Deep Learning Software Repositories

Martin White
College of William and Mary, martingwhite@gmail.com

Deep Learning Software Repositories

Martin White

Virginia Beach, VA

Master of Engineering, Old Dominion University, 2011
Master of Science, Old Dominion University, 2007
Bachelor of Science, Old Dominion University, 2004

A Dissertation presented to the Graduate Faculty
of The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
August 2017

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

[illegible]

Martin White

Approved by the Committee, August 2017

[illegible]

Committee Chair
Associate Professor Denys Poshyvanyk, Computer Science
College of William & Mary

[illegible]

Associate Professor Peter Kemper, Computer Science
College of William & Mary

[illegible]

Professor Evgenia Smirni, Computer Science
College of William & Mary

[illegible]

Professor Andreas Stathopoulos, Computer Science
College of William & Mary

Weizhen Mao for Giuliano Antoniol

Professor Giuliano Antoniol, Computer Engineering and Software Engineering
École Polytechnique de Montréal

ABSTRACT

Bridging the abstraction gap between artifacts and concepts is the essence of software engineering (SE) research problems. SE researchers regularly use machine learning to bridge this gap, but there are three fundamental issues with traditional applications of machine learning in SE research. Traditional applications are too reliant on labeled data. They are too reliant on human intuition, and they are not capable of learning expressive yet efficient internal representations. Ultimately, SE research needs approaches that can automatically learn representations of massive, heterogeneous, datasets in situ, apply the learned features to a particular task and possibly transfer knowledge from task to task.

Improvements in both computational power and the amount of memory in modern computer architectures have enabled new approaches to canonical machine learning tasks. Specifically, these architectural advances have enabled machines that are capable of learning deep, compositional representations of massive data depots. The rise of deep learning has ushered in tremendous advances in several fields. Given the complexity of software repositories, we presume deep learning has the potential to usher in new analytical frameworks and methodologies for SE research and the practical applications it reaches.

This dissertation examines and enables deep learning algorithms in different SE contexts. We demonstrate that deep learners significantly outperform state-of-the-practice software language models at code suggestion on a Java corpus. Further, these deep learners for code suggestion automatically learn how to represent lexical elements. We use these representations to transmute source code into structures for detecting similar code fragments at different levels of granularity—without declaring features for how the source code is to be represented. Then we use our learning-based framework for encoding fragments to intelligently select and adapt statements in a codebase for automated program repair.

In our work on code suggestion, code clone detection, and automated program repair, everything for representing lexical elements and code fragments is mined from the source code repository. Indeed, our work aims to move SE research from the art of feature engineering to the science of automated discovery.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

Thank you to the members of SEMERU, particularly Michele Tufano whose keen insight advanced this work in many ways. Thank you, Michele.

I want to thank Matías Martínez and Martin Monperrus for the opportunity to collaborate. I am almost embarrassed at how much I learned from them.

I want to thank my committee members—Giuliano Antoniol, Peter Kemper, Denys Poshyvanyk, Evgenia Smirni, and Andreas Stathopoulos—all researchers who I hold in the highest regard. I extend my *deepest* professional gratitude to my advisor. Thank you, Denys, for your time and guidance from the beginning (absolute zero) through the end.

To my children, this process required appreciable sacrifice on your part, but I hope the experience imparted wisdom that will shape decisions you make for the better. To my wife, Jessica, this work is at least as much your accomplishment as it is mine. I have learned my most valuable lessons from you.

Marty
August 2017

To Jessica, for everything

LIST OF TABLES

LIST OF FIGURES

Deep Learning Software Repositories

Vision

Imagine you are a software engineering (SE) researcher. You study artifacts that are produced and archived during the software development lifecycle. Suppose you are presented with a problem like detecting similar source code fragments such as methods or classes in a codebase. What is the first thing you do?

A reasonable first step is to think about how to represent fragments. However, considering the complexity of software repositories, there is another approach. Rather than specify how you want to represent an artifact, you simply declare what you want to represent, abstracting away the problem of determining the representation. Changing “how” to “what” requires learning the representation from the data, which—considering the complexity of software repositories—requires models with a lot of representation power.

There is an apparent discrepancy between the expressiveness that is produced and archived in repositories and the representation power of models that we use to reap information from the repositories. If we can improve the representation power, then SE researchers will begin to limn their data several automated transformations away from input space, unveiling previously hidden insight. SE research papers will shift from motivating feature sets to prefiguring search problems for empirically-based features. The nature of data collection will change from describing preprocessing pipelines to emphasizing the small amount of intuition required to preprocess artifacts. SE research, an empirical discipline, values intentions like gaining empirically-based insight and replacing intuition with search-based optimization. But how can we improve representation power? This dissertation moves SE research toward deep learning software repositories.

Chapter 1

Introduction

SE research concerns the analysis, design, implementation, maintenance, and evolution of software systems, but modern ever-growing software repositories are extraordinarily complex. The complexity stems in part from the considerable volume of unstructured data such as requirements, design documents, source code files, communication archives (e.g., email and online fora transcripts), test cases, and defect reports. Moreover, the data are typically unlabeled, i.e., not primed to answer a particular question, and labeling software artifacts to supervise a learner is exceedingly expensive and consistently inconsistent. Bridging the abstraction gap between artifacts and concepts is the essence of SE research problems such as code clone detection and automated program repair. SE researchers regularly use machine learning to bridge this gap.

1.1 Deep Learning Software Repositories

Traditional applications of machine learning in SE research begin by assembling a list of attributes to characterize observations, and the goal of a machine learning algorithm is to discover a useful representation from the data to perform a task. For instance, many SE problems can be cast as classification tasks. For a classification task, the goal of machine learning may be to infer a conditional probability distribution over class labels.

This inference problem often assumes the training data are labeled, but SE research needs approaches that can leverage the disproportionate amount of unlabeled data in software repositories and learn useful representations with limited supervision. These problems also give way to feature engineering, where an expert places a strong prior on the feature space, which is assumed to be the ideal perspective for the task. Considering the complexity of software repositories, SE research needs approaches based on the science of building machines to automatically discover discriminatory features as opposed to the traditional art of intuitive feature engineering. These approaches would learn kernels rather than simply apply kernels [20]. Finally, there is an implicit problem with traditional approaches using kernels, which is learning decision boundaries in a feature space that is one handcrafted transformation removed from input space. This design exemplifies shallow architectures, and the problem here is one of efficiency. In order to solve deeply complex problems like code clone detection, SE research needs approaches that are capable of learning hierarchical representations to manage the complexity. Ultimately, SE research needs approaches that can automatically learn representations of massive, heterogeneous, datasets in situ, apply the learned features to a particular task and possibly transfer knowledge from task to task.

Improvements in both computational power and the amount of memory in modern computer architectures have enabled new approaches to canonical machine learning tasks, which underpin many different applications in SE research [217]. Specifically, these architectural advances (with keen activators [57] and suitable initialization [190]) have enabled machines that are capable of learning deep, compositional representations of massive data depots. The rise of deep learning has ushered in tremendous advances in several fields. Given the complexity of software repositories, we presume deep learning has the potential to usher in new analytical frameworks and methodologies for SE research and the practical applications it reaches.

1.2 Contributions

In Sec. 1.1, we diagnosed a few problems with traditional applications of machine learning in SE research. Traditional applications are too reliant on labeled data. They are too reliant on human intuition, and they are not capable of learning expressive yet efficient internal representations. By “internal,” we mean traditional approaches learn $y = f(x)$ as opposed to deep, compositional representations like $y = f(g(h(x)))$ where f, g, and h are nonlinear. Thus, by “expressive,” we mean that $f \circ g \circ h$ is a highly nonlinear operator capable of modeling arbitrarily complex domains. By “efficient,” we mean features can be reused at progressively higher levels of abstraction, e.g., g learns from h and f learns from $g \circ h$.

Why is all of this important to SE? Software repositories are too complex to simply operate on labeled input (label dependence), once (shallow architecture), in a specific way (feature engineering) to effectively conceptualize artifacts.

This dissertation posits a new research paradigm for SE where low-level representations automatically acquire domain knowledge from unlabeled input, and this knowledge automatically informs high-level, invariant features. Additionally, in this new paradigm, the knowledge we learn in one domain can be transferred to another domain under certain conditions. Our research lies at the nexus of this rich new class of models and SE. We use representation learning [20], a significant departure from traditional approaches, to automatically extract useful information from the disproportionate amount of unlabeled data in software repositories. The value in circumventing feature engineering is twofold. Learning transformations of the data will drastically reduce the cost of modeling software artifacts, because the purpose is to reduce the amount of supervision upon improving knowledge of the domain, and software repositories store a lot of data to affect the improvement. The other benefit is better performance. Generally, learning algorithms are more efficient than humans at discovering correlations in high dimensional spaces. So, rather than rely on strong prior knowledge to propose the best perspective on an SE task, our work uses machines to discover the perspective that yields optimal performance.

The following list summarizes the main contributions of this dissertation:

- We introduce deep learning to SE research. Deep learning, a nascent field in machine learning, will provide the SE community with new ways to mine and analyze artifacts to support SE tasks.
- We show that deep learning induces high-quality software language models compared to state-of-the-practice models using a general measure of quality. Then we demonstrate its effectiveness at code suggestion.
- We introduce a learning-based paradigm for code clone detection.
- We introduce a learning-based approach to intelligently select and adapt statements in a codebase for automated program repair along with a set of novel metrics that are specific to the analysis of fix space navigation strategies.

1.3 Dissertation Overview

Chapter 2: Deep Learning Code Corpora for Code Suggestion

Deep learning subsumes algorithms that automatically learn compositional representations. The ability of these models to generalize well has ushered in tremendous advances in many fields such as natural language processing (NLP). Recent research in the SE community has demonstrated the usefulness of applying NLP techniques to software corpora [14, 15, 43, 47, 55, 141, 154, 156]. Hence, we motivate deep learning for software language modeling, highlighting fundamental differences between state-of-the-practice software language models and connectionist models. Our deep learners are applicable to source code files (since they only require lexically analyzed source code written in any programming language) and other types of artifacts. We show how a particular deep learner can remember its state to effectively model sequential data, e.g., streaming software tokens, and the state is shown to be much more expressive than discrete tokens in a prefix. Then we instantiate deep learners and show that deep learning induces high-quality models compared to n-grams and cache-based n-grams on a corpus of Java projects. We experiment with two of the models' hyperparameters, which govern their capacity and the amount of context they use to inform predictions, before building several committees of software language models to aid generalization. Then we apply the deep learners to code suggestion and demonstrate their effectiveness at a real SE task compared to state-of-the-practice models. Finally, we propose avenues for future work where deep learning can support model-based testing and improve software lexicons.

Chapter 3: Deep Learning Code Fragments for Code Clone Detection

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 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.

Chapter 4: Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities

In the field of automated program repair, the redundancy assumption claims large programs contain the seeds of their own repair [13, 120]. However, most redundancy-based program repair techniques do not reason about the repair ingredients—the code that is reused to craft a patch. We aim to reason about the repair ingredients by using code similarities to prioritize and transform statements in a codebase for patch generation. Our approach, DeepRepair, relies on deep learning to measure code similarities. Code fragments at well-defined levels of granularity in a codebase can be sorted according to their similarity to suspicious elements (i.e., code elements that contain suspicious statements) and statements can be transformed by mapping out-of-scope identifiers to similar identifiers in scope. We examined these new search strategies for patch generation with respect to effectiveness from the viewpoint of a software maintainer. Our comparative experiments were executed on six open-source Java projects including 374 buggy program revisions and consisted of 19,949 trials spanning 2,616 days of computation time. DeepRepair's search strategy using code similarities generally found compilable ingredients faster than the baseline, jGenProg, but this improvement neither yielded test-adequate patches in fewer attempts (on average) nor found significantly more patches than the baseline. Although the patch counts were not statistically different, there were notable differences between the nature of DeepRepair patches and baseline patches. The results demonstrate that our learning-based approach finds patches that cannot be found by existing redundancy-based repair techniques.

1.4 Bibliographical Notes

Portions of this dissertation have been previously published or have been submitted for publication and are under review at the time of this writing. Material from Chap. 2 was published and presented at the 12th Working Conference on Mining Software Repositories (MSR'15) [210] and the IEEE/ACM 37th International Conference on Software Engineering (ICSE'15) [207]. Material from Chap. 3 was published and presented at the IEEE/ACM 31st International International Conference on Automated Software Engineering (ASE'16) [209]. The ideas and results in Chap. 4 are currently under review at an international conference. Material from Chap. 4 was based on a collaborative research effort with Martin Monperrus from the University of Lille and INRIA and Matías Martínez from the University of Valenciennes. Parergons of the work in this dissertation included work published and presented at the 12th Working Conference on Mining Software Repositories (MSR'15) [111] and the IEEE 23rd International Conference on Program Comprehension (ICPC'15) [208].

Chapter 2

Deep Learning Code Corpora for Code Suggestion

The field of NLP has developed many prominent techniques to support speech recognition [78] and statistical machine translation [97], among many other applications. One critical component of many of these techniques is a statistical language model. The most prevalent class of statistical language models is simple Markov models called n-gram models (or “n-grams”) [59]. n-grams are useful abstractions for modeling sequential data where there are dependencies among the terms in a sequence. A corpus can be regarded as a sequence of sequences, and corpus-based models such as n-grams learn conditional probability distributions from the order of terms in a corpus. Corpus-based models can be used for many different types of tasks like discriminating instances of data or generating new data that are characteristic of a domain.

The terms in a sequence can represent different entities depending on the domain. In SE, sequential data emerge from countless artifacts, e.g., source code files and execution traces, where the terms can be software tokens or method calls, and the sequences can be lines of code or method call sequences. While software tokens and method calls characterize two different lexicons, statistical language models such as n-grams can be applied to corpora from each domain because the models represent simple arrangements of terms. Consequently, models like n-grams can be used to predict the next term in a sequence [87].

Recent research in the SE community has examined and successfully applied n-grams to formal languages, like programming languages, and SE artifacts [2, 4, 6, 29, 68, 142, 150, 196, 198]. The breadth of these applications in SE research and practice underscores the importance of the ability to effectively learn from sequential data in software repositories. However, there is an apparent discrepancy between the representation power of models like n-grams for reaping information from repositories and the expressiveness that is produced and archived in repositories.

Consider the characteristics of modern software repositories and the requirements that these characteristics impose on models. Software repositories are massive depots of unstructured data, so good models require a lot of capacity to be able to learn from the voluminous scale rather than saturate after observing a fraction of the data that are available. Specifically, the kind of conceptual information that is buried in software repositories is very complex, requiring expressive models to manage this complexity. Moreover, software artifacts are laden with semantics, which means approaches that depend on matching lexical elements are suboptimal. Finally, practical SE tasks like developing a feature or reproducing an issue require a lot of context—more context than short lists of the last two, three, and four terms in a sequence.

Capacity, expressiveness, semantics, and context are key concerns when mining sequential SE data and inducing software language models in particular. Nonetheless, n-grams have limited capacity [171]. They are not expressive, because they are simply smoothed counts of term co-occurrences [137]. They have trouble with semantics and generalizing beyond the explicit features observed in training [21, 140, 179]. Lastly, language models, including software language models, based on n-grams are quickly overwhelmed by the curse of dimensionality [21], so the amount of context is limited.

How can we improve the performance at SE tasks based on software language models? In order to improve the quality of software language models, we must improve the representation power of the abstractions we use, so the goal of this work is to marry deep learning and software language modeling. The purpose of applying deep learning to software language modeling is to improve the quality of the underlying abstractions for numerous SE tasks, viz. code suggestion [52, 68, 198], deriving readable string test inputs to reduce human oracle cost [2], predicting programmer comments to improve search over codebases and code categorization [142], improving error reporting [29], generating feasible test cases to improve coverage [196], improving stylistic consistency to aid readability and maintainability [4], code migration [144, 145, 146], synthesizing application programming interface (API) completions [167], code review [65], fault localization [166], and suggesting accurate method and class names [5].

Thus, we make the following contributions:

- We introduce deep learning to SE research, specifically, software language modeling. Deep learning, a nascent field in machine learning, will provide the SE community with new ways to mine and analyze artifacts to support SE tasks.
- We motivate deep learning algorithms for software language modeling by clearly distinguishing them from state-of-the-practice software language models.
- We show that deep learning induces high-quality software language models compared to state-of-the-practice models using an intrinsic evaluation metric.[1] Then we demonstrate its effectiveness at code suggestion.
- While we focus on applying one deep architecture to one SE task, we believe deep learning is teeming with opportunities in SE research. We identify avenues for future work, highlighting different ways that deep learning can be used to support practical SE tasks.

Sec. 2.1 will review background on software language modeling and deep learning for NLP. This section will define all the keywords (e.g., deep architecture, deep learning,

[1]An intrinsic evaluation measures the quality of a model independent of any application [87].

deep software language model) and affirm the purpose of introducing these state-of-the-art approaches to SE research. Sec. 2.2 will pinpoint how this new class of software language models is poised to perform better at SE tasks by emphasizing their capacity and expressiveness as well as their ability to model semantics and consider rich contexts. Sec. 2.3 will use perplexity (PP), an intrinsic evaluation metric, to compare the quality of this new class of software language models to a state-of-the-practice baseline, and Sec. 2.4 will measure the models at a real SE task. Sec. 2.5 will discuss threats to the validity of our work. Sec. 2.6 will describe avenues for future work. One avenue proposes using deep software language models to support objectives other than code coverage in model-based testing. Another avenue proposes using deep software language models to improve software lexicons. Sec. 2.7 summarizes the chapter.

2.1 Background and Related Work

In this section, we present background on statistical language models and preliminary research applying these models to software corpora. We focus on how current approaches to software language modeling can be improved, laying the foundation for Sec. 2.2, where we show how deep learning can realize these improvements. Then we define all the keywords associated with deep learning, before reviewing preliminary research applying deep learning to NLP.

2.1.1 Language Models

A statistical language model is a probability distribution over sentences in a language [87]. This ostensibly simple abstraction is remarkably effective for NLP tasks such as speech recognition and statistical machine translation. In statistical language modeling, our goal is to find a tractable representation p of a sentence s in a language with vocabulary $\mathcal{V}$ by way of a joint distribution over words w_i:

$$p(s) = p(w_1, \ldots, w_m) \equiv p(w_1^m) = \prod_{i=1}^{m} p(w_i|w_1^{i-1}) \approx \prod_{i=1}^{m} p(w_i|w_{i-n+1}^{i-1}) \tag{2.1}$$

where $w_i \in \mathcal{V} \cup \{\text{BOS}, \text{EOS}\}$. BOS and EOS denote the beginning and end of sentence tokens, respectively. The joint distribution simplifies under the Markov assumption. In practice, we generalize the model's maximum likelihood estimates using one of many smoothing techniques [38]. These probabilistic automata (or, more specifically, n-grams) measure the degree of membership of every conceivable sentence in the language. Sentences frequently observed—in a generative sense—in the training corpus are judged to be more fluent than sentences observed less frequently (or not at all). In other words, we expect a good model to assign a high probability to a representative test document, or, equivalently, in-domain out-of-sample cases should have low cross entropy H with respect to p:

$$H_p(s) \equiv H_p(w_1^m) = \mathbb{E}_q[-\log_2 p(w_1^m)] \approx -\frac{1}{m} \sum_{i=1}^{m} \log_2 p(w_i|w_{i-n+1}^{i-1}) \tag{2.2}$$

where $q \sim \mathcal{U}(0, m)$. Cross entropy (Eq. (2.2)) is an empirical estimate of how well a language model predicts terms in a sequence [6]. Likewise, PP $= 2^{H_p}$ estimates the average number of terms at each point in the test document [6]. In language modeling, PP is a proxy for quality, and—as noted by Tu et al. [198]—good quality language models show great promise in SE applications. Our goal is to introduce powerful abstractions novel to software language modeling using PP as empirical validation of their efficacy and capacity to support SE tasks.

2.1.2 Applications of Software Language Models

Hindle et al. [68] demonstrated that language models over software corpora emit a "naturalness" in the sense that real programs written by real people have useful statistical properties, encapsulated in statistical language models, that can leverage SE tasks. This work was an important first step in applying natural language abstractions to software corpora. But n-grams are simple approaches that do not have the capacity to learn representations that reliably generalize beyond the explicit features in a training corpus [140]. Furthermore, these models build limited domain abstractions, and they are quickly overwhelmed by the curse of dimensionality [8, 21, 68, 126, 191].

The expectation in software language modeling research is that performance at SE tasks will improve with models more sophisticated than n-grams [68]. The purpose of our work is to introduce compositional representations that are designed to process data in stages in a complex architecture. Each stage transforms internal representations as information flows from one layer of the architecture to the next [18]. The feature spaces in a deep learner are fundamentally different than the conditional probability tables that constitute an n-gram model, and the power lies in the fact that these efficient distributed representations generalize well [18, 133].

Allamanis and Sutton [6] estimated an n-gram from a software corpus with more than one billion tokens, but we regard the massive scale as an organic smoothing technique. The model's effectiveness is still subject to token distances in the corpus where clues behind the n-gram's relatively short prefix are elided from the model's context [21, 126, 170]. Moreover, the massive scale does not truly solve the problem of considering tokens' semantic similarity [21, 126, 170]. The approach for software language modeling that we present in Sec. 2.2 is designed to consider an arbitrary number of levels of context where context takes on a much deeper meaning than concatenated tokens in a prefix. In our work, a deep learner encodes context in a continuous-valued state vector, encapsulating much richer semantics.

Allamanis and Sutton [6] conducted experiments where they collapsed the vocabulary by having the tokenizer replace identifiers and literals with generic tokens, which was a novel way to measure the model's performance on structural aspects of the code. However, we regard this approach as feature engineering. In this case, the token types in the corpus are engineered to solve the specific problem of modeling syntax. But the essence of deep learning, which underpins our work, is to design approaches that can automatically discover these feature spaces [18, 17] to—for instance—capture regularities at the syntactic, type, scope, and semantic levels [68].

Although Allamanis' giga-token model over source code demonstrated improvements in quality, a drawback to estimating n-grams over a massive corpus is losing resolution in the model. Good resolution yields regularities "endemic" to particular granularities, e.g., methods, classes, or modules. Of course, if the training corpus is too small, then the language model will be brittle for any practical application [74, 171]. A cache-based language model [39, 104] is designed to solve this optimization problem by interpolating a static model with a dynamic cache component. Kuhn and De Mori [104] originally proposed cache-based language models for speech recognition. Recently, Tu et al. [198] applied cache models to software corpora:

$$p(w_i|w_{i-n+1}^{i-1}, c) = \lambda p_N(w_i|w_{i-n+1}^{i-1}) + (1-\lambda)p_C(w_i|w_{i-n+1}^{i-1}) \tag{2.3}$$

where c is the list of n-grams stored in the cache; λ is the interpolation weight; p_N is a static n-gram model; and p_C is a dynamic cache model. The cache component encapsulates endemic and specific patterns in source code.

The cache component is a mechanism for capturing local context. The context we recur in a deep learner will be an expressive continuous-valued state vector. This state vector is capable of characterizing domain concepts [18, 21] rather than simply storing auxiliary conditional probability tables such as p_C in Eq. (2.3). Consequently, cache-based language models still suffer from the inability to understand semantic similarity, because they are fundamentally look-up tables, whereas deep learners induce similar representations for token types used in similar ways [137].

2.1.3 Artificial Neural Networks

Connectionism subsumes an expansive and deep body of knowledge that pervades artificial intelligence, neuroscience, and philosophy. Connectionist models comprise neuron-like processing units. Each unit has an activity level computed from its inputs [70]. In a feed-forward topology, information in the artificial neural network flows from input units through hidden units to output units along connections with adjustable weights.

A neural network architecture specifies intrinsic characteristics such as the number of units in the input, hidden, and output layers as well as the number of hidden layers in the network. A deep architecture comprises many hidden layers. Supervised learning algorithms discriminatively train [26] the weights to achieve the desired input-output behavior [174], so the hidden units automatically learn to represent important features of the domain [70]. This process of training the weights in a deep architecture is known as deep learning, and we refer to software language models based on deep learning as deep software language models. Thus, deep learners, including deep software language models, comprise many levels of nonlinear transformations [18].

The canonical learning algorithm for neural networks is the back-propagation procedure [174], which allows an arbitrarily connected neural network to develop internal representations of its environment [174]. These neural activation patterns, or distributed representations [69], harness formidable and efficient internal representations of domain concepts. Units can participate in the representation of more than one concept, which gives way to representational efficiency (where different pools of units encode different concepts) and aids generalization [18, 71].

A simple two-layer feed-forward neural network, with one hidden layer and one output layer, cannot reliably learn beyond first-order temporal dependencies [183]. This architecture can be augmented with a short-term memory by recurring the hidden layer, encapsulating the network's state, back to the input layer. This continuous-valued state vector is fundamentally different than a discrete token in an n-gram's history. The directed cycle provides context for the current prediction. We can provide more context by extending the recurrence and considering an arbitrary number of levels of context. From a temporal perspective, this recurrent neural network (RtNN) can be viewed as a very deep neural network [67, 77, 157, 190, 191] where depth is the length of the longest path from an input node to an output node.

The purpose of the depth in this case is to reliably model temporal dependencies. The depth of an RtNN is evident when you unfold the recurrence in time and measure the path from any unit in the deepest state vector to any output unit. Deep architectures like RtNNs lie at the forefront of machine learning and NLP, but we are not indiscriminately introducing complexity. We expect these approaches will yield tremendous advances in SE as they already have in other fields.

2.1.4 Applications of Neural Network Language Models

Connectionist models for NLP go back at least as far as Elman [50], who used them to represent lexical categories, and Miikkulainen and Dyer [125], who developed a mechanism for building distributed representations for communication in a parallel distributed processing network. Bengio et al. [21] proposed a statistical model of natural language based on neural networks to learn distributed representations for words to allay the curse of dimensonality since one training sentence increases the probability of a combinatorial number of similar sentences [21]. Sequences of words were modeled by agglutinating the word representations of consecutive words in the corpus into a single pattern to be presented to the network. Bengio also constructed model ensembles by combining a neural network language model with low-order n-grams and observed that mixing the neural network's posterior distribution with an interpolated trigram ($n = 3$) improved the performance. This work also measured the performance of the model after adding direct connections from nodes in the projection layer to output nodes, but the topology of this network does not constitute a deep architecture. This model represents history by presenting n-gram patterns to the network, whereas our work is based on a network which considers an arbitrary number of contextual levels to inform predictions.

Our primary related work is the work by Mikolov [126], who excised the projection layer in Bengio's architecture [21] and added recurrent connections [86] from the hidden layer back to the input layer to form an RtNN. Representing context with recurrent connections rather than patterns of n-grams is what distinguishes Mikolov's recurrent architecture from Bengio's feed-forward architecture. Mikolov reported improvements using RtNNs over feed-forward neural networks [126] and implemented a toolkit [131] for training, evaluating, and using RtNN language models. The package implements several heuristics for controlling the computational complexity of training RtNNs [128]. Recently, Raychev et al. [167] proposed a tool based in part on Mikolov's package, RtNNs, and program analysis techniques for synthesizing API completions.

2.2 Deep Software Language Model

In this section, we specify a deep architecture for software language modeling and pinpoint how this new class of models is poised to improve the performance at SE tasks that use language models. We begin with the ubiquitous two-layer feed-forward neural network. As noted in Sec. 2.1, these models cannot reliably learn beyond first-order temporal dependencies, so Elman networks augment the architecture with a short-term memory mechanism [50]. RtNNs extend Elman networks by considering an arbitrary number of levels of context. RtNNs are state-of-the-art models for NLP, but they are expensive to train, so a number of heuristics have been developed to control the complexity [128]. One heuristic is designed to reduce the complexity of computing the posterior distribution for each training example by factorizing the output layer and organizing the tokens into classes [130, 181]. Another heuristic involves training a maximum entropy model with an RtNN by implementing direct connections between input units and output units [128].

A pattern is presented to a feed-forward neural network by setting the value of each unit in the network's input layer x to the pattern's corresponding value. For instance, given a software corpus $\mathcal{C}$ of lexically analyzed tokens, we can represent a token w in the vocabulary $\mathcal{V}_{\mathcal{C}}$ using one-hot encoding and set $w_i = x_i$. The token is projected onto a feature space $\mathcal{F}$ by an affine transformation $p_j = a_{ji}x_i + b_j$. This transformation (or pre-activation) is a fundamental point of divergence from models like n-grams. Then each p_j is transformed by a differentiable, nonlinear function f such that $z_j = f(p_j)$ where z_j are the units that comprise the hidden layer z. The size of z (i.e., $|z|$) is an example of a hyperparameter [18], and adjusting this hyperparameter will regulate the model's capacity such that models with larger hidden layers yield more capacity [19, 20]. Practical choices for f include the logistic sigmoid, the hyperbolic tangent, and the rectifier [57]. These activation functions enable highly nonlinear and supremely expressive models [191].

After learning weights from x to z, when a fresh token is presented to the network, the units z_j will fire with varying intensities—analyzing the learned features—and ascribe a point in $\mathcal{F}$ to the token, effectively inducing clusters of examples in $\mathcal{F}$. These clusters enable a connectionist software language model to generalize beyond simple Markov chains in $\mathcal{C}$ like n-grams and model semantics. The hidden units are transformed $q_k = \beta_{kj}z_j$ (omitting bias terms going forward for brevity) and activated by a function g in the output layer y such that $y_k = g(q_k)$. For multinomial classification problems, such as predicting the next token in source code, the softmax function activates values in the output layer such that $p(y_k|w) = g(q_k)$. In software language modeling, propagating a token $w(t)$ from x through z to y yields a posterior distribution over $\mathcal{V}_{\mathcal{C}}$, and the model predicts the next token $w(t+1)$ in a sequence:

$$\hat{w}(t+1) = \operatorname*{argmax}_k p(y_k|w(t)) \tag{2.4}$$

We require an algorithm for learning $\theta = \{a, \beta\}$ from $\mathcal{C}$, i.e., maximizing the likelihood function,

$$\mathcal{L}(\theta) = \prod_{t=1}^{|\mathcal{C}|} p(w(t+1)|w(t), \theta)$$

Equivalently, we can minimize the negative log-likelihood by training θ using stochastic gradient descent [108]. For each $w \in \mathcal{C}$, we compute the gradient of the error in the output layer, using a cross entropy criterion, and propagate this error back through the network [174], using the chain rule to evaluate partial derivatives of the error function with respect to the weights, before updating the weights. Overfitting is a key concern since these models have the capacity to learn very complex representations, so θ is typically regularized [21, 126, 179].

The immediate concern with the model (Eq. (2.4)) is the inability to reliably learn beyond first-order temporal dependencies. n-grams encode temporal dependencies by learning tables of smoothed conditional probability distributions over a large number of prefixes. On the other hand, connectionist models can represent histories much more compactly. Specifically, an Elman network [50] augments a feed-forward network with a simple short-term memory mechanism. The short-term memory is realized by copying the hidden state $z(t-1)$ back to the input layer $x(t)$ and learning more weights γ to influence the hidden activations. This recurrence provides context for the current prediction. Thus, the input layer x in an Elman network is essentially a concatenation of $w(t)$ and $z(t-1)$, i.e., $x_i(t) = (w(t), z(t-1))_i$. In a feed-forward network, the pre-activation in z took the form $p_j = a_{ji}x_i$, but after recurring the state vector, the pre-activation in z takes the form $p_j(t) = a_{ji}w_i(t) + \gamma_{jj}z_j(t-1) = \alpha_{ji}x_i(t)$, where α is simply a concatenation of a_{ji} and γ_{jj}, and the model becomes $\theta = \{\alpha, \beta\}$. The cost of learning an additional $\mathcal{O}(m^2)$ parameters, where $m = |z|$, is met with improved representation power over sequences of software tokens.

An RtNN (Fig. 2.1) extends the memory bank in an Elman network for an arbitrary number of levels of context—though there may be practical limits to the depth of the recurrence [22]. Therefore, because of the gradient problem [22], we typically truncate the

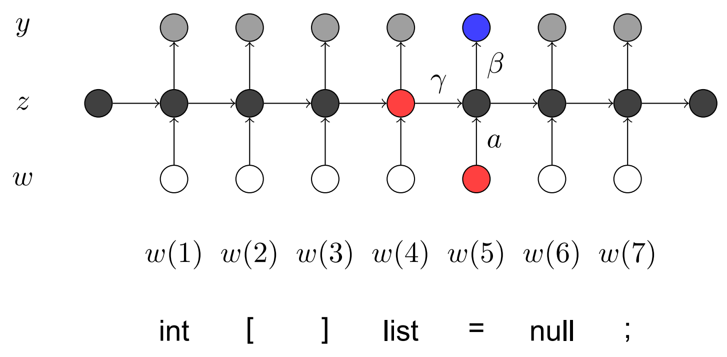

Figure 2.1: The depth of an RtNN is evident when the recurrence is unfolded in time. Time steps correspond to lexical elements in a corpus. Each node in the figure represents a vector of units. White nodes are one-hot term vectors; black nodes are continuous-valued state vectors; and gray nodes are posterior distributions over the vocabulary. The red nodes constitute the input layer at time $t = 5$. The blue node represents the posterior distribution for predicting $w(6)$.

back-propagation through time procedure [206]. Parenthetically, there are other ways to control this problem [126, 158, 178], but we omit these implementation details here. As the error is back-propagated through time in an RtNN, each level of temporal context has an abating amount of influence on training γ, which is shared across time. This weight sharing yields an efficient representation compared to n-grams, which are hampered by the curse of dimensionality as they try to encode deeper contexts. And persisting a sequence of state vectors is much more expressive, in terms of discriminative power, than hard-coded prefixes. Interestingly, if τ is the number of time steps the error is back-propagated through time, the network is still capable of learning information longer than τ steps [126].

However, while an RtNN is capable of learning powerful representations, it has some computationally expensive components, e.g., the posterior distribution in the output layer. Massive software repositories have a daunting challenge that arguably far exceeds the same problem in natural languages (including highly inflective natural languages), which is the size of the vocabulary. Computing the softmax function over extremely large vocabularies $|\mathcal{C}| \times \varepsilon$ times, where ε is the number of training epochs, is nontrivial. One solution to controlling this complexity is to factorize the output layer [59, 136, 140, 181]. Using class-based output layers has been shown to yield 15–30 times speed-up [126].

Direct connections are another implementation detail designed to improve performance. Bengio et al. [21] implemented direct connections from units in the projection layer to output units. The authors reported the connections did not help the model generalize from their relatively small corpus, but the connections did help reduce the training time. Mikolov [128] proposed direct connections from input units to output units and cast these connections as a maximum entropy model which can be trained with the neural network using stochastic gradient descent. The only change to the model specification is the addition of a term in the output pre-activation to account for the connections. The direct connections are reported to have yielded significant performance gains with respect to PP and word error rate [126].

Now, we are ready to present a deep architecture for software language modeling, specified in Eq. (2.5)–(2.7), without implementation details like class-based output layers and direct connections for clarity:

$$x_i(t) = (w(t), z(t-1))_i \tag{2.5}$$

$$z_j(t) = f(\alpha_{ji} x_i(t)) \tag{2.6}$$

$$y_k(t) = g(\beta_{kj} z_j(t)) \tag{2.7}$$

where $\alpha =$ concatenate(a, γ), $f(u_j) =$ sigmoid(u_j), and $g(u_k) =$ softmax(u_k). The model is similar to Eq. (2.4), except we present more than the current token to the network, i.e., $x(t)$ rather than simply $w(t)$.

2.3 Intrinsic Evaluation

The goal of our empirical study was to analyze deep learners for the purpose of evaluation with respect to effectiveness [211]. Our study was from the point of view of software developers in the context of real-world Java systems [211]. We used PP, an intrinsic evaluation metric that estimates the average number of tokens to choose from at each point in a sequence, as the criterion. We began by computing the PP of several different n-gram configurations by varying the order n and adding a dynamic cache component to establish a state-of-the-practice baseline in software language modeling. Then we instantiated several deep learners and computed the PP of these models with varying amounts of capacity and context over the same software corpus. Next, we selected the most performant architectures and interpolated several model instances to aid generalization and assess the performance of committees of software language models. Finally, deep learners are capable online learners, so we also measured the performance of models that learned as they tested on out-of-domain samples.

2.3.1 Methodology and Data Collection

To build the corpora for our study, we used the JFlex[2] scanner generator, packaged with a full Java lexical analyzer, to tokenize the source code in a repository of 16,221 Java projects cloned from GitHub. After tokenizing the files in each project, we sampled projects from the repository without replacement, querying enough projects to gather over seven million tokens. Then we randomly partitioned the projects into mutually exclusive training, development, and testing sets where approximately five million tokens were allotted for training, one million tokens for development, and one million tokens for testing.

The purpose of a training set is to learn a useful representation of the domain. For example, a high-quality software language model is useful, because it can effectively predict the next token in a sequence. In a supervised setting, the training set couples input and its corresponding target to guide the form of the representation. To learn a good model, the supervised learning algorithm presents a token $w(t)$ to the model and, in the case of deep software language models, the back-propagation through time algorithm (with a gradient descent step) trains the model using the next token $w(t+1)$ in the sequence.

Generally, the purpose of a development set is to govern hyperparameters such as the learning rate in gradient searches. The model is evaluated on the development set

[2]http://jflex.de/

Table 2.1: Code Corpora Statistics

Corpus	Projects	Tokens	Vocab.
Training	732	4,979,346	90,415
Development	125	1,000,581	19,816
Testing	173	1,364,515	32,124
Total	1,030	7,344,442	125,181

after each training epoch, and its relative performance on the development set can be used to judge convergence. It is important to note that data in the development set are not used to learn any of the model's parameters. In other words, none of the weights are modified as the model is evaluated on the development set.

Once the model is trained, it can be evaluated on a testing set. Notably, by partitioning the projects into mutually exclusive training, development, and testing sets, all of our experiments simulated new project settings (or greenfield development) [198]. Training and testing on distinct domains presents unique challenges for corpus-based models like software language models [198].

For each set of projects, we removed blank lines from the files and randomly agglutinated the source files to form a training corpus, a development corpus, and a testing corpus. Tab. 2.1 lists summary statistics for each corpus, including the number of projects used to form each corpus, the total number of tokens in each corpus, and the vocabulary size for each corpus. The total vocabulary size denotes the number of unique tokens in a concatenated corpus of all three corpora. From these corpora, we used standard text normalization techniques that are used in the NLP community [126]. We used regular expressions to replace integers, real numbers, exponential notation, and hexadecimal numbers with a generic NUM token. After replacing numbers, we replaced hapax legomenon in each corpus as well as every token in the development and testing corpora that did not appear in the training corpus with the UNK token to build an open vocabulary system [87] with a vocabulary size of 71,293.

Table 2.2: PP versus Order

Order	$\mathcal{B}$ack-off	$\mathcal{I}$nterpolated
2	29.9132	28.8362
3	20.9029	20.9299
4	20.0656	20.4990
5	19.8911	20.0741
6	19.9678	20.0497
7	19.9976	19.9929
8	20.0573	19.9815
9	20.0475	20.0147

Table 2.3: PP versus Cache

Cache	5-gram $\mathcal{B}$	8-gram $\mathcal{I}$
10	15.6914	15.6929
50	12.5881	12.5005
100	12.3170	12.2209
500	12.4920	12.3868
1,000	12.7552	12.6477
5,000	12.4912	13.3809
10,000	13.7828	13.6721

2.3.2 State-of-the-Practice Software Language Models

In practice, n-grams' maximum likelihood estimates are discounted [38], and the probability mass is redistributed using either back-off [92] or interpolation [79]. We used SRILM [189] to estimate back-off $(\mathcal{B})$ and interpolated $(\mathcal{I})$ n-grams from our training corpus, varying the order from two to nine. Each model was smoothed using modified Kneser-Ney [38] with an unknown token and no cutoffs. Tab. 2.2 lists PP versus order for each model. The models' results on the test corpus are virtually indistinguishable for this dataset, and both models appear to saturate near order five. This saturation is consistent with other studies on similar corpora [68]. With respect to PP, the most performant back-off model was the 5-gram and the most performant interpolated model was the 8-gram. We augmented these models with a unigram cache, varying the size of the cache from 10 to 10,000. The dynamic unigram cache model was linearly interpolated with the static n-gram model using a mixing coefficient of 0.05. Tab. 2.3 lists PP versus unigram cache size for both models. The 100-token unigram cache component effectively improved the performance for both the 5-gram back-off model and the interpolated 8-gram model. These performance gains from using a dynamic cache component are consistent with previous empirical studies [198].

Key result. We used the interpolated 8-gram model with a 100-token unigram cache as the baseline.

2.3.3 State-of-the-Art Software Language Models

After computing a baseline result using state-of-the-practice software language models, we trained several RtNNs. These models have expansive design spaces spanned by several hyperparameters. We chose to measure the performance by varying the size m of the hidden layer and the number of steps τ in the truncated back-propagation through time algorithm. To train and test RtNNs, we used the RNNLM Toolkit [131]. We instantiated 10 models with the same random seed, but we varied m from 50 to 500 units with sigmoid activations. The RNNLM Toolkit implements a simple mechanism to control the gradient problem by limiting the maximum size of gradients of errors that get accumulated in the hidden units [126]. For each model, we truncated the back-propagation through time algorithm at 10 levels, updating the context every 10 iterations, and factorized the output layer into 268 classes. We used the default starting learning rate of 0.1 and the default ℓ_2 regularization parameter of 10^{-6}. The learning rate was annealed during training by monitoring the model's performance on the development set. After each training epoch, PP on the development set was computed to govern the learning rate schedule [131]. Finally, to determine the number of direct connections from input nodes to output nodes, we built a frequency distribution of the token types in the training corpus. We found that 995 token types covered 80.0% of the tokens in the training corpus, so we set the number of direct connections equal to 1,000. All 10 deep learners—without a dynamic auxiliary component like a cache during testing—outperformed the baseline on our dataset, with the best results between 200 and 400 hidden units. Next, we selected the five models with m between 200 and 400. For each model, we varied the number of levels of context, keeping the same configuration for every other parameter (Tab. 2.4). For our dataset, the most performant models in our (m, τ) design space were (300, 20) and (400, 5).

Key result. Deep learning beat the baseline.

Table 2.4: PP versus Hidden Layer Size (m) and Depth (τ)

m	τ	PP
200	5	10.3744
	10	10.3168
	15	10.4778
	20	10.4465
250	5	10.2557
	10	10.4265
	15	10.2815
	20	14.3927
300	5	17.4354
	10	10.3297
	15	14.7124
	20	10.1960
350	5	10.4856
	10	10.3954
	15	10.6468
	20	10.2892
400	5	10.1721
	10	10.3338
	15	13.9920
	20	10.5023

2.3.4 Committees of Deep Software Language Models

Neural network language models are initialized with small random weights. Different instantiations will likely lead to models finding different local minima on the error surface, and models converging to different local minima may have different perspectives on the task. Therefore, we can construct committees of software language models by simply averaging $p(y|x)$ for each model instance [25]. Bengio et al. [21] reported performance gains by combining a neural network language model with an interpolated trigram, and the authors noted the performance gains suggest that the models make errors in different places. Likewise, Schwenk and Gauvain [179] interpolated neural network language models with back-off models to improve the performance in a speech recognition system. Mikolov [126] reported performance gains by combining several RtNN language models. We instanti-

Table 2.5: Committees of Software Language Models

Architecture	Seeds	Coefficients	PP
(300, 20)	1,2	0.50	9.6467
	1,2,3	0.33	9.5060
	1,2,3,4	0.25	9.4549
	1,2,3,4,5	0.20	9.3534
	1,2,3,4,5,$\mathcal{N}$	0.60	7.8512
(400, 5)	1,2	0.50	9.5775
	1,2,3	0.33	9.9305
	1,2,3,4	0.25	9.6265
	1,2,3,4,5	0.20	9.5326
	1,2,3,4,5,$\mathcal{N}$	0.60	7.9346

ated five models with 300 hidden units and 20 levels of context and five models with 400 hidden units and five levels of context with different random seeds. Tab. 2.5 lists the results of combining several software language models on our dataset. For example, the first row denotes the linear interpolation of two (300, 20) models—one model instantiated with random seed 1 and the other instantiated with random seed 2—where the coefficients in the mixture are 0.50. $\mathcal{N}$ denotes an interpolated 8-gram model with a 100-token unigram cache. So, the fifth row represents the combination of five deep models and an interpolated n-gram model, where the combination of deep models has a weight of 0.60 in the mixture and the n-gram has a weight of 0.40. The top performing committee achieves a cross-entropy score of 2.9729 bits. Recall these performance scores are computed using a training corpus of 732 randomly chosen projects, and the test corpus is another random collection of 173 out-of-domain projects. The committee improves the performance as compared to instances of each model.

Key result. Constructing committees of deep software language models can aid generalization and improve performance.

Table 2.6: Static, Dynamic, and Mixture Models

Architecture	Seed	Update	Coefficients	PP
(300, 20)	5	S	-	10.1686
		D	-	3.6518
		M	0.50	3.9856
(400, 5)	1	S	-	10.1712
		D	-	3.5958
		M	0.50	3.7480

2.3.5 Deep Software Language Models Online

Cache-based language models separate static concerns from dynamic concerns using a convex combination of components where a large static model is interpolated with a small dynamic model. In software language modeling, this small dynamic model has been used to capture local patterns in source code [198]. Neural network language models are capable of learning online by back-propagating the error for each test document and performing a gradient search thereby enabling adaptation. Tab. 2.6 lists the results of evaluating models online on our dataset. Static models denote conventional models. Dynamic models denote deep learners that learn as they test on out-of-domain samples. Mixture models denote a committee comprising the static and dynamic models using the corresponding mixture coefficient. For example, the third row represents the combination of one static model with 300 hidden units and 20 levels of context and one dynamic instance where the posterior distributions are equally weighted. Evaluating models online significantly improved the performance on our dataset.

Key result. The cross entropy scores for online models are on the order of two bits. When deep software language models are online, they can be incrementally trained. Thus, in new project settings, online learners are able to automatically adapt as the project is being developed. In the committee of static and dynamic models, the static component can be regarded as weak prior knowledge of the domain in new project settings, and the dynamic component acts as an incremental learner, which adapts as the project is being developed. Although the dynamic models performed noticeably better on our dataset, one potential benefit of sacrificing some of the performance gain by using a committee is that static models can serve as anchors and help prevent the model from being "poisoned," i.e., degenerated by learning unreliable information online.

2.4 Extrinsic Evaluation

In Sec. 2.3, our intrinsic evaluation compared the quality of deep software language models to state-of-the-practice models. In this section, we conduct an extrinsic evaluation [87] to measure the performance of deep software language models in re code suggestion and show that deep learning improves the performance. A code suggestion engine recommends the next token given the context [68, 198]. The goal of our empirical study was to analyze deep software language models for the purpose of evaluation with respect to effectiveness [211]. Our study was from the point of view of the software developer in the context of the same corpora listed in Tab. 2.1.

We examined the following research questions:

RQ1 Do deep learning models (Sec. 2.2) significantly outperform state-of-the-practice models (Sec. 2.1) at code suggestion on our dataset?

RQ2 Are there any distinguishing characteristics of the test documents on which the deep learning models achieve considerably better performance as compared to state-of-the-practice models?

2.4.1 Research Question 1

We used Top-k accuracy to compare deep software language models to state-of-the-practice models at code suggestion. Top-k accuracy has been used in previous code suggestion studies [68, 198]. Tab. 2.7 lists our Top-k results for the most performant static

Table 2.7: Top-k Accuracy

Model	Update	Top-1 (%)	Top-5 (%)	Top-10 (%)
8-gram $\mathcal{I}$	S	49.7	71.3	78.1
$\mathcal{N}$	D	4.8	69.5	78.5
(400, 5)	S	61.1	78.4	81.4
(300, 20)	D	72.2	88.4	92.0

and dynamic models of each model type. The deep learning models appear to outperform the n-grams at each level, so we designed comparative experiments to measure the statistical significance of our results. The treatments in our experimental design were the language models. The experimental units were the sentences in the test corpus, and the responses were the Top-k scores. The null hypothesis stated there was no difference in performance. The (two-tailed) research hypothesis stated there was a difference in performance. We tested these hypotheses at $\alpha = 0.05$ using the Wilcoxon test [180], a nonparameteric test, to determine whether the reported differences were statistically significant. Comparing the dynamic (300, 20) model to the 8-gram $\mathcal{I}$ model, we found $p \leq 2.2 \times 10^{-16} < 0.05 = \alpha$ in all three cases (i.e., Top-1, Top-5, and Top-10); therefore, we rejected the null hypothesis, suggesting that a statistically significant difference existed. We interpreted the difference as deep learning realizing an improvement at the code suggestion task. Regarding the effect size (Cliff's δ), we observed a medium effect size for Top-1, a large effect size for Top-5, and a medium effect size for Top-10.

Key result. Deep learning significantly outperformed n-grams at code suggestion on our dataset.

2.4.2 Research Question 2

After assessing the significance of applying deep learning to a real SE task, we conducted an exploratory study on the performance results. We sorted all the sentences in the test corpus by their Top-10% (according to the n-gram), i.e., the ratio of the number of tokens (including EOS) in the sentence suggested in the Top-10 divided by the total number of tokens in the sentence. We observed that the sentences at the top of the list with low Top-10 scores were relatively short in length. Some of these sentences only comprised annotations (e.g., @Before and @Test) and others only comprised keywords (e.g., else, try, and finally). Given the poor performance of n-grams on these test documents, we were interested in comparing the performance of deep software language models on these sentences. We designed another set of experiments to compare the performance of the two models; however, in these experiments, the experimental units were sentences of length one, two, or three, respectively. Each experiment compared the models' Top-k performances at each sentence length. The null hypothesis for each comparative experiment stated there was no difference in performance. The (two-tailed) research hypothesis stated there was a difference in performance. We tested these hypotheses as above. All comparisons yielded statistically significant differences where $p \leq 2.2 \times 10^{-16} < 0.05 = \alpha$; therefore, we rejected the null hypothesis (for each comparison) and interpreted the difference as improved performance. Regarding the effect size (Cliff's δ), we only found large effect sizes for Top-5 where the sentence length was equal to two and for Top-10 where the sentence length was equal to two or three.

Our results show that deep learning improves the performance at a SE task based on software language models. Moreover, there may be interesting cases in software corpora where deep learning outperforms models like n-grams. Sentences of length one or two tokens are arguably more relevant to software corpora than natural language corpora.

2.5 Threats to Validity

Internal Validity

Threats to internal validity can be related to confounding factors internal to a study that could have affected the results. In our study, we relied on the RNNLM Toolkit to train and evaluate deep software language models. While the toolkit is a reliable implementation that has been used in a number of NLP experiments [128, 130, 131], it is still an evolving project. However, our results and trends are in line with those that have been obtained in the field of NLP. Thus, we are confident that the results are reliable.

External Validity

Threats to external validity represent the ability to generalize the observations in a study. We do not claim that the obtained results can be observed across other repositories or projects, especially projects written in other programming languages. Additionally, our dataset is representative of only repositories hosted on GitHub, so we do not claim that the results generalize to all Java projects. GitHub's exponential growth and popularity as a public forge indicates that it represents a large portion of the open source community. While GitHub contains a large number of repositories, it may not necessarily be a comprehensive set of all open source projects or even all Java projects. However, we analyzed the diversity of the projects from the proposed metrics in Nagappan et al. [143] and compared our dataset to the projects available on Boa [49] and found 1,556 projects out of the 16,221 projects. We also analyzed the diversity of the 1,030 tokenized projects in our training, development, and testing corpora, and we were able to match 128 projects. Our entire dataset had a diversity score of 0.3455, and the subset that we used to conduct our language modeling experiments had a diversity score of 0.2208. According to our dimensions, these values suggest that approximately 10% of our entire dataset covers one-third of the open source projects, and approximately 10% of our corpus covers one-fifth of open source projects. In our diversity analysis, we considered six metrics: programming languages, developers, project age, number of committers, number of revisions, and number of programming languages. For the entire dataset, we had scores of 0.45, 0.99, 1.00, 0.99, 0.96, and 0.99, respectively. For the study corpora, we had scores of 0.38, 0.98, 1.00, 0.98, 0.92, and 1.00, respectively. These results indicate that both our dataset and our corpora have high-dimensional diversity coverage for the relevant dimensions to our study. Since we consider only Java projects, it is expected that our representativeness would be rather low in the programming languages dimension. Thus, our results are representative of a proportion of the open source community. Further evaluation of projects across other open source repositories and other programming languages would be necessary to validate our observations in a more general context. It is also important to note that we only consider open source projects.

Construct Validity

Threats to construct validity concern the relationship between theory and observation and relate to possible measurement imprecision when extracting data used in a study. In mining the Git repositories and collecting the projects for our analysis, we relied on both the GitHub API and the Git command line utility. These tools are under active development with a community supporting them. Additionally, the GitHub API is the primary interface to extract project information. We cannot exclude imprecisions due to the implementation of such an API.

2.6 Avenues for Future Work

There are two principal research components in our future work on deep software language modeling and using deep learning to mine sequential SE data. One research component examines extensions of the models. One set of extensions involves search problems, such as hyperparameter optimization, designed to improve deep learning-based approaches for mining sequential SE data. Another set of extensions involves entirely new architectures and models, such as stacked RtNNs [178] and recursive neural networks [186], for mining sequential SE data. The other research component examines applications of deep architectures to SE tasks. The first application, model-based testing, is an example of an SE task that can benefit from our deep software language models, where the raw features do not have to be words per se. As we noted above, the nature of the terms depends on the domain, and this application supports that claim. The next application, software lexicon, shows that deep software language models are not simply useful for their high-quality output. We can also use their internal representations and feature detectors to support SE tasks. Thus, the same model that serves as a code suggestion engine can also be used to improve the software lexicon.

Hyperparameter Optimization

Deep architectures comprise many levels of nonlinear transformations. Different subspaces in a deep architecture are trained as information flows forward and supervision propagates back through the network, but models like RtNNs entail a considerable number of hyperparameters for governing different facets of the architecture. For example, the size of the hidden layer, the number of levels before truncating the back-propagation through time algorithm, the number of classes for partitioning token types in the output layer, the learning rate, and the amount of regularization yield an expansive design space with new and important search problems for SE research to optimize these complex architectures over SE datasets for SE tasks. SE research has examined similar problems in different contexts [155]. Additionally, recent research in the machine learning community has proposed methodologies beyond a naive grid search for automatically configuring the optimal set of hyperparameters [23, 184], but these approaches have not been measured using SE datasets in the context of SE tasks.

Model-based Testing

Tonella et al. [196] demonstrated how interpolating a spectrum of low-order Markov models inferred from an event log can be used to improve code coverage, since “backing off” increases the likelihood of deriving feasible test cases. Conceptually, the work by Tonella et al. uses “naturalness” (Sec. 2.1) at different scales to improve code coverage, but we envision much more opportunity in model-based testing by exploiting the posterior distribution in the output layer of a deep software language model. While the posterior can specify natural event sequences, we can also infer unnatural event sequences from this model. Our work will segment the posterior's domain into a natural space and an unnatural space [111]. In this sense, we propose a novel interpretation of deep software language models as natural bits to support other aspects of software testing, e.g., destructive testing and regression testing. Moreover, the natural space in this model is not fixed. We envision a framework for adapting this space by dynamically toggling event sequences as natural or unnatural depending on the evidence to steer the model online according to specific objectives (other than code coverage).

Software Lexicon

Software maintenance is hard. Since program comprehension is one contributing factor, improving the software lexicon is one way to support critical maintenance tasks [161]. We envision a novel lexicon broker to negotiate commits with the expressed goal of supporting program comprehension by consolidating concepts and, to this end, serving as a recommendation engine in cases where developers' implementations can be improved. How can we enable this broker? While a deep software language model can effectively support many different SE tasks, the architecture's components may be used for other pertinent SE tasks such as learning software synonyms in massive repositories [73, 194, 195, 213, 214]. Recall the deep software language model embeds a token vector in a low-dimensional subspace using a linear projection a (Sec. 2.2). This transformation is perhaps best understood by thinking of the vector $a_{ji}w_i$ as a linear combination of the columns of a [197], i.e., $a_{ji}w_i = \sum_{j=1}^{n} w_j a_{\cdot j} = a_{\cdot N} \in \mathbb{R}^{m\times 1}$ where $1 \leq N \leq n$, $m = |z|$, and the last equality is because w is one-hot encoded. So, each column in a represents one token in the vocabulary. These are contextualized feature vectors that can leverage intelligent recommendations on improving the lexicon. After conducting our empirical validation (Sec. 2.3) and observing how well the deep software language models performed

Table 2.8: Querying Similar Terms

Query	Closest Tokens
getX	getY, getWidth
transient	native, volatile
@BeforeClass	@AfterClass, @Extension

on our dataset, we conducted a cursory study of the models' internal representations. The purpose of this small exploratory study was to begin to understand how these models can be analyzed to address other SE concerns, e.g., token similarity [195]. We extracted the token embeddings $a_{\cdot j} \in \mathbb{R}^{300\times 1}$ from one of our static (300, 20) models (Sec. 2.3). Tab. 2.8 lists the two closest tokens, using Euclidean distance, for three distinct queries. Of the 71,293 token types in the vocabulary (Sec. 2.3), the two closest tokens to "getX" were two other getter methods that appear to be related to position or size. Again, of the 71,293 token types, the closest tokens to "transient" were two other Java keywords. Finally, the two closest tokens to "@BeforeClass" were two other Java annotations. While these are intriguing anecdotes, this is very preliminary work in this space, but we believe these cursory observations warrant a deeper and much more rigorous examination in different SE contexts.

2.7 Summary

State-of-the-practice software language models are bound to the n-gram features that are apparent by simply scanning a corpus and aggregating counts of specific and discrete token sequences. On the other hand, deep learning uses expressive, continuous-valued representations that are capable of learning more robust models. We propose that SE research, with a wealth of unstructured data, is a unique opportunity to employ these state-of-the-art approaches. By empirically demonstrating that a relatively simple RtNN configuration can outperform n-grams and cache-based n-grams with respect to PP on a Java corpus, deep software language models are shown to be high-quality software language models, capable of showing great promise in SE applications [198]. We also demonstrate an improvement in performance at an SE task. Finally, we identify avenues for future work using deep software language models to conduct model-based testing and improve software lexicons. Computer vision, speech recognition, and other fields have occupied the attention of the approaches we present in this chapter. Our work is the first step toward deep learning software repositories.

Chapter 3

Deep Learning Code Fragments for Code Clone Detection

Abstraction is the most important word in 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 [121]. For example, software systems comprise modules; modules comprise classes; and classes comprise methods. Modules, classes, and methods represent different levels of granularity and enable the decomposition of a problem into specialized constituent solutions, i.e., constituents. A module encapsulates a coherent set of classes. Likewise, classes encapsulate specific services or utilities that are designed to aid software development around a particular concern. Software components can generally be characterized by their constituents at lower levels of granularity. Thus, a general procedure for representing a well-conceived software component is adequately representing its constituents.

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 [54, 95]. 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 [16, 110, 138], cloning is not necessarily harmful [91, 165]. Nor should clones necessarily be refactored [28, 96]. Nonetheless, the ability to automatically detect that two fragments are similar is critical in many applications [172], e.g., detecting library candidates [9, 44], aiding program comprehension [168], detecting malicious software [202], detecting plagiarism or copyright infringement [10, 27, 90], detecting similar applications [122, 200], detecting context-based inconsistencies [82, 114, 205], and searching for refactoring opportunities [42, 134, 135]. Roy and Cordy [172] classified clone detection techniques by their "internal source code representation," synthesizing a taxonomy of text-, token-, tree-, graph- and metrics-based techniques. In this dissertation, our newfangled approach to mining internal source code representations gives way to a new, learning-based paradigm.

The clone detection process begins by transforming source code in situ into representations suitable for assessing similarity [172, 173]. 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 [31, 46, 105, 115] 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[1] to how we can automatically represent fragments. However, it is reasonable to expect that software systems from different application domains and at different

[1] By "prior knowledge," we mean information coupled with (yet distinct from) training data.

stages of development yield unique patterns in 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" [20, 107], 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 twofold. 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. 3.2.1). This transformation from lexical elements to vectors is fundamentally different than the token abstraction used by token-based techniques (Sec. 3.1.1). Second, our representation learning-based approach is designed to learn discriminating features for fragments at different levels of granularity (Sec. 3.2.2) rather than depend on intuitive (yet threateningly myopic) features that are designed around the structural elements of a language like tree-based techniques (Sec. 3.1.1).

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. 3.2.1) and syntactic analysis with recursive neural networks (Sec. 3.2.2). The purpose of coupling the front end of the compiler with deep neural networks and deep learning (Sec. 3.1.3) 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.

Sec. 3.1 will review background on code clone detection and recursive deep learning. This section will define four types of clones and review different code clone detection techniques. Sec. 3.2 will present our novel, learning-based approach to encoding fragments at arbitrary levels of granularity. Sec. 3.3 will describe our empirical validation aimed at determining whether the idea of learning representations for fragments can be relevant for clone detection. Sec. 3.4 will present our results. Sec. 3.5 will consolidate threats to the validity of our work and lessons learned from training recursive deep learners on source code corpora. Sec. 3.6 will describe avenues for future work. One avenue proposes using different metric spaces to compute code similarities at scale. Another avenue proposes augmenting the learning algorithm with more information imputed by the compiler to solve other prediction problems. Sec. 3.7 summarizes the chapter.

3.1 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 [193]. Code clones (or clones) are two or more fragments that are similar with respect to a clone type [193]. A candidate is a clone pair reported by a clone detector [72]. 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. 3.1.1) is chock-full of handcrafted feature vectors to represent fragments. In our work (Sec. 3.2), 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. 3.1.3) 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.

3.1.1 Code Clone Detection

Generally, there are four clone types. Type I: Identical fragments except for variations in comments, white space, or layout [172]. Type II: Identical fragments except for variations in identifier names and literal values in addition to Type I differences [172]. 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 [172]. Type IV: Syntactically dissimilar fragments that implement the same functionality [172]. 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 [48, 83, 84, 85] 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 [10, 11, 12, 90, 110] 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 [173]. 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 [16, 80, 101, 215] measure the similarity of subtrees in syntactic representations. Our primary related work is the influential work by Jiang et al. [80] who presented Deckard, which transforms parse trees into “characteristic vectors” and clusters similar vectors (using Locality Sensitive Hashing [56]) 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 [53]. 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 modest number of specific features. Moreover, characteristic vectors approximate structural information while neglecting domain information rooted in identifiers [135]. In fact, there is generally no special treatment for identifiers and literal types in AST-based approaches [172]. Our work operates on identifiers and literal types.

Graph-based techniques [37, 53, 98, 102, 112] use static program analysis to transform code into a program dependence graph (PDG), an intermediate representation of data and control dependencies [51]. Gabel et al. [53] 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. [37] used a “geometry characteristic” of dependency graphs to measure methods’ similarities before combining method-level similarities to detect application clones in Android markets. They begin 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 basic structures of structured programming. 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 [37]. 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 [7].

Other detection approaches include the following. Davey et al. [44] ignored identifiers and operators and instead considered the frequency of keywords, indentation patterns, and line lengths to represent fragments. These feature vectors were passed to a self-organizing map to detect clones. Marcus and Maletic [115] examined identifiers and comments to identify implementations of similar high-level concepts. Nguyen et al. [148, 149, 160] extracted structural features from generic, graph-based representations to build characteristic vectors. Lee et al. [109] measured structural similarities like Deckard and proposed a multidimensional indexing structure to support fast inference. Kim et al. [94] proposed a semantic detection technique that compared programs’ memory states. Hermans et al. [66] proposed a text-based algorithm for detecting clones in spreadsheets.

Finally, Jiang and Su [81] 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.

3.1.2 Language Modeling

Our approach is based in part on language models (Sec. 2.1.1). Traditionally, statistical language models have been effective abstractions for NLP tasks. Recently, their effectiveness has suffused SE tasks (Sec. 2.1.1). A statistical language model is a tractable representation of sentences (e.g., lines of code or traces of method invocations) in a language (Sec. 2.1.1). Tractability is realized by decomposing a joint distribution and analyzing probabilistic automata such as n-grams (Eq. (2.1)). In Chap. 2, we showed that deep learning induces high-quality software language models, so we enlist RtNNs, which map the terms in source code to continuous-valued vectors called embeddings (Sec. 3.2.1). To the best of our knowledge, we are the first to propose language models and embeddings for code clone detection.

3.1.3 Recursive Deep Learning

Compositional learning algorithms typify deep learning [18, 107]. In Chap. 2, we 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 [210]. While RtNNs are powerful architectures for modeling sequences of terms, their generalization—the recursive neural network (RvNN) [58]—is capable of modeling arbitrary structures to, for instance, predict the sentiment of natural language sentences [187, 188]. 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 inher-

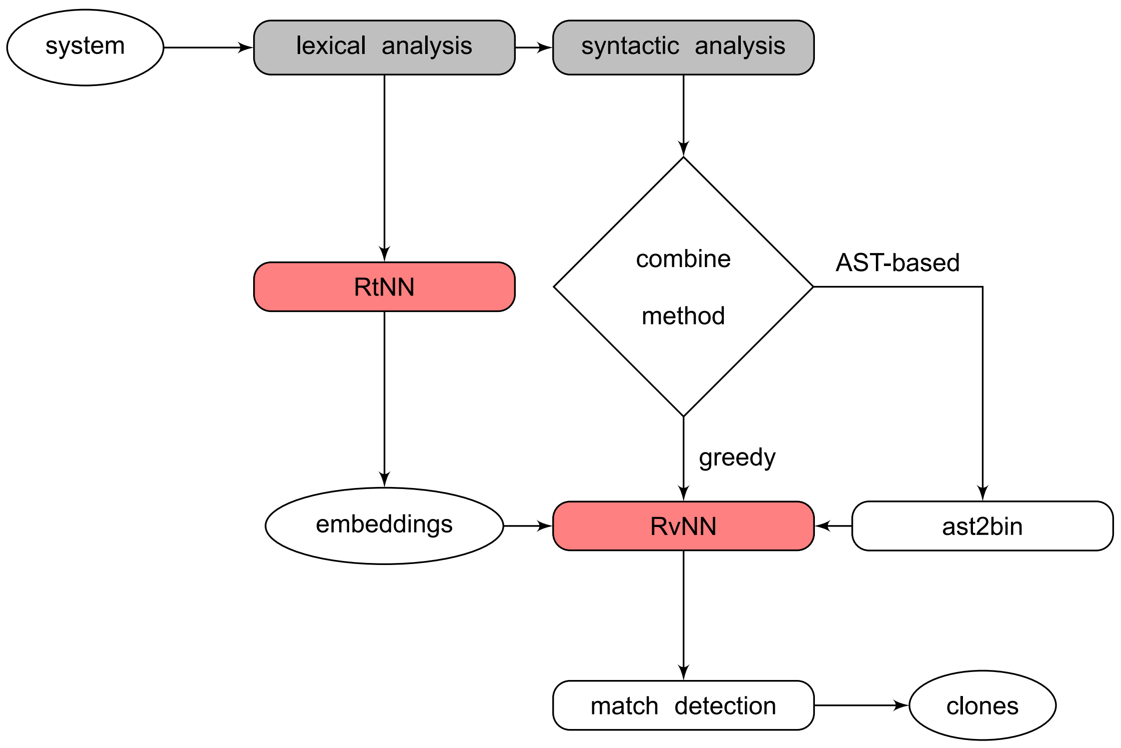

Figure 3.1: Our approach couples deep learners (red) to front end compiler stages (gray).

ently 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.

3.2 Deep Learning Code Fragments

In this section, we specify our learning-based approach (Fig. 3.1) in two parts. The first part (Sec. 3.2.1) 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 [126, 128, 129, 130, 131] and SE literature [210] for some of the technical de-

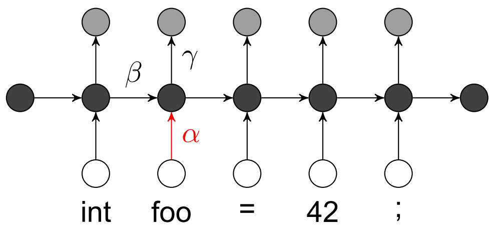

Figure 3.2: RtNN. White nodes are one-hot term vectors; black nodes are continuous-valued state vectors; and gray nodes are posterior distributions. We extract the matrix of embeddings represented by the red arc.

tails behind training and evaluating these models. The second part (Sec. 3.2.2) 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.

3.2.1 Deep Learning Code at the Lexical Level

An RtNN (Fig. 3.2) 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 [18], 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 [59, 126, 136, 140, 181]). The size of z (i.e., $|z| = n$ hidden units) is an example of a user-specified hyperparameter [18]. Adjusting n regulates the model's capacity [19, 20]. Recall from Chap. 2 that the depth of an RtNN is attributed to the recurrence [67, 77, 107, 157, 190, 191] 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 $z(i-1)$:

$$x(i) = [t(i); z(i-1)] \tag{3.1}$$

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.,

$$z(i) = f(\alpha t(i) + \beta z(i-1)) \tag{3.2}$$

where we omit the bias term. This state vector is multiplied by another matrix $\gamma \in \mathbb{R}^{m \times n}$ and normalized to compute a posterior distribution over terms,

$$y(i) = p(t|x(i)) = \text{softmax}(\gamma z(i)) \tag{3.3}$$

Eq. (3.1)–(3.3) specify an RtNN. Eq. (3.4) 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 $\underline{\underline{\text{inputs}}}$:

$$\underline{y(i)} = \text{softmax}(\gamma f(\alpha \underline{\underline{t(i)}} + \beta f(\alpha \underline{\underline{t(i-1)}} + \beta \underline{\underline{(\cdots)}}))) \tag{3.4}$$

The model $\theta = \{\alpha, \beta, \gamma\}$ is trained using a cross entropy criterion [25] but we omit the technical details here [126, 206]. In software language modeling, the model's output $y(i)$ can be used to predict the next term in a line of code as $\text{argmax}_k\, y_k(i)$ [210]. 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. (3.2). Each column of α corresponds to a term. The column space of α comprises semantic representations [5, 107, 210] for every term in $\mathcal{V}$ such that the model imputes similar vectors to terms used in similar ways in the corpus [107]. Given that each term is one-hot encoded when presented to the model, the matrix-vector product αt in Eq. (3.2) 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.

3.2.2 Deep Learning Code at the 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. 3.3–3.4). 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. 3.2.1), and the features for nonterminal nodes are not indicator-based occurrence counts (Sec. 3.1.1). 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. (3.5)–(3.7)). 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 [58] lies at the heart of our approach.

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 [3]. These constituents are called programming constructs, and a context-free grammar specifies the syntax of programming constructs [3]. The AST is one type of intermediate code that represents the hierarchical syntactic structure of a program [3]. 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 [3], suppose each AST node has a special attribute repr that stores a vector representation. This code[2] 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 [41] 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 [41], 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

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

Table 3.1: ast2bin Productions

Head	Body
⟨AnonymousClassDeclaration⟩	⟨ClassBodyElementList⟩
⟨ArgumentList⟩	⟨AnonymousClassDeclaration⟩ | ⟨Expression⟩ | ⟨Expression⟩ ⟨ArgumentList⟩
⟨ArrayCreation⟩	⟨ArrayType⟩ ⟨DimensionList⟩
⟨ArrayInitializer⟩	⟨ArrayInitializerList⟩
⟨ArrayInitializerList⟩	⟨Expression⟩ | ⟨Expression⟩ ⟨ArrayInitializerList⟩
⟨Block⟩	⟨StatementList⟩
⟨Branches⟩	⟨Expression⟩ ⟨Expression⟩ | ⟨Statement⟩ ⟨Statement⟩
⟨CatchClauseList⟩	⟨CatchClause⟩ | ⟨CatchClause⟩ ⟨CatchClauseList⟩
⟨ClassBodyElementList⟩	⟨ClassBodyElement⟩ | ⟨ClassBodyElement⟩ ⟨ClassBodyElementList⟩
⟨ClassInstanceCreation⟩	⟨ClassInstanceCreationHeader⟩ ⟨ArgumentList⟩
⟨ClassInstanceCreationHeader⟩	⟨Expression⟩ ⟨TypeList⟩
⟨CompilationUnit⟩	⟨CompilationUnitHeader⟩ ⟨TypeDeclarationList⟩
⟨CompilationUnitHeader⟩	⟨PackageDeclaration⟩ ⟨ImportDeclarationList⟩
⟨ConditionalExpression⟩	⟨Expression⟩ ⟨Branches⟩
⟨ConstructorInvocation⟩	⟨ArgumentList⟩
⟨DimensionList⟩	⟨Expression⟩ | ⟨Expression⟩ ⟨DimensionList⟩
⟨EnhancedForStatement⟩	⟨EnhancedForStatementHeader⟩ ⟨Statement⟩
⟨EnhancedForStatementHeader⟩	⟨FormalParameter⟩ ⟨Expression⟩
⟨ExpressionList⟩	⟨Expression⟩ | ⟨Expression⟩ ⟨ExpressionList⟩
⟨FieldDeclaration⟩	⟨FieldDeclarationHeader⟩ ⟨VariableDeclarationFragmentList⟩
⟨FieldDeclarationHeader⟩	⟨ModifierList⟩ ⟨Type⟩
⟨ForStatement⟩	⟨ForStatementHeader⟩ ⟨Statement⟩
⟨ForStatementHeader⟩	⟨Expression⟩ ⟨ExpressionList⟩
⟨IfStatement⟩	⟨Expression⟩ ⟨Branches⟩
⟨ImportDeclarationList⟩	⟨ImportDeclaration⟩ | ⟨ImportDeclaration⟩ ⟨ImportDeclarationList⟩
⟨InfixExpression⟩	⟨InfixExpressionList⟩
⟨InfixExpressionList⟩	⟨Expression⟩ | ⟨Expression⟩ ⟨InfixExpressionList⟩
⟨MemberValuePairList⟩	⟨MemberValuePair⟩ | ⟨MemberValuePair⟩ ⟨MemberValuePairList⟩
⟨MethodDeclaration⟩	⟨MethodDeclarationHeader⟩ ⟨Block⟩
⟨MethodDeclarationHeader⟩	⟨SignatureElementList⟩
⟨MethodInvocation⟩	⟨MethodInvocationHeader⟩ ⟨MethodInvocationBody⟩
⟨MethodInvocationBody⟩	⟨Identifier⟩ ⟨ArgumentList⟩
⟨MethodInvocationHeader⟩	⟨Expression⟩ ⟨TypeList⟩ | ⟨ClassName⟩ ⟨TypeList⟩
⟨ModifierList⟩	⟨ExtendedModifier⟩ | ⟨ExtendedModifier⟩ ⟨ModifierList⟩ | ⟨Modifier⟩ | ⟨Modifier⟩ ⟨ModifierList⟩
⟨NormalAnnotation⟩	⟨TypeName⟩ ⟨MemberValuePairList⟩
⟨ParameterizedType⟩	⟨TypeList⟩
⟨SignatureElementList⟩	⟨SignatureElement⟩ | ⟨SignatureElement⟩ ⟨SignatureElementList⟩
⟨SingleVariableDeclaration⟩	⟨VariableDeclarationHeader⟩ ⟨Expression⟩
⟨StatementList⟩	⟨Statement⟩ | ⟨Statement⟩ ⟨StatementList⟩
⟨SuperConstructorInvocation⟩	⟨SuperConstructorinvocationHeader⟩ ⟨ArgumentList⟩
⟨SuperConstructorinvocationHeader⟩	⟨Expression⟩ ⟨TypeList⟩
⟨SuperMethodInvocation⟩	⟨MethodInvocationHeader⟩ ⟨MethodInvocationBody⟩
⟨SwitchCaseItem⟩	⟨SwitchCase⟩ ⟨ExpressionList⟩
⟨SwitchCaseList⟩	⟨SwitchCaseItem⟩ | ⟨SwitchCaseItem⟩ ⟨SwitchCaseList⟩
⟨SwitchStatement⟩	⟨Expression⟩ ⟨SwitchCaseList⟩
⟨TryStatement⟩	⟨Block⟩ ⟨CatchClauseList⟩
⟨TypeDeclaration⟩	⟨TypeDeclarationHeader⟩ ⟨ClassBodyElementList⟩
⟨TypeDeclarationHeader⟩	⟨ModifierList⟩ ⟨TypeSignature⟩
⟨TypeDeclarationList⟩	⟨TypeDeclaration⟩ | ⟨TypeDeclaration⟩ ⟨TypeDeclarationList⟩
⟨TypeList⟩	⟨Type⟩ | ⟨Type⟩ ⟨TypeList⟩
⟨TypeSignature⟩	⟨Identifier⟩ ⟨TypeList⟩
⟨Variable⟩	⟨Type⟩ ⟨Identifier⟩
⟨VariableDeclarationExpression⟩	⟨VariableDeclarationHeader⟩ ⟨VariableDeclarationFragmentList⟩
⟨VariableDeclarationFragmentList⟩	⟨VariableDeclarationFragment⟩ | ⟨VariableDeclarationFragment⟩ ⟨VariableDeclarationFragmentList⟩
⟨VariableDeclarationHeader⟩	⟨ModifierList⟩ ⟨Variable⟩ | ⟨ModifierList⟩ ⟨Type⟩
⟨VariableDeclarationStatement⟩	⟨VariableDeclarationHeader⟩ ⟨VariableDeclarationFragmentList⟩

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:

$\langle \textit{IfStatement} \rangle ::= \langle \textit{Expression} \rangle \ \langle \textit{Branches} \rangle$

Table 3.2: Node Precedence

TypeDeclaration
MethodDeclaration
OtherType
ExpressionStatement
QualifiedName
SimpleType
SimpleName
ParenthesizedExpression
Block
ArtificialType

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

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 Blocks can have the form of a recursive list of statements:

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

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

Our complete set of productions is listed in Tab. 3.1.

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. 3.2 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.

- 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.

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. 3.3 (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. 3.3. We initialize this attribute for each terminal by using its lexical element to select the corresponding column in the matrix of embeddings

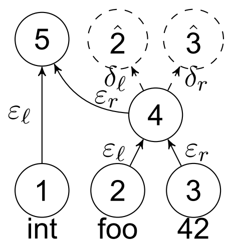

Figure 3.3: AST-based Encoding

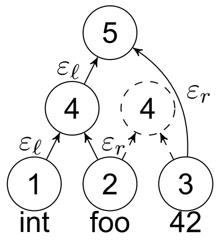

Figure 3.4: Greedy Encoding

α (Fig. 3.2). For example, if the lexical element int maps to the jth column of α, then repr for the PrimitiveType (1) in Fig. 3.3 is initialized such that $1.\text{repr} = \alpha_{\cdot j}$. This attribute is initialized to null for nonterminal nodes such as the VariableDeclarationFragment (4) and the VariableDeclarationStatement (5) in Fig. 3.3. At this juncture, we have used patterns mined at the lexical level (Sec. 3.2.1) to initialize a sequence of embeddings. Next, we use an autoencoder to combine embeddings. The canonical form of an autoencoder is a neural network with one input layer x, one hidden layer z, and one output layer y

$$z = g\left(\varepsilon x + \beta_z\right) \tag{3.5}$$

$$y = h\left(\delta z + \beta_y\right) \tag{3.6}$$

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; and $\beta_z \in \mathbb{R}^n$ 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. (3.2). The function g is a nonlinear vector function, and h is typically the identity function.

In Sec. 3.2.2, we claimed that our learner only accepts fixed-size inputs, prompting 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. 3.3, 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. (3.5), 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. 3.2.1). 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:

$$E(x_\ell, x_r; \varepsilon, \delta, \beta_z, \beta_y) = ||x_\ell - \hat{x}_\ell||_2^2 + ||x_r - \hat{x}_r||_2^2 \tag{3.7}$$

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. 3.3 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):

$$\text{5.repr} = g([\varepsilon_\ell, \varepsilon_r][\text{1.repr}; \text{4.repr}] + \beta_z)$$

As before, to train the model, we decode the representation (i.e., $y = h([\delta_\ell; \delta_r][\text{5.repr}] + \beta_y))$ and compare the reconstruction to the input (i.e., $x = [\text{1.repr}; \text{4.repr}]$) 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, the representation for the VariableDeclarationStatement (5), than the PrimitiveType (1). After computing the code for each nonterminal node in a forward pass, the backpropagation through structure algorithm [58] 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) [45]. 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. [187] 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" [3], 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 [3]. 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.

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.

Greedy Combinations for Clone Detection

Here we draw from an approach proposed by Socher et al. [187] 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. (3.7)) to encode first. For example, in Fig. 3.4, the first iteration derives two codes; the model does a better job at reconstructing $[1.\text{repr}; 2.\text{repr}]$ rather than the VariableDeclarationFragment $[2.\text{repr}; 3.\text{repr}]$. 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 [173]. On the other hand, for the greedy method, the clone granularity is generally “free,” i.e., it combines fragments without syntactic boundaries [173]. Second, training requires more computational resources for the greedy encoding than the AST-based encoding. 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.

3.3 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 [211]. Our study was from the point of view of software maintainers in the context of Ph.D. students and real-world Java systems [211]. 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 [32, 35, 99, 100, 192, 193, 201, 204]. Roy et al. [173] 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 [193]. 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. 3.3.2 and consolidate threats to the validity of our work in Sec. 3.5.

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.

RQ3 Are our representations suitable for detecting fragments that are similar with respect to a clone type?

RQ4 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 RQ3 and to synthesize qualitative data on code clones across two detection techniques for RQ4. Judging code clones is inherently difficult (even among experts [99, 201, 204]) because of imperfect definitions [35, 172] and the lack of oracles [204], so we developed a research instrument [175] to support consistent evaluations and control construct threats. We describe the guidelines used to manually examine candidates in Sec. 3.3.2.

3.3.1 Data Collection Procedure

Our subject systems included eight real-world Java systems (Tab. 3.3) used in previous studies [172]. We used ANTLR[3] to tokenize the source code and the RNNLM Toolkit [131] to train several RtNNs for each system, varying hidden layer sizes and depths [210]. We selected the highest quality model for each system, using PP (Chap. 2) as a proxy for quality, and extracted the matrix of embeddings (Fig. 3.2). Researchers have not established a correlation between intrinsic evaluation metrics such as PP 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. (3.2). For CAROL, our simplest system in terms of tokens and vocabulary size, we used 400.

Next, we used the Eclipse Java development tools[4] to build the AST for each file in every system. Each AST node[5] represents a programming construct, and we relied on the visitor design pattern[6] to traverse ASTs, identify nodes' types, and implement ast2bin (Sec. 3.2.2). 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

[3]http://www.antlr.org/
[4]http://www.eclipse.org/jdt/
[5]org.eclipse.jdt.core.dom.ASTNode
[6]org.eclipse.jdt.core.dom.ASTVisitor

Table 3.3: Subject Systems' Statistics

System	Files	LOC	Tokens	Vocab.
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

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. [82], setting minT to 50, stride to ∞, and similarity to 1.0, which correspond to standard choices in other tools.

3.3.2 Analysis Procedure

Research Question 3

After running the AST-based and greedy methods, the next step in the clone detection process [172, 173] (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 hyperparameters 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. [173] 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 [18, 19, 20, 190, 191]. 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 [72, 153, 177] have underscored several practical uses for file-level clone detection to include, inter alia, detecting similar projects and measuring third-party library reuse. Sec. 3.4 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 [175].

Research Question 4

RQ4 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 [173]. 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 [80], 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. 3.4 synthesizes qualitative data from the study.

Table 3.4: 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)

3.4 Empirical Results

Our RvNN implementation forked Socher et al. [187], which used L-BFGS [152] 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. 3.4 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. 3.4 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. 3.5 summarizes lessons learned from training these models on source code.

3.4.1 Research Question 3

Sampling candidates for each combining method and system, Tab. 3.5 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 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

Table 3.5: 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)

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. 3.5 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.

3.4.2 Research Question 4

For a traditional, structure-oriented technique, we selected Deckard [80]. 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.

Listing 3.1: Difference.java

```java
protected Collection<Resource> getCollection() {
  List<ResourceCollection> rcs = getResourceCollections();
  int size = rcs.size();
  if (size < 2) {
    throw new BuildException("The difference of " + size
      + " resource collection" + ((size == 1) ? "" : "s")
      + " is undefined.");
  }
  Set<Resource> hs = new HashSet<Resource>();
  List<Resource> al = new ArrayList<Resource>();
  for (ResourceCollection rc : rcs) {
    for (Resource r : rc) {
      if (hs.add(r)) {
        al.add(r);
      } else {
        al.remove(r);
      }
    }
  }
  return al;
}
```

Listing 3.2: Intersect.java

```java
protected Collection<Resource> getCollection() {
  List<ResourceCollection> rcs = getResourceCollections();
  int size = rcs.size();
  if (size < 2) {
    throw new BuildException("The intersection of " + size
      + " resource collection" + ((size == 1) ? "" : "s")
      + " is undefined.");
  }
  Iterator<ResourceCollection> rc = rcs.iterator();
  Set<Resource> s = new
       LinkedHashSet<Resource>(collect(rc.next()));
  while (rc.hasNext()) {
    s.retainAll(collect(rc.next()));
  }
  return s;
}
private Set<Resource> collect(ResourceCollection rc) {
  Set<Resource> result = new LinkedHashSet<Resource>();
  for (Resource r : rc) {
    result.add(r);
  }
  return result;
}
```

Figure 3.5: Replaced Control Statements in Apache Ant

Modified Control Flow

In ArgoUML 0.34, both GoNamespaceToDiagram and GoProjectToStateMachine extend the abstract class AbstractPerspectiveRule, which implements the interface PerspectiveRule. The interface specifies three methods: getRuleName, getChildren, and getDependencies; these are the only methods implemented in both classes. While getRuleName and getDependencies are Type I (micro) clones, getChildren has two different implementations. GoNamespaceToDiagram defines an ArrayList wrapped as a List, iterates through a list of Diagram objects, and performs checks before adding them to the list. On the other hand, GoProjectToStateMachine defines an ArrayList wrapped as a Collection and iterates through a list of Model objects, adding them to a collection. Our approach was robust against variations in syntax from the conditional statements. Deckard only reported similarities between the package declarations and import statements.

Replaced Control Statements

In Apache Ant 1.9.6, both Difference (List. 3.1) and Intersect (List. 3.2) extend BaseResourceCollectionContainer with their main functionality in the method getCollection (Fig. 3.5). The first eight lines of getCollection are Type I clones, but the classes differ on how

Listing 3.3: OnReplicateVisitor.java

```java
Object processCollection(Object collection,
      PersistentCollectionType type) throws
      HibernateException {
  SessionImpl session = getSession();
  Serializable key = getKey();
  CollectionPersister persister =
        session.getCollectionPersister( type.getRole() );
  session.removeCollection(persister, key);
  if ( collection!=null && (collection instanceof
        PersistentCollection) ) {
    PersistentCollection wrapper = (PersistentCollection)
          collection;
    wrapper.setCurrentSession(session);
    session.addNewCollection(wrapper);
  }
  else {
    // otherwise a null or brand new collection
    // this will also (inefficiently) handle arrays, which
    // have no snapshot, so we can't do any better
    //processArrayOrNewCollection(collection, type);
  }
  return null;
}
```

Listing 3.4: OnUpdateVisitor.java

```java
Object processCollection(Object collection,
      PersistentCollectionType type) throws
      HibernateException {
  SessionImpl session = getSession();
  Serializable key = getKey();
  CollectionPersister persister =
        session.getCollectionPersister( type.getRole() );
  if ( collection!=null && (collection instanceof
        PersistentCollection) ) {
    PersistentCollection wrapper = (PersistentCollection)
          collection;
    if ( wrapper.setCurrentSession(session) ) {
      CollectionSnapshot snapshot =
            wrapper.getCollectionSnapshot();
      if ( !SessionImpl.isOwnerUnchanged(snapshot,
            persister, key) ) {
        session.removeCollection(persister, key);
      }
      session.reattachCollection(wrapper, snapshot);
    }
    else {
      session.removeCollection(persister, key);
    }
  }
  else {
    // null or brand new collection
    // this will also (inefficiently) handle arrays, which
    // have no snapshot, so we can't do any better
    session.removeCollection(persister, key);
    //processArrayOrNewCollection(collection, type);
  }
  return null;
}
```

Figure 3.6: Reordered Data-dependent Statements in Hibernate

they populate the collection. Difference uses nested enhanced for-loops to iterate over the list of ResourceCollections, inspecting each resource in every collection, to calculate the difference between two or more nested ResourceCollections. Intersect uses an iterator, a while loop, and an enhanced for-loop to calculate the intersection of two or more nested ResourceCollections. Our learning-based paradigm detected the clone pair despite the classes using distinctly different control statements to loop over an iterable object. Deckard only detected similarities between the package declarations and import statements (not shown).

Reordered Data-dependent Statements

In Hibernate 2, for OnReplicateVisitor (List. 3.3) and OnUpdateVisitor (List. 3.4), Deckard detected similar fragments from the package declarations up to the method declaration for processCollection, the core method for each class (Fig. 3.6). Both classes extend the abstract class ReattachVisitor, and the required method processCollection is similar across

Listing 3.5: NonUniqueObjectException.java

```java
private final Serializable identifier;
private final Class clazz;
public NonUniqueObjectException(String message,
      Serializable id, Class clazz) {
  super(message);
  this.clazz = clazz;
  this.identifier = id;
}
public NonUniqueObjectException(Serializable id, Class
      clazz) {
  this("a different object with the same identifier value
        was already associated with the session", id,
        clazz);
}
public Serializable getIdentifier() {
  return identifier;
}
public String getMessage() {
  return super.getMessage() + ": " + identifier + ", of
        class: " + clazz.getName();
}
public Class getPersistentClass() {
  return clazz;
}
```

Listing 3.6: WrongClassException.java

```java
private final Serializable identifier;
private final Class clazz;
public WrongClassException(String msg, Serializable
      identifier, Class clazz) {
  super(msg);
  this.identifier = identifier;
  this.clazz = clazz;
}
public Serializable getIdentifier() {
  return identifier;
}
public String getMessage() {
  return "Object with id: " + identifier + " was not of
        the specified subclass: " + clazz.getName() + " ("
        + super.getMessage() + ")" ;
}
public Class getPersistentClass() {
  return clazz;
}
```

Figure 3.7: Overloaded Constructor in Hibernate

the implementations. We manually analyzed the two implementations and found that while the implementations are distinctly different, the methods are similar. One difference provides evidence that our learning-based approach is capable of handling reordered data-dependent statements: the method invocation session.removeCollection(persister, key) is placed in different positions relative to the first if statement.

Overloaded Constructor

In Hibernate 2, NonUniqueObjectException (List. 3.5) and WrongClassException (List. 3.6) have the same private fields and implement the same methods using similar syntax (Fig. 3.7). Discounting Type I and Type II variations, the few notable differences are reordered data-independent statements in the constructors, minor syntactic differences in getMessage, and one class overloads its constructor with an additional (one-line) method. Deckard did not report any similar fragments for this pair of files; however, Deckard did report clones between NonUniqueObjectException and UnresolvableObjectException, which overloads its constructor too.

Listing 3.7: MINFORecord.java

```java
private static final long serialVersionUID =
    -3962147172340353796L;
private Name responsibleAddress;
private Name errorAddress;
MINFORecord() {}
Record
getObject() {
  return new MINFORecord();
}
public
MINFORecord(Name name, int dclass, long ttl,
            Name responsibleAddress, Name errorAddress)
{
  super(name, Type.MINFO, dclass, ttl);
  this.responsibleAddress =
       checkName("responsibleAddress",
       responsibleAddress);
  this.errorAddress = checkName("errorAddress",
       errorAddress);
}
void
rrFromWire(DNSInput in) throws IOException {
  responsibleAddress = new Name(in);
  errorAddress = new Name(in);
}
void
rdataFromString(Tokenizer st, Name origin) throws
     IOException {
  responsibleAddress = st.getName(origin);
  errorAddress = st.getName(origin);
}
String
rrToString() {
  StringBuffer sb = new StringBuffer();
  sb.append(responsibleAddress);
  sb.append(" ");
  sb.append(errorAddress);
  return sb.toString();
}
public Name
getResponsibleAddress() {
  return responsibleAddress;
}
public Name
getErrorAddress() {
  return errorAddress;
}
void
rrToWire(DNSOutput out, Compression c, boolean canonical) {
  responsibleAddress.toWire(out, null, canonical);
  errorAddress.toWire(out, null, canonical);
}
```

Listing 3.8: SRVRecord.java

```java
private static final long serialVersionUID =
    -3886460132387522052L;
private int priority, weight, port;
private Name target;
SRVRecord() {}
Record
getObject() {
  return new SRVRecord();
}
public
SRVRecord(Name name, int dclass, long ttl, int priority,
          int weight, int port, Name target)
{
  super(name, Type.SRV, dclass, ttl);
  this.priority = checkU16("priority", priority);
  this.weight = checkU16("weight", weight);
  this.port = checkU16("port", port);
  this.target = checkName("target", target);
}
void
rrFromWire(DNSInput in) throws IOException {
  priority = in.readU16();
  weight = in.readU16();
  port = in.readU16();
  target = new Name(in);
}
void
rdataFromString(Tokenizer st, Name origin) throws
     IOException {
  priority = st.getUInt16();
  weight = st.getUInt16();
  port = st.getUInt16();
  target = st.getName(origin);
}
String
rrToString() {
  StringBuffer sb = new StringBuffer();
  sb.append(priority + " ");
  sb.append(weight + " ");
  sb.append(port + " ");
  sb.append(target);
  return sb.toString();
}
public int
getPriority() {
  return priority;
}
public int
getWeight() {
  return weight;
}
public int
getPort() {
  return port;
}
public Name
getTarget() {
  return target;
}
void
rrToWire(DNSOutput out, Compression c, boolean canonical) {
  out.writeU16(priority);
  out.writeU16(weight);
  out.writeU16(port);
  target.toWire(out, null, canonical);
}
public Name
getAdditionalName() {
  return target;
}
```

Figure 3.8: Inserted and Deleted Lines in dnsjava

Inserted and Deleted Lines

In dnsjava 2.0.0, both MINFORecord (List. 3.7) and SRVRecord (List. 3.8) extend the Record class and have a similar set of methods save a few additional getters since

SRVRecord has a few more private fields than MINFORecord (Fig. 3.8). Despite the syntactic differences between constructors and helper methods, there are evident similarities among the implementations. For example, both constructors execute a call to super before setting their private fields. Likewise, many of the helper methods have the same interface and are exclusively designed to set private fields. Deckard did not report any similar fragments between these classes yet Deckard did link some fragments of these classes to other files in the system.

Fragmentation

In JHotDraw 6, both draw/standard/ConnectionTool and jhotdraw/standard/ConnectionTool 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, splintering the files makes it difficult to detect these (strong) Type III clones.

3.5 Discussion

Internal Validity

We acknowledge the confounding configuration choice problem [204]. 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 viewpoint 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 [32]. We do not infer developer behavior from our results and understand that humans must be observed while using the approach [32]. 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 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.

3.6 Future Work

3.6.1 On Scaling Deep Learning for Clone Detection

Here, we draw from work in the machine learning community on semantic hashing [176] 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. (3.5)) as a nonlinear vector function. The function g is called an "activation" function [25], and there

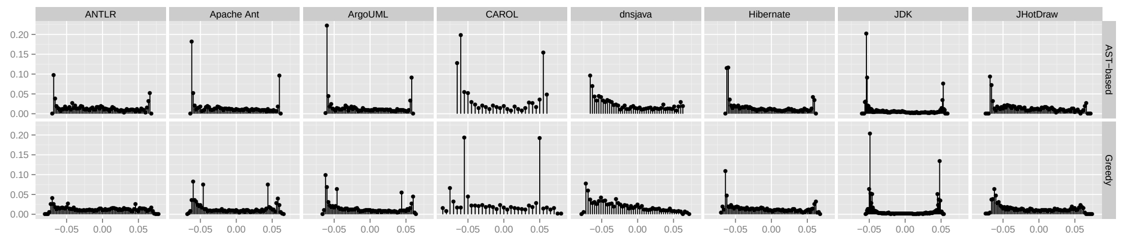

Figure 3.9: Relative Frequency Histograms of File-level Features

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. (3.5), would lie in the (approximately) linear part of tanh [64]. As the model trains, weights increase, drawing pre-activations away from zero and introducing nonlinearities [64]. When using tanh activations, weights may be directed positively or negatively away from zero. For instance, we initialized our RvNNs by sampling from (approximately) $\mathcal{U}$(-0.08,0.08) and used tanh activations. After the models were trained, we encoded each file in every system. Fig. 3.9 shows the relative frequency histograms of features; we used weight decay [64] 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 [103, 176]. 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 [103, 176]—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

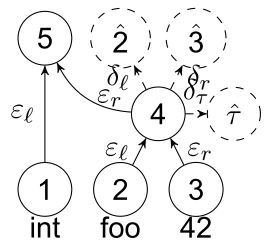

Figure 3.10: τ Decoder

measuring similarity with ℓ_2. To tune models for binary feature vectors, we plan to experiment with different learning heuristics. Salakhutdinov and Hinton [176] reported that semantic hashing (with their generative-based approach) was much faster than LSH in their experiments hashing natural language documents.

3.6.2 Type Prediction

Fig. 3.10 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. (3.5)). Socher et al. [187] 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. 3.10, if we present [2.repr; 3.repr] to the model, then we expect $\hat{\tau} =$ VariableDeclarationFragment. The only change to the criterion (Eq. (3.7)) is adding an expression to compute (cross-entropy) misclassification costs.

3.7 Summary

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 [173].

Chapter 4

Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities

In the field of automated program repair, many repair approaches are based on a common intuition: a patch can be composed of source code residing elsewhere in the repository or even in other projects. These approaches are called redundancy-based repair techniques, since they leverage redundancy and repetition in source code [13, 33, 34, 54, 68, 120, 147, 161]. For example, GenProg [60, 61, 106] reuses existing code from the same codebase, whereas CodePhage [182] transplants checks from one application to another. This intuition has been examined by at least two independent empirical studies, showing that a significant proportion of commits are indeed composed of existing code [13, 120].

The code that is reused to craft a patch is called a repair ingredient. Most redundancy-based repair techniques harvest repair ingredients at random and do not reason further about the optimal strategies to select repair ingredients. In other words, these repair techniques simply brute-force search for viable ingredients, randomly searching the codebase in a straightforward trial-and-error process. Although these techniques are able to find patches, their naive search and rigid application of repair ingredients means patches that use novel expressions are unattainable.

We aim to improve the reasoning about repair ingredients before they enter the repair pipeline (i.e., ingredient insertion, compilation, and test execution). Our key intuition is that a good ingredient does not come from just any location in the program but comes from similar code. We design our experiments to evaluate whether an approach built on this intuition can effectively improve patch generation.

We rely on deep learning to reason about code similarities. Our learning-based approach automatically creates a representation of source code that accounts for the structure and meaning of lexical elements. Our approach is completely unsupervised, and no top-down specification of features is made beforehand. We use the learned features to compute distances between code elements to measure similarities, and we use the similarities to intelligently select and adapt repair ingredients to generate a series of statement-level edits for program repair.

We design, implement, and evaluate DeepRepair, an approach for sorting and transforming program repair ingredients via deep learning code similarities. Our approach is implemented on top of Astor [117], an automatic software repair framework for Java. DeepRepair uses recursive deep learning [185] to prioritize repair ingredients in a fix space, and to select code that is most similar when choosing a repair ingredient to generate a patch. Additionally, DeepRepair addresses a major limitation of current redundancy-based repair techniques that do not adapt repair ingredients by automatically transforming ingredients based on lexical elements' similarities. For instance, given the simple ingredient

```
return FastMath.abs(y - x) <= eps;
```

the transformation may involve replacing the variable eps (which is out of scope at the modification point[1]) with the variable SAFE_MIN (which is in scope).

We assess the effectiveness of using code similarities to sort or transform repair ingredients for patch generation. To do so, we evaluate our approach along several dimensions to measure what aspects contribute and do not contribute to improved repair effec-

[1]Modification points are locations in the program where candidate patches would be applied.

tiveness. We compute a baseline; evaluate sorting (at different levels of granularity and scope) in isolation; evaluate transforming in isolation; and evaluate sorting with the ability to transform repair ingredients. We evaluate everything on six open-source Java projects including 374 buggy program revisions in Defects4J database version 1.1.0 [89, 88]. In summary, our evaluation consists of 19,949 trials spanning 2,616 days of computation time. DeepRepair's search strategy using code similarities generally found compilable ingredients (Sec. 4.2.3) faster than the baseline, jGenProg [116], but this improvement neither yielded test-adequate patches in fewer attempts (on average) nor found significantly more patches than the baseline. Although the patch counts were not statistically different, there were notable differences between the nature of DeepRepair's patches and jGenProg's patches (Sec. 4.4).

To sum up, we make the following noteworthy contributions:

- We introduce a novel deep learning-based approach to intelligently select and adapt repair ingredients for redundancy-based repair.
- We provide the implementation of our approach for Java in a publicly available tool.
- We introduce an evaluation protocol for redundancy-based repair, including a set of novel metrics that are specific to the analysis of ingredient selection strategies.
- We conduct an evaluation on 374 real bugs from the Defects4J benchmark showing that our algorithm finds patches that cannot be found by existing redundancy-based techniques.

Sec. 4.1 will review background on automated program repair and the redundancy assumption. This section will review two types of repair techniques, correct-by-construction and generate-and-validate. Generate-and-validate techniques rely on the redundancy assumption: large programs contain the seeds of their own repair [13, 120]. Sec. 4.2 will present our technical approach organized as a pipeline with three phases: language recognition, machine learning, and program repair. Sec. 4.3 will describe our empirical validation aimed at measuring effectiveness. Sec. 4.4 will present our results. Sec. 4.5 will discuss threats to the validity of our work. Sec. 4.6 summarizes the chapter.

4.1 Background and Related Work

4.1.1 Automated Program Repair

Automated program repair involves the transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification [139]. Behavioral repair techniques in particular change the behavior of a program under repair by changing its (source or binary) code [139]. For example, GenProg [60, 61, 106] changes the behavior of a program under repair according to a test suite (i.e., an input-output specification) by modifying the program's source code. This generate-and-validate technique searches for statement-level modifications to make to an AST. GenProg does not use symbolic reasoning. Indeed, GenProg does not reason about repair ingredients at all, since it selects ingredients from a pool of equiprobable statements.

A complimentary set of repair techniques leverage program analysis and program synthesis to repair programs by constructing code with particular properties [123, 124, 151, 212]. For example, SemFix [151] synthesizes (side-effect free) expressions for replacing the right-hand side of assignments or branch predicates to repair programs. Angelix [123] uses guided symbolic execution and satisfiability modulo theories (SMT) solvers to synthesize patches using angelic values (i.e., expression values that make a given test case pass). Nopol [212] either modifies an existing conditional expression or adds a precondition to a statement or block in the code.

Both types of techniques have distinct advantages. Generate-and-validate techniques have the advantage of operating at coarse granularity with the power to mutate statements, leveraging the intuition that using human-written code for fixes at coarse granularity leads to better quality patches. However, these techniques generally do not mutate code below statement-level granularity, so they do not change conditional expressions nor variables. Semantics-based techniques have the advantage of operating at fine granularity on expressions and variables, enabling them to synthesize a repair even if the patch code does not exist in the codebase [151]. However, they generally do not operate at higher levels of granularity, and scalability has been a key concern.

SearchRepair [93] draws from both generate-and-validate and semantics-based techniques. SearchRepair uses symbolic reasoning to search for code and semantic reasoning to generate candidate patches. Consequently, SearchRepair suffers the same scalability problems as semantics-based techniques and has only been demonstrated to work on small programs. In lieu of program semantics to search for similar code, we use a learning-based approach to query textually/functionally similar code at arbitrary levels of granularity. SearchRepair uses several software systems to repair small, student programs, whereas our approach uses the program under repair, and we aim to repair real software systems. SearchRepair depends on input-output examples to describe desired behavior, whereas we automatically learn features for distinguishing code fragments.

Recently, Yokoyama et al. [216] used code similarity to select code lines in code regions similar to the faulty code regions. Our work using code similarities differs in several important respects. They used small, fixed-sized code regions of 4, 6, and 8 lines, whereas we use code regions at arbitrary levels of granularity, making it possible to map a suspicious statement (i.e., a statement suspected to contain a bug [118]) to its method or class and query similar execution contexts or similar classes. Their similarity metric analyzed the longest common subsequence between two token sequences, whereas we use a learning-based clone detector that fuses information on structure and identifiers and is capable of finding more meaningful similarities than token-based techniques [209]. They used a collection of 24 bug-fix commits and defined a code coverage metric to evaluate their approach. We use a collection of 374 reproducible bugs and implemented our learning-based approach in an automatic software repair framework to measure effectiveness. Finally, our approach is capable of using similarities to transform repair ingredients.

Our approach relies in part on machine learning for the fix localization problem. Prophet [113] is a learning-based approach that uses explicitly designed code features to rank candidate repairs. We use representation learning [20, 107] to automatically learn how to encode fragments to detect similarities. Other approaches train on correct (student) solutions to specific programming tasks and try to learn task-specific repair strategies [24, 162]. For example, in massively open online courses, the programs are generally small and synthetic [63]. Finally, other approaches cannot transform statements. For example, Gupta et al. [63] use an oracle that rejects a fix if it does not preserve the identifiers and keywords present in the original statement. We use similarities to map the set of out-of-scope variables to a set of variables in scope.

4.1.2 Redundancy Assumption

Martinez et al. [120] empirically examined a critical assumption of GenProg that certain bugs can be fixed by copying and rearranging existing code. They validated the redundancy assumption by defining a concept of software temporal redundancy. A commit is temporally redundant if it is only a rearrangement of code in previous commits. They measured redundancy at two levels of granularity—line- and token-level granularity—and we draw key insights from their results at each level.

At line-level granularity, they found that most of the temporal redundancy is localized in the same file. Accordingly, we use a learning-based code clone detection approach, which is capable of detecting file-level clones, so the search will first look in the same file and similar files. Moreover, their token-level granularity results imply that many repairs need never invent a new token. Ergo, the tokens exist, but repair engines need to learn how to use them. Again, our key insight is to look to the learning-based code clone detection approach, which maps tokens to continuous-valued vectors called embeddings that we can use to measure similarities. Then we use the similarities to consider different tokens in different contexts.

We draw one more bit of insight from their empirical study. The authors note a tension between working with the line pool or the token pool [120], and they characterize this tension by the combination spaces of line- (smaller combination space) versus token-level (larger combination space) granularity. The essence of our work is to suppose there exists a manifold that is largely governed by coarse-grained snippets like lines yet can be parameterized by fine-grained snippets like tokens. This low-dimensional representation is where we will find repairs.

Barr et al. [13] examined a history of 15,723 commits to determine the extent to which the commits can be reconstructed from existing code. The grafts they found were mostly single lines, i.e., micro-clones, and they proposed that micro-clones are useful since they are the atoms of code construction [13]. The learning-based clone detection approach uses these micro-clones to compute the representation for fragments at higher levels of granularity, which we in turn use to assess similarities and determine which statements and values to assign to the source code modifications first. Their findings align with Martinez et al. [120] in that changes to a codebase contain snippets that already exist in the codebase at the time of the change.

4.2 Technical Approach

Our approach is organized as a pipeline comprising three phases: recognition, learning, and repair. The language recognition phase (Sec. 4.2.1) consumes source code of the application under repair and produces training data. The machine learning phase (Sec. 4.2.2) consumes the training data and produces encoders for encoding anything from a lexical element to a class such that similarities can be detected among the encodings. The program repair phase (Sec. 4.2.3) uses the encoders to query and transform code fragments in the codebase for patch generation. To explain our approach, we use the buggy program revision Math-63 (Revision ID: d2a5bc0) from the Defects4J database version 1.1.0 [89, 88] as a running example throughout this section.

4.2.1 Language Recognition Phase

Our approach begins with a program and its collection of source code files. For example, Math[2] is a library of lightweight, self-contained mathematics and statistics components. Math is a Maven[3] project with the standard directory layout. The src/main/java directory is the artifact producing source directory where the package hierarchy exists.

The first stage of our language recognition phase consumes this source directory and parses its contents to create an AST or any model for representing the code. There are no special requirements for the model provided its complete in the sense that it contains all the required information to derive compilable and executable programs. A well-formed, typed AST—an instance of a model—is sufficient since the visitor design pattern facilitates queries on the program under analysis such as the number of files, classes, and methods in the application sources. In our example, this stage produces a model for Math containing 459 files, 661 classes, and 4,983 methods.

The second stage of our language recognition phase consumes the model and uses a program processor to produce corpora at different levels of granularity. In this context, a program processor is simply a utility for performing a specific action. First, we create a file-level corpus by querying the model for all the files. Our program processor splays each file on its own line in the corpus by printing (from left to right) the terminal symbols [3] of the corresponding syntax tree. Simultaneously, we store keys that uniquely identify the Java source code files. Likewise, we use the program processor to build corpora at other levels of granularity, e.g., classes or methods, provided there is a way to uniquely identify the code elements. For example, Math files can be identified by their path; classes can be identified by their qualified name; and methods can be identified by concatenating the qualified name of their parent type and their signature. We build corpora at different levels of granularity because (in the next phase of our approach) we mine patterns at the level of classes, methods, and even lexical elements. These representations will be the

[2]http://commons.apache.org/proper/commons-math/
[3]https://maven.apache.org/

```
public static boolean equals(double x, double y) {
    return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
}
```

Figure 4.1: Suspicious Method in Math-63's MathUtils.java

mechanisms for querying and transforming repair ingredients.

The third stage of our language recognition phase normalizes the corpora by mapping some or all of the literal tokens to their respective type. For example, every floating point number in the Math corpus would be replaced by a generic symbol for floating point literal values. Normalizing the corpora is the last stage of recognition and staging data for learning.

4.2.2 Machine Learning Phase

The first stage of our machine learning phase involves training a neural network language model from the file-level corpus (Sec. 2.1.4). We require a neural network to learn representations for each term in the file-level corpus [21]. These models learn from the order of terms in a corpus, imputing continuous-valued vectors called "embeddings" to the terms in such a way that terms used in similar ways have embeddings that are close to each other in a feature space (Sec. 3.2.1). In this work, we use these embeddings to initialize the second stage of the learning phase.

The second stage of our machine learning phase involves training another learner to encode arbitrary streams of embeddings. We leverage work on recursive autoencoders [187, 188] and learning-based code clone detection (Chap. 3). To review the general, recursive learning procedure, consider a suspicious method in Math-63's class MathUtils (Fig. 4.1). Language recognition filters the method's return statement to the stream of terms in Fig. 4.2. Then our first machine learning stage maps the stream of terms to a stream of embeddings $x_\cdot \in \mathbb{R}^n$ indexed in Fig. 4.2 by nodes (0)–(7). There are seven pairs of adjacent terms. Each pair of adjacent terms are εncoded by agglutinating the embeddings $x = [x_\ell; x_r] \in \mathbb{R}^{2n}$ multiplying x by a matrix $\varepsilon = [\varepsilon_\ell, \varepsilon_r] \in \mathbb{R}^{n\times 2n}$ adding a

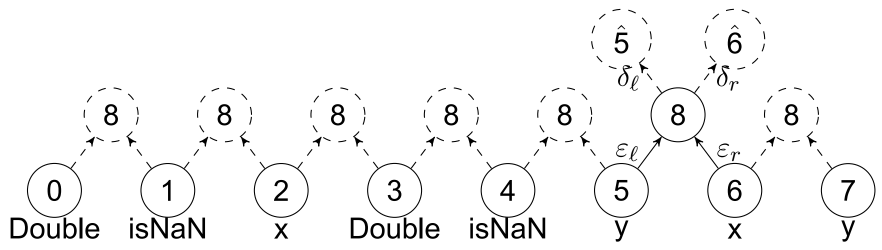

Figure 4.2: First Iteration of the Procedure to Encode a Stream of Lexical Elements

βias vector $\beta_z \in \mathbb{R}^n$ and passing the result to a nonlinear vector $f$unction f:

$$z = f\left(\varepsilon x + \beta_z\right) \tag{4.1}$$

For example, in Fig. 4.2, x_ℓ, x_r, and z may correspond to nodes (5), (6), and (8), respectively. The result z represents an encoding for the stream of two terms corresponding to x, e.g., "y x" in Fig. 4.2. Then z is δecoded by multiplying it by a matrix $\delta = [\delta_\ell; \delta_r] \in \mathbb{R}^{2n \times n}$ and adding a βias vector $\beta_y \in \mathbb{R}^{2n}$, i.e., $y = \delta z + \beta_y$. The output $y = [\hat{x}_\ell; \hat{x}_r] \in \mathbb{R}^{2n}$ is the model's reconstruction of the input (Chap. 3). As before (Sec. 3.2.2), training the model $\theta = \{\varepsilon, \delta, \beta_z, \beta_y\}$ involves measuring the Error between the original input vector x and the reconstruction y, i.e.,

$$E(x; \theta) = ||x_\ell - \hat{x}_\ell||_2^2 + ||x_r - \hat{x}_r||_2^2 \tag{4.2}$$

Concretely, the model is trained by minimizing Eq. (4.2). Training the model to encode streams with more than two terms requires recursively applying the autoencoder. To this end, we use the greedy procedure defined by Socher et al. [186], sketched in Sec. 3.2.2. In the first iteration of the procedure, each pair of adjacent terms are encoded (Fig. 4.2). The pair whose encoding yields the lowest reconstruction error (Eq. (4.2)) is the pair selected for encoding at the current iteration. For example, in Fig. 4.2, the pair "y x" is selected to be encoded first. As a result, in the next iteration, nodes (5) and (6) are replaced by node (8) and the procedure repeats. Upon deriving an encoding for the entire stream, the backpropagation through structure algorithm [58] computes partial derivatives of the (global) error function w.r.t. θ. Then the error signal is optimized using standard methods.

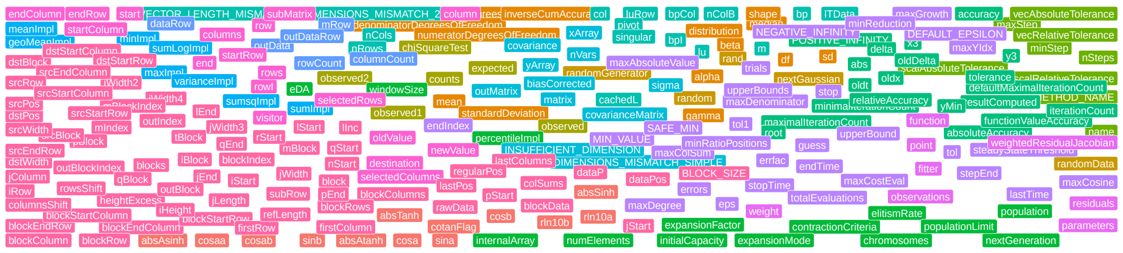

Figure 4.3: Sample of Math-63 identifiers' embeddings (best viewed in color) where positions have been jittered to avoid overplotting and colors correspond to clusters.

Our intent behind using this learning-based approach was manyfold. First, given that we aimed to assess the effectiveness of using similarities in sorting and transforming fix space elements for patch generation, this recursive learning procedure gave us the ability to evaluate similarity-based sorts at well-defined levels of granularity. The same model can be used to recognize similarities among classes, methods, or even identifiers. Second, the approach did not require intuition in the matter of engineering features for fragments since the approach automatically searched for empirically-based features.

The third and last stage of our machine learning phase involves clustering identifiers' embeddings. For example, we clustered Math-63 identifiers' embeddings and used t-SNE [199] to reduce their dimensionality $\mathbb{R}^n \to \mathbb{R}^2$. We plotted a handful of identifiers in Fig. 4.3 where terms with similar semantics appear to be proximate in the feature space. In this context, by similar "semantics," we mean terms are used in similar ways in the program (i.e., their token neighborhoods are similar). It is worth noting that the important distinction between inference and decision lies at the heart of the purpose for this learning stage. The learning phase induces models from source code, but the repair phase will require decisions on whether or not to transform ingredients. We cluster embeddings to operationalize a decision criterion. When transforming ingredients, the only identifiers that may replace an out-of-scope variable access are identifiers in its cluster (Sec. 4.2.3).

4.2.3 Program Repair Phase

Core Repair Loop

DeepRepair is based on a typical generate-and-validate repair loop à la GenProg. First, it begins in a traditional way by running fault localization to get a list of suspicious statements and their suspicious values. For example, running spectrum-based fault localization [1] on Math-63 yields eight statements with suspicious values greater than 0.1. These suspicious statements serve as modification points. For each modification point in sequence, a repair operator is used to transform that statement. Then, one tries to recompile the changed class (since repair operators do not guarantee a well-formed program after modification), and the tentative patch is validated against the whole test suite.

Repair Operators

In DeepRepair, the repair operators are addition of statements and replacement of statements. Contrary to Genprog and jGenProg [116], we do not use removal of statements because it generates too many incorrect patches [163]. Both addition and replacement are redundancy-based repair operators, since they need to select code from elsewhere in the codebase. In DeepRepair, the reused code are in ingredient pools, with three main pools, corresponding to whether the ingredient is in the same class as the modification point (local reuse), in the same package (package reuse), or anywhere in the codebase (global reuse).

DeepRepair departs from jGenProg on two fundamental points. First, the default jGenprog operators randomly pick one statement from the ingredient pool, whereas DeepRepair sorts the ingredients according to a specific criterion based on code similarity. Second, the default jGenprog operators reuse code “as is” at the modification point, without applying any transformation, so it could happen that the ingredient has variables that are not in the scope at the modification point (or wrongly typed), resulting in an uncompilable—obviously incorrect—candidate patch. On the contrary, DeepRepair has the ability to transform an ingredient so that it fits within the programming scope of the modification point. In DeepRepair, the sorting and transformation of ingredients are based on deep learning. A combination of a sorting and transformation technique is called a fix space navigation strategy. In this paper, we explore five novel navigation strategies implemented in DeepRepair.

Sorting Ingredients

When DeepRepair applies a repair operator, it sorts the available repair ingredients. DeepRepair prioritizes the ingredients that come from methods (resp. classes) that are similar to the method (resp. class) containing the modification point. For example, if the suspicious statement is the return statement in Fig. 4.1, then DeepRepair takes the parent method MathUtils::equals(double,double) and uses the method-level similarity list to sort methods in the codebase. Then, it extracts the statements from each similar method in order and enqueues them in a first-in first-out ingredient queue.

Transforming Ingredients

For a patch to be compiled, the ingredient must “fit” in the modification point in the sense that all variable accesses must be in scope. We refer to these “fit” ingredients as compilable ingredients. The key point of DeepRepair is its ability to transform ingredients, so a repair ingredient can be adapted to a particular context, resulting in uncompilable ingredients becoming compilable at the modification point. If a variable is out of scope, then DeepRepair examines the other identifiers in its cluster to determine whether any of them are in scope. For example, suppose a patch attempt consists of replacing the return statement in Fig. 4.1, and suppose we poll the following ingredient:

```java
return equals(x, y, 1) || FastMath.abs(y - x) <= eps;
```

The variable eps is out of scope at the modification point, but SAFE_MIN—a term in its cluster (Fig. 4.3)—is in scope, so we replace eps with SAFE_MIN in the ingredient. Applying this patch with a transformed ingredient yields a correct patch for Math-63 (Sec. 4.4.4). Transforming repair ingredients using clusters based on learned embeddings enables DeepRepair to create novel patches. These patches would be impossible to generate if only raw ingredients were used as done in jGenProg.

4.3 Empirical Validation

Redundancy-based program repair techniques search large repair spaces for fixes. One simplifying assumption in the theory of fix space navigation strategies is that fixes already exist somewhere in the codebase (Sec. 4.1.2). Invoking this redundancy assumption, in Sec. 4.3.1, we define our research questions, define the goal of our empirical study, establish the experimental baseline configuration, and state our hypotheses. Then we select the dependent variables and identify the independent variables that we change (i.e., the factors) as well as the independent variables we control at fixed levels. We conclude our plan with our experimental design: comparative experiments are characterized by treatments, experimental units (i.e., the objects to which we apply the treatments), and the responses that are measured. Next we describe our data collection procedures for each phase of our approach (Sec. 4.3.2). We conclude this section by specifying our analysis procedures for each one of our research questions (Sec. 4.3.3).

4.3.1 Experiment Scope and Plan

Our research questions included the following:

RQ5 Do code similarities based on deep learning improve fix space navigation as compared to a uniform random search strategy?

RQ6 Does ingredient transformation using embeddings based on deep learning effectively transform repair ingredients as compared to a default ingredient application algorithm that does not transform ingredients?

Table 4.1: Five DeepRepair Configurations Evaluated in Our Experiment

Code	Features
ED	(E)xecutable-level similarity ingredient sorting (D)efault ingredient application (no ingredient transformation)
TD	(T)ype-level similarity ingredient sorting (D)efault ingredient application (no ingredient transformation)
RE	(R)andom ingredient sorting (E)mbeddings-based ingredient transformation
EE	(E)xecutable-level similarity ingredient sorting (E)mbeddings-based ingredient transformation
TE	(T)ype-level similarity ingredient sorting (E)mbeddings-based ingredient transformation

RQ7 Does DeepRepair, our learning-based approach that uses code similarities and transforms repair ingredients, improve fix space navigation as compared to jGenProg [116]?

RQ8 Does DeepRepair generate higher quality patches than jGenProg?

The goal of our empirical study was to analyze ingredient search strategies for the purpose of evaluation with respect to effectiveness [211]. Our study was from the point of view of the software maintainer in the context of six open-source Java projects, a collection of (real) reproducible bugs, and a collection of JUnit test cases [211].

The baseline configuration, jGenProg, was the uniform random search strategy where ingredients were selected from a pool of equiprobable statements. We configured the baseline with a cache so the same modification instance (i.e., ingredient and operator instance) was never attempted more than once to improve its efficiency. The baseline was also configured with a default ingredient application algorithm that analyzed the variable accesses in a repair ingredient, matching accesses' names and types to variables in scope. If at least one variable access failed to match a variable in scope, then the ingredient was discarded. Tab. 4.1 lists the DeepRepair configurations.

We formalized our experiment into the following hypotheses:

H_0^{1a} Using code similarities generates the same number of test-adequate patches (on average) as jGenProg.

H_0^{1b} Using code similarities attempts[4] the same number of ingredients before finding a test-adequate patch as jGenProg.

H_0^{2a} Ingredient transformation using embeddings generates the same number of test-adequate patches as jGenProg.

H_0^{2b} Ingredient transformation using embeddings attempts the same number of ingredients before finding a test-adequate patch as jGenProg.

H_0^{3a} DeepRepair generates the same number of test-adequate patches as jGenProg.

H_0^{3b} DeepRepair attempts the same number of ingredients before finding a test-adequate patch as jGenProg.

H_0^{4} There is no significant difference in quality (i.e., correctness) between patches generated by DeepRepair and jGenProg.

We chose the following dependent variables: number of test-adequate patches and number of ingredients attempted. We chose the following independent variables (highlighting the factors): **ingredient search strategy**, **scope**, **clone granularity**, **variable resolution algorithm**, fault localization threshold, maximum number of suspicious candidates, and programming language. The fault localization threshold was fixed at 0.1; the maximum number of suspicious candidates was fixed at 1,000; and the language was fixed on Java. The random seed for each experimental configuration ranged from one to three.

We designed comparative experiments to measure the statistical significance of our results. Typically, empirical studies in the field report the number of test-adequate patches

[4]An attempt is defined to be a request sent to the fix space for an ingredient.

found, but we conducted a large-scale study, report the number of test-adequate patches found, and measure the statistical significance of differences to contextualize the results. For our quantitative study, the treatments in our experimental design were the ingredient search strategies. The experimental units were the buggy program revisions, and the responses were the number of test-adequate patches found and the number of ingredients attempted before finding a test-adequate patch. We consolidate threats to the validity of our work in Sec. 4.5.

4.3.2 Data Collection Procedure

The first stage of our recognition phase involved creating a model of source code (Sec. 4.2.1). We used Spoon [159], an open-source library for analyzing and transforming Java source code, to build a model for each buggy program revision.

Given the models, we implemented program processors for querying program elements at three levels of granularity: file-, type-, and executable-level granularity. Types included classes and interfaces, and executables included methods and constructors, but we omitted anonymous blocks. We only queried top-level types and executables to control the number of similarities to be computed. In this context, by “top-level,” we mean types or executables that did not have a parent type or executable, respectively. For example, a nested class would not be included, but its enclosing class may be included. Then we extracted the yield [3] from each program element’s syntax tree along with a key for uniquely identifying the element to build three corpora (Sec. 4.2.1). So each line of a type-level corpus corresponded to a class or interface in the program. The only normalization we performed was replacing character, float, integer, and string literal values with their respective type.

The first stage of our learning phase involved inducing neural network language models from the normalized file-level corpora (Sec. 4.2.2). We used word2vec [127, 132, 133] to learn embeddings for each buggy program revision. We selected word2vec over other architectures because the models can be trained quickly, and we only used the language models' embeddings to initialize the embeddings for the recursive autoencoders rather than randomly initialize them. We used the skip-gram model and set the size of the word vectors to 400 in accordance with previous studies using similar subject systems [209]. We set the maximum skip length between words to 10, used a hierarchical softmax to optimize the computation of output vectors' updates [169], and trained each model for 20 iterations. The language models enabled us to transform the file-level corpus for each program revision into streams of embeddings.

Then we trained recursive autoencoders to encode streams of embeddings. The encoders used tanh activations (i.e., $f ::=$ tanh in Eq. (4.1)), used L-BFGS [152] to optimize costs in batch mode, and trained for up to 50 epochs. After an encoder was trained on a revision's file-level corpus, we used it to encode every type and executable in the revision's type- and executable-level corpora, respectively. Given the encodings, we computed the pairwise Euclidean distance between each pair of types in the type-level corpus and each pair of executables in the executable-level corpus to measure similarities for each program revision.

Next, we extracted the term embeddings from the trained encoder and clustered them using k-means. For each revision, to determine k, we used simulated annealing, initializing both k and the temperature to be the square root of the corresponding vocabulary size. The reason we chose the square root of the vocabulary size is because of its effectiveness in related contexts that categorize words [131]. The objective we optimized was minimizing the number of points with negative silhouette values. Thus, at the end of the learning phase, each buggy revision had a cached list of executable- and type-level similarities as well as a categorization for identifiers.

The final phase of our technical approach involved automatically repairing buggy program revisions (Sec. 4.2.3). Our subject systems comprised six open-source Java projects including 374 buggy program revisions in Defects4J database version 1.1.0 [89, 88]. We could not build the Spoon model for Mockito bugs 1–21, which was likely because of

Table 4.2: Project Statistics

Project	Files	LOC	Tokens	Vocab.
Apache commons-lang	221	48,890	420,000	4,672
Apache commons-math	845	97,130	830,000	8,450
Closure compiler	937	247,300	1,449,000	26,490
JFreechart	952	130,300	921,800	9,008
Joda-Time	316	81,640	736,300	5,989
Mockito	680	44,990	309,500	5,735

missing or incompatible dependencies. In Tab. 4.2, we report median values since each project has several buggy program revisions.

To run program repair experiments, we used Astor [117], an automatic software repair framework for Java. Within the Astor framework, we leveraged GZoltar [30], a spectrum-based fault localization tool, to compute the Ochiai formula [1] for statements' suspicious values.

Each trial corresponded to a seeded treatment, which was a factorial of strategy and scope. Previous empirical studies indicated that fragment locality matters in software maintenance and evolution, so we analyzed three different levels of scope: local, package, and global [120]. For local scope, Astor builds the ingredient search space by amalgamating the distinct set of classes that contain at least one suspicious statement. For package scope, Astor computes the distinct set of packages that contain at least one suspicious statement and builds the ingredient search space using the set of classes in those packages. For global scope, Astor builds the ingredient space using all classes from the application under repair. We configured Astor to not stop at the first patch found and—starting from the original program—to continue searching for other patches until reaching a three-hour time limit. In total, we ran 20,196 trials (374 buggy program revisions $\times$ 6 configurations $\times$ 3 levels of scope $\times$ 3 random seeds) on subclusters comprising 64 compute nodes running Red Hat Enterprise Linux 6.2. Each compute node was a Dell PowerEdge C6100 serving two Intel Xeon X5672 quad-core processors at 3.2 GHz with 12 MB L3 cache and at least 48 GB of 1333 MHz main memory. Each trial was allocated two (hyper-threaded) cores and evolved one program variant for three hours using three repair operators: InsertAfterOp, InsertBeforeOp, and ReplaceOp. We did not include RemoveOp in our operator space because we only focused on repair operators that reuse code. We also wanted to guard against meaningless patches that simply remove functionality.

4.3.3 Analysis Procedure

Research Question 5

We analyzed the effectiveness of fix space navigation strategies in two parts. The first part analyzed effectiveness at generating test-adequate patches. We used the non-parametric Wilcoxon test with a Bonferroni correction to compare the number of test-adequate patches using jGenProg versus the strategy using code similarities at executable- and type-level granularity. We used Wilcoxon since the test-adequate patch counts could be paired, and we used a Bonferroni correction since we performed several tests simultaneously at different levels of scope and strategy. To complement our statistical analysis on number of test-adequate patches found, we also computed the difference between the set of jGenProg patches and the set of DeepRepair patches. Specifically, if D is the set of DeepRepair patches, and J is the set of jGenProg patches, then we compute and report $|D \setminus J| \,/\, |D|$, the percentage of DeepRepair patches that were not found by jGenProg. The second part analyzed the number of attempts to generate test-adequate patches. We used the non-parametric Mann-Whitney test with a Bonferroni correction to compare the number of attempts to generate test-adequate patches using jGenProg and using code similarities at executable- and type-level granularity. To complement our analysis on number of attempts, we also plotted the number of attempts to generate compilable ingredients.

Research Question 6

We analyzed effectiveness in two parts. We used the Wilcoxon test with a Bonferroni correction to compare the number of test-adequate patches using jGenProg versus the uniform random search strategy with the embeddings-based ingredient transformation algorithm. The algorithm gives jGenProg the ability to transform repair ingredients containing variable accesses that are out of scope. We also computed the difference between the set of jGenProg patches and the set of DeepRepair patches. Additionally, we used the Mann-Whitney test with a Bonferroni correction to compare the number of attempts to generate test-adequate patches and plotted the number of attempts to generate compilable ingredients.

Research Question 7

Our experimental design for RQ7 was virtually identical to our design for RQ5 except here we compared jGenProg to the strategy using code similarities with the embeddings-based variable resolution algorithm.

Research Question 8

We used correctness as a proxy for quality. Three judges evaluated the same random sample of 30 (15 jGenProg and 15 DeepRepair) patches to assess correctness. Martinez et al. [119] defined the correctness of a patch to be one of three values: correct, incorrect, or unknown. Correct denotes a patch is equivalent (according to the judge's understanding) to the human-written patch. Judges were also prompted for their confidence in their correctness rating where confidence was one of four values: high, moderate, slight, and none. Additionally, for reproducibility, judge's also assessed the readability of each patch, where the readability of a patch was either easy, medium, or hard in accordance with previous studies on patch correctness [116, 164]. We define readability to be the subjective qualification of how easily the patch can be understood. Readability is a subjective aggregation of different quantitative metrics: patch size in number of lines, number of involved variables, number of method calls, and the types of AST elements being inserted or replaced. There is no accepted quantitative aggregation of these metrics, and we think that a quantitative aggregation would be project- and even bug-dependent. In addition to the random sample, judges also evaluated the patches generated by jGenProg that were not found by DeepRepair and patches generated by DeepRepair that were not found by jGenProg.

4.4 Empirical Results

We ran 20,196 repair trials, 247 of which were killed. The 19,949 trials that finished took a total of 2,616 days of computation time. Zero patches were found for Closure, Mockito, and Time bugs. The baseline configuration found test-adequate patches for 48 different bugs. Six other bugs were unlocked by DeepRepair configurations. The trials found 19,832 different test-adequate patch instances and attempted 406,443,249 ingredients.

4.4.1 Research Question 5 (Analysis of ED and TD Strategies)

Tab. 4.3 lists the bugs for which test-adequate patches were found at (L)ocal, (P)ackage, and (G)lobal scope. jGenProg found test-adequate patches for 48 bugs. The treatments ED and TD found patches for 40 and 38 bugs, respectively.

Since many configurations found more than one patch for the same bug, we compared the patch counts between jGenProg and the two treatments ED and TD to see whether one configuration was more productive than another. We failed to reject the null hypothesis H_0^{1a} at each level of scope. Next, we analyzed the sets of patches, and we observed that approximately 99%, 25%, and 36% of DeepRepair's patches for Chart, Lang, and Math were not found by jGenProg. This result means that DeepRepair finds alternative patches.

We also counted the number of attempts needed to find each patch. Fig. 4.4 shows descriptive statistics for the number of attempts to find a test-adequate patch. We failed

Table 4.3: Patches Found at (L)ocal, (P)ackage, and (G)lobal Scope

ProjectID	BugID	jGenProg	ED	TD	RE	EE	TE
Chart	1	LPG	LPG	LPG	LPG	LPG	LPG
	3	LPG	LPG	LPG	LPG	LPG	LPG
	5	LPG	LPG	LPG	LPG	LPG	LPG
	7	LPG	LPG	LPG	LPG	LPG	LPG
	9	-	-	-	-	P	-
	12	G	LPG	LPG	G	LPG	LPG
	13	LPG	LPG	LPG	LPG	LPG	LPG
	14	LPG	LPG	LPG	LPG	LPG	LPG
	15	LPG	LPG	LPG	LPG	LPG	LPG
	18	L	-	-	L	-	-
	25	LPG	LPG	LPG	LPG	LPG	-
	26	LPG	LPG	LPG	LPG	LPG	LPG
Lang	7	LPG	LPG	LPG	LPG	LPG	LPG
	10	LPG	LPG	LPG	LPG	LPG	LPG
	20	LPG	LPG	LPG	LPG	LPG	LPG
	22	LPG	LPG	LPG	LPG	LPG	LP
	24	LPG	LPG	L	LPG	L	L
	27	LPG	LPG	LPG	LPG	LPG	LPG
	38	PG	-	-	PG	-	-
	39	LPG	LPG	LPG	LPG	L	-
Math	2	LPG	LPG	LPG	LPG	LPG	LPG
	5	LPG	LPG	LPG	LPG	LPG	LPG
	6	LP	LPG	LPG	LP	LP	G
	7	L	-	-	-	-	-
	8	-	-	-	G	LPG	LPG
	18	L	-	-	P	-	-
	20	LPG	LPG	LPG	LPG	LPG	LPG
	22	LPG	LPG	-	LPG	LPG	-
	24	-	-	-	LP	-	-
	28	LPG	LPG	LPG	LPG	LPG	LPG
	32	L	LPG	-	LP	-	-
	40	LPG	LPG	LP	LPG	LPG	L
	44	P	-	-	-	-	-
	49	LPG	LPG	LPG	LPG	LPG	LPG
	50	LPG	LPG	LP	LPG	LP	LP
	53	LPG	LPG	LPG	LPG	LPG	LPG
	56	L	LPG	LP	LPG	LP	-
	57	G	L	LP	G	-	LP
	58	-	-	-	LP	-	-
	60	G	LP	LPG	G	LP	LP
	63	-	-	-	LPG	LPG	LPG
	64	L	-	-	-	-	-
	70	LPG	LPG	LPG	LPG	LPG	LPG
	71	LPG	-	-	LPG	-	-
	73	LPG	LPG	LPG	LPG	LPG	LPG
	74	PG	-	-	PG	-	-
	77	LPG	L	LPG	LPG	L	LPG
	78	LPG	LPG	LPG	LPG	LPG	LPG
	80	LPG	LPG	LPG	LPG	P	LPG
	81	LPG	LPG	LPG	LPG	LPG	LPG
	82	-	-	-	-	PG	G
	84	LPG	LP	LP	LPG	LP	LP
	85	LPG	LPG	LPG	LPG	LPG	LPG
	98	LPG	LPG	LPG	LPG	LPG	LPG
Total	54	48	40	38	49	42	38

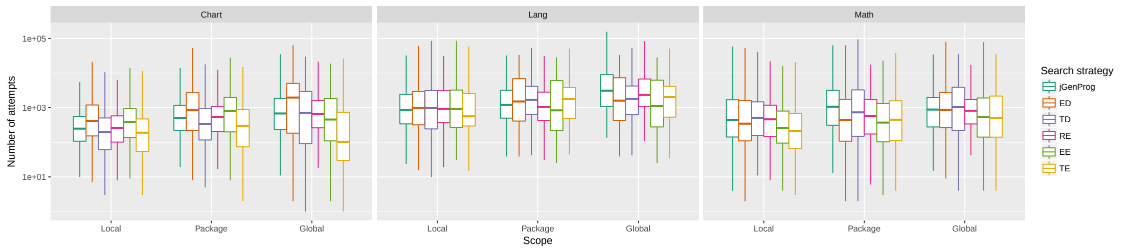

Figure 4.4: Number of Attempts to Find a Test-adequate Patch

to reject the null hypothesis H_0^{1b} at each level of scope. We also counted the number of attempts needed to find each compilable ingredient (as defined in Sec. 4.2.3). Fig. 4.5 shows descriptive statistics for the number of attempts to find a compilable ingredient at each level of scope for jGenProg, ED, and TD. Generally, sorting the fix space using code similarities results in fewer attempts before finding a compilable ingredient.

Key result. DeepRepair's search strategy using code similarities generally finds compilable ingredients faster than jGenProg, but this improvement neither yields test-adequate patches in fewer attempts (on average) nor finds significantly more patches than jGenProg. Yet there were notable differences between DeepRepair and jGenProg patches.

4.4.2 Research Question 6 (Analysis of RE Strategy)

The treatment RE found test-adequate patches for 49 bugs. Comparing the patch counts, we failed to reject the null hypothesis H_0^{2a} at each level of scope. Analyzing the set difference, 53%, 3%, and 53% of DeepRepair's patches for Chart, Lang, and Math were not found by jGenProg. We also failed to reject the null hypothesis H_0^{2b} at each level of scope. This result is illustrated in Fig. 4.4, which shows descriptive statistics for the number of attempts to find a test-adequate patch for jGenProg and RE.

Key result. DeepRepair's search strategy using the embeddings-based ingredient transformation algorithm neither yields test-adequate patches in fewer attempts (on average) nor finds significantly more patches than jGenProg, but there were notable differences between DeepRepair and jGenProg patches.

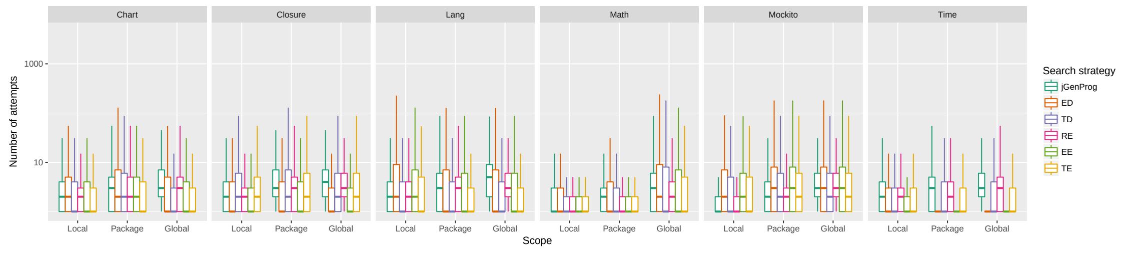

Figure 4.5: Number of Attempts to Find a Compilable Ingredient

4.4.3 Research Question 7 (Analysis of EE and TE Strategies)

The treatments EE and TE found test-adequate patches for 42 and 38 bugs, respectively. Comparing the patch counts, we failed to reject the null hypothesis H_0^{3a} at each level of scope. Analyzing the set difference, 99%, 28%, and 51% of DeepRepair's patches for Chart, Lang, and Math were not found by jGenProg. Fig. 4.4 shows descriptive statistics for the number of attempts to find a test-adequate patch for jGenProg, EE, and TE. We failed to reject the null hypothesis H_0^{3b} at each level of scope. Fig. 4.5 shows descriptive statistics for the number of attempts to find a compilable ingredient at each level of scope for jGenProg, EE, and TE.

Key result. DeepRepair's search strategy using code similarities with the embeddings-based ingredient transformation algorithm generally finds compilable ingredients faster than jGenProg, but this improvement neither yields test-adequate patches in fewer attempts (on average) nor finds significantly more patches than jGenProg. Once more, the DeepRepair configurations appear to find a complementary set of patches.

4.4.4 Research Question 8 (Manual Assessment)

After independently evaluating each sample patch, three judges discussed and resolved conflicts in terms of correctness. Five DeepRepair patches and five jGenProg patches were evaluated to be correct. We failed to reject the null hypothesis H_0^4. Although judges did notice differences in the patches generated by the approaches, no significant difference in readability was reported.

In addition to the random sample, we manually examined the following specific sets of patches. There were three bugs patched exclusively by jGenProg. We examined jGenProg's patches for these bugs and found that all of them were clearly incorrect. On the other hand, there were six bugs patched exclusively by DeepRepair configurations. We also examined these patches and report some of our findings below.

Chart-9. The human-written patch was

```
--- a/data/time/TimeSeries.java
+++ b/data/time/TimeSeries.java
@@ -941,7 +941,7 @@ public class TimeSeries extends Series implements Cloneable,
    Serializable {
                endIndex = -(endIndex + 1); // this is first item AFTER end period
                endIndex = endIndex - 1;    // so this is last item BEFORE end
            }
+           if ((endIndex < 0) || (endIndex < startIndex)) {
-           if (endIndex < 0) {
                emptyRange = true;
            }
            if (emptyRange) {
```

None of the identifiers in the human-written patch were new (cf. Sec. 4.1.2), but the conditional expression (endIndex < 0) || (endIndex < startIndex) was novel w.r.t. the codebase, so generate-and-validate techniques that cannot generate new code would never find this patch. The selection statement that DeepRepair generated passed the test suite, but it was incorrect (to the best of our knowledge). Notably, the DeepRepair patch contained the expression (endIndex < 1) || (endIndex > LAST_WEEK_IN_YEAR) with a syntactic structure that resembled the human-written expression. DeepRepair's conditional expression was novel w.r.t. the codebase as it was generated by transforming an ingredient. It was originally (result < 1) || (result > LAST_WEEK_IN_YEAR) but DeepRepair recognized similarities in how the identifiers, result and endIndex, were used in the codebase, so it replaced result—a variable out of scope—with endIndex.

Math-58. The human-written patch was

```
--- a/optimization/fitting/GaussianFitter.java
+++ b/optimization/fitting/GaussianFitter.java
@@ -118,7 +118,7 @@ public class GaussianFitter extends CurveFitter {
          */
         public double[] fit() {
             final double[] guess = (new ParameterGuesser(getObservations())).guess();
+            return fit(guess);
_            return fit(new Gaussian.Parametric(), guess);
         }

         /**
```

The patched class, GaussianFitter.java, fits a Gaussian to observations. As before, none of the identifiers in the human-written patch were new, but the statement was novel w.r.t. the codebase. DeepRepair generated the following patch:

```
--- a/optimization/general/LevenbergMarquardtOptimizer.java
+++ b/optimization/general/LevenbergMarquardtOptimizer.java
@@ -159,6 +159,7 @@
             }
         }
         double coeff1 = 0;
+        previousCost = FastMath.max(previousCost, lmPar);
         for (int j = 0; j < (solvedCols); ++j) {
             coeff1 += (work1[j]) * (work1[j]);
         }
```

The patched class, LevenbergMarquardtOptimizer.java, solves a least squares problem using the Levenberg-Marquardt algorithm. The statement inserted in the DeepRepair patch was a novel statement w.r.t. the codebase. The original ingredient was taken from the same file, but it contained a variable access that was out of scope at the modification point. The ingredient was transformed by substituting previousCost, a proximate term in scope, for the out-of-scope variable. The result was a patch that set the variable previousCost equal to the maximum of its previous value or the Levenberg-Marquardt parameter. The root cause in the failing test case was GaussianFitter throws a NonStrictlyPositiveException when the standard deviation is not a positive number. GaussianFitter uses LevenbergMarquardtOptimizer, but we expect the patch is incorrect. The readability is hard since it involves a nontrivial optimization procedure.

Math-63. The human-written patch was

```
--- a/util/MathUtils.java
+++ b/util/MathUtils.java
@@ -414,7 +414,7 @@ public final class MathUtils {
      * @return {@code true} if the values are equal.
      */
     public static boolean equals(double x, double y) {
+        return equals(x, y, 1);
-        return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
     }

     /**
```

Again, none of the identifiers in the human-written patch were new, but the statement was novel w.r.t. the codebase. DeepRepair generated the following patch:

```
--- a/util/MathUtils.java
+++ b/util/MathUtils.java
@@ -181,7 +181,7 @@
     }

     public static boolean equals(double x, double y) {
-        return ((Double.isNaN(x)) && (Double.isNaN(y))) || (x == y);
+        return (equals(x, y, 1)) || ((FastMath.abs((y - x))) <= (SAFE_MIN));
     }

     public static boolean equalsIncludingNaN(double x, double y) {
```

The ingredient was selected from a similar method:

```
/**
 * Returns true if both arguments are equal or within the range of allowed
 * error (inclusive).
 *
 * @param x first value
 * @param y second value
 * @param eps the amount of absolute error to allow.
 * @return {@code true} if the values are equal or within range of each other.
 */
public static boolean equals(double x, double y, double eps) {
    return equals(x, y, 1) || FastMath.abs(y - x) <= eps;
}
```

But the variable eps in the ingredient was not in scope at the modification point. DeepRepair recognized similarities between how the identifiers eps and SAFE_MIN are used in the codebase, so it replaced eps with SAFE_MIN. As a result, both the human-written patch and DeepRepair's patch invoke equals(x, y, 1) which returns true if x and y are equal or within the range of allowed errors (inclusive). In this case, the range of allowed error is defined to be zero floating point numbers between the two values, so the values must be the same floating point number or adjacent floating point numbers. Therefore, when equals(x, y, 1) is true, both the human-written patch and the DeepRepair patch return true since the conditional expression in the DeepRepair patch short-circuits. When equals(x, y, 1) is false, the human-written patch returns false, and since SAFE_MIN is defined to be the smallest normalized number in IEEE 754 arithmetic (i.e., 0x1.0p-1022), the DeepRepair patch returns false. Hence, the DeepRepair patch is semantically equivalent to the human patch and considered correct.

Key result. There are apparent, critical differences observed in DeepRepair's patches compared to jGenProg, which unlock new bugs—that would otherwise have not been patched—by reusing and transforming repair ingredients. Our future work aims to extensively analyze more results to understand which defect classes can be unlocked with DeepRepair's search strategies.

4.5 Threats to Validity

Internal Validity

DeepRepair relies on deep learning to compute similarities among code elements at different levels of granularity. Learning-based code clone detection has been evaluated at multiple levels of granularity with promising results (Chap. 3). Additionally, we manually examined small random samples of similar code fragments at executable- and type-level granularity from each project to validate some degree of textual/functional similarity in accordance with previous studies. However, we do not claim to have used optimal settings for training each program revision or even each project. We also acknowledge the confounding configuration choice problem [204]. We did not adopt arbitrary configurations and tried to justify each configuration in our approach.

External Validity

In our experiments we evaluate DeepRepair on 374 buggy program revisions (in six unique software systems) from the Defects4J benchmark. One threat is that the number of bugs may not be large enough to represent the actual differences between DeepRepair and jGenProg. While more systems with real bugs can help mitigate this threat, we would like to point out that a subset of Defects4J programs was used not only in program repair experiments [116, 212] but also recent testing studies [76, 89]. Moreover, we used a subset of programs and bugs from Defects4J that were used in prior experiments and made an effort to include new bugs for Mockito and Closure. Hence, our experiment is de facto the largest program repair experiment for Java.

Construct Validity

Our empirical evaluation is similar to all previous studies on program repair in that the programs under repair must be repaired at one single location. DeepRepair and jGenProg do not target buggy program revisions with multiple faults. Also, prior work noted some impact of flaky test cases on program repair [116]. If the failing test is flaky, the repair technique may conclude that a correct patch has been found. If a passing test case is flaky, the repair technique may conclude that a patch has introduced a regression while it may not necessarily be the case, which in turn may result in an underestimation of the performance of a repair technique. While we did not detect any flaky tests in our benchmarks during the experiments, their potential presence could impact both DeepRepair and jGenProg.

Conclusion Validity

DeepRepair and jGenProg have random components. For example, jGenProg chooses the statements and mutations randomly based on rules during the search. To this end, it is possible that different runs of DeepRepair and jGenProg would produce somewhat different patches. However, our experiments are as computationally extensive as they could have potentially been within our means, consisting of 19,949 trials spanning 2,616 days of computation time. Finally, running several trials of DeepRepair and jGenProg on every buggy revision generates dozens of different patches. Thus, we could not manually analyze all the generated patches (as this would require years of manual work); however, we randomly sampled a subset for manual evaluation. In order to minimize bias, three authors inspected randomly sampled patches generated by DeepRepair and jGenProg.

4.6 Summary

We introduced a novel learning-based algorithm to intelligently select and transform repair ingredients in a generate-and-validate repair loop based on the redundancy assumption (Sec. 4.1.2). DeepRepair takes a novel perspective on the ingredient selection problem: it selects ingredients from similar methods or classes where similarity has been inferred with deep unsupervised learning. Moreover, many repairs need the usage of a new token in order to make an ingredient compilable at a specific modification point. DeepRepair is the first approach ever, to the best of our knowledge, that transforms repair ingredients using identifiers' similarities in order to expand the search space. We conducted a computationally intensive empirical study, and found that DeepRepair did not significantly improve effectiveness using a new metric (number of attempts), but DeepRepair did generate many patches that cannot be generated by existing redundancy-based repair techniques.

Chapter 5

Conclusion

Software repositories are extraordinarily complex. Consider the complex domain analyses, intricate designs, mazelike implementations, and ambiguous reports that are produced and archived during the software development lifecycle. Reconciling (concrete) artifacts and (abstract) concepts to support SE tasks is the essence of SE research. However, there is an apparent discrepancy between the expressiveness that is engineered and submitted in repositories and the representation power of models like simple Markov models for reaping this information from repositories.

This dissertation makes three main contributions:

Chapter 2 We introduce deep learning to SE research. Deep learning will provide the SE community with new ways to mine and analyze artifacts to support SE tasks. Our empirical study demonstrates that deep learning induces high-quality software language models compared to state-of-the-practice models using an intrinsic evaluation metric. Then we demonstrate its effectiveness at code suggestion using a set of Java projects downloaded from GitHub.

Chapter 3 We introduce a new paradigm for code clone detection, an important problem for software maintenance and evolution. Many clone detection 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. Our novel learning-based approach to code clone detection fuses information on structure and identifiers and uses the data in repositories to automate the step of specifying how to represent source code. Our empirical validation asks and analyzes the question of whether our representations are suitable for detecting similar code fragments. We found that our learning-based approach finds pairs mapping to all four clone types and detects clones that were either undetected or suboptimally reported by a prominent clone detection tool.

Chapter 4 We introduce a novel deep learning-based approach to intelligently select and adapt program repair ingredients for redundancy-based repair. Our approach is the first to reason on the fix space by sorting ingredients using textual/functional similarities and the first to transform repair ingredients using identifiers' similarities, thereby expanding the search space. We also introduce an evaluation protocol for redundancy-based repair, including a set of novel metrics that are specific to the analysis of ingredient selection strategies. We conducted a large-scale empirical study on 374 real, reproducible bugs that spanned 2,616 days of computation time. Our algorithm found patches that cannot be found by existing redundancy-based techniques. Indeed, sorting and transforming program repair ingredients using code similarities enables patches that would be impossible to generate if restricted to intact ingredients in the codebase.

In summary, our work on code suggestion, code clone detection, and automated program repair has introduced new ways to mine and analyze artifacts for SE tasks such as feature location [40], defect prediction [203], cross-lingual question retrieval [36], generating API usage sequences for a given natural language query [62], fixing common C language errors [63], and fault localization [75], among many others. Deep learners remain one hypothesis set, but novel ways to use the improved representation power will give way to elegant, yet practical and impactful, applications in the future in SE research.

Appendix A

Intermediate Code Statistics

Table A.1: Intermediate Code Metrics

Metric	Description
ASTChildren	Number of children of nonterminal nodes
ASTChildrenMoreThan2	Number of children of nonterminal nodes having more than two children
ASTMaxDepth	Maximum number of edges from a node to the AST root
ASTNonTerminal	Number of nonterminal nodes
ASTNonTerminal1Child	Number of nonterminal nodes with degree one
ASTNonTerminalMoreThan2	Number of nonterminal nodes with degree greater than two
ASTTerminal	Number of terminal nodes
ASTUniqueNonTerminal	Number of different nonterminal nodes
FBTArtificialNonTerminal	Number of nonterminals generated by the modified grammar
FBTMaxDepth	Maximum number of edges from a node to the FBT root
FBTNonTerminal	Number of nonterminal nodes
FBTTerminal	Number of terminal nodes (after merging adjacent, identical literals with the same parent)
FBTUniqueNonTerminal	Number of unique nonterminal nodes

Table A.2: Intermediate Code Statistics

System	Metric	min	q_1	θ	μ	q_3	max
ANTLR 4	ASTChildren	1	1	2	2	3	4,009
	ASTChildrenMoreThan2	3	3	3	5	3	4,009
	ASTMaxDepth	2	7	8	10	11	26
	ASTNonTerminal	2	40	110	465	420	9,559
	ASTNonTerminal1Child	1	13	41	183	152	4,597
	ASTNonTerminalMoreThan2	0	6	20	124	80	4,440
	ASTTerminal	2	43	130	590	496	14,130
	ASTUniqueNonTerminal	2	15	18	20	26	50
	FBTArtificialNonTerminal	0	16	52	289	190	9,076
	FBTMaxDepth	1	11	19	66	37	4,040
	FBTNonTerminal	1	42	121	568	476	13,980
	FBTTerminal	2	43	122	569	477	13,990
	FBTUniqueNonTerminal	1	18	24	26	33	61
Apache Ant 1.9.6	ASTChildren	1	1	2	2	2	512
	ASTChildrenMoreThan2	3	3	3	4	5	512
	ASTMaxDepth	2	8	10	11	14	49
	ASTNonTerminal	2	58	148	278	312	3,762
	ASTNonTerminal1Child	1	22	54	104	116	1,387
	ASTNonTerminalMoreThan2	0	10	26	55	60	939
	ASTTerminal	2	62	158	302	340	3,914
	ASTUniqueNonTerminal	2	16	22	22	28	47
	FBTArtificialNonTerminal	0	24	64	124	142	1,700
	FBTMaxDepth	1	13	22	29	36	188
	FBTNonTerminal	1	60	155	297	331	3,871
	FBTTerminal	2	61	156	298	332	3,872
	FBTUniqueNonTerminal	1	21	30	29	38	60
ArgoUML 0.34	ASTChildren	1	1	2	2	2	415
	ASTChildrenMoreThan2	3	3	3	4	5	415
	ASTMaxDepth	4	8	10	11	13	65
	ASTNonTerminal	4	48	105	239	236	10,220
	ASTNonTerminal1Child	1	17	38	92	89	4,621
	ASTNonTerminalMoreThan2	0	8	17	40	39	1,015
	ASTTerminal	3	50	112	246	250	7,822
	ASTUniqueNonTerminal	3	15	20	20	25	45
	FBTArtificialNonTerminal	0	19	42	97	100	2,393
	FBTMaxDepth	2	11	17	24	29	423
	FBTNonTerminal	2	49	110	244	248	7,810
	FBTTerminal	3	50	111	245	249	7,811
	FBTUniqueNonTerminal	2	20	27	27	33	57
CAROL 2.0.5	ASTChildren	1	1	2	2	2	45
	ASTChildrenMoreThan2	3	3	4	5	5	45
	ASTMaxDepth	4	7	10	11	14	21
	ASTNonTerminal	6	38	94	174	234	1,285
	ASTNonTerminal1Child	1	13	32	66	87	466
	ASTNonTerminalMoreThan2	0	6	14	28	37	220
	ASTTerminal	6	44	101	179	245	1,335
	ASTUniqueNonTerminal	4	14	21	20	28	38
	FBTArtificialNonTerminal	0	15	38	70	94	504
	FBTMaxDepth	4	10	17	20	26	57
	FBTNonTerminal	5	42	100	178	243	1,323
	FBTTerminal	6	43	100	179	244	1,324
	FBTUniqueNonTerminal	4	18	27	27	36	49

Table A.2: Intermediate Code Statistics continued

System	Metric	min	q_1	θ	μ	q_3	max
dnsjava 2.0.0	ASTChildren	1	1	2	2	2	263
	ASTChildrenMoreThan2	3	3	3	5	5	263
	ASTMaxDepth	4	8	9	10	12	39
	ASTNonTerminal	11	69	152	298	344	2,681
	ASTNonTerminal1Child	1	24	58	104	122	928
	ASTNonTerminalMoreThan2	2	16	36	63	79	682
	ASTTerminal	12	78	184	361	415	4,781
	ASTUniqueNonTerminal	7	15	22	22	28	40
	FBTArtificialNonTerminal	4	33	82	156	175	1,958
	FBTMaxDepth	6	16	25	32	36	275
	FBTNonTerminal	11	77	176	349	391	3,711
	FBTTerminal	12	78	176	350	392	3,712
	FBTUniqueNonTerminal	11	21	29	29	36	53
Hibernate 2	ASTChildren	1	1	2	2	2	249
	ASTChildrenMoreThan2	3	3	4	5	5	249
	ASTMaxDepth	4	7	10	10	12	24
	ASTNonTerminal	6	46	98	254	230	12,650
	ASTNonTerminal1Child	1	19	42	103	86	5,284
	ASTNonTerminalMoreThan2	1	9	18	45	39	2,090
	ASTTerminal	6	51	109	274	242	13,030
	ASTUniqueNonTerminal	5	14	19	20	26	39
	FBTArtificialNonTerminal	1	24	50	121	102	5,623
	FBTMaxDepth	3	13	19	27	28	271
	FBTNonTerminal	5	50	108	272	240	12,980
	FBTTerminal	6	51	109	273	242	12,980
	FBTUniqueNonTerminal	5	16	26	25	33	53
JDK 1.4.2	ASTChildren	1	1	2	2	2	882
	ASTChildrenMoreThan2	3	3	4	5	5	882
	ASTMaxDepth	3	6	10	11	14	50
	ASTNonTerminal	4	29	96	321	306	7,446
	ASTNonTerminal1Child	1	9	32	113	108	2,855
	ASTNonTerminalMoreThan2	0	5	16	51	47	1,098
	ASTTerminal	4	34	113	355	330	7,980
	ASTUniqueNonTerminal	4	9	20	19	27	46
	FBTArtificialNonTerminal	0	14	47	140	134	3,134
	FBTMaxDepth	2	10	20	31	35	969
	FBTNonTerminal	3	33	109	348	325	7,811
	FBTTerminal	4	34	110	348	326	7,812
	FBTUniqueNonTerminal	3	13	25	25	35	61
JHotDraw 6	ASTChildren	1	1	2	2	2	126
	ASTChildrenMoreThan2	3	3	4	5	5	126
	ASTMaxDepth	4	7	10	10	12	21
	ASTNonTerminal	5	51	71	144	159	1,798
	ASTNonTerminal1Child	1	21	26	57	60	823
	ASTNonTerminalMoreThan2	0	10	15	26	31	366
	ASTTerminal	5	59	86	157	178	1,797
	ASTUniqueNonTerminal	4	17	18	19	22	37
	FBTArtificialNonTerminal	0	27	40	68	75	810
	FBTMaxDepth	3	13	16	21	24	130
	FBTNonTerminal	4	58	85	155	173	1,785
	FBTTerminal	5	59	86	156	174	1,786
	FBTUniqueNonTerminal	4	19	23	24	31	51

Bibliography

[1] R. Abreu, P. Zoeteweij, and A. van Gemund. An evaluation of similarity coefficients for software fault localization. PRDC'06.

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

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

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

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

[6] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. MSR'13.

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

[8] E. Arisoy, T. Sainath, B. Kingsbury, and B. Ramabhadran. Deep neural network language models. WLM'12.

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

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

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

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

[13] E. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. FSE'14.

[14] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. de Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *TOSEM*, 23(1), 2014.

[15] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. Methodbook: Recommending move method refactorings via relational topic models. *TSE*, 40(7), 2014.

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

[17] Y. Bengio. Deep learning of representations: Looking forward. SLSP'13.

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

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

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

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

[22] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *TNN*, 5(2):157–166, 1994.

[23] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 13(1):281–305, 2012.

[24] S. Bhatia and R. Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR*, abs/1603.06129, 2016.

[25] C. Bishop. *Pattern Recognition and Machine Learning*. 2006.

[26] C. Bishop and J. Lasserre. Generative or discriminative? Getting the best of both worlds. *Bayesian Statistics*, 8:3–24, 2007.

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

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

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

[30] J. Campos, A. Riboira, A. Perez, and R. Abreu. GZoltar: An eclipse plug-in for testing and debugging. ASE'12.

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

[32] 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.

[33] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. ICSE'13.

[34] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. FSE'10.

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

[36] G. Chen, C. Chen, Z. Xing, and B. Xu. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. ASE'16.

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

[38] S. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. ACL'96.

[39] P. Clarkson and A. Robinson. Language model adaptation using mixtures and an exponentially decaying cache. ICASSP'97.

[40] C. Corley, K. Damevski, and N. Kraft. Exploring the use of deep learning for feature location. ICSME'15.

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

[42] 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.

[43] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk. Enhancing software traceability by automatically expanding corpora with relevant documentation. ICSM'13.

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

[45] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6), 1990.

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

[47] B. Dit, M. Revelle, and D. Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *EMSE*, 18(2), 2013.

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

[49] R. Dyer, H. Nguyen, H. Rajan, and T. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. ICSE'13.

[50] J. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

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

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

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

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

[55] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia. On integrating orthogonal information retrieval methods to improve traceability recovery. ICSM'11.

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

[57] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. AISTATS'11.

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

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

[60] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. ICSE'12.

[61] C. Le Goues, W. Weimer, and S. Forrest. Representations and operators for improving evolutionary software repair. GECCO'12.

[62] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API learning. FSE'16.

[63] R. Gupta, S. Pal, A.Kanade, and S. Shevade. Deepfix: Fixing common C language errors by deep learning. AAAI'17.

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

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

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

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

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

[69] G. Hinton. Learning distributed representations of concepts. COGSCI'86.

[70] G. Hinton. Connectionist learning procedures. *AI*, 40(1-3):185–234, 1989.

[71] G. Hinton, J. McClelland, and D. Rumelhart. Parallel distributed processing: Explorations in the microstructure of cognition, Vol. 1. chapter Distributed Representations. 1986.

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

[73] M. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. MSR'13.

[74] B. Hsu. *Language Modeling for Limited-data Domains*. PhD thesis, 2009.

[75] X. Huo, M. Li, and Z. Zhou. Learning unified features from natural and programming languages for locating buggy source code. IJCAI'16.

[76] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. ICSE'14.

[77] O. İrsoy and C. Cardie. Opinion mining with deep recurrent neural networks. EMNLP'14.

[78] F. Jelinek. *Statistical Methods for Speech Recognition*. 1997.

[79] F. Jelinek and R. Mercer. Interpolated estimation of Markov source parameters from sparse data. In *Proceedings of the Workshop on Pattern Recognition in Practice*, 1980.

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

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

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

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

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

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

[86] M. Jordan. Serial order: A parallel distributed processing approach. Technical report, Institute for Cognitive Science, University of California, San Diego, 1986.

[87] D. Jurafsky and J. Martin. *Speech and Language Processing*. 2 edition, 2009.

[88] R. Just, D. Jalali, and M. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. ISSTA'14.

[89] R. Just, D. Jalali, L. Inozemtseva, M. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? FSE'14.

[90] 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.

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

[92] S. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *TASSP*, 35(3):400–401, 1987.

[93] Y. Ke, K. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search. ASE'15.

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

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

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

[97] P. Koehn. *Statistical Machine Translation*. 2010.

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

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

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

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

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

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

[104] R. Kuhn and R. De Mori. A cache-based natural language model for speech recognition. *TPAMI*, 12(6):570–583, June 1990.

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

[106] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *TSE*, 38(1), 2012.

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

[108] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Backpropagation applied to handwritten zip code recognition. *NECO*, 1(4):541–551, 1989.

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

[110] 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.

[111] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining Android app usages for generating actionable GUI-based execution scenarios. MSR'15.

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

[113] F. Long and M. Rinard. Automatic patch generation by learning correct code. POPL'16.

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

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

[116] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *EMSE*, 2016.

[117] M. Martinez and M. Monperrus. Astor: A program repair library for Java (demo). ISSTA'16.

[118] M. Martinez and M. Monperrus. ASTOR: Evolutionary automatic software repair for Java. *CoRR*, abs/1410.6651, 2014.

[119] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *EMSE*, 20(1):176–205, 2015.

[120] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches. ICSE Companion'14.

[121] S. McConnell. *Code Complete*. 2 edition, 2004.

[122] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. ICSE'12.

[123] S. Mechtaev, Y. Jooyong, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. ICSE'16.

[124] S. Mechtaev, Y. Jooyong, and A. Roychoudhury. DirectFix: Looking for simple program repairs. ICSE'15.

[125] R. Miikkulainen and M. Dyer. Natural language processing with modular neural networks and distributed lexicon. *Cognitive Science*, 15:343–399, 1991.

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

[127] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

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

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

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

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

[132] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. NIPS'13.

[133] T. Mikolov, W. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. NAACL-HLT'13.

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

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

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

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

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

[139] M. Monperrus. Automatic software repair: A bibliography. Technical report, Inria, 2015.

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

[141] E. Moritz, M. Linares Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers. Export: Detecting and visualizing API usages in large source code repositories. ASE'13.

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

[143] M. Nagappan, T. Zimmermann, and C. Bird. Diversity in software engineering research. FSE'13.

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

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

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

[147] H. Nguyen, A. Nguyen, T. Nguyen, T. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. ASE'13.

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

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

[150] T. Nguyen, A. Nguyen, H. Nguyen, and T. Nguyen. A statistical semantic language model for source code. FSE'13.

[151] T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. ICSE'13.

[152] J. Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35(151), 1980.

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

[154] F. Palomba, M. Linares Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. User reviews matter! Tracking crowdsourced reviews to support evolution of successful apps. ICSME'15.

[155] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms. ICSE'13.

[156] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. Parameterizing and assembling IR-based solutions for SE tasks using genetic algorithms. SANER'16.

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

[158] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. ICML'13.

[159] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *SPE*, 46:1155–1179, 2015.

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

[161] D. Pierret and D. Poshyvanyk. An empirical exploration of regularities in open-source software lexicons. ICPC'09.

[162] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay. Sk_p: A neural program corrector for MOOCs. SPLASH Companion 2016.

[163] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. ICSE'14.

[164] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. ISSTA'15.

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

[166] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. ICSE'16.

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

[168] M. Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, 2005.

[169] X. Rong. word2vec parameter learning explained. *CoRR*, abs/1411.2738, 2014.

[170] R. Rosenfeld. A maximum entropy approach to adaptive statistical language modeling. *CSL*, 10(3):187–228, 1996.

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

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

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

[174] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

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

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

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

[178] Jürgen Schmidhuber. Learning complex, extended sequences using the principle of history compression. *NECO*, 4(2):234–242, 1992.

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

[180] D. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. 2 edition, 2000.

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

[182] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. PLDI'15.

[183] N. Sinha and M. Gupta. *Soft Computing and Intelligent Systems: Theory and Applications*. 1 edition, 1999.

[184] J. Snoek, H. Larochelle, and R. Adams. Practical Bayesian optimization of machine learning algorithms. NIPS'12, pages 2951–2959. 2012.

[185] R. Socher. *Recursive Deep Learning for Natural Language Processing and Computer Vision*. PhD thesis, 2014.

[186] R. Socher, C. Lin, A. Ng, and C. Manning. Parsing natural scenes and natural language with recursive neural networks. ICML'11.

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

[188] 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.

[189] A. Stolcke. SRILM—An extensible language modeling toolkit. INTERSPEECH'02.

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

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

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

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

[194] Y. Tian, D. Lo, and J. Lawall. Automated construction of a software-specific word similarity database. CSMR-WCRE'14.

[195] Y. Tian, D. Lo, and J. Lawall. SEWordSim: Software-specific word similarity database. ICSE Companion'14.

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

[197] L. Trefethen and D. Bau. *Numerical Linear Algebra*. 1997.

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

[199] L. van der Maaten and G. Hinton. Visualizing high-dimensional data using t-SNE. *JMLR*, 9:2579–2605, 2008.

[200] M. Linares Vásquez, A. Holtzhauer, and D. Poshyvanyk. On automatically detecting similar Android apps. ICPC'16.

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

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

[203] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. ICSE'16.

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

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

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

[207] M. White. Deep representations for software engineering. volume 2 of *ICSE'15*.

[208] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk. Generating reproducible and replayable bug reports from Android application crashes. ICPC'15.

[209] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. ASE'16.

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

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

[212] J. Xuan, M. Martínez, F. DeMarco, M. Clément, S. Lamelas, T. Durieux, Daniel Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *TSE*, 43(1):34–55, 2016.

[213] J. Yang and L. Tan. Inferring semantically related words from software context. MSR'12, pages 161–170, 2012.

[214] J. Yang and L. Tan. SWordNet: Inferring semantically related words from software context. *EMSE*, 19(6):1856–1886, December 2014.

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

[216] H. Yokoyama, Y. Higo, K. Hotta, T. Ohta, K. Okano, and S. Kusumoto. Toward improving ability to repair bugs automatically: A patch candidate location mechanism using code similarity. SAC'16, pages 1364–1370, 2016.

[217] D. Zhang and J. Tsai. Machine learning and software engineering. *Software Quality Journal*, 11(2):87–119, 2003.