufficient to model the non-consecutive interaction within n-grams present in the programs. This demands for powerful models that can capture long-term interactions.

Inspired by recent success of deep neural networks for learning features in other domains like Computer Vision (CV) and NLP, we are motivated to use such networks that enable us to learn the representations for the programs. Tasks like image classification [6], sequence prediction [7], image-captioning [8] and machine translation [9] are witnessing impressive results by leveraging the ability of deep neural networks to learn features from large amount of data. Similarly, deep learning has been used in domains of and Programming Languages (PL) and Compilers. Areas like program synthesis [10], induction, iterative compilation and compiler optimization [11] have benefited significantly with the use of deep neural networks.

In this paper, we employ a character-level language model to map the characters in a source-code to continuous-valued vectors called *embeddings*. We use these program embeddings as *deep features* for plagiarism detection in programming assignments. These deep features can capture the non-consecutive interaction within

n-grams present in programs at a syntactic level. These features are hierarchical in nature and learned through a series of non-linear transformations modeled by deep neural networks. These features are non-linear, *generic* which can generalize well [12]. We demonstrate the fact that these deep features are generic. By generic we mean that although they are learned using the task of sequence-prediction, they are directly applied on different dataset and to a different task like code plagiarism detection without the need to fine-tune on each individual problem-set.

Recently, character-level Recurrent Neural Networks (char-RNN) have been shown to generate text [13] and Linux C code [14]. Recent works also show that RNNs are particularly good at modeling syntactical aspects, like parenthesis pairing, indentation, etc. We prefer RNNs since they are inherently suitable for modeling sequences, attributed to their iterative nature. However, RNNs are difficult to train because of the *vanishing gradient* and *exploding gradient* [15] problems. To alleviate these problems, the Long short-term memory (LSTM) [16] units have been proposed. These LSTM units have special gated mechanism that can retain the previous state and memorize new information at the current time step. With the help of memory and different gates in LSTM, it enables us to capture non-consecutive interactions at syntactic level [17]. Hence, we use LSTM units in char-RNN to learn feature representations for programs. The detailed architecture of char-RNN model is described in Sec. IV-B. The contributions of our paper are as follows:

- We learn features that can capture non-contiguous interactions among n-grams present in the source-code, using deep neural networks.
- We demonstrate the generic nature, robustness and the superiority of our deep features when compared to source-code metrics and textual features in the task of plagiarism detection.

We evaluate the performance of the proposed features to detect plagiarism in programming assignments. Comparing our proposed features with popular handcrafted features (both source-code metrics and textual features), we report *f1-score* improvement of 9.5% for binary classification (copy/non-copy) and 5% for three-way classification (copy/partial-copy/non-copy) tasks respectively. For more details on classification tasks and results, refer to Sec. IV-E.

II. Background and Related Work

A. Language Modeling

The char-RNN used in our approach is an implicit statistical language model. A statistical language model is a probability distribution over sequences of words in a sentence. The goal of statistical language modeling is to predict the next word in textual data given context [18]. Traditional language models use the chain rule to model joint probabilities over word sequences as:

$$p(w_1, \ldots, w_N) = \prod_{i=1}^{N} p(w_i | w_1, \ldots, w_{i-1}) \qquad (1)$$

Many NLP tasks are performed using these statistical language models. Very recently, they have been put to use for many Software Engineering (SE) tasks such as code suggestions and fixing programming errors [19]. Hindle et al. [5] were the first to apply NLP techniques to source-code. They demonstrated that code written by humans is also likely to be repetitive and predictable like natural language. Hence, the code written by humans can be successfully modeled by statistical language models. Such models can be used to assist humans in code suggestions and design.

B. Related Work

Existing code plagiarism detection techniques can be put into four categories [20]. They are metric based, token based, tree based and Program Dependency Graph (PDG) based. However, each of these methods has some limitations. Metric based approaches fail because most of the metrics are highly sensitive to minor edits. Token based approaches cannot be applied to a repository containing multiple program submissions from different problem-sets. Abstract Syntax Tree (AST) based approaches produce many false-positives because of abstraction of the original code. PDG based approaches are costly as they involve comparing graphs.

In [3], the authors proposed a system that is based on properties of assignments that course instructors use to judge the similarity of two submissions. They proposed 12 textual features like similarity in comments and white-space etc. However, both MOSS and JPlag filter out these features when performing their analysis. Their main motivation is to use these features as cues in plagiarism detection. This system uses neural network based techniques to measure the relevance of each feature in the assessment. Their focus is on detecting plagiarized pairs within a single problem set. However, our proposed method is independent of problem set. The approaches proposed in [21] are based on searching similar n-grams or small character sequences (strings) between two source-codes. However, n-grams have their own limitations. Since they are simply frequency count of term co-occurrences, they are limited by the amount of context they consider [22].

In all the papers mentioned above, the main idea is to hand-craft certain features to capture similarity between two code submissions by exploiting either syntactical and stylistic (spaces, comments) aspects. Some of these are highly based on knowledge from generic programming constructs. Importantly, such handcrafted features are suitable only for specific tasks. This feature engineering itself is a fundamental point of difference of our paper, when compared with other papers. Here, we learn discriminative features from the data, which are generic in nature as opposed to specifying and engineering a set of specific features. A recent paper [23] is closely related to ours, in

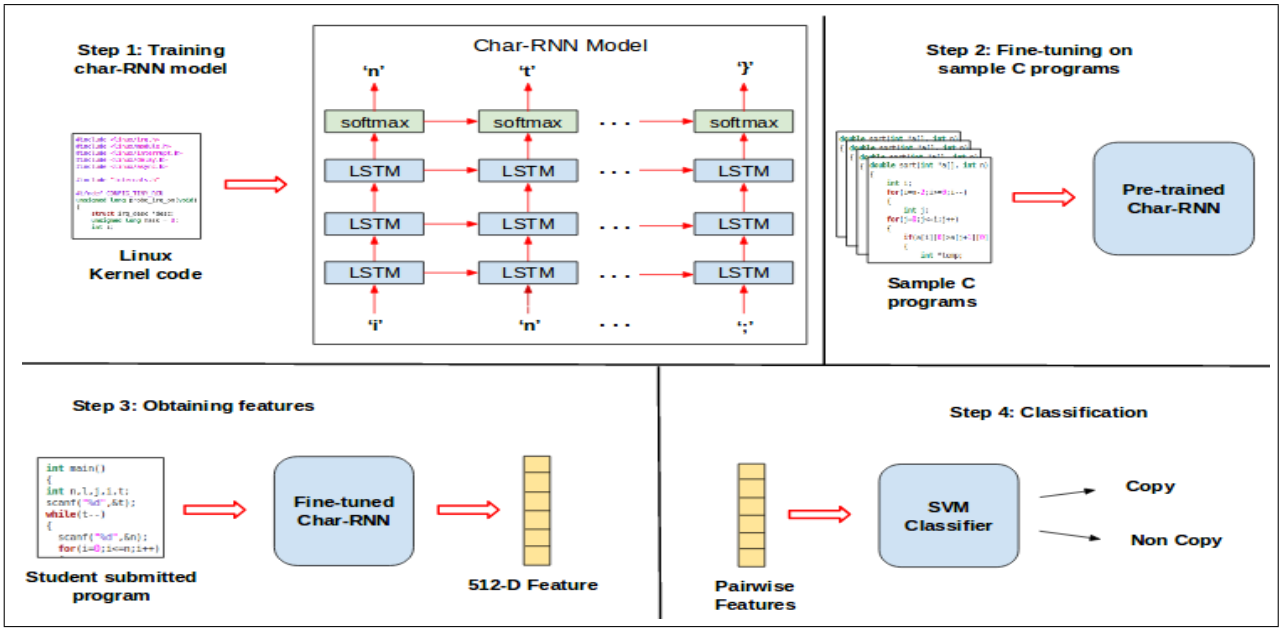

Figure 2. Our proposed approach step by step. First, we train a char-RNN model on the Linux Kernel source-code, then we fine-tune it to some sample C programs. Later, we use this fine-tuned model to obtain embeddings for programming solutions submitted by students and use these embeddings as features to detect plagiarized cases.

which the authors proposed a Tree-based Convolutional Neural Network (TBCNN), that can be applied on program AST. They use this architecture to classify programs by functionality and also to detect code snippets of certain patterns. However, features extracted from AST alone cannot be used for plagiarism detection, since we may loose some important cues like misspelled comments and white spaces [3].

III. Approach

Our proposed approach can be summarized into four steps as demonstrated in Figure 2.

A. Step 1: Train a char-RNN model

The first step in our approach is to exploit the "naturalness" in source-code by treating it as natural language and training the char-RNN [14] model on *Linux Kernel* source-code. A char-RNN is a character-level language model that accepts a sequence of characters as input and the model is trained to predict the next character in sequence at each time step. Consider there are V unique characters in Linux Kernel source-code. Each of the characters from the source-code is encoded as one-hot-vectors of V-dimensions. Now, a sequence of one-hot vectors $x = \{ x_1, x_2, \ldots, x_T \}$ are fed as input to a recurrent neural network, where T is the last time-step of the sequence. The model first computes a sequence of hidden vectors at each hidden layer. Let $h^L = \{ h_1, h_2, \ldots, h_T \}$ be the sequence of N-dimensional hidden vectors computed by the model at last layer. The last layer activations from the model are projected using a $[V \times N]$ weight matrix W_y to a sequence of output vectors $y = \{ y_1, y_2, \ldots, y_T \}$ to predict for next character in the sequence. However, these output vectors hold unnormalized log probability for next character in the sequence, which is normalized by employing the softmax function at the output layer. The model is trained using cross-entropy loss and we use LSTM units in place of vanilla RNN in our char-RNN. The parameter settings of the model and other training details are mentioned in Sec. IV-B.

B. Step 2: Fine-tuning char-RNN model

The second step is to use this pre-trained char-RNN model to fine-tune it to sample C programs. For this task, we have considered programming submissions from 4 problem-sets picked from 104-class program dataset [23]. In this step, the model is adapted to less complex C programs (unlike Linux kernel source-code). More details on fine-tuning are presented in Sec. IV-B.

C. Step 3: Obtaining Program Embeddings

The third step is to use this fine-tuned model to obtain feature representations for programming solutions submitted by students. The programs used here are picked from the dataset we created (see Sec. IV-A) and importantly, these are completely different from the ones used to fine-tune the model in the previous step. We consider the hidden-vectors from the last LSTM layer as the feature representations of the programs. A program is represented by the average of last layer LSTM hidden vectors from each character (i.e at hidden vector at each time step) similar to [24]. The embeddings obtained from the model are 512-D vectors. More details on features are given in Sec. IV-C.

D. Step 4: Classification

From the individual program feature representations, we construct pair-wise features and classify the submissions

Table I
OVERALL INFORMATION OF DATASETS USED. THE DATASET D2 REFERS TO THE 104-CLASS PROGRAM DATASET

Dataset	#problem sets	#code submissions	language
D1 (*this paper*)	22	4,700	C
D2 [23]	104	52,000	C

as copy/partial-copy/non-copy cases. Figure 3 demonstrates the process of construction of pair-wise features from individual program pair. These details are mentioned in Sec. IV-C. However, plagiarism cases only make up a small percentage student submissions in our dataset. The dataset is highly class imbalanced. To overcome class imbalance we use standard techniques like class-weighing scheme. These details are mentioned in Sec. IV-E.

IV. EXPERIMENTS

A. Datasets

The dataset we appropriately adopted is a collection of assignment solutions submitted by students from an introductory C programming course. We call this dataset as D1. Our dataset is formed out of 22 problem-sets. In each problem-set there are 70 to 250 student submissions. Program solutions falling in the same problem-set are functionally consistent. In total, there are about 4,700 submissions. The questions asked in the problem-sets range from more specific (as in case of tree-traversal) to diverse. Each problem-set is of varying difficulty, with student solutions ranging from 50 to 400 lines of code. The programs in our dataset are obfuscated in the following ways: variable name changing, careful conversion of while loop into for loop, dead code injection etc.

To train a classifier for plagiarism detection, data pairs (pairs of student's submissions) are needed. The labels make sense only for a pair of submissions and not for individual student submission. Explicitly enumerating all possible pairs for submissions in each problem set and annotating them is not feasible. Hence, we relied on MOSS scores and handcrafted features [25] for creating pairs. The constructed data pairs were annotated by teaching assistants. In total, there are about 3,600 program pairs. Out of these around 80 pairs are plagiarized, 110 pairs are partial-copy and the rest are non-copy. Table I provides the information about the datasets used.

For training our character-RNN model, we used the source-code of *Linux Kernel*. All the files are shuffled and concatenated to form a 6.2 Million character long dataset. During preprocessing stage, all the unicode characters are removed from the dataset. For fine-tuning our pre-trained char-RNN model, we used submissions from 104-class program dataset.

B. Char-RNN Training and Fine-tuning

We trained the char-RNN model on the source-code of *Linux Kernel* for sequence prediction task. Cross-entropy loss was used to train the model. Our char-RNN model consists of three hidden LSTM layers stacked on top of each other. Each of the layer consists of 512 LSTM cells/units. We used mini-batch stochastic gradient descent with RMSProp [26]. All the other hyper-parameters are same as mentioned in [14]. This pre-trained model was then fine-tuned to adapt it to the less complicated C/C++ program submissions, taken form from 4 problem-sets from 104-class program dataset. We call this dataset as D2. During fine-tuning, we freeze the weights of first two hidden layers and only the last hidden layer weights are allowed to update. Early stopping is used based on the validation performance.

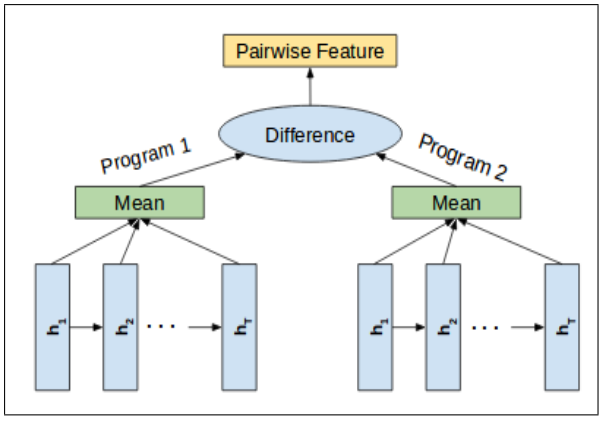

Figure 3. A schematic diagram showing our pairwise feature construction from two program submissions. The hidden vectors are from the last layer of char-RNN model.

C. Constructing Pairwise Representations

Given the feature representations of two individual programs from a problem-set, the pairwise features are constructed by taking the element-wise difference between individual program feature representations. Figure 3 demonstrates the process of construction of pair-wise features from individual program pairs.

D. Evaluation Measures

To analyze the classifier performance in supervised learning approach, we compute per-label *precision*, *recall* and *f1-score*. Suppose a label y is present in the ground-truth of m_1 instance pairs and it is predicted for m_2 pairs while testing out of which m_3 instance pairs are correct. Then its precision will be = m_3 /m_2 and recall will be = m_3 /m_1. Using these two measures, we get the percentage F1-score as $F1 = 2.P.R/(P + R)$, where P is precision and R is recall. After calculating precision, recall and f1-score metrics for each label individually, we calculate their unweighted mean.

E. Results

After obtaining pairwise feature representations for given program pairs, we train a Support Vector Machine (SVM) classifier in a variety of settings namely using deep features, full feature set (source-code metrics + textual features) and also using only textual features. We also treat the classification task as:

Table II
RESULTS OF BINARY CLASSIFICATION ON OUR DATASET. OBSERVE THE SIGNIFICANT BOOST IN F1-SCORE BY USING OUR PROPOSED DEEP FEATURES.

Features	Precision	Recall	F1-score
Textual features [3]	0.590	0.860	0.640
Source-code metrics [25]	0.670	0.800	0.715
Textual features + Source-code metrics	0.745	0.830	0.785
Deep features (*this paper*)	0.840	**0.940**	0.880
Deep features + Textual features + Source-code metrics (*this paper*)	**0.855**	**0.940**	**0.890**

Table III
RESULTS OF THREE-CLASS CLASSIFICATION ON OUR DATASET. OBSERVE THE SIGNIFICANT BOOST IN F1-SCORE BY USING OUR PROPOSED DEEP FEATURES.

Features	Precision	Recall	F1-score
Textual features [3]	0.413	0.490	0.423
Source-code metrics [25]	0.430	0.570	0.460
Textual features + Source-code metrics	0.440	0.573	0.463
Deep features (*this paper*)	0.470	0.653	0.513
Deep features + Textual features + Source-code metrics (*this paper*)	**0.490**	**0.660**	**0.543**

- a binary classification task (including only copy and non-copy cases).
- a three-way classification (includes partial-copy cases along with copy and non-copy cases).

1) Feature Comparison: We compare the proposed deep features with various other hand-crafted features like source-code metrics [25] and also few textual features [3]. The textual features are as follows: Difference in Length of Submissions (DLS) - the difference between the lengths of each student submission, Similarity as Measured by Diff (SMD) - the number of lines of common code in original submissions, Similarity of Comments (SOC) - the number of comments in common and Similarity in String Literals (SSL) - the similarity between the two sets of literals. To these features, we also add Edit Distance (ED) between two programs as an extra feature. These features are extracted at syntactic level of a source-code and are inherently pairwise features. The source-code metrics are typically a summary of the internal program representation. They consist of various elements that can capture basic block characteristics, method , control flow characteristics, frequency of variables and constants. Basic block features describe a program based on the number of basic blocks, basic blocks with successors, predecessors etc. Method features capture information related to calls in the methods. However, both textual features and source-code metrics fail to capture some of the essential aspects of a program like non-contiguous interactions at syntax level, needed to detect plagiarized program pairs.

2) Binary Classification: In the first task, we consider plagiarism detection as a binary classification problem. So we removed the instances belonging to the class of partial-copy cases from the dataset and trained an SVM classifier. Then, we opted for 4-fold cross-validation, training on 3/4 of the dataset and testing on the remaining 1/4. However, even after regularizing we found that the classifier was choosing to predict the class with the highest frequency. We use a class weighing scheme to alleviate the effect of class imbalance. The performance of classifier on the test-set using different features is shown in table II. Using our proposed deep features, we observe an improvement of 9.5% in f1-score when compared to source-code metrics+ textual features and an improvement of 24% in f1-score when compared to textual features alone.

Table IV
PER-LABEL PERFORMANCE IN BINARY CLASSIFICATION SETTING USING DEEP FEATURES + TEXTUAL FEATURES + SOURCE-CODE METRICS.

Class label	Precision	Recall	F1-score
Not copy	1.000	0.990	0.990
Copy	0.710	0.890	0.790

Table V
PER-LABEL PERFORMANCE IN THREE-WAY CLASSIFICATION SETTING USING DEEP FEATURES + TEXTUAL FEATURES + SOURCE-CODE METRICS.

Class label	Precision	Recall	F1-score
Not copy	0.970	0.890	0.930
Partial copy	0.110	0.300	0.170
Copy	0.390	0.790	0.530

Table IV shows per-label performance using deep features along with textual features and source-code metrics. Given below are the details of f1-score on per-label basis namely for copy and non-copy cases. The f1-score values obtained using textual features alone are 0.960 and 0.310 for non-copy and copy cases respectively. Using source-code metrics alone, the f1-score values are 0.980 and 0.450 for non-copy and copy cases respectively. By combining the textual features and source-code metrics, the f1-score values slightly increased and are 0.990 and 0.580 for non-copy and copy cases respectively. However, using deep features alone, we observed a significant boost in the f1-score values for copy cases. The f1-scores are noted to be 0.990 and 0.770 respectively.

3) Three-way Classification: As mentioned earlier, we also wanted to include the instances from the class of partial copy cases to our dataset. We treat the problem as a three-way classification problem and proceed by training a SVM classifier. The performance of classifier on the test-set using different features is shown in table III. Using our proposed deep features, we observe an improvement of 5% in f1-score when compared to source-code metrics+ textual features and an improvement of 9% in f1-score when compared to textual features alone.

Table V shows per-label performance using deep features along with textual features and source-code metrics. Given below are the details of f1-score on per-label basis namely for copy, partial-copy and non-copy cases. The f1-score values obtained using textual features alone are 0.890, 0.080 and 0.300 for non-copy, partial-copy and copy cases respectively. Using source-code metrics alone, the f1-score values are 0.910, 0.110 and 0.390 for non-copy, partial-copy and copy cases respectively. However, using deep features alone, we observed a significant boost in the f1-score values for copy cases. The f1-scores are noted to be 0.920, 0.140 and 0.480 respectively.

V. Conclusion

In this paper, we have explored the possibility of learning generic representations for source-code. Although the proposed deep features are learned using the task of sequence-prediction, they can be directly applied on different dataset and to a different task like code plagiarism detection without the need to fine-tune on each individual problem-set. Our experiments suggest that these features are very promising.

References

[1] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *ACM SIGMOD*, 2003. 1

[2] L. Prechelt *et al.*, "Finding plagiarisms among a set of programs with jplag," *J. UCS*, 2002. 1

[3] S. Engels, V. Lakshmanan, and M. Craig, "Plagiarism detection using feature-based neural networks," in *SIGCSE*, 2007. 1, 2, 3, 5

[4] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Transactions on Reliability*, 2016. 1

[5] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *ICSE, 2012*. 1, 2

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012. 1

[7] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013. 1

[8] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015. 1

[9] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014. 1

[10] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," *arXiv preprint arXiv:1611.01855*, 2016. 1

[11] C. Cummins, H. Leather, P. Petoumenos, and R. Mayr, "Deep learning for compilers," *ICSA*, 2017. 1

[12] Y. Bengio *et al.*, "Learning deep architectures for ai," *Foundations and trends® in Machine Learning*, 2009. 2

[13] I. Sutskever, J. Martens, and G. E. Hinton, "Generating text with recurrent neural networks," in *ICML*, 2011. 2

[14] A. Karpathy, J. Johnson, and L. Fei-Fei, "Visualizing and understanding recurrent networks," *arXiv preprint arXiv:1506.02078*, 2015. 2, 3, 4

[15] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *ICML*, 2013. 2

[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, 1997. 2

[17] S. Y *et al.*, "Deep lstm based feature mapping for query classification," in *HLT-NAACL 2016*. 2

[18] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, 2010, p. 3. 2

[19] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning." in *AAAI*, 2017. 2

[20] M. Chilowicz, E. Duris, and G. Roussel, "Syntax tree fingerprinting: a foundation for source code similarity detection," in *ICPC 2009*. 2

[21] E. Flores, A. Barrón-Cedeño, P. Rosso, and L. Moreno, "Towards the detection of cross-language source code reuse," *Natural Language Processing and Information Systems*, 2011. 2

[22] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," *arXiv preprint arXiv:1602.02410*, 2016. 2

[23] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing." in *AAAI*, 2016. 2, 3, 4

[24] D. Tang, B. Qin, and T. Liu, "Document modeling with gated recurrent neural network for sentiment classification." in *EMNLP*, 2015. 3

[25] J. Yasaswi, S. Kailash, A. Chilupuri, S. Purini, and C. V. Jawahar, "Unsupervised learning based approach for plagiarism detection in programming assignments," ser. ISEC '17. ACM, 2017. 4, 5

[26] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012. 4