Learning-based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection

Lutz Büch
Institute of Computer Science
Heidelberg University, Germany
lutz.buech@informatik.uni-heidelberg.de

Artur Andrzejak
Institute of Computer Science
Heidelberg University, Germany
artur.andrzejak@informatik.uni-heidelberg.de

***Abstract*—Code clone detection remains a crucial challenge in maintaining software projects. Many classic approaches rely on handcrafted aggregation schemes, while recent work uses supervised or unsupervised learning. In this work, we study several aspects of aggregation schemes for code clone detection based on supervised learning. To this aim, we implement an AST-based Recursive Neural Network. Firstly, our ablation study shows the influence of model choices and hyperparameters. We introduce error scaling as a way to effectively and efficiently address the class imbalance problem arising in code clone detection. Secondly, we study the influence of pretrained embeddings representing nodes in ASTs. We show that simply averaging all node vectors of a given AST yields strong baseline aggregation scheme. Further, learned AST aggregation schemes greatly benefit from pretrained node embeddings. Finally, we show the importance of carefully separating training and test data by clone clusters, to reliably measure generalization of models learned with supervision.**

***Index Terms*—Code Clone Detection, Abstract Syntax Trees, Embeddings, Recursive Neural Network, Siamese Network**

I. Introduction

Code clones can make up significant parts of large software systems [1], [2]. This problem is likely to become even more relevant in the future, due to today's ease of online collaboration and code distribution via open-source code repositories [3], question-answering forums [4], and app stores [5]. Besides clone detection, the notion of code similarity can be used in other applications like bug detection [2], [6], [7], performance prediction of code fragments [8], and information retrieval in software contexts [9].

In 2007, three surveys [10]–[12] described, categorized and evaluated the existing approaches to code clone detection. The algorithms fell into the categories of text-based, token-based, tree-based, PDG-based, metrics-based, and hybrid. The first four categories allow for an increasing degree of variation in the code fragments: from simple formatting differences to renaming of variables, over reordering of statements and other restructuring.

In recent years, more and more learning-based approaches to code clone detection have been studied [13]–[18]. One important aspect of learning-based techniques are embeddings of code tokens, as learning-based models usually depend on some continuous vector representation for input. Raw code and its traditional derivatives come in different forms, however always as a sequence, tree, or other graph of discrete tokens.

A recent study [13] comprehensively investigates the usefulness of different code representations and their combinations. Embeddings based on identifiers, Abstract Syntax Trees (ASTs), Bytecode and Control Flow Graphs are used for code clone classification (and detection of clone type) both in isolation, and as combined representations. One finding indicates that using AST-based representations strikes the best balance between precision and recall. Further, identifier-based representations can serve to complement AST-based representations.

In this work, we study representations derived from ASTs by learning from a clone/non-clone supervision signal. We use both AST node type and content (identifiers and literal values) to create the node representation. We implement a Siamese Network as a way to share weights of two instances of Recursive Neural Networks, that aggregate the ASTs of two given Java methods. We put a special focus on generalizability. We show that if training and test sets of clone and non-clone pairs are not separated by clusters, one does not measure the model's capability to generalize to other clone clusters.

We show that a strong baseline of aggregation is to simply average all node vectors. We also compare different hyperparameter settings and model variants. We find that the most important factor for a good model is a pretrained embedding. We show how error scaling solves the class imbalance problem of supervised code clone detection.

II. Background

Embeddings in Software Engineering

Design of learning approaches requires additional care if the entities of interest include sets of discrete tokens. These have to be mapped to a representation that makes them accessible to learning algorithms in an effective way, and that captures the semantics of these tokens. Often, this representation is a continuous vector of fixed length. Mappings of discrete tokens to real-valued vectors of fixed length are called *embeddings* or *distributed representations*.

Embeddings have found their way into the software engineering research literature as a way of facilitating neural network based models. Most embeddings in the software engineering literature represent method names, instances of API calls, bytecode or general code tokens [13], [16]–[21]. Others use word embeddings for natural language artifacts

[22], [23] or even mixed vocabularies of code tokens and natural language words [24].

Many embeddings are directly pretrained with an instance of the *word2vec* algorithm [19], [22], [23]. Others are trained with graph embedding algorithms [13], derived from training token-level Recurrent Neural Networks as language models for code [13], [17], [25], [26]. They are used to encode sequences, or the learned embeddings for the tokens themselves are used. Some embeddings are not pretrained but are jointly learned with the larger downstream task [20]. Sometimes, token-level embeddings are aggregated to sequence-level representations using recursive auto-encoders [13], [17], [25].

Researchers have also investigated the semantics captured in these embeddings, either by vector arithmetic yielding word analogies [19], [20] (as in the follow-up paper to word2vec method [27]), or by plotting a neighbourhood-preserving 2D projection [23]. In Section IV-H we present a qualitative assessment of the proposed content embedding which exploits such techniques.

Word2vec skip-gram

The word2vec [28] algorithm introduced in 2013 is currently one of the most popular approaches for training embeddings. It comes in two variants: continuous-bag-of-words and skip-gram. In the skip-gram model, a shallow neural network is trained to predict whether a given token fits in a given context. What constitutes an acceptable context for a token is defined by observing data in a corpus (e.g., natural language texts or a code repository). If a token occurs in a context in the data, it is deemed ”natural”, and thus, acceptable in terms of the learning goal. Other, ”unnatural” contexts are forged by altering natural contexts to generate negative training data. This process is called ”negative sampling”.

In more detail, in the skip-gram model a shallow (3-layer) neural network is trained to predict for a given token T the probability for any other token in our vocabulary of being in the context of T. The hidden layer of this network has k neurons, and the training adjusts the values of their (input-side) weights. Thus, in total, an $n \times k$ matrix w is learned, where n is the size of the vocabulary. The embedding for the token T is represented by a vector with k components, namely a row of w corresponding to the token T.

Recursive Neural Networks

Recursive (and Recurrent) Neural Networks differ from feed-forward Neural Networks in that the implicit computation graph is not static and the input is not of fixed shape. Instead, the mode of execution is iterative and goes over the whole input structure. In a Recurrent Neural Network (RNN, sometimes RtNN), inputs are sequences of tokens, and the RtNN reads in one token at a time, while updating an internal state. In a Recursive Neural Network (RNN, sometimes RvNN), the computation graph mirrors exactly the structure of the input tree. In the execution, one node is read at a time, while the states of the children are combined and updated in the current node. Therefore, the states are initialized at each leaf node. As a consequence, the computation tree will become as deep as the input trees.

Long Short Term Memory

In Deep Learning, there is the danger of exploding and vanishing gradients [29], because of high numbers of subsequent multiplications introduced by backpropagation through many layers. Memory-based models like the Long Short Term Memory unit (LSTM) [30] and Gated Recurrent Unit (GRU) [31] have shown to circumvent the problem of vanishing and exploding gradients. An LSTM implements an explicit memory structure that the network learns to use to model long term dependencies in the data. There are several gates that interact to compute a new state from input and a previous state. These gates can ”trap” the error and prevent unwanted divergence. The LSTM was originally conceptualized only for Recurrent Neural Networks which model sequential data, and only then generalized to Recursive Neural Networks [32] (i.e. tree topology). In this work we assume the latter scenario.

AST-based Code Clone Detection

As one baseline in our evaluation, we use the well-known AST-based algorithm Deckard [33]. This algorithm recursively aggregates tree patterns in ASTs until finally one characteristic vector at the root of the tree represents the whole tree. Deckard discards nodes of irrelevant type and all node content such as identifier names. The goal of Deckard was a robust and scalable code clone detection algorithm.

The theory developed in [33] shows that the Euclidean distance of Deckard’s vectors approximates the tree edit distance of the according pair of ASTs. In our evaluation against Deckard as a baseline, we discovered that the comparison of Deckard’s vectors performs significantly better (in terms of AUC), when one uses the cosine similarity instead. Note that this comparison is not the same as the final output of Deckard, which involves blocking and locality sensitive hashing. We intercepted the Deckard pipeline at the point where it outputs the tree vectors. Also, Deckard was designed to discover clones at varying levels of granularity, whereas our test data only defines clones at method level.

III. Approach

We use a Siamese Network [34] at the top level of our recursive aggregation approach. It compares the output of two identical Recursive Neural Networks with shared weights that encode two code fragments. The Siamese Network tries to maximize the cosine similarity for clones and decrease the cosine similarity for non-clones (attempting to make the similarity at least orthogonal). Figure 1 shows a schematic overview of one forward and one backward pass while training the Siamese Network. The error that is observed in the training set is backpropagated to the two instances of the Recursive Neural Network corresponding to the two trees.

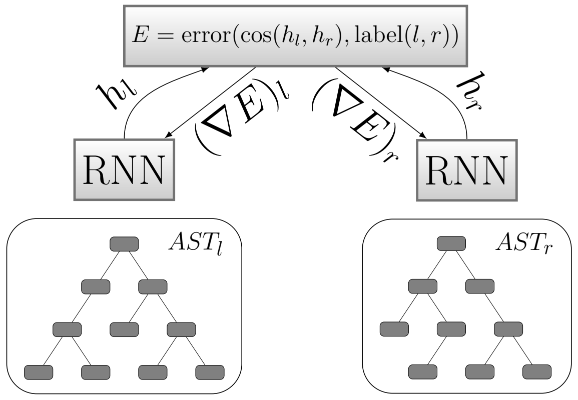

Fig. 1. The Siamese network during training

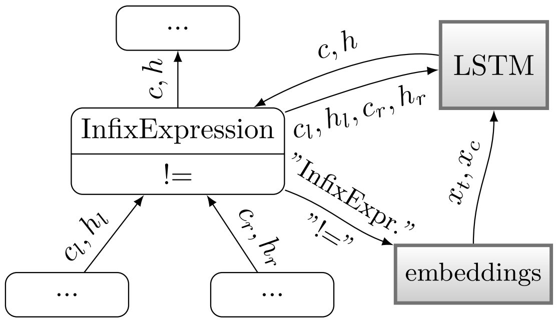

Fig. 2. Flow of data during evaluation of the RNN

A. Recursive Neural Network

Our Recursive Neural Network is designed to process ASTs. These are normalized to binary trees where each node holds two vectors. In Section IV-C we detail how the tree structure is derived and normalized. The two vectors represent one discrete label each - the *node type* and *node content*.

With *node types* we denote the abstract names that occur when deriving an AST, like "StringLiteral", "InfixExpression", "ForStatement" or "FieldAccess". These are defined by the grammar the parser uses. We define *node contents* as values that reflect the user-defined entities, like variable identifiers, class or method names or literal values. E.g., "Hello, World!", "HashMap", "false", "1024" or "getParent". Some of the content adheres to a grammar of its own (e.g., variable names in Java), while in other contexts it can be arbitrary (e.g., the content of a StringLiteral). These values are mapped to fixed length real vectors with the help of a lookup table. In Sections IV-C and IV-E we explain the details of the lookup tables and how we pretrain their parameters to obtain useful embeddings.

B. LSTM

We build on the definition of the n-ary tree-structured LSTM (conceived for constituency trees), as well as the original Torch7 implementation[1] of [32]. In our case, the trees are binary, there are (up to) two input vectors at every node, and only one output at the root node.

1) Forward pass: Our Recursive Neural Network operates as follows when reading in a tree (see Fig. 2). The single LSTM unit traverses the tree. At each node, it combines its own output at the children nodes, and the node type and content of the current node. For the child nodes, it receives the hidden states h_l, h_r and the cell states c_l and c_r. The node type and node content are fed in the respective lookup tables that output the according vector representations x_t and x_c. These six vectors c_l, h_l, c_r, h_r, x_t and x_c are the input to the LSTM cell and will influence the hidden state h and cell state c of the current node as follows (i, f_k and u are the input, forget and update gates of the LSTM, respectively):

$$i = \sigma\left(\sum_{p=t,c} W_p^{(i)} x_p + \sum_{p=l,r} U_p^{(i)} h_p + b^{(i)}\right) \tag{1}$$

$$f_k = \sigma\left(\sum_{p=t,c} W_p^{(f)} x_p + \sum_{p=l,r} U_p^{(f)} h_p + b^{(f)}\right), \tag{2}$$

$$\text{where } k \in \{l, r\}$$

$$u = \tanh\left(\sum_{p=t,c} W_p^{(u)} x_p + \sum_{p=l,r} U_p^{(u)} h_p + b^{(u)}\right) \tag{3}$$

$$c = i \odot u + \sum_{k=l,r} f_k \odot c_k \tag{4}$$

$$h = \tanh(c). \tag{5}$$

Note that this variant does not include an output gate, since this simpler variant performed better in our preliminary experiments. We base our implementation on the original implementation of [32].

Starting at the leaves, the network recursively aggregates the whole tree into a single vector representation. Each missing child or node content is simply represented as a zero vector. I.e., for a leaf node, the vectors c_l, h_l, c_r and h_r are all zero. The vector representation of the tree is the hidden state vector h of the LSTM at the root node.

2) Backward pass & weight update: When two trees AST_l, AST_r have been processed by the Recursive Neural Network, the Siamese Network will determine the error compared to the desired outcome (see Fig. 1). It will compute the relative contribution of each dimension for both trees and feedback these errors to the respective instances of the Recursive Neural Network:

$$\text{error}(s, l) = \begin{cases} 1 - s & l = \text{clone} \\ \max(0, s - m) & \text{otherwise} \end{cases} \tag{6}$$

where $s = \cos(h_l, h_r)$ and $l = \text{label}(l, r)$. The margin m is set to 0 by default. In the evaluation we explore other values for m.

[1]https://github.com/stanfordnlp/treelstm

After repeating this for many pairs, the Recursive Neural Network can average the error gradient at each output dimension for every AST in the training set. This error gradient is then used to compute the error gradient with respect to the weights of the network. This is done by the process of backpropagation through structure [35]. Finally, with this information, any optimization algorithm can take over updating the weights in order to minimize the error. In Section IV-G details our particular choices for the optimization parameters.

IV. Evaluation

A. Data collection

To implement our supervised learning schema, we need a supervision signal. We do not learn to predict a label for every tree (like cluster id), but we consider tree pairs as the data instances. We label each pair with *clone* or *non-clone*. To obtain ground-truth data for training and testing, we turn to the well-known BigCloneBench benchmark dataset [36] of method-level Java code clones.

For each clone cluster, we assemble methods that are confirmed as mutual clones, with a minimal confidence of 2 or greater. Here *minimal confidence* refers to the attribute of the clone data table in BigCloneBench. It is the minimum difference in true positive vs. false positive votes by the judges for a given clone pair. We exclude trees with more than 1000 nodes or with a depth greater than 28, because of limitations of our implementation. This eliminates 227 candidates (94 because of depth, 48 because of size, 85 because of both). We limit each clone cluster to 20 representative methods and discard clusters that have less than 5 methods after filtering for minimal confidence.

We only add methods that are not isomorphic to another method that is already in the cluster, with respect to their AST representation after pre-processing. These would only result in trivial matches, since they would necessarily always have the same vector representation. Note that this automatically removes all Type-1 code clones from both training, validation and test data, since these correspond to pairs of isomorphic ASTs.

This results in 609 methods distributed over 33 clone clusters. The ratio of Type-4 clones (as defined in BigCloneBench) in our data is 93.3%. We split this data into training, validation and test sets, such that methods of the same clusters belong to the same set. We use the ratios of 2/3 of clusters for training, and 1/6 for validation and testing, respectively. These ratios do not translate to the numbers of clones and non-clones, because of the combinatorics involved and since clusters do not have the same size. For cross-validation, we assemble three different splits of the data with non-overlapping validation and test sets (see Table II).

Importantly, this split is along the lines of clone *clusters*, not only individual pairs. We can only hope to measure real generalizability if the methods we evaluate on are not the same methods we trained our model on. Moreover, they should not even originate from the same cluster, or we run the risk that the model will pick up on patterns within a cluster, that may not be transferable to other clusters. This could bias the evaluation and over-estimate the performance. To make this point very clearly, we included the alternative mode of evaluation which does not do this careful distinction, see Section IV-H.

B. Imbalanced classes

Because of the combinatorial way of defining clones and non-clones, non-clones quickly outnumber clones by orders of magnitude. If we add all pairs to a training set despite of this fact, we end up having very imbalanced classes. This can lead to suboptimal learning, since the network is mainly punished for allowing any degree of similarity to non-clones, while the reward for yielding high similarity for clones is marginal. One way of addressing this issue is to downsample the non-clone class, as proposed in [18].

We do not choose to perform up- or downsampling to address this imbalance, however. Instead, we downscale the errors for non-clones. The most costly operation are the forward and backward passes over the ASTs by the RNN, which is unavoidable. The passes through the Siamese network, however, are negligible in cost. That provides the opportunity to consider all pairs without great cost and therefore exploit more information from the errors.

Consequently, to address the class imbalance, we scale the errors occurring when considering non-clones such that they have the same magnitude as those coming from clones. We do so by dividing the errors occurring for non-clone pairs by the ratio of number of non-clones to clones.

Different from the relatedness data from the NLP domain used in [32], our training data considers every possible pair, since we can deduce the label for a pair by membership in clusters. Therefore, we deviate from the mini-batch gradient descent implementation of [32] and implement learning with gradient descent - computed on the entire epoch of data.

C. Data preprocessing

We parse the raw Java source code with the Eclipse Java development tools (JDT) to obtain the AST trees. We make three modifications before using these ASTs (see Fig. 4). First, we cut out the node that carries the method name, to make sure the model cannot rely on the method name.

Second, we introduce additional nodes to obtain an equivalent binary tree. For a node with more than two children, we first halve the number of children. We assign the bigger half to a new left child, and the others (if more than one) to another new right child node. We repeat this step recursively for both children, as long as the smaller set of children still exceeds two. For an example with five children, refer to node *MethodDeclaration* in Fig. 4. Note that the sixth child, SimpleName["main"], is discarded. By this operation, we introduce additional depth of $\lceil log(n) \rceil - 1$ at a node with n children. This makes the trees necessarily bigger and deeper (see Table I). Note that Table I shows the statistics w.r.t. all parsed trees, not the final selection of 609 trees.

Finally, we map rare node contents to a generic *UNKNOWN* token. The node contents obey a Zipf distribution - there

```java
public static void main(String args[]){
    System.out.println("Hello, World!");
}
```

Fig. 3. *Hello World* code

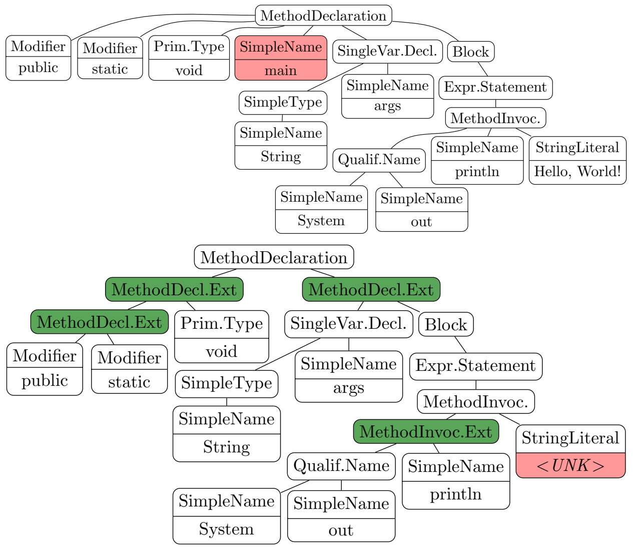

Fig. 4. AST tree of *Hello World* code, before and after preprocessing

TABLE I
STATISTICS OF TREE SIZES BEFORE AND AFTER BINARIZATION

	Node counts		**Tree depths**	
	Original	**Binarized**	**Original**	**Binarized**
min	13	15	5	7
5%	43	50	7	10
25%	80	93	9	13
median	130	151	11	16
average	203.98	240.66	11.81	17.16
75%	220	259	13	20
95%	577	681	19	28
max	6,554	8,117	55	70

are some very frequent contents, like ”float” or ”true”, and many one-off contents, like ”Mail was recorded successfully.”. There are 32k distinct contents, where 21k occur in just one method each (29k in one cluster each), another 5k in exactly two methods. We discard any token that occurs in less than 6 code clone clusters or in less than 50 clone pairs within any cluster. This leaves us with 1032 contents of the initial 32k. This prevents the model from overfitting by memorizing certain contingencies.

D. Quality metric AUC

For our evaluation we use Area Under the Curve (AUC) for the Receiver Operating Characteristic (ROC). AUC computes the integral of the function mapping false positive rates to the corresponding true positive rate for a given ranking of instances. Here, we order the pairs of methods by the cosine similarity of their representation vectors. AUC allows for another, more intuitive interpretation: it is the probability of ordering a randomly drawn pair of one clone and one non-clone correctly. That is, assigning a higher similarity score to the clone pair than to the non-clone pair. Such AUC gives us a good overall summary of the quality of the vectors, in terms of how well their similarity is able to separate the clone and non-clone pairs. In addition, it does not require a decision threshold, like precision, recall or F-measure do. So it is a good metric to estimate just the quality of the representations, because it is not influenced by any subsequent heuristic that picks a decision threshold.

Note that our study systematically under-estimates the performance in a real world application. Since we collect our data to filter out trivial clones resulting from matching isomorphic trees, we never encounter Type-1 code clones. This is on purpose, since Type-1 clones are trivially handled by any AST-based approach. The vast majority of clones in our data are Type-4 clones (93.3%). See [36] for a description of clone types.

E. Pretraining embeddings

In order for an AST to be consumable by a Recursive Neural Network, its nodes have to be mapped to vectors. This problem has been faced for a long time by the NLP (Natural Language Processing) community. The lookup tables that map the discrete values of node type and content to real vectors, do not necessarily need any attention, other than setting an arbitrary dimensionality.

Similarly as the weights of the LSTM unit, they can be updated to a useful state by the backpropagated error. However, to speed up learning, it makes sense to initialize their values in a useful way. In the ablation study in Section IV-H, we compare the performance of the RvNN model with and without pretrained embeddings. See Section II for existing work on embeddings in Software Engineering.

The word2vec approach [28] has been shown to work well in NLP contexts and is able to capture semantics in a way that can be reflected in vector arithmetic [27]. In particular, word2vec makes sure that tokens that can replace each other in many naturally occurring contexts will end up with similar representations. This provides the network with a starting point to generalize beyond arbitrary and insignificant naming choices (e.g., ”dir” versus ”path”). It also can help to account for the variation in Type-2 code clones which may differ in identifier names and literal values, in addition to trivial formatting variations.

We use the word2vec skipgram model and make three modifications. These are motivated by the relative benign nature of programming code, as compared to natural language. Code underlies a well-defined grammar, therefore we can always parse it into an AST tree. We use this fact to regularize the contexts that are extracted from the data to train the skipgram model.

First, we fix the context size and view contexts as ordered tuples, rather than bags of tokens. Second, in the case of node types, we define the context not by a window, but by the local neighborhood of the node (parent, children). That means that

TABLE II
Split of supervised data in training, validation and test set of clusters in three ways for cross-validation

Fold	Training clusters	Validation cl.	Testing cl.	# Training pairs	# Validation pairs	# Testing pairs
0	4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 29	39, 43, 45, 42, 41, 38	33, 32, 30, 31, 34, 36	3,277 pos. 64,619 neg.	1,140 pos. 6,000 neg.	1,140 pos. 6,000 neg.
1	4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 31, 32, 33, 34, 36, 38, 39, 41, 42, 43, 45	23, 30, 29, 26, 25, 24	22, 21, 20, 19, 17, 18	3,482 pos. 69,289 neg.	1,070 pos. 5,600 neg.	1,005 pos. 5,100 neg.
2	18, 19, 20, 21, 22, 23, 24, 25, 26, 29, 30, 31, 32, 33, 34, 36, 38, 39, 41, 42, 43	11, 12, 13, 14, 15, 17	4, 5, 6, 7, 10, 45	3,920 pos. 82,400 neg.	841 pos. 4,109 neg.	796 pos. 3,482 neg.

there is an ordered context of three elements. In the case of node contents, we use an ordered context of four elements (two preceding and two succeeding tokens). Finally, the total possible number of positive and negative samples is relatively small. Therefore, we consider its entirety, instead of statistical sampling and smoothing w.r.t. the relative frequencies.

F. Baselines

We first compare the supervised learning of representations against a very simple approach which exploits the pretrained embeddings. In this approach every AST is represented by the average over all of its node vectors, and compared to the representation of other ASTs by cosine similarity. This approach is very cheap, easy to understand and does not require further learning, given the pretrained embeddings.

We also compare our algorithm against the well-known Deckard code clone detection tool [33]. An important commonality with Deckard is that our algorithm summarizes ASTs to a single vector of fixed length to make them comparable by a similarity measure. That makes it possible to evaluate both with the aggregate quality metric of AUC. Five of the 609 trees in our data were not parseable by Deckard.

Note that we do not use the whole Deckard clone detection pipeline (with blocking, LS hashing, ...), but solely the pipeline part for computing the Deckard vector representation of a given Java fragment. Also, Deckard is designed to discover cloned code fragments at different levels of granularity. Our evaluation uses only code clones from BigCloneBench, which are clones on the method level. We consider different methods with the same AST as identical, since these Type I clones would only mean trivial clones for both our approach and Deckard. Thereby, we under-estimate the overall performance to detect clones, for both Deckard and our approach.

We observed that the Deckard vectors yield a better separation of clones from non-clones if cosine similarity is used instead of the intended Euclidean norm. For this reason, we report both performances.

Note that the baseline approaches do not require (supervised) learning. For the sake of compatibility, we evaluate them on the same three test sets we constructed for the supervised cross validation.

TABLE III
The performance w.r.t. different dimensionalities; baselines

	dim.	#params	AUC$_0$	AUC$_1$	AUC$_2$	∅AUC
	300	752,294	0.799	0.771	0.903	**0.824**
	200	345,094	0.766	0.861	0.876	**0.834**
	150	201,494	0.764	0.841	0.929	**0.845**
	100	97,894	0.690	0.840	0.857	**0.795**
	50	34,294	0.613	0.817	0.793	**0.741**
	30	20,054	0.588	0.768	0.783	**0.713**
Emb. ∅	14	10,694	0.784	0.752	0.885	**0.807**
D. cos.	300	–	0.565	0.746	0.798	**0.703**
D. Eucl.	300	–	0.543	0.653	0.602	**0.599**

TABLE IV
The performance for dimensionality 150 w.r.t. other changes

	#params	AUC$_0$	AUC$_1$	AUC$_2$	∅AUC
10×neg. scal.	201,494	0.744	0.817	0.912	**0.824**
margin $m = 0.9$	201,494	0.750	0.793	0.885	**0.810**
Emb. ∅ (14D)	10,694	0.784	0.752	0.885	**0.807**
margin $m = 0.5$	201,494	0.726	0.813	0.856	**0.798**
with output gate	249,194	0.700	0.763	0.822	**0.761**
no pretraining	201,494	0.668	0.807	0.763	**0.746**
*Unaware split**	*201,494*	*0.990**	*0.997**	*0.991**	***0.993****

G. Hyperparameters

We follow the choices in the implementation of [32]. We employ the learning rate of 0.05, and a separate learning rate for the embeddings of 0.1. As an optimizer, we use *adagrad*. A difference to [32] is that we implement gradient descent instead of mini-batch gradient descent, as explained in Section IV-B.

We run each experiment for 500 epochs. We use the evaluation of AUC on the validation set to retrospectively choose the best performing model state. However, the performance on the validation set can sometimes make chaotic jumps in the first few dozen epochs. Therefore, we exclude the first 100 epochs from consideration.

The standard hyperparameters for our experiments are as follows. The node content embedding is 10-dimensional, the node type embedding 4-dimensional. We initialize both with separate word2vec-like pretraining (see Section IV-E). The dimensionality of hidden and cell states is 150 per default. Below, we report the results for runs with one changed hyperparameter at a time.

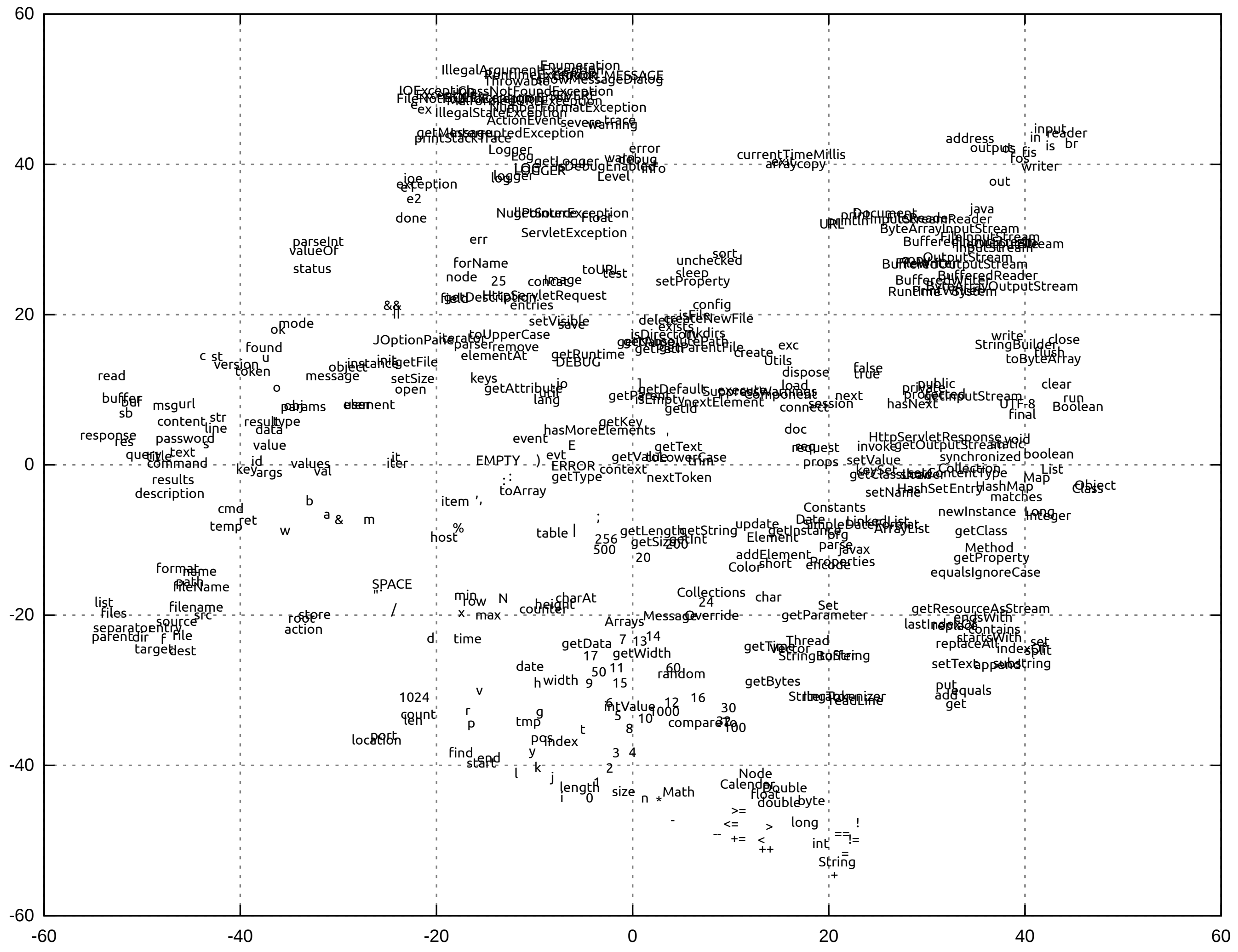
Fig. 5. T-SNE projection of content embedding vectors

H. Results

We evaluate each model by cross-validating with the available data. See Table II for the three different splits. We report the AUC on each test set, that is evaluated on the best model, as measured on the validation set (see Section IV-G). Finally, we average these three AUC values to get a measure for the overall performance.

Influence of dimensionality: In Table III we list the result of running experiments with varying dimensionality for the hidden and cell states h and c. The matrices $U_p^{(*)}$ grow quadratically in size with growing dimensionality, increasing the total number of parameters quickly.

More parameters help to converge to a vector representation that implies similarities for method pairs that reflect more closely the clone/non-clone relation. The AUC on the training data itself converges to about 0.92 up to 1.00, over all experiments. Note that the training does not optimize the model for AUC directly, but for high cosine similarity for clones and low similarity for non-clones. At a certain amount of complexity, the model is able to overfit to the particularities in the training data to such a degree, that it is to the detriment of the performance on unseen clone clusters. The tipping point seems to be a dimensionality of about 150. Before and after that point, performance degrades.

The reported parameters include 364 parameters for the 4D embedding of 91 node types and 10,330 parameters for the 10D embedding of the 1,032 node contents and the *UNKNOWN* token. In most cases, the number of these weights is small compared to the LSTM weights. However, in the case of the 50D or 30D models, these numbers actually are of the same order of magnitude as the LSTM weights.

Influence of embeddings: To measure the impact of pretraining the node type and node content embeddings we ran an experiment in which the pretraining was skipped. Instead, the lookup tables where initialized with normally distributed random numbers. Table IV shows the impact of the pretraining of embeddings. The average performance drops from 0.845 to only 0.746 AUC. We included also the baseline that uses the pretrained embeddings ("Emb. ∅ (14D)"). It can surpass many of the non-optimally configured models. Especially, it outperforms the otherwise optimally configured model, that starts training with no pretrained embeddings. From this comparison we conclude that a good pretrained embedding has a higher impact on performance than a complex model on top of it.

Variation of network layout: In Section III-B1 we introduced the definitions for our LSTM variant which does not use an output gate. We ran an experiment to evaluate the influence of an additional output gate. The LSTM is changed by introducing the two following equations, one overriding the former definition of h:

$$o = \sigma\left(\sum_{p=t,c} W_p^{(o)} x_p + \sum_{p=l,r} U_p^{(o)} h_p + b^{(o)}\right) \tag{7}$$

$$h = o \odot \tanh(c) \tag{5*}$$

These definitions introduce a significant number of parameters and the model with an output gate performs much worse. The number is not as high as in the 200D or 300D models, but the internal structure of this LSTM variant is more convoluted. Apparently this is enough to overfit to the training data.

Variation of the error function: We observed in our preliminary experiments that the class imbalance between the classes clone and non-clone does indeed have an adverse effect on the learning progress. All above experiments use the error scaling discussed in Section IV-B. We tried an experiment with a scaling factor of 10. That is, we add a factor of 10 to the down-scaled error for non-clones. This setting is an intermediate between full scaling and no scaling. Full scaling works best, as seen in Table IV, but the learning curve indicates convergence is faster and more stable for non-clone errors scaled by 10.

A similar observation can be made when varying the margin m from Equation 6, which is set to 0 per default. The performance is not optimal for the other values, but we get faster convergence and, in the case of the moderate value $m = 0.5$, more stable learning curves.

Importance of cluster-aware data splits: In supervised learning of code clones, individual labeled instances are pairs of code fragments. We noted in Section IV-A that we carefully split our labeled data into training, validation and test sets, based on clusters. Therefore, each method of a given cluster will *only* appear in pairs that belong to either training, validation or test set. In this sense, the splitting is aware of the clusters the methods belong to. We decided to use this approach as otherwise we would get an overestimation of the performance on unseen data.

We conducted an experiment to provide evidence for this claim. We created three new splits of the data into training, validation and test sets - this time, simply randomly drawing pairs of methods. However, the pairs are still drawn only and kept the same over the training epochs. Also, each pair only occurs in either training, validation or test set, or is not drawn at all. We made it so that the numbers of positive and negative pairs coincide with the numbers in Table II. So for the first split, the training set would consist of 3,277 positive and 64,619 negative pairs. Just this time, the clones are drawn from all 33 clusters, and the non-clones may have methods from any two of the 33 clusters. Since this experiment is conducted on different data, we distinguish it by marking the results with asterisks.

Fig. IV shows that the performance in terms of AUC is in fact much higher than in the other cases (see row labelled "unaware split"). These values are actually on average equal to the AUC measured on the training set. The reason for this is that here, the model is trained on the same clusters as it is evaluated on. Whereas if we ensure that the test phase only includes unseen clusters, the AUC in testing cannot reach the AUC of the training. We conclude that evaluating supervised code clone detection on the same clusters gives misleadingly good results, that cannot translate to unseen clone clusters, as shown by the other experiments.

Qualitative assessment of content embedding: The pretrained embeddings offer a considerable performance gain and even make for a good baseline, when simply averaging these vectors over all nodes. A qualitative evaluation described below shows that these embeddings convey potentially useful information. We visualize the local neighborhoods of the 10-dimensional content vectors via T-SNE [37] in Fig. 5, for which we use a perplexity of 15. We only include the most frequently used contents to keep the diagram readable. Concretely, we chose as a cut-off point contents that are used in at least 16 clusters or 50 methods, which is true for a total of 459 of the 1032 content strings we use in the embeddings. This excludes the more idiosyncratic or domain specific strings that only occur in smaller number of specific methods or clusters. Note that we display the string consisting of one space character as "SPACE" and the empty string as "EMPTY".

Some striking structures are visible in the diagram. We can clearly identify distinct clusters of strings that are associated with exceptions (on the top, in the middle). Other clusters deal primarily with I/O streams (top right), operators and primitive types (bottom), and filenames (left bottom). In the context of code clones, this helps to account for variation found in Type-2 clones. Those vary only in the choice of a variable name that is swapped out for a similar name. Further, this puts variation in the scope of detectability, where types are changed to get an implementation with higher precision, changes due to API renaming, or replacing of a caught exception type for a more or less specific other type.

There are only a total of 91 different node *types* including the artificial ones introduced for the binarization. Consequently, their embedding vectors lie very sparsely in the 4-dimensional representation space and its visualization does not allow for a similarly rich qualitative interpretation.

V. Related Work

A lot of recent work uses neural networks of different kinds to compute continuous vector representations for code fragments to solve different tasks.

Code Clone Detection

The closest related work [16] also uses a Siamese Network of Recursive LSTM to learn a representation of code in a supervised way. Further, the framework learns to transform this continuous vector representation to a binary hash code, which is compared by Hamming distance. The models are trained on BigCloneBench and OJClone and evaluated in terms of precision, recall, and $F1$ for a fixed decision threshold and compared to state-of-the-art approaches and representations. In this work, we put a focus on generalizability and show that only a careful distinction of training, validation and test pairs by clusters guarantees a realistic measurement of generalization. We also offer an ablation study of model variants and hyper-parameters.

Oreo [14] uses a Siamese Network architecture to predict code clones, where Java code fragments are represented by software metrics. Filtering by similar size and sufficient overlap of so-called ”action tokens” is employed as a blocking method. The training supervision signal may be provided by any state-of-art code clone detector; here SourcererCC [38]. The authors show that the resulting model is highly scalable, competitive in performance, and outstanding in the so-called ”twilight zone” of code clones (moderately Type-3 and beyond).

DeepSim [15] represents Java methods by matrices encoding their control flow and data flow graphs. It then pretrains the layers of a feed-forward network by a Stacked Denoising Auto-Encoder. The learned representation is compared to a fine-tuned version obtained by supervised learning with a Siamese architecture, and to state-of-the-art approaches.

CCLearner [18] aggregates two ASTs in token-frequency lists of eight categories of tokens. It then computes the dice index for all eight pairs of lists. A Neural Network with two hidden layers learns to predict the supervision signal clone/non-clone from this. CCLearner uses only one clone cluster, #4, from BigCloneBench for training, and 9 remaining clusters for testing. Our approach does not rely on preselected categories of tokens or hand-crafted features. Instead, it learns a non-linear aggregation scheme by a Recursive Neural Net.

The algorithm in [17] is an unsupervised learning approach for code clone detection. The authors train a language model for the tokens at the leaf nodes with a Recurrent Neural Network. Then they train a recursive auto-encoder to summarize the vectors of length n representing two child nodes into a single vector of length n, by minimizing the reconstruction error. The resulting vector representing the whole AST is used for comparison by the Euclidean norm. Our approach uses a supervision signal from known clones and non-clones and can directly learn to distinguish these in a training set.

Other tasks

Code2vec [20] introduces a vector representation for ASTs of Java methods, to predict their method names. The basic units under consideration are paths between two leaf nodes. These are treated as distinct tokens. Together with the two leaf nodes they build a path context. Each of the three tokens of a path context is mapped to a vector by an embedding. A Neural Network learns to assign attention weights to all path contexts found in an AST. These are used to compute a weighted sum of the path context vectors. The network is trained to maximize the cosine similarity to the representation of the method's name, from an embedding. Our approach does not rely on any hand-crafted feature of an AST; the features emerge from the supervised training.

A Siamese Network has been used by [22] to link different types of software engineering artifacts in natural language (requirements and design descriptions). The two network copies that the Siamese Network combines are Recurrent Neural Networks, reading in two sequences of Natural Language tokens. These tokens are represented by an embedding trained with a word2vec skip-gram algorithm. Our approach reads tree-structured data instances (ASTs) originating from the same domain and aims at retrieving clones.

VI. Future Work

To prevent overfitting, we use closed vocabulary for node contents, i.e. restrict node contents to those with strings occurring more frequently. In future work our approach could be extended to work with an open token vocabulary by modeling node content strings via a Recurrent Neural Network. This would allow generalization of vector representations to handle unseen strings in a meaningful way, and is compatible with the remainder of the proposed method.

Another possible future work relates to model interpretability and understanding about how these models operate in detail. In particular, an interesting challenge is to study whether it is possible to distill certain aggregation schemes as explicit and transparent heuristics. These could serve as ”stand-alone” features of ASTs.

Our model does not include any hand-crafted features that were created with the task of code clone detection in mind. Therefore, the continuous vector representation of code fragments can be used for a whole range of tasks. It would be particularly interesting to observe whether using a model pretrained on the code clone detection task shows faster or better convergence on unrelated tasks such as generating natural language summaries of code fragments. The practice of such transfer learning is applied in fields like computer vision [39] and NLP [40].

Conversely, it might be worthwhile to study whether Code Clone Detection can benefit from representations pretrained on a different task, via transfer learning, or jointly trained via multitask learning.

VII. Conclusion

In this paper, we study a data-driven approach to deriving non-linear aggregation schemes for abstract syntax trees. We use the cosine similarity of the vector representations of method pairs to evaluate on a test set of clone clusters from BigCloneBench.

We find that performance in terms of AUC increases until the dimensionality of 150, after which point overfitting sets in. We find further evidence for the usefulness of pretrained embeddings in learning-based approaches for software engineering. A good and simple baseline is to simply average node vectors, which especially does not involve further learning.

We show that error scaling is a good way to address the class imbalance problem. Our results also imply that in supervised learning of code clone detection, it is important to split training and test data by clusters, to get a useful estimation of generalization to unseen clusters.

References

[1] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE, 1995, pp. 86–95.

[2] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 48–62.

[3] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on github," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 84, 2017.

[4] L. An, O. Mlouki, F. Khomh, and G. Antoniol, "Stack overflow: a code laundering platform?" in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 283–293.

[5] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 37–54.

[6] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone smells in software evolution," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 24–33.

[7] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[8] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *Parallel Architectures and Compilation Techniques (PACT), 2006 International Conference on*. IEEE, 2006, pp. 114–122.

[9] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, 2009.

[10] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[11] R. Koschke, "Survey of research on software clones," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[12] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," vol. 33, no. 9, 2007.

[13] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *International Conference on Mining Software Repositories*, 2018.

[14] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. Lopes, "Oreo: Detection of clones in the twilight zone," *arXiv preprint arXiv:1806.05837*, 2018.

[15] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 141–151.

[16] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[17] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[18] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 249–260.

[19] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 438–449.

[20] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *arXiv preprint arXiv:1803.09473*, 2018.

[21] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[22] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 3–14.

[23] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[24] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 404–415.

[25] 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*. ACM, 2017, pp. 763–773.

[26] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, "A deep neural network language model with contexts for source code," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 323–334.

[27] 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*, 2013, pp. 3111–3119.

[28] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[29] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber ***et al.***, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.

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

[31] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[32] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[33] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[34] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," in *Advances in Neural Information Processing Systems*, 1994, pp. 737–744.

[35] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Neural Networks, 1996., IEEE International Conference on*, vol. 1. IEEE, 1996, pp. 347–352.

[36] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 131–140.

[37] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[38] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1157–1168.

[39] L. Shao, F. Zhu, and X. Li, "Transfer learning for visual categorization: A survey," *IEEE transactions on neural networks and learning systems*, vol. 26, no. 5, pp. 1019–1034, 2015.

[40] L. Mou, Z. Meng, R. Yan, G. Li, Y. Xu, L. Zhang, and Z. Jin, "How transferable are neural networks in nlp applications?" *arXiv preprint arXiv:1603.06111*, 2016.