Deep Learning Code Fragments for Code Clone Detection

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk
Department of Computer Science
College of William and Mary
Williamsburg, Virginia, USA
{mgwhite, mtufano, cvendome, denys}@cs.wm.edu

ABSTRACT

Code clone detection is an important problem for software maintenance and evolution. Many approaches consider either structure or identifiers, but none of the existing detection techniques model *both* sources of information. These techniques also depend on generic, handcrafted features to represent code fragments. We introduce learning-based detection techniques where everything for representing terms and fragments in source code is mined from the repository. Our code analysis supports a framework, which relies on deep learning, for automatically linking patterns mined at the lexical level with patterns mined at the syntactic level. We evaluated our novel learning-based approach for code clone detection with respect to feasibility from the point of view of software maintainers. We sampled and manually evaluated 398 file- and 480 method-level pairs across eight real-world Java systems; 93% of the file- and method-level samples were evaluated to be true positives. Among the true positives, we found pairs mapping to all four clone types. We compared our approach to a traditional structure-oriented technique and found that our learning-based approach detected clones that were either undetected or suboptimally reported by the prominent tool Deckard. Our results affirm that our learning-based approach is suitable for clone detection and a tenable technique for researchers.

CCS Concepts

•**Software and its engineering → Reusability;**

Keywords

code clone detection, machine learning, deep learning, neural networks, language models, abstract syntax trees

1. INTRODUCTION

Abstraction is the most important word in software engineering (SE). Accordingly, software repositories are replete with abstractions, which give software engineers the ability to manage complexity by separating concerns and handling different details at different levels. Abstractions at all levels of granularity are complemented by implementations. These implementations can be developed from scratch, or they can be *cloned* from existing code fragments [1,2]. If existing code provides a reasonable starting point for the implementation, then a software engineer may clone the code by copying and pasting the fragment. Another way that clones can be introduced in a software system is when an engineer unknowingly develops an implementation that is similar to an existing one. Copying and pasting code and subsequently modifying the copied fragment may yield textually similar code fragments where the similarities can be characterized by their syntax. On the other hand, when an engineer unknowingly develops an implementation that is similar in intent to something that already exists, she may create clones that are functionally similar yet syntactically different.

Detecting clones is an important problem for software maintenance and evolution. Although prior work has demonstrated several adverse impacts of code cloning [3–5], cloning is not necessarily harmful [6, 7]. Nor should clones necessarily be refactored [8, 9]. Nonetheless, the ability to automatically detect that two fragments are similar is critical in many applications [10], e.g., detecting library candidates [11, 12], aiding program comprehension [13], detecting malicious software [14], detecting plagiarism or copyright infringement [15, 16], detecting context-based inconsistencies [17–19], and searching for refactoring opportunities [20–22]. Roy and Cordy [10] classified clone detection techniques by their "internal source code representation," synthesizing a taxonomy of text-, token-, tree-, graph-, and metrics-based techniques. In this paper, our newfangled approach to mining internal source code representations gives way to a new, *learning-based* paradigm.

The clone detection process begins by transforming in situ code into representations suitable for assessing similarity [10, 23]. For instance, to represent fragments, traditional tree-based clone detection tools depend on **handcrafted features** that are tightly coupled to **generic** programming constructs. In this respect, the domain information that is rooted in **identifiers** [24–27] is discarded, breaking the link between information that can be learned at *both* the lexical level and syntactic level. Moreover, declaring features (e.g., the occurrence counts of programming constructs) applies a great deal of prior knowledge to how we can automatically represent fragments. However, it is reasonable to expect that software systems from different application **domains** and at different stages of **development** yield *unique* patterns in

ASE'16, September 03-07, 2016, Singapore, Singapore
© 2016 ACM. ISBN 978-1-4503-3845-5/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2970276.2970326

source code that would be revealing for problems like code clone detection. *Yet these patterns are not necessarily captured using approaches that establish a generic feature space*, and the only way these useful, latent features can be descried is by using perspectives of code that are learned, i.e., learning the representations themselves. Automatically learning the representations, or "representation learning" [28, 29], relaxes the prior knowledge used to transform raw data like source code into suitable representations, automating what has been a manual step in the detection process. Mining effective source code features, analyzing the language of identifiers in source code, analyzing syntactic patterns, and engineering approaches that can adapt to changing repositories are fundamental SE research problems. Engineering a clone detection approach that considers all of these concerns is what motivates our work. Our key result is a new set of techniques that *fuse* and *use*. We fuse information on structure and identifiers in code and use the data in repositories to automate the step of specifying transformations.

Our key insight to representing code fragments for code clone detection is two-fold. First, our approach maps the terms in fragments to continuous-valued vectors such that terms used in similar ways in the source code repository map to similar vectors (Sec. 3.1). This transformation from terms to vectors is fundamentally different than the token abstraction used by token-based techniques (Sec. 2.1). Second, our representation learning-based approach is designed to learn discriminating features for fragments at different levels of granularity (Sec. 3.2) rather than depend on intuitive (yet limited) features that are designed around the structural elements of a language like tree-based techniques (Sec. 2.1).

The essence of our approach goes back to **abstraction** and handling different details at different levels in SE. We propose exploiting this guiding principle in software construction, so our techniques for modeling source code exploit empirically-based patterns in *structures* of terms in code just as language modeling has exploited patterns in *sequences* of terms. To this end, we pair lexical analysis with recurrent neural networks (Sec. 3.1) and syntactic analysis with recursive neural networks (Sec. 3.2). The purpose of coupling the front end of the compiler with deep neural networks and deep learning (Sec. 2.3) is to provide a framework for linking patterns mined at the lexical level (by modeling how terms are used) with patterns mined at the syntactic level (by modeling how fragments are composed). Clone detection is one important application of this framework.

2. BACKGROUND AND RELATED WORK

A **code fragment** (or **fragment**) is a contiguous segment of source code, specified by the source file and the lines where the segment begins and ends [30]. **Code clones** (or **clones**) are two or more fragments that are *similar* with respect to a **clone type** [30]. A **candidate** is a clone pair reported by a clone detector [31]. We introduce learning-based detection techniques where everything for representing terms and fragments is mined from the source code repository. *Indeed, our approach aims to move clone detection from the art of feature engineering to the science of automated discovery.* To substantiate our progress against this goal, consider the following difference. The related work (Sec. 2.1) uses handcrafted feature vectors to represent fragments. In our work (Sec. 3), this handcrafting is supplanted by methods for automatically discovering empirically-based features. This supplantation is evidenced by the fact that feature vectors in traditional approaches generally lend themselves to interpretation. A feature may correspond to the occurrence count of a programming construct in a tree-based technique or a measure of central tendency in a metrics-based technique, etc. Alternatively, our feature vectors do *not* lend themselves to interpretation. Why? We use a special type of machine learning (Sec. 2.3) that shifts our clone detection approach from an imperative style *Here is how I want to represent fragments* to a declarative style *Here is what I want to represent.* Hence, our work does not replace existing techniques but rather provides a completely new perspective.

2.1 Code Clone Detection

Generally, there are four clone types. **Type I:** Identical fragments except for variations in comments and layout [10]. **Type II:** Identical fragments except for variations in identifier names and literal values in addition to Type I differences [10]. **Type III:** Syntactically similar fragments that differ at the statement level. The fragments have statements added, modified, or removed with respect to each other, in addition to Type II differences [10]. **Type IV:** Syntactically dissimilar fragments that implement the same functionality [10]. Type I, II, and III clones indicate textual similarity whereas Type IV clones indicate functional similarity.

Recall that detection techniques generally begin by representing code before measuring similarity, and these techniques can be classified by their source code representation. Text-based techniques [32–35] apply slight transformations to code and measure similarity by comparing sequences of text. Consequently, text-based techniques are limited in their ability to recognize two fragments as a clone pair even if the difference between them is as inconsequential as a systematic renaming of identifiers.

Token-based techniques [5, 15, 36–38] mollify the scrupulous text-based rule by operating at a higher level of abstraction. These techniques lexically analyze the code to produce a stream of tokens and compare subsequences to detect clones. Matching subsequences of tokens generally improves the recognition power, but the token abstraction has a tendency to admit more false positives [23]. Our learning-based approach differs from token-based techniques in at least two ways. First, the token abstraction maps each term to a (discrete) class, which effectively *bins* the terms, whereas our approach maps terms to continuous-valued vectors in a feature space where similarities are encoded as distances. Second, our approach incorporates context (e.g., syntax) beyond the token abstraction as tree-based techniques do.

Tree-based techniques [3, 39–41] measure the similarity of subtrees in syntactic representations. Our primary related work is the influential work by Jiang et al. [41] who presented Deckard, which transforms parse trees into "characteristic vectors" and clusters similar vectors (using Locality Sensitive Hashing [42]) to detect clones. We use abstract syntax trees (ASTs) rather than parse trees, and while Deckard distinguishes between "relevant" and "irrelevant" nodes, we regard every nonempty node in an AST as relevant. *Designating a subset of nodes as relevant amounts to handcrafting an abstraction for fragments.* Each component of a characteristic vector represents the occurrence count of relevant nodes in the corresponding subtree, so the vector's dimension is the number of tree patterns deemed relevant to approximate a given tree [43]. This feature engineering represents a fundamental point of divergence in our work where we learn discriminating features from the data as opposed to declare a priori a number of specific features. Moreover, characteristic vectors approximate structural information while neglecting domain information rooted in identifiers [22]. In fact, there is generally no special treatment for identifiers and literal types in AST-based approaches [10]. Our work operates on identifiers and literal types.

Graph-based techniques [43–47] use static program analysis to transform code into a program dependence graph (PDG), an intermediate representation of data and control dependencies [48]. Gabel et al. [43] augmented Deckard with semantic information derived from PDGs; they mapped subgraphs to related structured syntax (defining significant nodes to be those that descend from the parent statement class) and then detected clones using Deckard. Chen et al. [47] used a "geometry characteristic" of dependency graphs to measure methods' similarities before combining method-level similarities to detect application clones in Android markets. They began by extracting methods from Android application packages and transforming each method to a control flow graph (CFG). They compute a "centroid" for each method by mapping every CFG node to a three-dimensional point where the dimensions represent structures. Our work uses more resolution by operating on AST nodes rather than basic blocks. They only use the CFG part of the PDG whereas our approach is designed to learn models of how terms are composed at any level of granularity. By mapping methods in a three-dimensional space of engineered features, Chen et al. place an extraordinarily strong prior on the source code representation [47]. They imply the definition of the centroid can be extended so the centroid can be impacted by the `invoke` statement, but this augmentation constitutes more feature engineering designed to improve the performance for the specific application of Android app clone detection. Our unique approach obviates the need to engineer this kind of feature, since handcrafting is time-consuming and limited [49].

Other detection approaches include the following. Davey et al. [11] ignored identifiers and operators and instead considered the frequency of keywords, indentation pattern, and length of each line to represent fragments. These feature vectors were passed to a self-organizing map to detect clones. Marcus and Maletic [25] examined identifiers and comments to identify implementations of similar high-level concepts. Nguyen et al. [50–52] extracted structural features from generic, graph-based representations to build characteristic vectors. Lee et al. [53] measured structural similarities like Deckard and proposed a multidimensional indexing structure to support fast inference. Kim et al. [54] proposed a semantic detection technique that compared programs' abstract memory states. Hermans et al. [55] proposed a text-based algorithm for detecting clones in spreadsheets. Finally, Jiang and Su [56] presented EqMiner, a novel approach to identifying functionally equivalent fragments. EqMiner runs fragments with random inputs and defines functional equivalence in terms of I/O behavior. Our work aims to detect both textual and functional similarity at compile time.

2.2 Language Modeling

Our approach is based in part on language models. A statistical **language model** is a probability distribution over sentences in a language [57]. Traditionally, statistical language models have been effective abstractions for natural language processing (NLP) tasks. Recently, their effectiveness has suffused SE tasks such as code suggestion [58–60], deriving readable string test inputs to reduce human oracle cost [61], predicting comments to improve search over code bases and code categorization [62], improving error reporting [63], generating feasible test cases to improve coverage [64], improving stylistic consistency to aid readability and maintainability [65], code migration [66–68], synthesizing API completions [69], code review [70], fault localization [71], and suggesting method and class names [72].

A statistical language model is a tractable representation of sentences (e.g., lines of code or traces of method invocations) in a language. Tractability is realized by decomposing a joint distribution and analyzing probabilistic automata, e.g., n-grams: $p(s) = \prod_{i=1}^{m} p(w_i|w_1^{i-1}) \approx \prod_{i=1}^{m} p(w_i|w_{i-n+1}^{i-1})$. However, n-grams suffer apparent limitations [73, 74]. For example, since they are simply smoothed counts of term co-occurrences, they are limited in their ability to generalize beyond the explicit features observed in training [29, 75–77]. They are also limited by the amount of context they consider [75], yet the novel approach of casting applications in terms of "naturalness" in SE contexts has spawned many applications of this technology to new problems. We enlist a particular type of language model to map the terms in source code to continuous-valued vectors called *embeddings* (Sec. 3.1). *To the best of our knowledge, we are the first to propose language models and embeddings for clone detection.*

2.3 Deep Learning

Compositional learning algorithms typify **deep learning** [29, 78], a nascent field of machine learning. These state-of-the-art learning algorithms have seeded new approaches in computer vision and NLP. NLP in particular has realized substantial improvements applying deep learners, e.g., the recurrent neural network (RtNN) [79–83], to corpora. Our prior work considered the limitations of n-grams and aimed to improve the representation power of the abstractions (e.g., software language models) we use in SE research by examining RtNNs on software corpora for a code suggestion engine [84]. While RtNNs are powerful architectures for modeling sequences of terms, their generalization—the *recursive* neural network (RvNN) [85]—is capable of modeling arbitrary structures to, for instance, predict the sentiment of natural language sentences [86, 87]. In our work, we cast clone detection as a recursive learning procedure designed to *adequately represent fragments that serve as constituents* of higher-order components. Of course, recursive learning is inherently compositional, which gives way ipso facto to deep learning. Hence, the purpose of deep learning in this new application is to synchronize the source code representation that we use in the clone detection process with the manner in which the code is conceptually organized. *To the best of our knowledge, we are the first to propose a deep, compositional, learning-based detection approach capable of inducing representations at different levels of granularity.*

3. DEEP LEARNING CODE FRAGMENTS

In this section, we specify our learning-based approach (Fig. 1) in two parts. The first part (Sec. 3.1) describes how we use a particular type of language model, an RtNN, to map each term in a fragment to an embedding. We rely on related work from the NLP community [79–83] and SE lit-

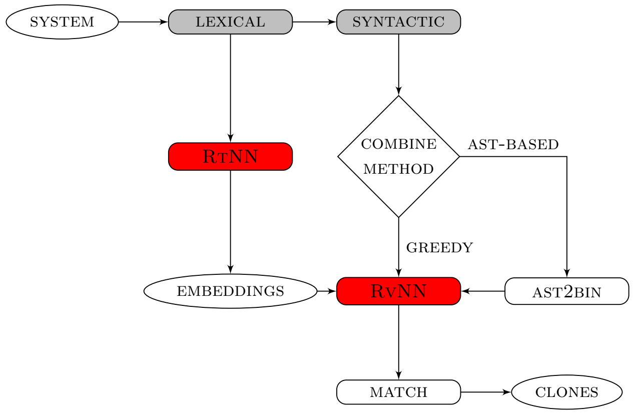

Figure 1: Coupling deep learners to front-end stages

erature [84] for some of the technical details behind training and evaluating these models. The second part (Sec. 3.2) describes how we use the language's grammar and a recursive learning procedure, implemented as an RvNN, to encode arbitrarily long sequences of embeddings to characterize fragments. We are not going to specify the features for modeling these fragments at different levels of granularity; *the purpose of using deep learning is to automate this manual step.*

3.1 Deep Learning Code: Lexical Level

An RtNN (Fig. 2) is a deep learner that is well suited for modeling sequences of terms in a source code corpus with vocabulary $\mathcal{V}$ where $|\mathcal{V}| = m$ terms. Let n, a user-specified hyperparameter [78], be the number of hidden units. An RtNN comprises an input layer $x \in \mathbb{R}^{m+n}$, a hidden layer $z \in \mathbb{R}^n$, and an output layer $y \in \mathbb{R}^m$ (assuming away heuristics such as class-based output layers [76, 83, 88–90]). Adjusting n regulates the model's capacity [28, 91]. The depth of an RtNN is attributed to the recurrence [29, 92–95] where the hidden state is copied back to the input layer, so the input layer in an RtNN agglutinates the current term $t(i)$ and the previous state $z(i-1)$:

$$x(i) = [t(i); z(i-1)] \tag{1}$$

This input vector is multiplied by a matrix $[\alpha, \beta] \in \mathbb{R}^{n \times (m+n)}$ and passed to a nonlinear vector function f, i.e.,

$$z(i) = f(\alpha t(i) + \beta z(i-1)) \tag{2}$$

This state vector is multiplied by another matrix $\gamma \in \mathbb{R}^{m \times n}$ and normalized to compute a posterior over terms,

$$y(i) = p(t|x(i)) = \text{softmax}(\gamma z(i)) \tag{3}$$

Eq. (1)–(3) specify an RtNN. Eq. (4) highlights its depth by making its composition a bit more explicit to show how its $\underline{\text{output}}$ is a highly nonlinear function of its previous $\underline{\underline{\text{inputs}}}$:

$$\underline{y(i)} = \text{softmax}(\gamma f(\alpha \underline{\underline{t(i)}} + \beta f(\alpha \underline{\underline{t(i-1)}} + \beta(\underline{\underline{\cdots}})))) \tag{4}$$

The model $\theta = \{\alpha, \beta, \gamma\}$ is trained using a cross entropy criterion [96] but we omit the technical details here [83, 97]. In software language modeling, the model's output $y(i)$ can be used to predict the next term in a line of code as $\text{argmax}_k\, y_k(i)$ [84]. However, deep learners are not simply useful for their output; their internal components are useful too. In this work, the most important component of RtNN-based software language models is the matrix of embeddings $\alpha \in \mathbb{R}^{n \times m}$ in Eq. (2). Each column of α corresponds to a term. The column space of α comprises semantic representations [29, 72, 84] for every term in $\mathcal{V}$ such that the model imputes similar vectors to terms used in similar ways in the corpus [29]. Given that each term is one-hot encoded when presented to the model, the matrix-vector product αt in Eq. (2) amounts to mapping any term in $\mathcal{V}$ to a column in α thereby mapping sequences of terms in fragments to sequences of embeddings. Thus, to represent fragments, we encode arbitrarily long sequences of embeddings.

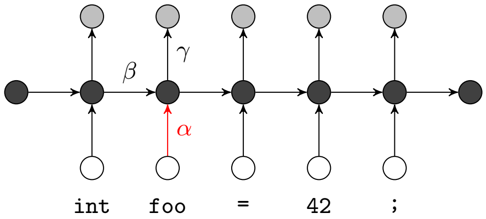

Figure 2: RtNN. White nodes are one-hot term vectors; black nodes are continuous-valued states; gray nodes are posterior distributions. We extract the matrix of embeddings represented by the red arc.

3.2 Deep Learning Code: Syntactic Level

Our learning-based archetype diverges from traditional techniques. Given a fragment, information will flow up from the terminal nodes through the nonterminal nodes to the root of a hierarchical structure (Fig. 3–4). This bottom-up flow of information is like the procedures for computing characteristic vectors in traditional structure-oriented techniques or computing metrics in metrics-based techniques. However, we mine vector representations for terminal nodes (Sec. 3.1), and the features for nonterminal nodes are not indicator-based occurrence counts (Sec. 2.1). The feature space is induced by learning to discriminate fragments (Sec. 3.2.2). Furthermore, after information is synthesized in a bottom-up traversal to compute characteristic vectors or metrics, traditional techniques terminate and pass the source code representations to a match detection algorithm to find similar fragments. *In a way, we regard the bottom-up flow of information as necessary—but not sufficient—to adequately represent fragments.* Hence, our termination condition is fundamentally different. In our approach, the procedure for mining representations terminates when the model has converged to a solution such that it can adequately represent programming constructs at different levels of granularity (Eq. (5)–(7)). This criterion where information at the lexical level is transmitted from terminals to a structure's root and a supervised signal is broadcasted from the root back through the structure [85] lies at the heart of our approach.

3.2.1 *From ASTs to Full Binary Trees*

The front end of a compiler decomposes a program into *constituents* and produces intermediate code according to the syntax of the language [98]. These constituents are called programming constructs, and a context-free grammar specifies the syntax of programming constructs [98]. The AST is one type of intermediate code that represents the hierarchical syntactic structure of a program [98]. Ultimately, our goal is to specify learning-based techniques for encoding arbitrarily long sequences of lexical elements. Since the nonterminal nodes in ASTs subsume sequences of lexical elements [98], suppose each AST node has a special attribute *repr* that stores a vector representation, a code[1] that characterizes the node and, by extension, the sequence of lexical elements the node subsumes. We mine the codes in such a way that similar sequences have similar codes. One learning-based technique is based on the AST, a tree representation that can have an arbitrary number of levels comprising nodes with an arbitrary number of children, but herein lies the problem. Our learner only accepts fixed-size inputs (Sec. 3.2.2), so we transform the AST to a full binary tree to fix the size of the input, and we apply the learner recursively to model the structure at different levels.

The degree [99] of an AST node is either zero, one, two, or greater than two. By definition, AST nodes with degree zero or two satisfy the property of nodes in a full binary tree [99], but subtrees rooted at nodes with degree one (Case I) or greater than two (Case II) must be transformed in order to refashion the local subtree into a full binary tree. The first step of our transformation is to scan the AST and delete metadata (e.g., Javadoc nodes in ASTs for Java fragments) as well as nodes for empty anonymous class declarations, empty array initializers, empty blocks, empty classes, empty compilation units, and empty statements. As we scan the AST for empty nodes, we also look for sequences of identical literal types with the same parent. The learner will encode pairwise combinations of AST nodes; therefore, we avoid encoding pairs of the same literal type by visiting nonterminal nodes, inspecting their children, and collapsing adjacent, identical literal types to one instance. For example, `true true`, `true true true`, etc. all become `true`. Collapsing these sequences also helps control the depth of the binary tree at the risk of losing some resolution.

Next, to obtain a binary tree, subtrees rooted at Case II nodes (i.e., nodes with degree greater than two) need to be reorganized so the children are suitably arranged. We defined a grammar-based approach, for each nonterminal type, to systematically reorganize the children of Case II nodes. For example, IfStatement instances can have either two or three children. For this nonterminal type, we defined a new grammar that only produces binary subtrees (assuming away the syntax of Expression and Statement nodes) since every production body has either one or two constructs. To do so, we augmented the language's grammar by introducing new artificial nonterminal types such as Branches:

$$\langle \mathit{IfStatement} \rangle ::= \langle \mathit{Expression} \rangle \; \langle \mathit{Branches} \rangle$$

$$\langle \mathit{Branches} \rangle ::= \langle \mathit{Statement} \rangle \; [\; \langle \mathit{Statement} \rangle \;]$$

For nonterminal types with arbitrary maximum degree (e.g., Block nodes) we organized their children into binary lists. Since the children of Block nodes are represented by a sequence of statements, we replaced the original production

$$\langle \mathit{Block} \rangle ::= \{\; \langle \mathit{Statement} \rangle \;\}$$

with a new production where Block nodes can have the form of a recursive list of statements:

$$\langle \mathit{Block} \rangle ::= \langle \mathit{StatementList} \rangle$$

$$\langle \mathit{StatementList} \rangle ::= \langle \mathit{Statement} \rangle \; [\; \langle \mathit{StatementList} \rangle \;]$$

After we transform each Case II instance using the new grammar, we obtain a binary tree from the original AST, but the binary tree may or may not be a *full* binary tree since nodes may have one and only one child. In other words, we need to handle Case I nodes (i.e., nodes with degree one). We traverse the binary tree in a top-down manner, and when we reach a Case I node, we merge the node and its child into one node. Then we recursively continue the transit from the new merged node. The top-down visit ensures that instances of parent nodes with one and only one child are eventually merged into one node. Our merging procedure is governed by a precedence list that assigns a value to each nonterminal type. When merging two nodes, the precedence value is used to decide whether to assign the current node type or the child type to the new node. Tab. 1 shows the precedence list we defined where types higher in the list have higher precedence. When two nodes have the same precedence value—which may be the case with two *OtherType* nodes—the merge keeps the parent node. This design decision comes from the observation that the parent node is typically more expressive and representative of the programming construct than the child node. We determined the list upon several empirical observations. In particular, with this order, we ensure the following.

Table 1: Precedence

TypeDeclaration
MethodDeclaration
OtherType
ExpressionStatement
QualifiedName
SimpleType
SimpleName
ParenthesizedExpression
Block
ArtificialType

- Certain levels of granularity are protected and never overwritten by other nodes.
- When merging two nodes, more expressive types are preferred over more general types such as ParenthesizedExpression and Block.
- Artificial nonterminal nodes, created in the previous step to handle Case II nodes, will never replace nonterminal types in the original grammar.

The implications for protecting certain levels of granularity are apparent in SE applications such as clone detection where (for example) our approach is capable of representing and thereby reporting clones at well-defined abstraction boundaries to better support software maintainers.

3.2.2 From Full Binary Trees to Olive Trees

Now we describe how we transform a full binary tree to what we informally call an *olive tree*, which is the result of converting intermediate code to a full binary tree and then annotating this tree with mined representations. Consider the statement `int foo = 42;`. The AST for this statement is already a full binary tree depicted in Fig. 3 (1)–(5). Suppose again that each AST node has a special attribute repr, e.g., 2.repr stores the representation for the SimpleName (2) in Fig. 3. We initialize this attribute for each terminal by using its lexical element to select the corresponding column in the matrix of embeddings α (Fig. 2). For example, if the lexical element `int` maps to the jth column of α, then repr for the PrimitiveType (1) in Fig. 3 is initialized such that 1.repr $= \alpha_{\cdot j}$. This attribute is initialized to null for nonterminal nodes such as the VariableDeclarationFragment (4) and the VariableDeclarationStatement (5) in Fig. 3. At this juncture, we have used patterns mined at the lexical level (Sec. 3.1) to initialize a sequence of embeddings. Next, we use an autoencoder to combine embeddings. The canonical

[1] We use *code* to refer to source code, intermediate code, and representations. The context will always disambiguate the term.

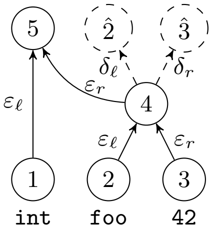

Figure 3: AST-based

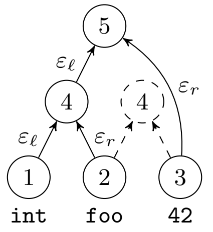

Figure 4: Greedy

form of an autoencoder is a neural network with one input layer x, one hidden layer z, and one output layer y

$$z = g\left(\varepsilon x + \beta_z\right) \quad (5)$$

$$y = h\left(\delta z + \beta_y\right) \quad (6)$$

where $\varepsilon = [\varepsilon_\ell, \varepsilon_r] \in \mathbb{R}^{n\times 2n}$ is the εncoder; $\delta = [\delta_\ell; \delta_r] \in \mathbb{R}^{2n\times n}$ is the δecoder; $\beta_z \in \mathbb{R}^n$ and $\beta_y \in \mathbb{R}^{2n}$ are βiases. The tie that binds patterns mined at the lexical level with patterns mined at the syntactic level is n, which is the same n that governed the size of the hidden layer z in Eq. (2). g is a nonlinear vector function, and h is typically the identity function. In Sec. 3.2.1, we claimed that our learner only accepts fixed-size inputs, which prompted the transformation of ASTs to full binary trees. Concretely, the input to the autoencoder is a vector of two sibling nodes' codes, i.e., $x = [x_\ell; x_r] \in \mathbb{R}^{2n}$. For example, to compute the representation for the VariableDeclarationFragment (4) in Fig. 3, we would present $x = [2.\text{repr}; 3.\text{repr}]$ to the model. Constricting the size of the hidden layer (i.e., $|z| = n < 2n$) coerces the model into learning a compressed representation of its input. This compression, z in Eq. (5), serves as the mined representation that we store in the nonterminal node's repr attribute. Essentially, the model embeds the input in a lower-dimensional feature space just as the language model embedded one-hot term vectors (Sec. 3.1). In other words, the language model transforms lexical elements to embeddings, and the autoencoder compresses any two embeddings to a vector with the same dimensions as a term embedding. The output $y = [\hat{x}_\ell; \hat{x}_r] \in \mathbb{R}^{2n}$ is referred to as the model's *reconstruction* of the input. Training the model involves measuring the distance between the original input vector and the reconstruction:

$$E(x_\ell, x_r; \varepsilon, \delta, \beta_z, \beta_y) = ||x_\ell - \hat{x}_\ell||_2^2 + ||x_r - \hat{x}_r||_2^2 \quad (7)$$

If the model can effectively learn discriminating features of the input, then it will be able to generalize and faithfully reconstruct *any* input vector sampled from the domain.

We just demonstrated how conventional autoencoders can compress modest sequences of two lexical elements, but to support clone detection, we learn codes for *much more*. Since the code for every node in the tree has the same size, we can apply the autoencoder recursively, an RvNN, to model the full binary tree at different levels. The autoencoder that we used to compress the SimpleName (2) and NumberLiteral (3) in Fig. 3 can be applied recursively insofar as the code for the VariableDeclarationFragment (4) is coalesced with the code for the PrimitiveType (1) and presented to the same model to compute the code for the VariableDeclarationStatement (5): $5.\text{repr} = g([\varepsilon_\ell, \varepsilon_r][1.\text{repr}; 4.\text{repr}] + \beta_z)$. As before, to train the model, we decode the representation (i.e., $y = h([\delta_\ell; \delta_r][5.\text{repr}] + \beta_y)$) and compare the reconstruction to the input (i.e., $x = [1.\text{repr}; 4.\text{repr}]$) to adjust the weights. But now the error is a (weighted) sum of all reconstruction errors where larger programming constructs will have more influence on shaping the representation for the fragment. For example, the VariableDeclarationFragment (4) has a greater influence on tuning 5.repr than the PrimitiveType (1). After computing the code for each nonterminal node in a forward pass, the backpropagation through structure algorithm [85] computes partial derivatives of the (global) error function with respect to the model's components. Then the error signal is optimized using standard methods. Once the deep learner has converged after a number of epochs, we inlay the full binary tree with the representations to produce an olive tree.

Why is deep learning a good approach for clone detection? Techniques that analyze identifiers generally use Latent Semantic Analysis (LSA) [100]. Deep learning has three apparent advantages over LSA. First, autoencoders are *nonlinear* dimensionality reducers. Second, recursively applying an autoencoder operates on input with *several* nonlinear transformations as opposed to using one linear decomposition of the input. Third, the recursion considers the *order* of terms. On the other hand, techniques that analyze structure discard identifiers, which we use as prior knowledge. Rather than use generic structural elements, our learning framework bases its representation on the discriminative power of identifiers and literal types, so even when the syntax is only weakly similar, deep learning can still recognize similarities among terms.

Socher et al. [86] applied recursive autoencoders to natural language sentences for sentiment analysis. The novelty in Socher's work was the semi-supervised augmentation designed to train the model to classify the sentiment of sentences using sentence-level labels. We use recursive autoencoders to learn representations, instantiated as syntactic-level attributes, of arbitrary sized code fragments. One final remark on the nature of the attributes that we use: in compiler parlance, an attribute (i.e., a quantity associated with a programming construct) is said to be "synthesized" or "inherited" [98], but the attribute we mine in this work is technically *neither*. A synthesized attribute for a node is computed from the attribute values for the node and the node's children whereas an inherited attribute is computed from the node, its parent, and its siblings [98]. However, in our work, attributes are synthesized in a bottom-up traversal, but then the training algorithm will adjust the attributes in a top-down manner as the errors for general programming constructs are divvied up among their *constituents*.

3.2.3 *Olive Trees for Clone Detection*

Once the model is trained, inference is straightforward. Recognizing a clone pair amounts to comparing the representations for two fragments, which can be at different levels of granularity. Specifically, given a fragment, we build the AST and then transform the AST to a full binary tree. If there are k terminal nodes in the full binary tree, then there will be $k - 1$ nonterminal nodes. As a result, encoding the sequence requires $k - 1$ matrix-vector multiplications each followed by the application of a vector function to derive the representation for the fragment. Naturally, the topology of the full binary tree governs the order in which the nodes' representations are combined. For the specific application of code clone detection, all that is required is a threshold for comparing two representations to determine whether their propinquity classifies them as a clone pair; the threshold completes the clone detection specification.

3.2.4 *Greedy Combinations for Clone Detection*

Here we draw from an approach proposed by Socher et al. [86] for combining pairwise representations in a greedy manner. First, we summarize the training procedure. For each fragment, we build the AST, but rather than transform the AST as before, we encode each pair of adjacent terminal nodes. Then we select the pair with the lowest reconstruction error (Eq. (7)) to encode first. For example, in Fig. 4, the first iteration derives two codes; the model does a better job at reconstructing [1.repr; 2.repr] rather than the VariableDeclarationFragment [2.repr; 3.repr]. The next iteration substitutes the chosen pair with their new parent and then computes the pairwise reconstruction errors again, selecting the pair with the minimum error. If there are k terminal nodes covering the fragment, then this procedure repeats until a representation has been computed for $k-1$ nonterminal nodes. Once the ad hoc tree is in place, the model is trained as before with the backpropagation through structure algorithm and a standard optimization method.

Once the model is trained, inference again is straightforward. Given a fragment, we build the AST and then greedily encode nodes until deriving a code for a node that subsumes the fragment. This code is compared to other greedily encoded fragments using a threshold to detect code clones. One important note on the training and inference procedures for greedily encoding nodes is that we do not *need* to build the AST. In fact, since the language model stores an embedding for every term in the corpus, we can operate directly on the concrete fragment. The reason we build the AST is to filter lexical elements such as punctuation to control the depth of the tree that we use for training and inference.

There are some remarkable differences between the two combining methods. First, for the AST-based method, the clone granularity is *generally* "fixed," i.e., it combines fragments within syntactic boundaries [23]. On the other hand, for the greedy method, the clone granularity is *generally* "free," i.e., it combines fragments without syntactic boundaries [23]. Second, training requires more computational resources for the greedy method than the AST-based method. The AST-based method has $k-1$ matrix-vector products to compute whereas the greedy method has $k-1$ (generally) dense matrix-matrix products to compute. Third, since the greedy method is trained without *explicit* knowledge of the syntax, it does not need to build the AST, so the model may better handle syntactically invalid fragments. Despite the differences, the methods together reify a new, learning-based paradigm for code clone detection.

4. EMPIRICAL VALIDATION

The **goal** of our empirical study was to **analyze** our source code representations for the **purpose** of evaluating them for code clone detection with **respect** to *feasibility* from the **point of view** of software maintainers in the **context** of Ph.D. students and real-world Java systems [101]. Our intent for establishing feasibility as the quality focus was twofold. First, we are not only presenting an innovative approach to transforming source code but also introducing the idea of framing clone detection as a *robust* learning problem. Hence, we seek to provide some understanding of the practical relevance of this new perspective. Second, given a new approach to clone detection, the evaluation in and of itself is a formidable task beset by undecidable problems and variable human judgment [30,102–108]. Roy et al. [23] highlight a number of factors that make evaluating and comparing detection tools challenging, including—but not limited to—the diverse nature of detection techniques, the lack of standard similarity definitions, the absence of benchmarks, the diversity of target languages, and the sensitivity of tuning parameters. Further, many clone detection tools are not available. Indeed, the community's knowledge of code clone detection tools' performances on real-world systems is limited [30]. In this respect, our experimental design, analysis, and reporting are consistent with current studies in the field. We discuss limitations of our empirical study in Sec. 4.2 and consolidate threats to the validity of our work in Sec. 6.

Table 2: Subject Systems' Statistics

System	Files	LOC	Tokens	$\lvert\mathcal{V}\rvert$
ANTLR 4	514	104,225	701,807	5,826
Apache Ant 1.9.6	1,218	136,352	888,424	16,029
ArgoUML 0.34	1,908	177,493	1,172,058	17,205
CAROL 2.0.5	184	12,022	80,947	2,210
dnsjava 2.0.0	196	24,660	169,219	3,012
Hibernate 2	555	51,499	365,256	5,850
JDK 1.4.2	4,129	562,120	3,512,807	45,107
JHotDraw 6	984	58,130	377,652	4,803

Notwithstanding the challenges, we aimed to determine whether the idea of learning representations for fragments can be *relevant* for clone detection and a tenable technique for researchers. We examined the following questions.

RQ1 Are our representations suitable for detecting fragments that are similar with respect to a clone type?

RQ2 Is there evidence that our compositional, learning-based approach is capable of recognizing clones that are undetected or suboptimally reported by a traditional, structure-oriented technique?

Considering our goal and questions, we intended to estimate the precision of our approach at different levels of granularity to answer RQ1 and to synthesize qualitative data on code clones across two detection techniques for RQ2. Judging code clones is inherently difficult (even among experts [102,104,107]) because of imperfect definitions [10,106] and the lack of oracles [107], so we developed a research instrument [109] to support consistent evaluations and control construct threats. We describe the guidelines used to manually examine candidates in Sec. 4.2.

4.1 Data Collection Procedure

Our subject systems included eight real-world Java systems (Tab. 2) used in previous studies [10]. We used ANTLR to tokenize the source code and the RNNLM Toolkit [80] to train several RtNNs for each system, varying hidden layer sizes and depths [84]. We selected the highest quality model for each system, using perplexity [57] as a proxy for quality, and extracted the matrix of embeddings (Fig. 2). Researchers have not established a correlation between intrinsic evaluation [57] metrics such as perplexity and the quality of model components like the matrix of embeddings. However, anecdotally, we have observed interesting patterns in good models induced from Java corpora where embeddings for similar terms are collocated in feature space. For each system except CAROL, we used a hidden layer size of 500, i.e., $z \in \mathbb{R}^{500}$ in Eq. (2). For CAROL, our simplest system in terms of tokens and vocabulary size, we used 400.

Next, we used the Eclipse Java development tools to build the AST for each file in every system. Each AST node represents a programming construct, and we relied on the visitor design pattern to traverse ASTs, identify nodes' types, and implement ast2bin (Sec. 3.2.1). Empirically, we found 25 different programming constructs that have at least one Case II instance, so we implemented productions (using 30 different artificial types) to handle each construct and verified that our ast2bin procedure transformed the 9,688 ASTs across our eight systems to full binary trees. The roots in all but 17 of these trees were CompilationUnit nodes. The others were rooted at TypeDeclaration nodes. To generate method-level corpora, we used a MethodVisitor, collecting methods with 10–50 LOC. We only considered methods with no more than 50 LOC to focus the method-level evaluation on small code fragments and complement the coarse, file-level evaluation.

Given the embeddings, we induced an ad hoc, annotated, full binary tree for each file using the greedy method. Then we used the embeddings and the AST-based full binary trees to induce an olive tree for each file. Our experimental design planned to compare results from our approach to the state-of-the-practice, so we ran Deckard on our systems. To configure Deckard, we used the settings proposed by Jiang et al. [17], setting minT to 50, stride to ∞, and similarity to 1.0, which correspond to standard choices in other tools.

4.2 Analysis Procedure

RQ1. After running the AST-based and greedy methods, the next step in the clone detection process [10,23] (and the first step in our analysis procedure) was to select a similarity metric and threshold. We selected the ℓ_2 norm to measure the similarity of fragments' codes. For the AST-based method, we used the same file-level threshold 1.0E-5 for each system. For the greedy method, the distances were dispersed across several orders of magnitude, so we selected file-level thresholds such that the number of candidates was approximately equal to the number proposed by the AST-based method for each project. Likewise, we used general thresholds for methods. Our selections were not optimized—in accordance with our goal of *evaluating feasibility* rather than *improving effectiveness*. In other words, we are studying the feasibility of a new, learning-based paradigm for code clone detection; improving the effectiveness of learning-based techniques by tuning project-dependent *hyper*parameters such as the size of the embeddings or the threshold for classification constitutes a different problem.

Given the lack of oracles for our systems, we set out to manually examine random samples of candidates. To provide a reasonable scope for the manual evaluation, we settled on assaying file- and method-level candidates using two-author agreement. Two Ph.D. students evaluated file- and method-level samples for each combining method and every system. If our approach performed well on several hundred oracled pairs at multiple levels of granularity, then it is sensible to conclude that our source code representations are suitable for clone detection. To support consistent evaluations, we adapted the taxonomy of editing scenarios designed by Roy et al. [23] to model clone creation and be general enough to apply to any level of granularity. In our scenario-based evaluation, both participants were presented with samples and instructed to compare them systematically—i.e., top-down from Scenario I to Scenario IV where clones created by the scenarios correspond to one of the four clone types—to assess each sample as a true positive or false positive. After independently evaluating the samples, authors' disagreements were discussed and resolved.

In addition to providing a reasonable scope for the manual evaluation, another reason why we examined file-level samples is we expected the coarse granularity (a mixture of compilation units and types) to be harder for our recursive learning procedure, which amounts to applying the chain rule for partial derivatives. Larger fragments yield deeper trees, but training deep architectures is notoriously difficult [28, 78, 91–93]. Consequently, if the RvNN is capable of producing good results at coarse granularity, then it is reasonable to expect its representations at lower levels of granularity are effective, and we substantiate this claim with our method-level evaluation. Moreover, empirical studies [31, 110, 111] have underscored several practical uses for file-level clone detection to include, inter alia, detecting similar projects and measuring third-party library reuse. Sec. 5 reports estimates of the precision of our approach.

Measuring recall is a common limitation to many clone detection studies. We considered using a synthetic clone benchmark, but our approach is based on learning from how terms are used in a corpus. By using a mutation-analysis procedure, we would increase our control over estimating recall, but we would reduce the degree of realism, which risks setting real influential factors (e.g., patterns mined at the lexical level) outside the scope of the study [109].

RQ2. Our second research question was intended to frame an exploratory study on our results as compared to state-of-the-practice results where differences may admit important practical impacts and theoretical advances. From a software maintainer's point of view, a detection technique that is capable of reporting clones at fixed levels of granularity is useful [23]. For example, given an oracled pair of file clones, it would be ideal for a detection technique to report the files as clones rather than splinter the compilation units and report their constituents as clones. Structure-oriented techniques like Deckard try to account for similar code of any size with ad hoc, user-provided input, e.g., the width of a sliding window [41], but automated support for this practical concern is not designed into the approach as it is in our work. Automatically reporting clones at a fixed level without requiring input from the user (beyond specifying the level) would be a notable strength of our compositional, learning-based paradigm where information is communicated between generalized constructs such as types and specialized constructs such as statements to train the model. To provide a reasonable scope for the exploratory study, we settled on file-level pairs. Sec. 5 synthesizes qualitative data from the study.

5. EMPIRICAL RESULTS

Our RvNN implementation forked Socher et al. [86], which used L-BFGS to optimize costs in batch mode. We trained each model for at least 30 epochs on one compute node serving two Intel Xeon E5-4627 v2 processors at 3.3 GHz. Tab. 3 reports the average training time (in seconds) per epoch. Once a model is trained, inference at any level of granularity amounts to matrix multiplications, so Tab. 3 reports the average time (in seconds) to infer the representation of a file. These results contained outliers, so we also report the median time in parentheses. Sec. 6 summarizes lessons learned from training these models on source code.

RQ1. Sampling candidates for each combining method and system, Tab. 4 reports the ratio of true positives as well as the total number of samples used to build the estimate. Altogether, we sampled and manually evaluated 398 file-level

Table 3: Performance Results

System	Training (sec)		Inference (sec)	
	AST-based	Greedy	AST-based	Greedy
ANTLR	443	3,516	3.21 (1.18)	33.36 (1.96)
Apache Ant	813	3,476	3.31 (1.76)	25.20 (3.10)
ArgoUML	1,018	3,868	2.58 (1.24)	16.35 (1.80)
CAROL	34	116	0.88 (0.48)	4.87 (0.95)
dnsjava	148	1,169	3.63 (2.16)	30.67 (4.30)
Hibernate	277	1,077	2.49 (1.17)	17.70 (1.70)
JDK	2,977	14,965	3.46 (1.19)	35.06 (1.80)
JHotDraw	336	792	1.67 (0.93)	6.40 (1.19)

Table 4: Precision Results

System	File-level		Method-level	
	AST-based	Greedy	AST-based	Greedy
ANTLR	97% (30)	100% (30)	100% (30)	100% (30)
Apache Ant	92% (24)	93% (30)	100% (30)	100% (30)
ArgoUML	90% (30)	100% (30)	100% (30)	100% (30)
CAROL	100% (1)	100% (10)	100% (30)	100% (30)
dnsjava	47% (30)	100% (30)	73% (30)	87% (30)
Hibernate	100% (13)	100% (20)	53% (30)	70% (30)
JDK	90% (30)	100% (30)	100% (30)	100% (30)
JHotDraw	100% (30)	100% (30)	100% (30)	100% (30)

pairs from a pool of 1,573 candidates and 480 method-level pairs from a pool of 60,474 candidates. 93% of the file-level samples were evaluated to be true positives where 16 of the 27 false positives came from one configuration (dnsjava, AST-based). Then we applied the model that was trained on the file corpus to the method corpus. 93% of the method-level samples were evaluated to be true positives. Once more, neither file- nor method-level thresholds were optimized. For systems that had less than or equal to 30 candidates (after applying the generic threshold), we manually evaluated every candidate. For instance, Hibernate (AST-based) only had 13 file-level pairs with distances below the threshold, and all 13 candidates were true positives. For systems that had more than 30 candidates, we sampled 30 of them. In one case (CAROL, AST-based), the threshold on file-level pairs was too strict. Nonetheless, Tab. 4 provides empirical evidence that our learning-based paradigm is feasible for real-world systems. Among the file-level true positives, we found pairs mapping to *all* four clone types: I (43), II (191), III (132), and IV (5). As expected, the distances were near zero for Type I clones, and there was more dispersion for the other types. Four of the five Type IV clones were found by the AST-based method. We placed several examples of true positives and false positives in our online appendix [112].

RQ2. For a traditional, structure-oriented technique, we selected the prominent tool Deckard [41]. For the exploratory study, we queried the file-level true positives and filtered them to remove pairs with at least one file that had less than 50 tokens and to remove Type I and Type II pairs. We focused the exploratory study on how Deckard reported fragments in the remaining pairs, and we found evidence that pairs were either undetected or suboptimally reported.

Undetected. In Hibernate, our approach detected NonUniqueObjectException and WrongClassException, which were evaluated to be Type III clones. Both classes have the same private fields and implement the same methods using similar syntax. Discounting Type I and Type II variations, the few notable differences are reordered data independent statements in the constructors, minor syntactic differences in a getter, and one class overloads its constructor. Deckard did not report any similar fragments for this pair. Another Hibernate pair, NonstrictReadWriteCache and ReadOnlyCache, implement the same interface, but the placement of their methods is noticeably different. Deckard detected the similarity from the package declarations through the field declarations, but these classes share many more points of commonality. In ArgoUML, our approach detected GoNamespaceToDiagram and GoProjectToStateMachine. Both classes extend AbstractPerspectiveRule, which implements PerspectiveRule. Two of the three methods in the interface are Type I clones. For the third method, one class defines an ArrayList wrapped as a List and iterates through a list of Diagrams, *conditionally* adding Diagrams to the List. The other class defines an ArrayList wrapped as a Collection and iterates through a list of Models, adding Models to the Collection where conditional checks appear to be abstracted away. Deckard only reported similarity between the package declarations and import statements. Similarly, in Apache Ant, Deckard detected similarities in the front matter of Difference and Intersect, but the classes have more similarity. Both classes extend BaseResourceCollectionContainer with their main functionality in the method getCollection. The first seven lines of getCollection are Type II clones, but then the classes differ on how they populate the collection. Difference uses a for loop to iterate over a list of ResourceCollections whereas Intersect uses a while loop. Our approach detected the clone pair despite the classes using distinctly different control statements. Finally, our approach detected MINFORecord and SRVRecord in dnsjava. Despite some syntactic differences, there are evident similarities, yet Deckard did not report any similar fragments between these classes. In sum, our approach detected pairs with strong and weak syntactic similarity that were undetected by Deckard; we placed several examples in our online appendix [112].

Suboptimally reported. In JHotDraw, our approach detected two instances of ConnectionTool, which were evaluated to be Type III clones. The two instances share most of their source code (with identical syntax) except for small numbers of additional lines (in some cases one line) in different locations throughout the files. These were larger files, which indicates that our approach is capable of handling gaps throughout a pair of large files and detecting their similarity. Deckard reported nearly 20 clone pairs that covered most of the files; however, from a software maintainer's point of view, this fragmentation makes it difficult to detect these strong Type III file clones.

6. DISCUSSION

Internal validity. We acknowledge the confounding configuration choice problem [107]. We did not adopt arbitrary configurations and tried to *justify* each configuration in our approach. We also tried to justify our Deckard configuration.

External validity. From the point of view of software maintainers, two Ph.D. students conducted the evaluation on eight real-world software systems. Thus, we believe everything to be representative.

Construct validity. We recognize that analytical studies such as our empirical validation cannot adequately evaluate the behavior of the developers while using a tool based on our approach [105]. We do not infer developer behavior from our results and understand that humans must be observed while using the approach [105]. Finally, to mitigate mono-method bias, two judges used a uniform set of guidelines to measure the similarity of code fragments.

Lessons learned. While our results affirm that deep learning is suitable for clone detection, reducing training times is one area that needs more attention. To this end, we identified some corrective action. First, we removed files with

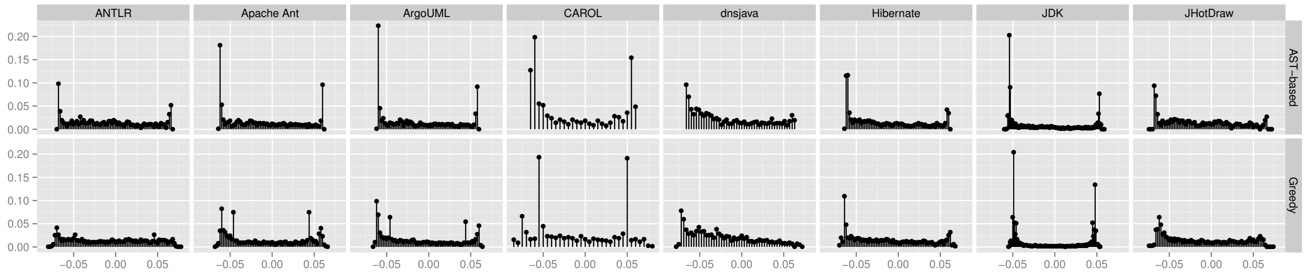

Figure 5: Relative frequency histograms of file-level features

more than 4,000 lexical elements from the training set, but we should have been more aggressive with this cutoff. Extremely large files significantly bogged down training times, and they may not be effective examples for the recursive learning algorithm. Second, we should have sorted files by their size before feeding the training set to the learning algorithm to improve worker utilization. Since we optimize the objective in batch mode, the order used to process examples is inconsequential to learning, yet the order can significantly impact times when training for several epochs. Third, training RvNNs is embarrassingly parallel, so we are modifying the implementation to run on a cluster of compute nodes.

7. FUTURE WORK

On scaling deep learning for clone detection. Here, we draw from work in the machine learning community on semantic hashing [113] and show how a seemingly innocuous machine learning detail in a deep learner can have important practical impacts in SE. In Sec. 3.2.2, we casually described g (Eq. (5)) as a nonlinear vector function. g is called an "activation" function [96], and there are a number of activation functions used in practice, e.g., $g := \tanh$. Models are initialized with small random weights, which implies the "pre-activation," e.g., $\varepsilon x + \beta_z$ in Eq. (5), would lie in the (approximately) linear part of tanh [114]. As the model trains, weights increase, drawing pre-activations away from zero and introducing nonlinearities [114]. When using tanh activations, weights may be directed positively or negatively away from zero. For instance, we initialized our RvNNs by sampling from (approximately) unif$(-0.08, 0.08)$ and used tanh activations. After the models were trained, we encoded each file in every system. Fig. 5 shows the relative frequency histograms of features; we used weight decay [114] for regularization. Distributions across the 16 configurations reveal an interesting bimodal structure. Suppose we select some $\lambda \in \mathbb{R}$ so features greater than λ map to 1 and features less than or equal to λ map to 0. λ transforms continuous-valued feature vectors into binary codes, allowing us to measure the similarity of fragments in a different metric space. Thus, given a fragment, clones can be detected by looking in small Hamming balls around the fragment for other fragments in the repository, a computation that can be optimized by fast algorithms on modern computer architectures [113, 115]. Not only would the binary codes enable fast search because measuring similarity amounts to finding fragments that only differ by a few bits but they would also require less memory [113, 115]—a key concern for massive repositories. While conducting our empirical study, we noticed a significant amount of Type I and II cloning in JHotDraw, so we transformed JHotDraw's (greedy) file-level feature vectors using $\lambda = 0$. 14 of 30 (47%) samples were evaluated to be true positives (all Type III clones), which was noticeably worse than measuring similarity with ℓ_2. To tune models for binary feature vectors, we plan to experiment with different learning heuristics. Salakhutdinov and Hinton [113] reported that semantic hashing (with their generative-based approach) was much faster than LSH in their experiments hashing natural language documents.

Type prediction. Fig. 6 shows how the original model $\{\varepsilon, \delta, \beta_z, \beta_y\}$ can be augmented with another decoder δ_τ trained to predict the type τ of a programming construct or fragment given its code (z in Eq. (5)). Socher et al. [86] used a similar design to analyze sentiment in a semi-supervised way with manually generated multinomial distributions. Our augmentation would not require manually generated, coarse-grained labels like the sentiment task because the types here are automatically imputed by the compiler. For example, in Fig. 6, if we present [2.repr; 3.repr] to the model, then we expect $\hat{\tau}$ = VariableDeclarationFragment. The only change to the criterion (Eq. (7)) is adding an expression to compute (cross-entropy) misclassification costs.

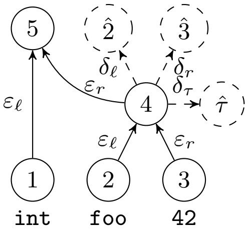

Figure 6: τ Decoder

8. CONCLUSION

We introduced a completely new way to detect code clones. Our learning-based paradigm diverges from traditional structure-oriented techniques in at least two important ways. First, terms—including identifiers—influence how fragments at different levels of granularity are represented. Second, our techniques are designed to automatically discover discriminating features of source code whereas traditional structure-oriented and metrics-based techniques use fixed transformations. Our results indicate that learning how to represent fragments for clone detection is feasible. We found that our techniques detected file- and method-level pairs mapping to all four clone types and evidence that learning is robust enough to detect similar fragments with reordered data independent declarations and statements, data dependent statements, and control statements that have been replaced [23]. Our online appendix is publicly available [112].

9. ACKNOWLEDGMENTS

We thank Lingxiao Jiang from Singapore Management University and Massimiliano Di Penta from the University of Sannio for their insightful comments that improved the paper. We thank Jeffrey Svajlenko and Chanchal Roy from the University of Saskatchewan for sharing data from their study [30]. This material is based upon work supported by the National Science Foundation under Grant No. 1525902.

10. REFERENCES

[1] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. ISESE'04.

[2] M. Gabel and Z. Su. A study of the uniqueness of source code. FSE'10.

[3] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. ICSM'98.

[4] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. METRICS'02.

[5] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *TSE*, 32(3), 2006.

[6] C. Kapser and M. Godfrey. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *EMSE*, 13(6), 2008.

[7] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? *EMSE*, 17(4-5), 2012.

[8] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. ESEC/FSE'05.

[9] D. Cai and M. Kim. An empirical study of long-lived code clones. FASE/ETAPS'11.

[10] C. Roy and J. Cordy. A survey on software clone detection research. Technical report, Queen's University, 2007.

[11] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *IJAST*, 1(3/4), 1995.

[12] J. Bailey and E. Burd. Evaluating clone detection tools for use during preventative maintenance. SCAM'02.

[13] M. Rieger. *Effective clone Detection Without Language Barriers*. PhD thesis, 2005.

[14] A. Walenstein and A. Lakhotia. The software similarity problem in malware analysis. Dagstuhl Seminar Proceedings, 2007.

[15] B. Baker. On finding duplication and near-duplication in large software systems. WCRE'95.

[16] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-independent clone detection applied to plagiarism detection. SCAM'10.

[17] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. ESEC/FSE'07.

[18] Lucia, D. Lo, L. Jiang, and A. Budi. Active refinement of clone anomaly reports. ICSE'12.

[19] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can i clone this piece of code here? ASE'12.

[20] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. Xiao: Tuning code clones at hands of engineers in practice. ACSAC'12.

[21] N. Milea, L. Jiang, and S. Khoo. Scalable detection of missed cross-function refactorings. ISSTA'14.

[22] N. Milea, L. Jiang, and S. Khoo. Vector abstraction and concretization for scalable detection of refactorings. FSE'14.

[23] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *SCP*, 74(7), 2009.

[24] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. WCRE'99.

[25] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. ASE'01.

[26] F. Deissenbock and M. Pizka. Concise and consistent naming [software system identifier naming]. IWPC'05.

[27] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. ICPC'06.

[28] Y. Bengio, A. Courville, and P. Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[29] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553), 2015.

[30] J. Svajlenko and C. Roy. Evaluating clone detection tools with bigclonebench. ICSME'15.

[31] K. Hotta, J. Yang, Y. Higo, and S. Kusumoto. How accurate is coarse-grained clone detection?: Comparison with fine-grained detectors. IWSC'14.

[32] J. Johnson. Identifying redundancy in source code using fingerprints. CASCON'93.

[33] J. Johnson. Visualizing textual redundancy in legacy source. CASCON'94.

[34] J. Johnson. Substring matching for clone detection and change tracking. ICSM'94.

[35] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. ICSM'99.

[36] B. Baker. A program for identifying duplicated code. In *Computer Science and Statistics*, 1992.

[37] B. Baker. Parameterized pattern matching: Algorithms and applications. *JCSS*, 52(1), 1996.

[38] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *TSE*, 28(7), 2002.

[39] W. Yang. Identifying syntactic differences between two programs. *SPE*, 21(7), 1991.

[40] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. WCRE'06.

[41] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. ICSE'07.

[42] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. VLDB'99.

[43] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. ICSE'08.

[44] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. SAS'01.

[45] J. Krinke. Identifying similar code with program dependence graphs. WCRE'01.

[46] C. Liu, C. Chen, J. Han, and P. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. KDD'06.

[47] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. ICSE'14.

[48] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3), 1987.

[49] I. Arel, D. Rose, and T. Karnowski. Research frontier: Deep machine learning–a new frontier in artificial intelligence research. *CIM*, 5(4), 2010.

[50] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. FASE'09.

[51] N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, and T. Nguyen. Complete and accurate clone detection in graph-based models. ICSE'09.

[52] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone management for evolving software. *TSE*, 38(5), 2012.

[53] M. Lee, J. Roh, S. Hwang, and S. Kim. Instant code clone search. FSE'10.

[54] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: Memory comparison-based clone detector. ICSE'11.

[55] F. Hermans, B. Sedee, M. Pinzger, and A. van Deursen. Data clone detection and visualization in spreadsheets. ICSE'13.

[56] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. ISSTA'09.

[57] D. Jurafsky and J. Martin. *Speech and Language Processing*. 2 ed., 2009.

[58] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. ICSE'12.

[59] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. FSE'14.

[60] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn. Cacheca: A cache language model based code suggestion tool. ICSE'15.

[61] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. ICST'13.

[62] D. Movshovitz-Attias and W. Cohen. Natural language models for predicting programming comments. ACL'13.

[63] J. Campbell, A. Hindle, and J. Amaral. Syntax errors just aren't natural: Improving error reporting with language models. MSR'14.

[64] P. Tonella, R. Tiella, and D. Nguyen. Interpolated n-grams for model based testing. ICSE'14.

[65] M. Allamanis, E. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. FSE'14.

[66] A. Nguyen, T. Nguyen, and T. Nguyen. Lexical statistical machine translation for language migration. ESEC/FSE'13.

[67] A. Nguyen, T. Nguyen, and T. Nguyen. Migrating code with statistical machine translation. ICSE Companion'14.

[68] A. Nguyen, H. Nguyen, T. Nguyen, and T. Nguyen. Statistical learning approach for mining api usage mappings for code migration. ASE'14.

[69] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. PLDI'14.

[70] V. Hellendoorn, P. Devanbu, and A. Bacchelli. Will they like this? evaluating code contributions with language models. MSR'15.

[71] B. Ray, V. Hellendoorn, Z. Tu, C. Nguyen, S. Godhane, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. *CoRR*, abs/1506.01159, 2015.

[72] M. Allamanis, E. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. FSE'15.

[73] R. Rosenfeld. Two decades of statistical language modeling: Where do we go from here? 88(8), 2000.

[74] A. Mnih and Y. Teh. A fast and simple algorithm for training neural probabilistic language models. ICML'12.

[75] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *JMLR*, 3, 2003.

[76] F. Morin and Y. Bengio. Hierarchical probabilistic neural network language model. AISTATS'05.

[77] H. Schwenk and J. Gauvain. Training neural network language models on very large corpora. HLT'05.

[78] Y. Bengio. Learning deep architectures for AI. *FTML*, 2(1), 2009.

[79] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, and S. Khudanpur. Recurrent neural network based language model. INTERSPEECH'10.

[80] T. Mikolov, S. Kombrink, A. Deoras, L. Burget, and J. Cernocký. Rnnlm - recurrent neural network language modeling toolkit. ASRU'11.

[81] T. Mikolov, S. Kombrink, L. Burget, J. Cernocký, and S. Khudanpur. Extensions of recurrent neural network language model. ICASSP'11.

[82] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocký. Strategies for training large scale neural network language models. ASRU'11.

[83] T. Mikolov. *Statistical Language Models Based on Neural Networks.* PhD thesis, 2012.

[84] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. MSR'15.

[85] C. Goller and A. Küchler. Learning task-dependent distributed representations by backpropagation through structure. ICNN'96.

[86] R. Socher, J. Pennington, E. Huang, A. Ng, and C. Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. EMNLP'11.

[87] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. EMNLP'13.

[88] J. Goodman. Classes for fast maximum entropy training. *CoRR*, cs.CL/0108006, 2001.

[89] A. Mnih and G. Hinton. Three new graphical models for statistical language modelling. ICML'07.

[90] Y. Shi, W. Zhang, J. Liu, and M. Johnson. Rnn language model with word clustering and class-based output layer. *EURASIP*, 1, 2013.

[91] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533, 2012.

[92] I. Sutskever, J. Martens, and G. Hinton. Generating text with recurrent neural networks. ICML'11.

[93] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. ICML'13.

[94] M. Hermans and B. Schrauwen. Training and analysing deep recurrent neural networks. NIPS'13.

[95] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio. How to construct deep recurrent neural networks. *CoRR*, abs/1312.6026, 2013.

[96] C. Bishop. *Pattern Recognition Machine Learning.* 2006.

[97] P. Werbos. Backpropagation through time: what it does and how to do it. 78(10), 1990.

[98] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* 2 ed., 2006.

[99] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* 3 ed., 2009.

[100] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6).

[101] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction.* 2000.

[102] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. WCRE'03.

[103] R. Koschke. Survey of research on software clones. Dagstuhl Seminar Proceedings, 2007.

[104] R. Koschke. Frontiers of software clone management. FoSM'08.

[105] J. Carver, D. Chatterji, and N. Kraft. On the need for human-based empirical validation of techniques and tools for code clone analysis. IWSC'11.

[106] D. Chatterji, J. Carver, and N. Kraft. Claims and beliefs about code clones: Do we agree as a community?: A survey. IWSC'12.

[107] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. ESEC/FSE'13.

[108] J. Svajlenko, J. Islam, I. Keivanloo, C. Roy, and M. Mia. Towards a big data curated benchmark of inter-project code clones. ICSME'14.

[109] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *EMSE*, 14(2), 2009.

[110] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue. Finding file clones in freebsd ports collection. MSR'10.

[111] J. Ossher, H. Sajnani, and C. Lopes. File cloning in open source java projects: The good, the bad, and the ugly. ICSM'11.

[112] https://sites.google.com/site/deeplearningclone/.

[113] R. Salakhutdinov and G. Hinton. Semantic hashing. *IJAR*, 50(7), 2009.

[114] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning.* 2 ed., 2009.

[115] A. Krizhevsky and G. Hinton. Using very deep autoencoders for content-based image retrieval. ESANN'11.