Deep Learning in Software Engineering

Cody Allen Watson

Williamsburg, Virginia, USA

Bachelor of Arts, Wofford College, 2015
Master of Science, College of William & Mary, 2017

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 2020

APPROVAL PAGE

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

Doctor of Philosophy

Cody Watson

Cody Watson

Approved by the Committee, August 2020

DP

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

[illegible]

Andreas Stathopoulos, Professor, Computer Science
College of William & Mary

Robert Michael Lewis

Robert Michael Lewis, Associate Professor, Computer Science
College of William & Mary

Zhenming Liu

Zhenming Liu, Assistant Professor, Computer Science
College of William & Mary

[illegible]

Massimiliano Di Penta, Associate Professor, Computer Science
University of Sannio

ABSTRACT

Software evolves and therefore requires an evolving field of Software Engineering. The evolution of software can be seen on an individual project level through the software life cycle, as well as on a collective level, as we study the trends and uses of software in the real world. As the needs and requirements of users change, so must software evolve to reflect those changes. This cycle is never ending and has led to continuous and rapid development of software projects. More importantly, it has put a great responsibility on software engineers, causing them to adopt practices and tools that allow them to increase their efficiency. However, these tools suffer the same fate as software designed for the general population; they need to change in order to reflect the user's needs. Fortunately, the demand for this evolving software has given software engineers a plethora of data and artifacts to analyze. The challenge arises when attempting to identify and apply patterns learned from the vast amount of data.

In this dissertation, we explore and develop techniques to take advantage of the vast amount of software data and to aid developers in software development tasks. Specifically, we exploit the tool of deep learning to automatically learn patterns discovered within previous software data and automatically apply those patterns to present day software development. We first set out to investigate the current impact of deep learning in software engineering by performing a systematic literature review of top tier conferences and journals. This review provides guidelines and common pitfalls for researchers to consider when implementing DL (Deep Learning) approaches in SE (Software Engineering). In addition, the review provides a research road map for areas within SE where DL could be applicable. Our next piece of work developed an approach that simultaneously learned different representations of source code for the task of clone detection. We found that the use of multiple representations, such as Identifiers, ASTs, CFGs and bytecode, can lead to the identification of similar code fragments. Through the use of deep learning strategies, we automatically learned these different representations without the requirement of hand-crafted features. Lastly, we designed a novel approach for automating the generation of assert statements through seq2seq learning, with the goal of increasing the efficiency of software testing. Given the test method and the context of the associated focal method, we automatically generated semantically and syntactically correct assert statements for a given, unseen test method.

We exemplify that the techniques presented in this dissertation provide a meaningful advancement to the field of software engineering and the automation of software development tasks. We provide analytical evaluations and empirical evidence that substantiate the impact of our findings and usefulness of our approaches toward the software engineering community.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

This Dissertation and the work it encompasses would not have been possible without the aid and support of many individuals. First, I would first like to express my gratitude to my adviser, Dr. Denys Poshyvanyk, for everything he has done for me throughout my Ph.D. studies. Denys is an incredible adviser, trusted mentor, compassionate person and dear friend. Denys believed in me, when I had trouble believing in myself and pushed me to strive toward goals I never thought I could achieve. I am truly blessed to have the opportunity to learn from Denys and would not be where I am today without him. I would also like to thank Dr. Angela Shiflet and Dr. George Shiflet for their guidance and impact in helping me achieve my goals of attending graduate school. They are a major inspiration toward my passion in becoming an undergraduate mentor and educator. I would also like to thank my distinguished committee members, Andreas, Robert, Zhenming and Massimiliano for their mentorship and feedback throughout the writing of this dissertation.

This dissertation is also a product of the effort and work done by my friends and intelligent collaborators. Special thanks to Michele for all of your patience and hard work on our research projects, you taught me a lot and I value your wisdom. Thanks to Kevin, in many ways I look up to you as an incredible researcher and teacher. Thanks David for your friendship, passion and dedication to our work. Thanks Carlos for always making the lab a great place to be. Thanks Nathan, its encouraging to see the excitement of the next generation of SEMERU researchers. Lastly, thanks to Marty, Gabriele and Massimilano for your guidance and expertise, it is a privilege to know such brilliant individuals.

I want to thank my parents Mark and Roberta, my siblings Sarina, Gavin, and my grandmother Dolores. Mom and Dad, words cannot express my love and gratitude for all you did for me. It amazes me that two people could be so loving and caring while also sacrificing to give their children every opportunity to succeed. Sarina and Gavin, the gifts you have and challenges you have overcome are an inspiration to me and to those around you. Thank you Grandma, your unwavering support gave me a solid foundation to accomplish my goals. Thank you to Harris and Kathy Settle, your love and encouragement is invaluable to me.

Most importantly, I want to thank my lovely wife Laura. You leave me speechless in the way you support, encourage, care for, motivate, challenge and love me. I could have never completed my doctoral studies without you. You celebrated with me in my accomplishments and supported me when I was struggling. The sacrifices you made allowed me to accomplish my dreams and I am forever thankful.

To my wife Laura - All of my accomplishments rest in the love and support you have given me. I am so grateful for the gift that you are, and I thank God every day for the opportunity to share my life with his beloved daughter. I love you.

LIST OF TABLES

LIST OF FIGURES

Deep Learning in Software Engineering

Chapter 1

Introduction

1.1 Overview

Software engineering (SE) research investigates questions pertaining to the design, development, maintenance, testing and evaluation of software systems. As software applications pervade a wide range of industries, both open- and closed-source code repositories have grown to become unprecedentedly large and complex. This results in an increase of unstructured, unlabeled, yet important data including requirements, design documents, source code files, test cases and defect reports. Previously, the software engineering community has applied traditional machine learning (ML) techniques to identify interesting patterns and unique relationships within this data. Unfortunately, the process of implementing ML techniques can be time consuming and is limited with regard to the representations that can be manually engineered from analyzed data. With recent improvements in computational power and the amount of memory available in modern computer architectures, an advancement to traditional ML approaches has arisen called Deep Learning (DL). DL represents a fundamental shift in the manner by which machines learn patterns from data by automatically extracting salient features for a given computational task as opposed to relying on human intuition. DL approaches are characterized by architectures comprised of several layers that perform mathematical transformations, according to sets of learnable parameters, on data passing through them. These computational layers and parameters form models that can be trained for specific tasks by updating the parameters according to a model's performance on a labeled set of training data. Given the immense amount of data in software repositories that can serve as training data, DL techniques have ushered in advancements across a range of tasks in software engineering research.

The use of DL as a tool for software development can allow for effective and efficient software production since it has the potential to extend to every part of the software life cycle. However, the SE community has taken advantage of this tool without much regard for its limitations. This is in part due to the mystical, black-box nature of DL algorithms, which are difficult to visualize and often include complex parameterization. The challenges of applying DL models differs from that of customary approaches since a majority of the complexity associated with the implementation is composed of selecting the appropriate model architecture and correctly evaluating the model's performance. The selection and implementation of these model architectures can be a convoluted task because they are heavily dependent on the large numbers of parameters, which can have significant bearing on model performance. Therefore, SE researchers have been met with frustration when trying to determine when DL is beneficial, as well as how to apply the most effective architecture. In this dissertation, we look to demystify the use of DL algorithms and provide clarity for when a DL solution may be applicable. Our goal is not to limit the use of DL in SE research, but rather to identify the limitations so that software engineers can use this tool to its fullest potential without exceeding its capabilities.

The clear purpose for the use of DL in SE is automation. These models do not merely automate a SE task, rather, they use data to automate the generation of a solution to the SE task. This shifts the burden of engineering a solution from the developer to the machine. This monumental shift in responsibility reduces the amount of human error and can more accurately account for the patterns within the data that are undetectable from a developer's perspective. The applications of DL to improve and automate SE tasks and the dire need of developer support for creating, testing, and maintaining DL projects more broadly points to an apparent synergy between ongoing research in SE and DL. This dissertation will serve as an example of this synergy and will exemplify the power of DL in SE when applied within the proper constraints, thus facilitating the further usage of DL within the SE research community.

1.2 Motivation - Why Deep Learning?

The use of statistically based approaches to model software and its artifacts is a well established notion. The practicality of using these statistical approaches intrinsically involves the use of natural language processing (NLP), which brings up the question "is software natural?" Many SE papers have studied the idea of "natural" software [174, 77, 126, 225, 101] to justify or refute the use of natural language models on software. However, the introduction of canonical machine learning (ML) and eventually DL algorithms, shored up the weaknesses of NLP methods in two significant ways. The first is the ability to account for longer-range dependencies of variable length data, while NLP models can only account for the immediate context. The second is the capability of learning a richer notion of similarity across different contexts of data, while NLP algorithms are more rigid in their ability to detect similar data [20]. Both limitations of NLP algorithms significantly hinder the ability to effectively model software repositories. Therefore, a large influx of canonical ML and DL based strategies were applied to SE tasks to evaluate the increased performance from NLP.

Prior to the use of DL strategies, canonical ML approaches were at the forefront of applications to SE tasks. These algorithms require the user to manually identify and compile expressive features to describe the data, which the ML algorithm will then use to generate meaningful representations. For example, consider the task of image classification where the goal is to determine the type of an animal within a picture. Although not strictly a SE task, image classification highlights the principals of feature engineering that are inherent to the design of canonical ML based approaches. Designing a machine learning algorithm with hand-crafted features to accomplish such a task can be cumbersome, as a human would have to extract the salient image features (e.g., lines, colors, etc) as shown in figure 1.1. These features will likely be important for recognizing patterns delineating different classes of data. Furthermore, even if effective features are discovered, the class divisions of the data may be difficult to learn via simple (e.g. linear, logistic) functions and may instead require a more complex kernel that could be difficult or effectively impossible to properly learn without a richer representation of data features. This process, known as feature engineering, can be lengthy and lead to a large amount of technical debt for real-world ML systems [238].

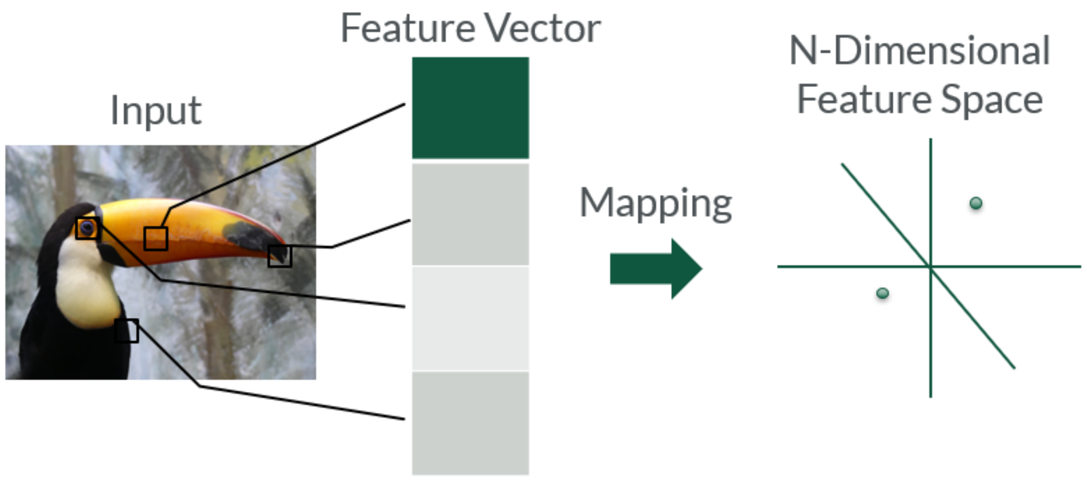


Figure 1.1: Feature engineering an image of a tucan.

SE research can require the analysis of complicated, feature rich data, so the reliance on feature engineering is not always plausible. Therefore, there is a need for automated approaches which can extract as many, or as few, salient features as necessary to accurately classify data. DL has shown to be an effective solution for the feature engineering problem while maintaining the ability to accurately represent large, heterogeneous, datasets in situ. 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 learn kernels rather than simply applying kernels [45]. Along with the advancement toward automatic feature engineering, DL algorithms transition from the use of shallow architectures to the use of hierarchical representations. This advancement in the complexity of the architecture allows SE researchers to better conceptualize software repositories. Thus, we can more accurately model software artifacts toward the goal of automating SE tasks for developers.

While these DL techniques bring with them unprecedented advancements, they also bring with them unprecedented and opaque complexity. Thus, SE researchers need to gain a better understanding of where DL has been both effectively and ineffectively used to support SE tasks in order to learn successful data collection, processing, architectures, representations used, as well as neglected tasks that serve as prime candidates for future DL applications with large impact. The rapid adoption of DL based approaches has outpaced the analysis of applicability and limitations. SE researchers have applied this new technique before contemplating the applicability to the specific SE task or the limitations which can be difficult to determine. There is a need within the SE community to stop and evaluate what SE tasks DL has been applied to, and more importantly, what challenges were faced when applying these complex architectures. With this evaluation, DL algorithms can be more accessible to SE researchers with the understanding of how and when they can be applied to SE tasks.

1.3 Applications

The purpose of the work presented in this dissertation focuses on two major goals, both of which propel the use of DL for SE (DL4SE) research:

1. To propagate the knowledge and experience from previous implementations of DL in SE that will further drive the analysis of complex SE repositories

2. To demonstrate the analysis and automation of software engineering tasks for developers through the use of DL algorithms, which removes the obstacle of manual feature extraction

To exemplify the need for continued research in DL4SE, we present work in the domain of software maintenance and software testing. These domains reflect back on the idea that DL can be applied to every faction of the software development life cycle. Although these works display advancement within their particular factions of the software development life cycle, they also represent the idea of automating developer focused tasks through DL modalities. We will show that through the numerous amount of software artifact data available, we can build successful systems that learn high-level, invariant features used specifically for the tasks of clone detection and test code generation.

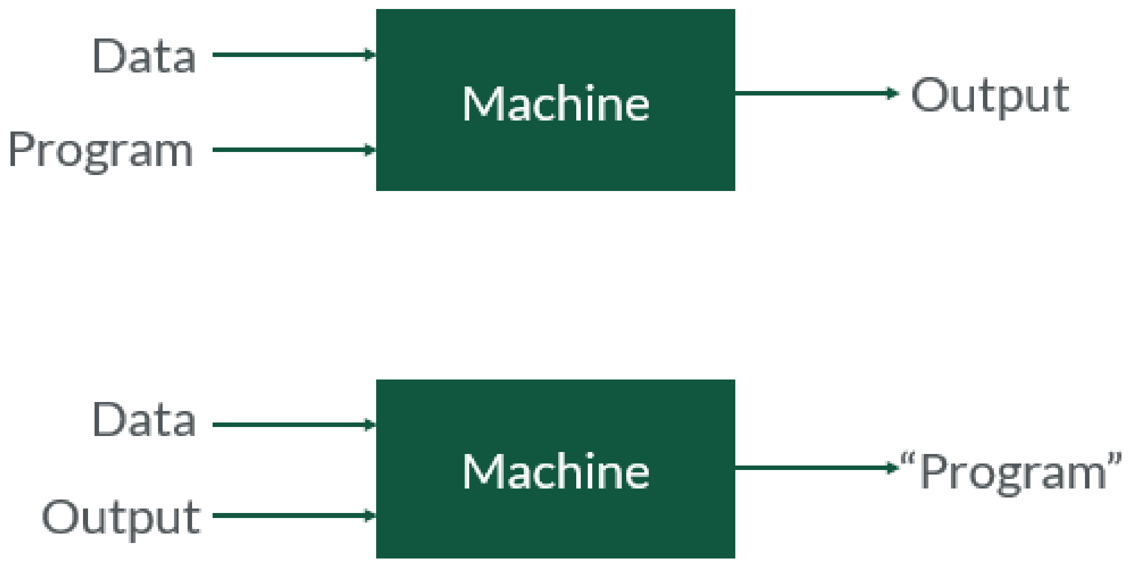


Figure 1.2: Traditional Programming vs. Software 2.0 Programming.

The overarching idea of these two areas contributes to the changing dynamic of software development and thus software engineering, coined by Andrej Karpathy as Software 2.0 [143]. Software 2.0 represents a transition away from traditional programming and moves toward automated programming through the use of DL. Figure 1.2 shows how traditional software engineering has been built on an idea of decomposition through the instructions of programming. We take a larger task, break it down into smaller tasks and solve those smaller tasks. Programs are engineered to represent how we decompose the problem into these smaller, solvable tasks. A traditional program will accept data as input, then perform a set of instructions to manipulate, identify, classify and arrange the data into a meaningful solution we call an output. DL turns this paradigm on its head. Now, rather than writing a set of instructions to interact with the data, we merely give the machine the data and our expected output for this data. In turn, the machine writes the program to manipulate, identify, classify and arrange the data. Now the output of the machine is the algorithm for generating the desired output.

Finally, we will highlight an emerging field of SE dedicated to the advancement and aid of developers implementing DL models. We refer to this field as SE4DL (Software Engineering for Deep Learning). Although our work focuses on the realm of DL4SE, the flaws and limitations seen in this realm points to a need for effective software maintenance and testing for DL architectures. Thus, we hope this dissertation serves as a medium between SE and DL to demonstrate that not only does SE research benefit from the powerful algorithms of DL, but that DL implementations can benefit from practices of SE.

1.4 Contributions and Outline

To accomplish the first goal set out by this dissertation, we design and conduct a systematic literature review (SLR) of DL4SE. This SLR serves to quantify and qualify the use of DL approaches to solve SE tasks. The intellectual contribution of this work is to determine novel patterns and proceedings pertaining to the components of learning [14]. This analysis should aid future SE researchers to systematically decompose and apply a DL approach to a SE task. Additionally, we note the challenges referenced by authors relating to the data preparation and creation of the DL model. Together, these findings reveal a research roadmap that identifies potential future research directions for DL in SE. Additionally, we provide guidelines for the implementations of these complex models along with some intuition on which complex architectures could be best suited for different types of SE artifacts. Although not explicitly a part of the DL4SE SLR, we also report a number of papers related to SE4DL. We briefly explore how a variety of works have addressed some of the unique problems that DL implementations have. Particularly, this body of work aims to better visualize and understand the inner-workings of a DL algorithm as well as provide a better testing structure to ensure the model's accuracy.

To accomplish the second goal set out by this dissertation, we design, build and empirically evaluate two DL based approaches which aim to aid developers in software maintenance and software testing. In particular, these approaches automate the detection of similar code fragments based on multiple representations of source code and the construction of semantically and syntactically correct assert statements within test methods. We demonstrate the intellectual contributions of these works by:

- Mining large, open-source, software repositories for distinct software artifacts.
- Effectively synthesizing SE artifacts into feature rich inputs to our DL based approaches.
- Packaging a DL framework to train, test and inference our DL model.
- Evaluating our DL model to ensure end-to-end automation that aids developers in the tasks of clone detection and assert statement generation.

In **Chapter 2** we discuss the background related to DL as it is applied in this dissertation. We begin by introducing related works for SLRs and the resources currently available to SE researchers when applying DL solutions. Additionally, we review related work pertaining to the task of code clone detection and test case generation. This provides the reader with the necessary information to understand the DL models implemented in chapters 4 and 5.

In **Chapter 3** we perform a systematic literature review for the SE community, which unveils a prominent research roadmap of DL4SE. This study involves SE related works from major international conferences and journals to gather the most representative collection of papers from all areas of SE. This SLR also addresses the challenges and methods of implementing and reporting on the use of DL based solutions in SE. This should serve as a resource for researchers in SE who wish to implement a DL model but are inquisitive about what problems can be addressed and what is vital when reporting their findings. Lastly, this work reveals the emerging field of SE we call SE4DL. We find that many researchers have identified problems with DL implementations and are using unique SE methods to better test and understand these complex models.

In **Chapter 4** we introduce our novel technique for the identification of similar code snippets through the use of multiple code representations. This technique involves the mining of source code from 10 compiled Java projects as well as 46 Java libraries to determine if we can identify similar code snippets across different libraries. We then subject the source code to four different types of abstractions that include identifiers, Abstract Syntax Trees, Control Flow Graphs and Bytecode. We conjectured that each code representation can provide a different, yet orthogonal view of the same code fragment, thus enabling a more reliable detection of similarities in code.

In **Chapter 5** we developed a novel technique for automatically generating assert statements for test methods. We mined millions of Java test methods, which contain the use of the Junit assert package. We then identified the particular focal method for each assert statement within the test method. With this information, we made use of a neural machine translation (NMT) model to *translate* test methods without an assert statement into semantically and syntactically correct assert statements. Our model, along with the implementation of a copy mechanism, was able to automatically learn assert statements even when the model was unable to naturally resolve certain tokens.

In **Chapter 6** we discuss future directions of research pertaining to the introduced topic of *software 2.0* as well as SE4DL. Both of these areas within software engineering are heavily motivated by this dissertation and this work will serve as a medium for those desiring to research in either of these areas. Lastly, we will summarize the contributions and findings of this dissertation.

In addition to the work presented in this dissertation, the author has worked on a variety of research topics within software engineering over the course of his career as a doctoral student including: (i) Bug-fixes [262] (ii) Code-changes [260] (iii) Software Testing [265] and (iv) Software evolution [201].

Chapter 2

Background

2.1 Deep Learning

In this chapter, we look to expound upon the complex statistical tool of DL and how it applies to the study of SE. We begin with a brief explanation as to how and why DL was first applied to unlabeled software artifacts found in online repositories. Once the SE community discovered the representational power of these DL models, their use to automate SE tasks dramatically increased. Next, we provide information regarding the use and practicality of SLRs in the field of SE. Lastly, we discuss the implementation details and theory behind two DL approaches used in this dissertation. We anticipate that the background information provided here will foster a smooth transition into the work presented in chapters 3, 4 and 5.

The progression toward the use of DL stems from the ability to automatically generate features. This removes the substantial manual effort required to generate the hand-crafted features that are required for canonical ML approaches. DL models also distinguish themselves from other statistically based models in their ability to capture long range dependencies within data and learn feature rich representations of software artifacts [20]. DL representations can be viewed as an extension of the perceptatron, an idea that was first proposed in 1965 [132]. However, the coined phrase of *Deep Learning* was not introduced until 1986 [74]. Although the introduction of DL was fairly early, its debut for modeling complex software repositories was not presented until 2015 by White et al. [286]. This work stimulated the use of DL to model software artifacts, demonstrating some of the advantages over information retrieval and natural language models.

The introduction of DL into SE brought about a series of works applying these models to a variety of tasks. White et al. [284] used DL, and in particular, representation learning via a recursive autoencoder for the purpose of code clone detection. The authors represent each source code fragment as a stream of identifiers and literal types (from the AST leaf nodes). Another work by Lam et al. [158] focuses on bug localization using both DL and Information Retrieval (IR) techniques. Rather than manually defining features, the IR technique, revised Vector Space Models (rVSM), collects these features which capture textual similarity. Then, a deep neural network learns how to map terms from the bug reports to tokens within the source code. Allamanis et al. [17] proposed an approach for suggesting meaningful class and method names. To accomplish this, identifiers in code are assigned a continuous vector, which considers the local context as well as long range dependencies by a neural log-bilinear context model. Then, identifiers which have similar vectors or embeddings will also appear to have similar contexts. However, in order to capture the global context of source code tokens, features are engineered which requires correct configuration and representation when integrated into the model. Building upon this previous work, Allamanis et al. [22] used an Attentional Neural Network (ANN) combined with convolution on the input tokens for assigning descriptive method names. Their approach allows for automatic learning of translation invariant features. Lastly, Gu et al. [109] used DL to avoid feature engineering in order to learn API usage scenarios given a natural language query. The approach encodes a query or annotation of the code into a fixed length context vector. This vector helps to decode an API sequence which should correlate with the query. Thus, once the model is trained, a natural language query will result in the correct API usage scenario for the context given in the query. We present only a few DL approaches to demonstrate the breadth and depth of DL applications in SE.

In the present day, there are a variety of different models and DL strategies to address many SE tasks. We perform a SLR to gain insight into which specific DL strategies have been implemented and provide some guidance on how to implement DL in SE for future SE research. Additionally, this dissertation presents two impactful DL approaches to detect code similarity and generate assert statements within test methods. We will review these two DL frameworks, which make use of recursive autoencoders, recurrent neural networks and neural machine translation.

2.2 Systematic Literature Reviews

Systematic literature reviews (SLR) have been a staple for the medical community for the process of summarizing and organizing critical aspects of medical research [1]. Since then, authors Kitchenham and Charters have adopted similar guidelines and procedures for conducting SLRs in the software engineering domain [11]. The purpose of SLRs is threefold: (i) to summarize existing evidence concerning a technology, (ii) to identify gaps or areas for continued investigation by researchers and (iii) to provide a framework or background for new research endeavors. Our work encompasses the gathering and evaluation of empirical research that uses DL in SE.

Many SLRs have been performed in the past on a variety of SE topics including effort estimation [240, 288, 267], fault prediction [128, 117, 223], software testing [80, 229] and many others [30, 98, 51]. These works attempt to summarize the landscape of research within their particular domain at the current time. They map the current approaches and techniques applied to a specific SE task and identify key challenges for researchers to continue exploring. However, none of these SLRs investigate the use of a single construct applied to multiple software engineering tasks. Rather, the software engineering task is the focal point of the review and a thorough analysis of the approaches and techniques are reported.

The study that has the closest relation to our work is by Allamanis *et al.* [19], who performed a SLR on machine learning (ML) and probabilistic models toward programming languages. The work the authors survey includes techniques that have approached the modeling of source code similar to the modeling of natural language. After the identification of such approaches, the authors begin to summarize how these models have been applied to SE tasks such as clone detection and program synthesis. They then highlight the unique challenges and limitations of these models when applied to the aforementioned SE tasks.

In our SLR, we systematically analyze the work done in applying DL to a particular SE task. Similar to Allamanis *et al.*, we look to identify techniques that have been used as solutions to numerous SE problems. We specifically look for DL based implementations and do not consider the broader body of work consisting of ML based approaches. Since DL has quickly risen as a popular tool to model complex software artifacts, we wanted to give researchers a resource that would summarize the body of work already performed, identify important challenges and limitations when applying these models, and distill out future research opportunities to apply DL in SE. To the best of our knowledge, no SLR exists for these applications.

2.3 Recursive Autoencoders

In this section, we present the architecture for recursive autoencoders which are used extensively in chapter 4. Recursive auto-encoders are a type of deep believe network (DBN) that creates a representation of features from an input vector [246, 247]. In a DBN, the layers of nodes communicate with previous and subsequent layers, but there is no communication between nodes of the same layer, which remains true for recursive autoencoders. To better understand what a recursive autoencoder does, it is important to recognize the function of this type of DL model. A recursive autoencoder will take a sequence of tokens as input, and for many tasks in SE, this results in an input sequence of source code tokens. These tokens are then abstracted and compressed into a smaller representation, however, this representation can also be decoded to reconstruct the original input. An overview of this algorithm can be seen in figure 2.1 where x_4 and x_5 are combined into a n-dimensional feature vector z, which can be decoded into $\hat{x}_4$ and $\hat{x}_5$. The decoding process should result in the original input tokens represented by x_4 and x_5.

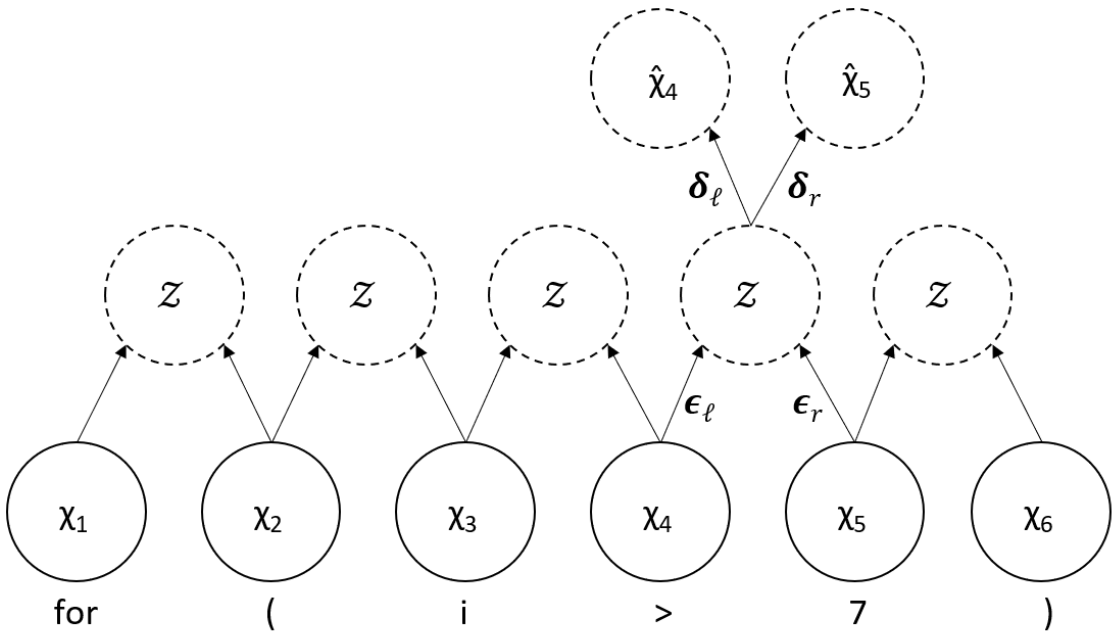


Figure 2.1: Recursive Autoencoder

A recursive autoencoder performs this compression at multiple different levels. Rather than encoding the entire sequence at one time, the sequence is compressed over a series of time steps. In figure 2.1 this results in the combination of two representations into the single encoding z at time step 1. Then, in the next time step, the algorithm combines the encodings z and x_4 to create a new encoding. This eventually results in a single encoding that can represent an entire sequence. The purpose of this model is to represent data with a smaller and unique feature vector. Then, feature vectors can be mapped into an n-dimensional space and clustered based on their distance to other mapped vectors. Training a recursive autoencoder measures the amount of loss there is between the original input sequence and the decoded output sequence. This loss is minimized over many different input encodings and subsequent decodings via a learning algorithm such as back propagation. In chapter 4, we will revisit the specific implementation applied to the task of clone detection.

2.4 Recurrent Neural Networks

Chapter 5 explores the use of recurrent neural networks for the purpose of learning the generation of assert statements in test methods. We look to provide background information regarding these complex models here. Recurrent neural networks (RNN) have, in many ways, advanced the way that we represent and embed software artifacts. Unlike the recursive autoencoders, this type of DL model considers the context of previous tokens in order to predict the next token. Additionally, recurrent neural networks are a type of supervised learning, meaning that the dataset we use to train these models are labeled. These labels are what we train the model to predict when given a particular input sequence. The type of RNN this work uses is a long short-term memory network (LSTM) which was discovered in 1997 [127]. This network outperformed multiple traditional models for tasks such as speech and handwriting recognition [236, 171]. Due to its success, these models were quickly identified as a tool for advancements in machine translation [252].

Unlike NLP models, which were the primary models used to represent source code before DL, the context of every previous token is considered. Figure 2.2 found in White et al. [286] describes how a RNN unrolled provides meaningful context to the next token predicted by the model. Here, the token $\omega(5)$ is being predicted by the model. The red node γ is the previous token's hidden state, which is comprised of the hidden state for every token before it. The current hidden state (grey node) computes its non-linear activation as a function of the input token (α) and the previous hidden state (γ). This process results in the next predicted token. This type of model is very effective in representing sequential tokens that are dependent on the tokens before them. Therefore, these DL models are an excellent choice for modeling the software artifact of source code. The applicability of RNNs to software artifacts has been applied to many different SE tasks, including software traceability [111], API usage [109], bug-fixes [262], mutation operators [265], malware detection [168], test input generation [146] and code clone detection [284]. In this work, we utilize the power of RNNs for the task of neural machine translation.

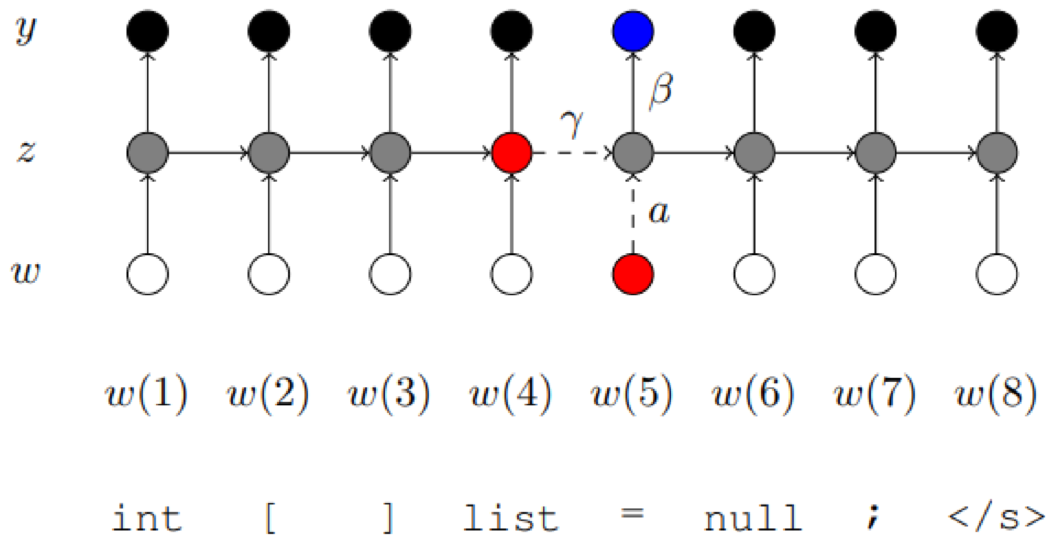

y
z
w
β
γ
a
$w(1)$ $w(2)$ $w(3)$ $w(4)$ $w(5)$ $w(6)$ $w(7)$ $w(8)$
int [ ] list = null ; </s>

Figure 2.2: RNN Unfolded

2.5 Neural Machine Translation

Neural machine translation is the DL framework presented in chapter 5. Our NMT implementation is comprised of the RNNs discussed in the previous section. Specifically, NMT is an end-to-end deep learning approach that specializes in modeling sequential data in order to perform a specific task [204]. Therefore, it is commonly used for the tasks involving the need for automated translation, text summarization and image captioning. NMT was introduced by Kalchbrenner et al. [140], Sutskever et al. [253], and Cho et al. [64] who defined RNN models for machine translation. The most famous architecture and the model we incorporate in chapter 5 is by Bahdanau et al. [32] published in 2015. NMT is comprised on an encoder that will encode a sequence of tokens into a fixed-length vector and a decoder that will translate that embedding back into *translated* tokens. A common example of this pipeline can be seen when translating Chinese to English. Figure 2.3 visualizes this process of translation.

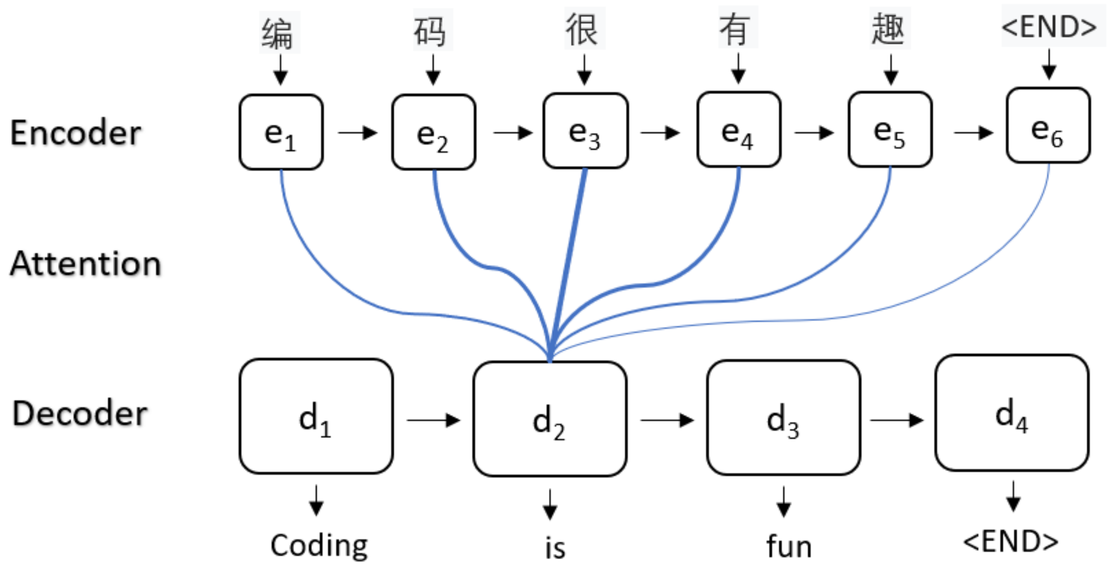


Figure 2.3: Neural Machine Translation Model

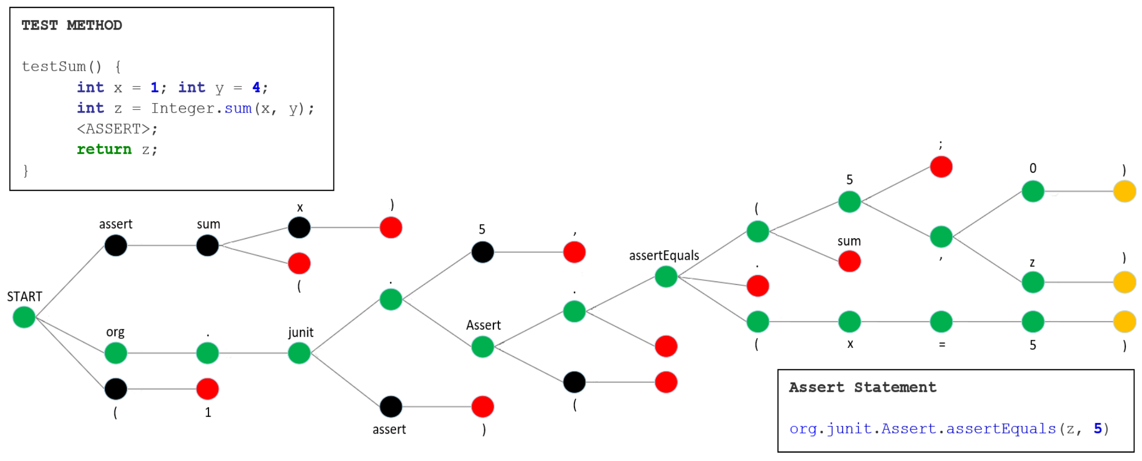
To begin the NMT translation process, it is important to understand the components that comprise this complex framework. Let's begin with the first component known as the bi-directional encoder. The bi-directional encoder is comprised of two RNNs. The first RNN learns the sequence of tokens in a standard forward pass, while the second RNN learns the sequence of tokens in a reverse pass. These models then combine their final states and create a single embedding, which is fed to the decoder. The decoder is the next major component of the NMT framework and is comprised of another RNN. This RNN predicts the set of output tokens, given the embedding created by the bi-directional encoder. The decoder not only considers the input embedding, but also a context vector and any tokens which have already been resolved by the decoder. The context vector represents the impact that the input tokens have on the prediction of the output tokens. In figure 2.3 this is shown by the lines with varying thickness. The thicker lines represent a greater influence a token in the embedding has on the current token the decoder is attempting to generate. This context vector is created by the attention mechanism incorporated into the model. All components of the model are trained together and are optimized by minimizing the negative log-likelihood of the targeted terms, while the weights are updated using stochastic gradient decent.

2.5.1 Open Vocabulary Problem

NMT suffers from a number of limitations which constrain its use and implementation. One such problem NMT faces is the open vocabulary problem. The NMT model can only consider a finite amount of tokens that will populate the source and target sequences. NMT models generate output from a probability distribution over words. This means that a large number of words to consider significantly slows down the optimization of the model. The oxford English dictionary estimates there to be around 250,000 unique English words, and many of these are uncommon or obsolete [2]. However, when modeling source code, developers are not limited to a finite vocabulary. Therefore, to take advantage of the NMT framework when modeling source code, we must artificially limit the vocabulary used. A number of processes to limit the vocabulary have been adopted in a variety of SE works [62, 115, 265, 262].

2.5.2 Multiple Predictions with Beam Search

One limitation of the NMT model is the ability to extract different, possible translations based on a single input sequence. Traditionally, NMT models decode the output token in a greedy way, choosing the token with the highest probability at the current time step. However, this may not lead to the correct result in every case. To increase the model's accuracy and effectiveness, we utilize a strategy named beam search [226]. When using beam search, rather than always choosing the token with the highest probability, this method keeps the top k hypothesis at each time step for what the decoded token could be.



```
TEST METHOD

testSum() {
       int x = 1; int y = 4;
       int z = Integer.sum(x, y);
       <ASSERT>;
       return z;
}
```

```
Assert Statement

org.junit.Assert.assertEquals(z, 5)
```

Figure 2.4: Beam Search Example

Therefore, rather than the model generating a single sequence of the most probable source code tokens, the model can generate k sequences as potential translations. Formally, let $\mathcal{H}_k$ be the set of k hypothesis decoded until time step t:

$$\mathcal{H}_k = \{(\alpha_1^1 \dots \alpha_t^1), (\alpha_1^2 \dots \alpha_t^2), (\alpha_1^k \dots \alpha_t^k)\}$$

At each time step $t+1$, there are V possible new tokens for each hypothesis. Therefore, we have a total of $k \cdot V$ potential hypotheses.

$$\mathcal{C}_{t+1} := \bigcup_{i=1}^{k} \{(\alpha_1^i, \dots, \alpha_t^i, 1), \dots, (\alpha_1^i, \dots, \alpha_t^i, V)\}$$

The candidate sets are decoded such that we keep k sequences that represent the highest probability candidates. This process happens at every time step until we reach a special end token that signifies the process has stopped. In our work, we consider k translations to potential assert statements. Since beam search is an approximate search algorithm, the larger the k the more likely the algorithm will generate a correct translation. However, there is no way to automatically determine the proper translation from the set of k hypothesis and thus puts a greater strain on the developer.

Consider the example shown in figure 2.4 where we show a beam width of $k = 3$. This example is taken in the context of our NMT model that predicts assert statements. Assuming the input to the NMT model is the test method in the upper-left corner of the figure, the beam search mechanism attempts to predict 3 possible assert statement translations. Beginning with the START token, beam search starts by generating the top-3 candidates for the first token (*i.e.*, **org, assert, (**). At time step 2, beam search extends all three hypotheses, as the top-3 highest probability tokens (**sum, ., 1**) follows the previous tokens found in each of the three beams. However, at time step 3, the third beam does not generate one of the top-3 highest probability tokens, so the beam is terminated (denoted by red circle). This process continues until the algorithm encounters an end of sequence token. The resulting sequences can be seen by following the green nodes until the yellow terminating node is reached.

2.6 Automatically Generating Assert Statements

In this section, we provide context and motivation for the learned generation of an assert statement for a particular test method which will be discussed in chapter 5. This work lies within the SE task of software testing, which is a vital component of the software life cycle. Software testing is widely studied in software engineering and focuses on topics from automatic test code generation [29] to automatic generation of test data for software [153]. However, software testing is an incredibly complex and systematic process that is often articulated through the use of hierarchical testing levels. Figure 2.5 shows this hierarchy where the first level is unit testing. This type of testing focuses on individual units of software with the intention of validating each unit as it is designed.

Although unit tests are a fundamental step in the process of software testing, the ability to test individual components of functionality is a crucial step toward more complex types of software testing. Assert statements provide the ability to capture the logic and functionality of the underlying code base within unit tests. Therefore, generating meaningful

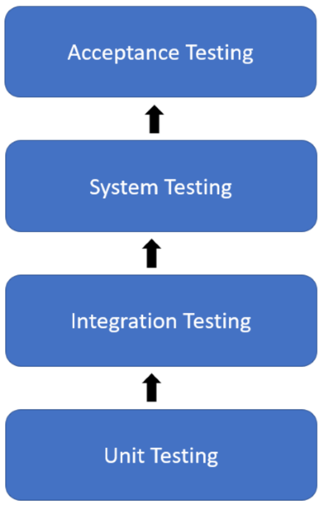


Figure 2.5: Software Testing Hierarchy

assert statements is necessary to detect potential faults. Despite the importance of unit tests, they can be incredibly time consuming and a laborious process for the developer. In order to reduce this burden on developers, automated tools such as Evosuite [94], Randoop [213] and Agitar [5], attempt to automatically generate entire test suites.

While several automated tools such as Evosuite [94], Randoop [213] and Agitar [5] have been proposed to automatically generate test methods, these tools embed their own methods for synthesizing assert statements. Evosuite, one of the most popular automated test generation tools, uses a mutation-based system to generate appropriate assert statements. In particular, it introduces mutants into the software and attempts to generate assert statements able to kill these mutants. Evosuite also tries to minimize the number of asserts while still maximizing the number of killed mutants [94].

Randoop is another automated test generation tool that creates assertions with *intelligent* guessing. This technique applies feedback-directed random testing by analyzing execution traces of the statement it creates. Essentially, a list of *contracts*, or pieces of logic that the code must follow, are used to guide the generation of assert statements. These contracts are very similar to user defined assert statements. However, the contracts only provide the logic. The Randoop program creates a syntactically correct assert statement that tests the user's provided logic pertaining to the test method. Differently from Evosuite and Randoop, ATLAS applies a deep learning-based approach, with the goal of mimicking the behavior of expert developers when writing assert statements.

Recent research has evaluated the ability of state-of-the-art automated techniques for test generation to capture real faults [239, 26]. They found that the quality of the generated assert statements are one of the primary reasons that automatically generated tests failed to uncover real faults. Another recent study conducted by Shamshiri *et al.* [239] highlights the importance of high-quality, complex assert statements when detecting real faults. The authors compare the abilities of both exception and assert statements as fault finding mechanisms. They note that three automated approaches (Evosuite, Randoop, Agitar) detect more faults through the use of an assert statement, however, insufficiencies in the automatically generated asserts led to many bugs going undetected [239]. Thus, this work makes two important conclusions: (i) assert statements are an important component of automated test case generation techniques, as they are the main vehicle through which faults are detected; (ii) the current quality of assert statements generated by automated testing techniques are often not of high enough quality to detect real faults.

A complimentary study performed by Almasi *et al.* [26], tested Evosuite and Randoop's ability to detect 25 real faults in an industrial software system. The study found that Evosuite was able to detect 56.4% and Randoop was able to detect 38.0% of the faults within that system. Of particular note was the author's qualitative evaluation, which showed that nearly half of the undetected faults could have been detected with a more appropriate assert statement. Additionally, the authors solicited feedback from developers, asking them how the generated tests could be improved. The general consensus among the respondents was that automated testing strategies failed to generate meaningful assert statements. The authors also presented hand-written test methods to developers for comparison, and they commented that "*the assertions are meaningful and useful* ***unlike the generated ones***". These findings demonstrate a clear need for automated techniques that generate *meaningful* assert statements to complement existing test generation approaches.

One of the limitations that tools such as Evosuite, Randoop and Agitar have is that they rely on heuristics or "intelligent" randomness in order to generate assert statements. This type of generation does not take into account the learnable components of test and focal methods, and thus leads to more simplistic asserts. Therefore, we leverage a NMT-based DL technique to generate asserts that can test the complexities contained within the context of the test and focal method. This strategy results in an assert which possesses the ability to accurately evaluate the logic of the focal method, leading to more useful unit tests and a higher quality test suite.

Chapter 3

Systematic Literature Review in Deep Learning for Software Engineering

3.1 Introduction

Software engineering (SE) research investigates questions pertaining to the design, development, maintenance, testing and evaluation of software systems. As software applications pervade a wide range of industries, both open- and closed-source code repositories have grown to become unprecedentedly large and complex. This results in an increase of unstructured, unlabeled, yet important data including requirements, design documents, source code files, test cases and defect reports. Previously, the software engineering community has applied traditional machine learning (ML) techniques to identify interesting patterns and unique relationships within this data to automate or enhance many tasks typically performed by developers. Unfortunately, the process of implementing ML techniques can be a tedious exercise in careful feature engineering, wherein researchers experiment with identifying salient pieces of data that can be leveraged to help solve a given problem or automate a given task. With recent improvements in computational power and the amount of memory available in modern computer architectures, an advancement to traditional ML approaches has arisen called Deep Learning (DL). Deep Learning represents a fundamental shift in the manner by which machines learn patterns from data by *automatically* extracting salient features for a given computational task, as opposed to relying upon human intuition. Deep Learning approaches are characterized by architectures comprised of several layers that perform mathematical transformations on data passing through them. These transformations are controlled by sets of learnable parameters that are adjusted using a variety of learning and optimization algorithms. These computational layers and parameters form models that can be trained for specific tasks by updating the parameters according to a model's performance on a labeled set of training data. Given the immense amount of unstructured data in software repositories that can serve to uncover hidden patterns, DL techniques have ushered in advancements across a range of tasks in software engineering research including automatic software repair [263], code suggestion [107], defect prediction [276], malware detection [169], feature location [69], among many others [187, 271, 179, 285, 289, 114, 258, 182]. We refer to this emerging field of research that leverages deep learning to improve or automate SE tasks as Deep Learning for Software Engineering (DL4SE).

Given the successes of DL approaches on benchmark computer vision (CV) and natural language processing (NLP) tasks, they have become quite popular as a general form of software development in many industries. That is, developers and engineers are continuing to leverage DL-based systems to solve problems that are difficult to tackle analytically via traditional algorithm design or programming (self-driving vehicles, facial recognition, etc.). Unfortunately, the very characteristics that make DL attractive and highly effective carry with them trade-offs related to interpretability and testability. In other words, while these models can often perform well for complex tasks, current challenges exist in interpreting *why* these models make the decisions they do, and in *testing* them to find where they fall short. Furthermore, the definition of a "development" cycle for DL models is still emergent, and understanding the best practices and outstanding challenges is essential to ensure that researchers and practitioners properly allocate resources to supporting DL developers and solving the most pressing challenges. For example, DL models are inherently data driven, wherein most of the implementation effort is composed of selecting the appropriate model and acquiring or creating a large labeled dataset to train the model. However, the selection and implementation of these model architectures can be a complex task due to the relatively large number of parameters, which can have significant bearing on model performance that must be selected. Additionally, the collection and preparation of a suitable dataset, and the subsequent representation of this data in a DL framework [164], can comprise a substantial effort in and of itself. Proper data collection, preprocessing and representation can be crucial to the implementation of effective DL models as the level of abstraction of the input data has the potential to reveal or hide latent features that the model could learn. The above aspects show the complexities behind the engineering for DL-related projects, and clearly highlight the need for established best practices and intuitive automated developer tools that provide developers an effective platform for creating and reusing DL models. We refer to the area of research that aims to improve development practices for DL-based systems as Software Engineering for Deep Learning (SE4DL). While research in SE has begun to move toward supporting DL systems, particularly in the area of testing [258, 273, 145], a clear enumeration of current practices and open challenges is needed to help guide the SE and DL research communities towards the most impactful problems.

The clear applications of DL to improve and automate SE tasks and the dire need of developer support for creating, testing, and maintaining DL projects more broadly points to a clear synergy between ongoing research in SE and DL. However, in order to effectively chart the most impactful path forward for research at the intersection of these two fields, researchers need a clear map of what has been done, what has been successful, and what can be improved. For instance, researchers would benefit from knowing which SE areas DL has been applied to and what types of models are optimal when analyzing different types of SE data. There are many DL algorithms, all posing different trade-offs and benefits within the context of their applications. SE researchers could gain immense benefit from understanding past successes (and failures) of these applications to further advance automation of new SE tasks. On the other hand, DL researchers and practitioners would benefit from understanding the open challenges and engineering practices related to the emerging development cycle of DL models. A better understanding of these existing challenges, alongside a characterization of the DL development cycle is absolutely essential to ensure more reliable and practical DL implementations in industry, and more fluid reproducible research in Deep Learning.

In an effort to map and guide research at the intersection of DL and SE, we conducted a systematic literature review to identify and systematically enumerate the synergies between the two research fields. We do this through the creation of a detailed taxonomy of past work on DL techniques applied to SE tasks, and subsequently identified open challenges and best practices for applying DL techniques to SE-related data. Additionally, we analyzed the impacts of these DL based approaches and discuss the difficulties encountered with reproducibility of these studies.

We organized our work around four types of Research Questions (RQs) that are fundamentally centered upon the *components of learning*. That is, we used the various components of the machine learning process as enumerated by Yaser Abu-Mostafa's book, *Learning From Data* [12], to help ground the creation of our taxonomy and exploration of the topic. Our overarching interest was to identify best practices and promising research directions for applying DL frameworks to SE contexts, as well as to identify the direction of emerging work that applies principles from SE research to improve the DL development process. Clarity in these respective areas will give researchers the tools necessary to effectively apply DL models to SE tasks.

To answer the RQs, we focused primarily on the primary studies classified as DL4SE. We created a taxonomy of these studies that highlights important concepts and methodologies applied to the types of software artifacts analyzed, the learning models implemented and the evaluation of these approaches. We discovered that while DL in SE has been successfully applied to many SE tasks, there are common pitfalls and missing details when implementing these complex models. Therefore, in addition to the taxonomies that describe how the components of learning have been addressed, we provide insight into commonly missed steps and how to avoid them when considering a DL approach. From this, we can provide the SE community with important guidelines for applying DL models that address issues such as sampling bias and data snooping, overfitting and underfitting of the learning models, and the over engineering of DL architectures. Our goal is to empower SE researchers in their ability to create more robust and reliable DL approaches that address interesting SE tasks.

With regards to SE4DL, we found initial research that has begun to address the following DL development tasks of (i) requirements engineering, (ii) design, (iii) implementation, (iv) testing, and (v) maintenance of software. Specifically, we found that recent research has focused on the testing and interoperability of these models. This stems from the inherent abstract nature of DL models and their ability to model target functions which are intractable for human minds. We discussed these works at length and attempted to give insight into the troublesome limitations when testing DL approaches. However, since these approaches do not actually implement a DL based approach, we did not include them in our DL4SE classifications. Our entire SLR process can be found in figure 3.2, which explains the process for developing the RQs and describes the steps taken in order to synthesize the necessary information gathered from the primary studies.

The remainder of our survey is structured as follows: Section 3.2 discusses the synthesis of the RQs this SLR looks to answer, which are paralleled to the components of learning; Section 3.3 explains our systematic process for extracting and analyzing the primary studies that comprise our taxonomy of DL in SE; Section 3.4.1 begins the generation of our taxonomy by analyzing the SE tasks that DL has been applied to; Section 3.4.2 analyzes the type of data that DL approaches use and how that data is being represented to the DL model; Section 3.4.4 focuses on the learning models that are used to address SE tasks; Section 3.4.5 discusses the impacts and performance of DL approaches in SE; Section 3.4.6

begins a discussion around the factors that contribute to the difficulty of reproducibility of DL applications in SE; Section 3.5 provides future researchers with steps that should be taken and common pitfalls when applying DL in SE; Section 3.6 discusses a new emerging field in SE, where SE concepts are applied to the field of DL engineering; Section 3.7 focuses on the primary studies which were found by our search string but are not included in the taxonomy of DL in SE; finally, Section 3.8 summarizes our results and highlights interesting, potential directions for the field of DL4SE.

3.2 RQ Synthesis

The synthesis and generation of research questions is a quintessential step to any systematic literature review (SLR). When studying how DL has affected SE, we wanted to formulate RQs that would naturally highlight a taxonomy of DL in SE, establish coherent guidelines for applying DL to SE tasks and address common pitfalls when implementing these complex models. Therefore, we sought to allow the components of learning to guide the development of our RQs and tailor those questions to specifically analyze DL used in the context of SE. The components of learning are a ML formalization introduced by Yaser Abu-Mostafa [13] as a way to enumerate the conditions for learning. We also wanted to address what components of learning the primary studies in SE are missing or are not adequately reporting on. Failure to adequately consider or report the components of learning when applying a DL approach in SE could lead to difficulty in reproducibility and threaten the validity of the results.

The first component of learning is an unknown target function, which represents the relationship that is desired to be learned. This component is closely linked to the task that the learning approach is being applied to. From analysis of the target function to be learned, one can infer the input and output of the model, the type of learning, hypothetical features to be extracted from the data and potential architectures to be used. To encompass the essence of this component of learning, our first RQ was:

RQ1: What types of SE tasks have been addressed by DL-based approaches?

In understanding what SE tasks have been analyzed, we were able to naturally present a taxonomy of what tasks have yet to be explored using a DL based approach. We were also able to infer why certain SE tasks may present unique challenges for DL models and the target users of these DL approaches, given the SE task they address.

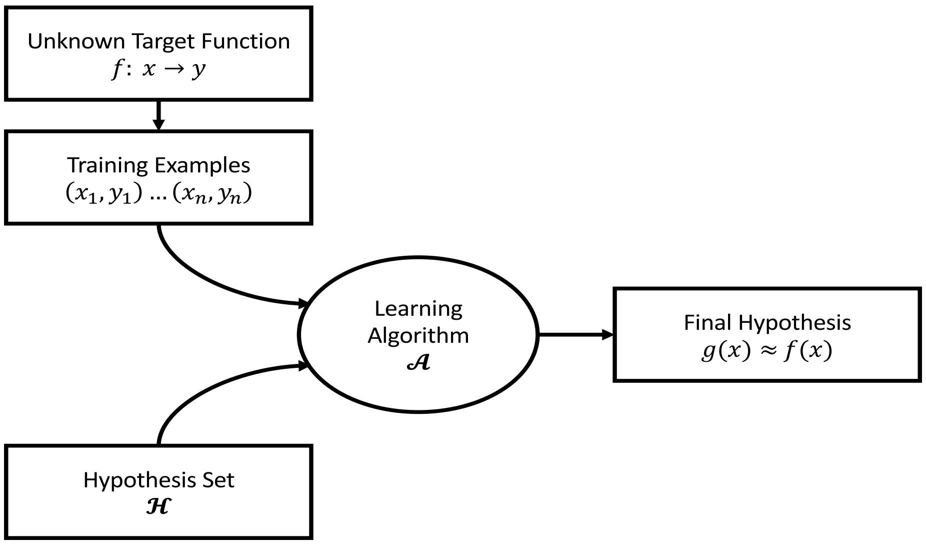


Figure 3.1: The Components of Learning

A natural next step from analyzing the unknown target function is to explore the data being presented to the model to learn this unknown target function. Here, we primarily focused on studying the input and output training examples and the techniques used in DL approaches to prepare the data for the model. An understanding of the training examples presents greater insight into the target function while also giving further intuition about what the potential features to be extracted are and DL architectures that can be used to extract those features. A taxonomy of the data used, how it was extracted and how it was preprocessed should encourage new ideas or techniques to better capture the features of the data points, resulting in a more accurate model of the target function. Thus, our second RQ was:

RQ2: How are software artifacts being extracted, prepared and used in DL based approaches for SE tasks?

Within this RQ we explored multiple facets of the extraction and preprocessing steps by breaking this RQ down into three sub-question. The first sub-question discussed the different types of data that are being used in DL based approaches. Given the plethora of software artifacts stored in online repositories, it is important to know which of those artifacts are being analyzed and modeled with a DL architecture. Our second sub-question addressed how data is being extracted and pre-processed into a format that the DL model can appropriately learn from. Analyzing the extraction of the data will present researchers with potential tools and techniques to mine similar types of data for different DL applications within SE. Additionally, the representation of data is somewhat dependent on the DL architecture and its ability to extract features from that representation, which gives importance to the discussion of the relationship between DL architectures and the data they process. Our third sub-question explored what type of exploratory data analysis is conducted to help inform model design and training. It is important to note that DL algorithms are data hungry, but the quality of the learned hypothesis is a product of the quality of data the model learns from. Therefore, we looked to explore techniques to mitigate problems associated with the data, such as sampling bias and data snooping.

Next, we encompassed both the third and fourth components of learning, the learning algorithm and hypothesis set, into a single research question because we felt that they were highly related to one another. The learning algorithm is the process of navigating the hypothesis set for the best fit model to the data. These are primarily mathematical models that use a probability distribution over the input to appropriately approximate the optimal hypothesis from the hypothesis set. The hypothesis set is a set of all hypotheses, based on the learning algorithm, that the input can be mapped to. This set changes because it is a function of the possible outputs given the input space, and is dependent on the learning algorithm's ability to model those possible outputs. Together the learning algorithm with the hypothesis set is referred to as the learning model, thus, our third RQ asked:

RQ3: What deep learning models are used to support SE tasks?

We also wanted to make an important distinction within the learning model to address the variety of architectures used and the optimizations applied to those architectures in order to model the appropriate hypothesis set. We broke down RQ3 into three sub-questions, which addressed the aforementioned attributes of the learning model separately. Firstly, we asked what types of model architectures are used to perform automated feature engineering of the data within various SE tasks. Here, we looked at the variations of architectures to relate them to the data they analyzed and the task they addressed. We also used this RQ to present how the type of architecture chosen to model a particular target function relates to the types of features that are being extracted from the data. Secondly, we looked into the different learning algorithms and training processes that are used in order to optimize the models. This is where we addressed the variety of different learning algorithms whose responsibility is to properly capture the hypothesis set for the given input space. The different optimization algorithms and training processes used to tune the weights of the model are an important step for finding the target hypothesis that best represents the data. Lastly, we analyzed the methods used to combat overfitting and underfitting. This question addressed one of the major concerns with using DL algorithms and their tendency to overfit or underfit the data they model. Within SE, we wanted to know what techniques the authors are implementing to combat these well understood limitations.

Our fourth RQ addressed the component of learning known as the final hypothesis. The final hypothesis is the target function learned by the model that is used to predict the target of previously unseen data points. Measuring the effectiveness of DL models can encourage the use of DL in areas of SE that DL has not yet been applied. We also felt that we could provide an indication toward the advantages of certain data formatting, DL architectures or techniques that have been successful for certain SE tasks in the past. Thus, our fourth RQ was:

RQ4: How well do DL tasks perform in supporting various SE tasks?

The purpose of this RQ is to analyze the effectiveness of DL applied to SE tasks, however, this can be a difficult and complex. We attempted to answer this overarching RQ by first addressing the "baseline" techniques that are used to evaluate new DL models and discovering what metrics are used in those comparisons. An evaluation that contains a comparison with a "baseline" approach, or even non-learning based solution, is important for determining the increased effectiveness of applying a new DL framework. Additionally, the metrics used to quantify the comparison between DL approaches should be known so that future implementations of DL in SE know how to quantify their increased effectiveness. Our second sub-question explored how DL based approaches are impacting the automatization of SE tasks and in which way these models promote generalizability to different sets of data. Generalizability of DL approaches in SE is vital for the usability of these models. If the state of the art DL approach is only applicable within a narrowly defined set of circumstances, then there may still be room for improvement on this solution. Lastly, we wanted to explore common mistakes made by overengineering the DL model and analyzing the trade-off of model complexity with accuracy. We look at DL architectures through the lens of Occam's Razor principal, which states that the least complex model that is able to learn the target function is the one that should be implemented [224].

Our last RQ encompassed all of the components of learning by asking a discussion question related to the reproducibility of DL approaches in SE. SE as a field has made a significant effort toward publishing works that are reproducible. However, DL models can be very complex and we wanted to determine if there were important details left out of the primary studies we analyzed. Therefore, we asked the question:

RQ5: What common factors contribute to the difficulty when reproducing DL studies in SE?

Our goal was to identify common components of implementing a DL based approach which are absent from the primary studies. Particularly, we addressed two important attributes: what primary studies are not reproducible and why, as well as what primary studies are not replicable and why. Reproducibility is the ability to take the exact same model with the exact same dataset from the primary study and produce the same results. Replicability is to follow the methodology described by the primary study such that a similar implementation can be generated potentially on a different dataset. This RQ helps the SE community to understand what factors are being insufficiently described or absent all together within the DL approach, leading to difficulty in reproducing or replicating the primary study.

Lastly, we looked to generate a set of guidelines, not only for the implementation of DL in SE, but also how to report the model in a way that values reproducibility. These guidelines start with the identification of the SE task to be studied and provide a step by step process through evaluating the new DL approach. Due to the high variability of DL approaches and the SE tasks they are applied to, we only provide generalizable steps that should be performed for future implementations of DL in SE. In addition, we provided checkpoints throughout this process that address common pitfalls or mistakes that future SE researchers can avoid when implementing these complex models. Our hope is that adhering to these guidelines will lead to future DL approaches in SE with an increased amount of clarity and reproducibility.

3.3 Methodology

In this paper we performed a systematic literature review of deep learning in software engineering, which discusses the open issues involving the use of these complex models to address software engineering tasks. We applied a systematic methodology to our literature review in order to uphold the integrity of our analysis and provide a reproducibility aspect to this work. Within the field of SE, SLRs have become a standardized practice to communicate the past and present state of a particular area within SE. These standards were developed by Kitchenham's work, *Guidelines for Performing Systematic Literature Reviews in Software Engineering* [149], which contains the guidelines and strategies used in this review.

The full breakdown of our methodology in conducting the SLR can be seen in figure 3.2. As is described in Kitchenham's guidelines, we synthesized research questions (RQs) before beginning the searching process, which helped to naturally guide the SLR procedure and focused the search only to primary studies that helped answer these RQs. From here, we performed the following steps when conducting our review:

- Searched for primary studies
- Filtered studies through our inclusion criteria
- Performed snowballing and manual additions to the body of work
- Applied our exclusion criteria and quality analysis of the remaining studies
- Extracted relevant data and analyzed that data from the final set of studies
- Synthesized and created a taxonomy of DL in SE

3.3.1 Searching for Primary Studies

The next step in the methodology was to begin searching for primary studies that would aid in the ability to properly address the synthesized RQs. We first began by identifying venues we would like to search. We chose venues which have a high standard for peer reviewed research, as well as a high influence over the field of SE. Therefore, we considered research from the SE conferences of ICSE, ASE, FSE, ICSME, ICST, MSR, and ISSTA. In addition, we considered research from the SE journals of TSE, TOSEM, EMSE, and IEEE Software. Lastly, we considered the top conferences dedicated to machine learning and deep learning implementations. We did this in order to capture papers focused on the creation of a DL approach, that also apply that approach to a SE task. The conferences we considered in the

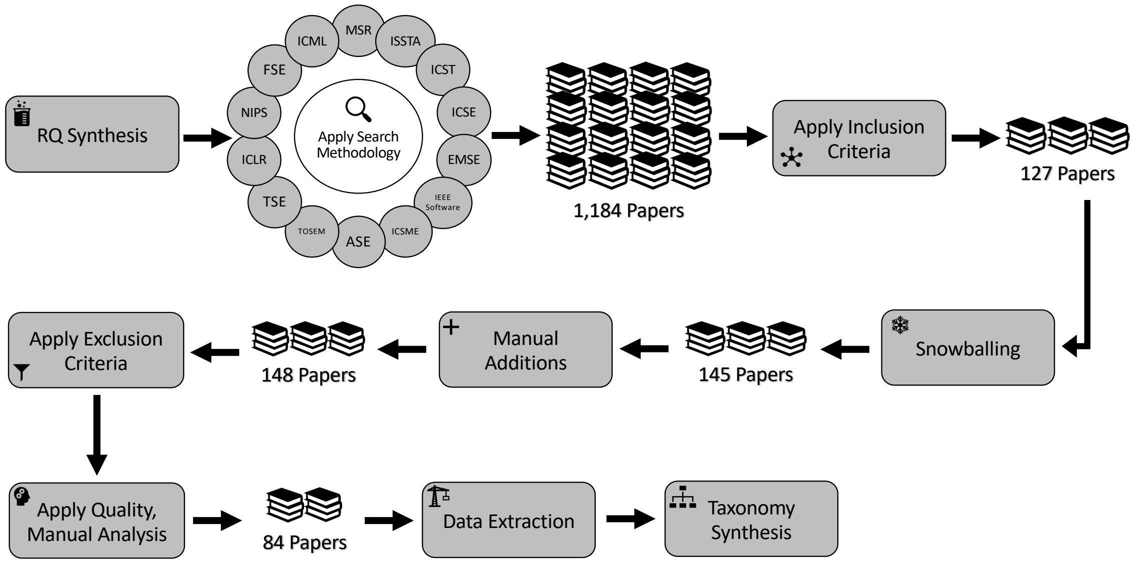


Figure 3.2: SLR Methodology

ML/DL field were ICML, ICRL, and NIPS. Once we established the venues to search, we developed a search string to query each database for relevant research. The development of the search string was determined based on terms that relate to the RQs we wanted to answer. We empirically evaluated multiple search strings using a variety of derived terms and found the string *("Deep" OR "Learning" OR "Neural")* to return the most relevant primary studies. At this point, we were ready to begin querying the respective conferences and journals for primary studies. Most of the venues we searched were found in four online databases: arXiv, IEEE Xplore, ACM Digital Library and SpringerLink. The fourth database we searched was Google Scholar, which was mainly used to access the primary studies contained within the ML/DL conferences.

The four major databases we considered provided a mechanism for advanced searching that can be used in conjunction with the search string. Although there are nuances amongst each database, the method for gathering studies was consistent. The search string provided an initial filtering of the primary studies, then additional features of the advanced search allowed us to add discriminatory criteria. These additional features are how we limited our search results by year and venue. In addition to the three major databases, we searched

Google Scholar for papers within ICML, ICRL and NeurIPS through the Publish or Perish software [266]. To search these three conferences, we had to augment our search string with SE terms that would only return papers addressing a SE task. We gathered these terms from ICSE (International Conference on Software Engineering), since they are used as topics for technical track paper submissions. We iterated through each term and appended it to our search string. The results of that search were manually inspected for relevant papers to SE. After searching the databases with the specified search string, our initial results yielded 1,184 relevant studies.

3.3.2 Filtering Through Inclusion Criteria

In this step of the SLR, we defined the inclusion criteria that determined which primary studies would be included in our taxonomy. To an extent, part of our inclusion criteria was already used to extract the primary studies. The year and the venue were used in advanced searches to filter out extraneous research that our search string returned. However, after the search phase concluded, we filtered our results based on a more strict set of inclusion criteria. This ensured that the primary studies would be a useful addition to the taxonomy and would contain the necessary information to answer the RQs. A full list of the inclusion and exclusion criteria can be viewed in table 3.1.

Table 3.1: SLR Inclusion and Exclusion Criteria

Inclusion Criteria	**Exclusion Criteria**
Study is published in the year range 2009-2019.	Study was published after January 1, 2009 or before April 1, 2019.
Study clearly defines a SE task that is addressed with a DL based approach.	Study does not solve a SE task.
Study was published in the predefined venues.	Study is outside the scope of software engineering or is published in an unaccepted venue.
Paper must identify and address the implementation of the DL framework.	Paper does not fully implement, evaluate or address the implementation of a DL approach.

The primary studies gathered through the use of the search string, the snowballing method, or through manual addition were subjected to the same criteria for inclusion. Three of the primary authors divided the works and labeled them for inclusion based on reading the abstract and methodology of the paper. A second author then reviewed each classification. If the classifications between the two authors were in disagreement, then all authors reviewed and discussed this work until a unanimous consensus was achieved. At the end of this phase of the SLR, we were able to include 127 primary studies for consideration.

3.3.3 Snowballing and Manual Additions

Our next step in the process was snowballing and the inclusion of manually added papers. After extracting primary studies from the specified venues and subjecting those studies to our inclusion criteria, we performed snowballing on the resulting studies. Snowballing helped to ensure we were gathering as much related research as possible. Our snowballing process looked at every reference within the studies passing our inclusion criteria and determined if any of those references also passed our inclusion criteria. Lastly, using our expert knowledge of the field, we manually added any related works that were missed by our systematic process. We performed this manual addition for completeness of the survey and made note of which studies were manually added in our taxonomy. From performing snowballing, we were able to add 18 new pieces of research to include in our SLR. In addition, we manually added three relevant pieces of work that were not captured by either our search string or snowballing. This resulted in 148 primary studies.

3.3.4 Exclusion Criteria and Quality Analysis

At this stage, we applied our exclusion criteria to determine if there were any primary studies which were inadequate to be a part of our taxonomies. This involved a significant manual analysis to closely analyze how each study incorporated the use of DL when analyzing a SE task. Particularly, we found many studies that only used a DL based approach as a means for evaluation. We also found instances where DL was discussed as an idea or part of the future work of a study. We therefore excluded these works to ensure that every paper we analyzed implemented and evaluated a full DL approach to address a SE task. This resulted in 84 studies, all of which were included in the data extraction and taxonomy synthesis phases. In figure 3.3 we show the breakdown of venues that our final 84 studies came from.

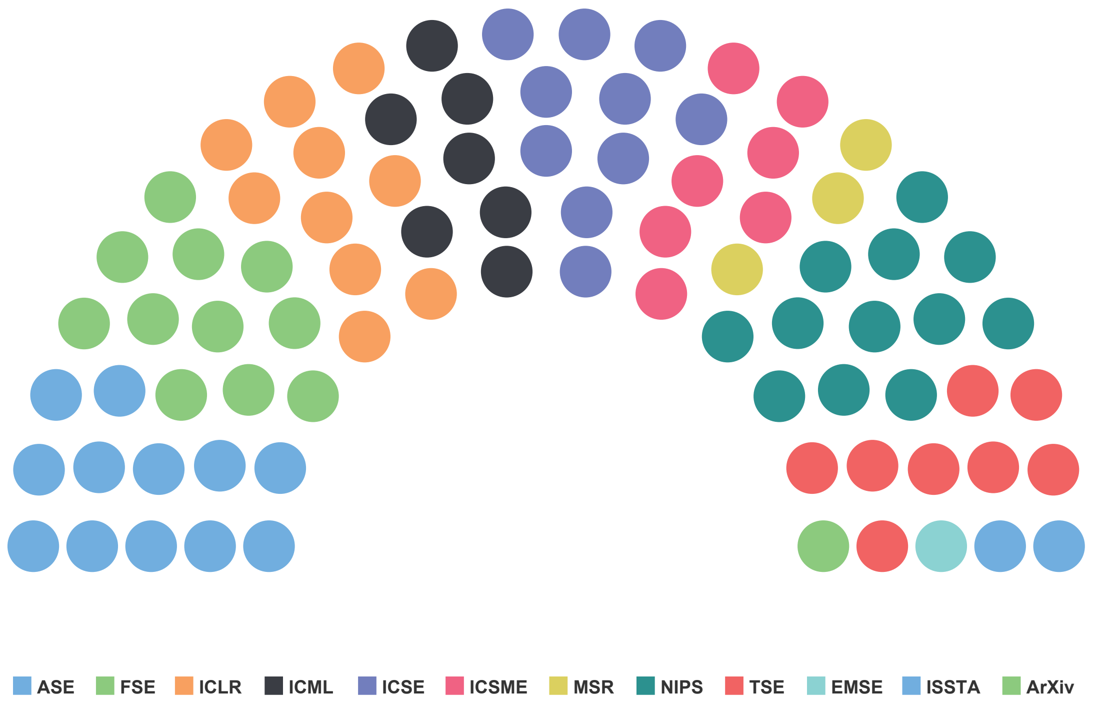


Figure 3.3: Venue distribution of DL4SE

3.3.5 Data Extraction and Analysis

Our next step in the SLR methodology was to extract the necessary data from the primary studies. This step involved the development of an extraction form where the authors agreed about specific information to extract from reading the papers. We would then use this information to create complete taxonomies of DL in SE. Each author was tasked with extracting data from a subset of the 84 primary studies classified as DL4SE. However, the results of the extraction phase were confirmed by at least two additional authors to ensure that all important details were gathered and that papers were properly represented by the taxonomy. We then performed data analysis of our extracted data to determine the types of relationships between features. We explain our process of data analysis in the next section.

3.3.5.1 Data Analysis

Our final task was to analyze the data extracted from the SLR process. We performed Exploratory Data Analysis (EDA) in conjunction with Confirmatory Data Analysis (CDA), in order to obtain descriptive statistical attributes of our dataset. The mined facts and associations allowed us to ascertain and support the conclusions of this SLR.

The Systematic Literature Review of DL4SE required formal statistical modeling to refine the answers for the proposed research questions and to help formulate new hypotheses to be addressed in the future. Hence, we introduced DL4SE-DA, a set of statistical processes and data mining pipelines that uncovered hidden relationships among the literature reporting a DL based approach in SE. Such hidden relationships were collected and analyzed to illustrate the state-of-the-art DL techniques employed in the SE context.

Our DL4SE-DA is a simplified version of the classical Knowledge Discovery in Databases, or KDD [92]. The KDD process extracts knowledge from a DL4SE structured database. This structured database was the product of multiple iterations of data gathering and collection from the inspected literature. The KDD involves five stages:

1. **Selection**. This stage was led by the taxonomy process explained in the beginning of this section. After collecting all the papers and creating the taxonomies, we organized the data into 35 features or attributes that you find in the repository. In fact, we manually engineered features from the DL4SE papers. Some of the features we considered were: venue, year published, type of paper, metrics, data-scale, type of tuning, learning algorithm, SE data, and so on.

2. **Preprocessing**. We applied a preprocessing technique that transformed the features into the correct type (nominal), removed outliers (papers that do not belong to the DL4SE), and re-inspected the papers to extract missing information produced by the normalization process. For instance, we normalized the feature "metrics" into "MRR", "ROC or AUC", "BLEU Score", "Accuracy", "Precision", "Recall", "F1 Measure", and "Other Metrics". "Other Metrics" refers to unconventional metrics found during the extraction. Similarly, the same normalization was applied to other features like "SE Data" and "Reproducibility Types". This separation into more detailed classes contributes to a better understanding and classification of the paper by the data mining tasks or methods.

3. **Transformation**. In this stage, we omitted the use of any data transformation method. We performed a Principal Component Analysis (PCA) to reduce 35 features into 2 components for visualization purposes.

4. **Data Mining** In this stage, we used two distinct data mining tasks, Correlation Analysis and Association Rule Learning. We decided that the goal of the KDD process should be oriented to uncover hidden relationships on the extracted features Correlations and Association Rules. A clear explanation is provided in the subsection "Data Mining Tasks for the SLR of DL4SE".

5. **Interpretation/Evaluation** We used the Knowledge Discover to automatically find patterns in our papers that resemble "actionable knowledge". This actionable knowledge was generated by conducting a reasoning process on the data mining outcomes. This reasoning process produced an argument support analysis formally found in our online repository [280].

We used RapidMiner [10] as our software tool to conduct the data analysis and the procedures and pipelines for using this tool are published in our repository. Before carrying out any mining tasks, we decided to address a basic descriptive statistics procedure for each feature. These statistics exhibit basic relative frequencies, counts, and quality metrics. We revised 4 quality metrics: ID-ness, Stability, Missing, and Text-ness. ID-ness measures to what extent a feature resembles an ID, where as the "title" is the only feature with a high ID-ness value. Stability measures to what extent a feature is constant (or has the same value). Missing refers to the number of missing values. Finally, Text-ness measures to what extent a feature contains free-text. The results for each feature could be navigated once the SRL process was ran in RapidMiner. We employed two well-established data mining tasks to better understand our data: Correlations and Association Rule Learning.

3.3.5.2 Correlations

Due to the nominal nature of the SLR features, we were unable to perform classic statistical correlations on our data (i.e. Pearson's correlation). However, we adapted an operator based on attributes' information dependencies. This operator is known as "mutual information" [190]. Similar to covariance or Pearson's correlation, we were able to represent the outcomes of the operator in a confusion matrix.

The mutual information measures to what extent one feature knows about another one. High mutual information values represent less uncertainty; therefore, we built arguments such as "the deep learning architecture used on a paper is less uncertain (or more predictable) given a particular SE task" or "the reported architectures within the papers are mutually dependent with the Software Engineering task".

The difference between correlation and association rules depends on the granularity level of the data (e.g., paper, feature, or class). The correlation procedure was performed at the feature level, while the association rule learning was performed at the class or category level.

3.3.5.3 Association Rule Learning

For generating the association rules, we employed the classic FP-Growth algorithm (FP stands for Frequent Pattern). The FP-Growth computed frequently co-occurring items in our transactional dataset (see DL4SE-Dataset.csv in [280]) with a minimum support of 0.95. These items comprise each class per feature. For instance, the feature "Loss Function" exhibited a set of classes (or items) such as "NCE", "Max Log Likelihood", "Cross-Entropy", "MSE", "Hinge Loss", "N/A-Loss", and so on. The main premise of the FP-Growth can be summarized as: if an item set is frequent (i.e. MSE, RNN), then all of its item subsets are also frequent (i.e. MSE and RNN).

Once the FP-Growth generated the item sets (e.g. MSE, Hinge Loss), the algorithm started to check the item sets' support in continuous scans of the database (DB). Such support measures the occurances of the item set in the database. The FP-Growth scans the DB, which brings about a FP-tree data structure. We recursively mined the FP-tree data structure to extract frequent item sets [118]. Nonetheless, the association rule learning requires more than the FP-tree extraction.

An association rule serves as an if-then statement (or premise-conclusion) based on the frequent item set pattern. Let's observe the following rule mined from our dataset: "Given that an author used Supervised Learning, we can conclude that their approach is irreproducible with a support of 0.7 and a confidence of 0.8". We observe the association rule has an antecedent (i.e., the item set Supervised Learning) and a consequent (i.e.,the item set Irreproducible). These relationships are mined from item sets that usually have a high support and confidence. The confidence is a measure of the number of times that premise-conclusion statement is found to be true. We tuned both parameters to be greater than 0.7.

We refute the idea that association rule learning avoids spurious correlations. Regardless, we organized the rules into an interconnected net of premises/conclusions to find explanations around techniques and methodologies reported on the papers. Any non-logical rule is disregarded as well as rules that possess a lower support and confidence.

The results of our exploratory data analysis and source of the information provided in this SLR can be found in our online repository [280].

3.3.6 Taxonomy Synthesis

In the next section, we discuss the resulting taxonomies created from the primary studies found through our systematic approach. We use these taxonomies in order to answer the RQs presented in the beginning of this work. These RQs naturally define a larger taxonomy that researchers can use to determine what types of SE tasks can be better studied using certain DL based approaches, as well as look for future applications of DL to model complex software artifacts. We oriented the results of our RQs to provide an understanding about the process of applying a DL based approach in SE.

3.4 Results

3.4.1 RQ1: What types of SE tasks have been addressed by DL-based approaches?

In this RQ, we explored what types of SE tasks DL based approaches are applied to. DL models are excellent at identifying an appropriate target hypothesis given complex, hierarchical, abstract representations of data points. This makes them useful models to apply to SE tasks, as these models can automatically extract useful, hierarchical features within SE data that developers or researchers could not find manually. These features can be used to establish relationships between the facets of input data and target variables. Those relationships can then be used in the prediction of a target, given an input data point that was previously unseen by the model.

Out of the 84 papers we analyzed for this SLR, we found 23 separate SE tasks where a DL based approach had been applied. Figure 3.4 gives a visual breakdown of how many SE tasks we found within these 84 primary studies across a 10 year period. Unsurprisingly, there was very little work done between the years of 2009 and 2014. However, even after the DL breakout paper of AlexNet by Krizhevsky et al. [156] in 2012 showing the usefulness of DL in image classification, no work on SE tasks took place until 3 years later in

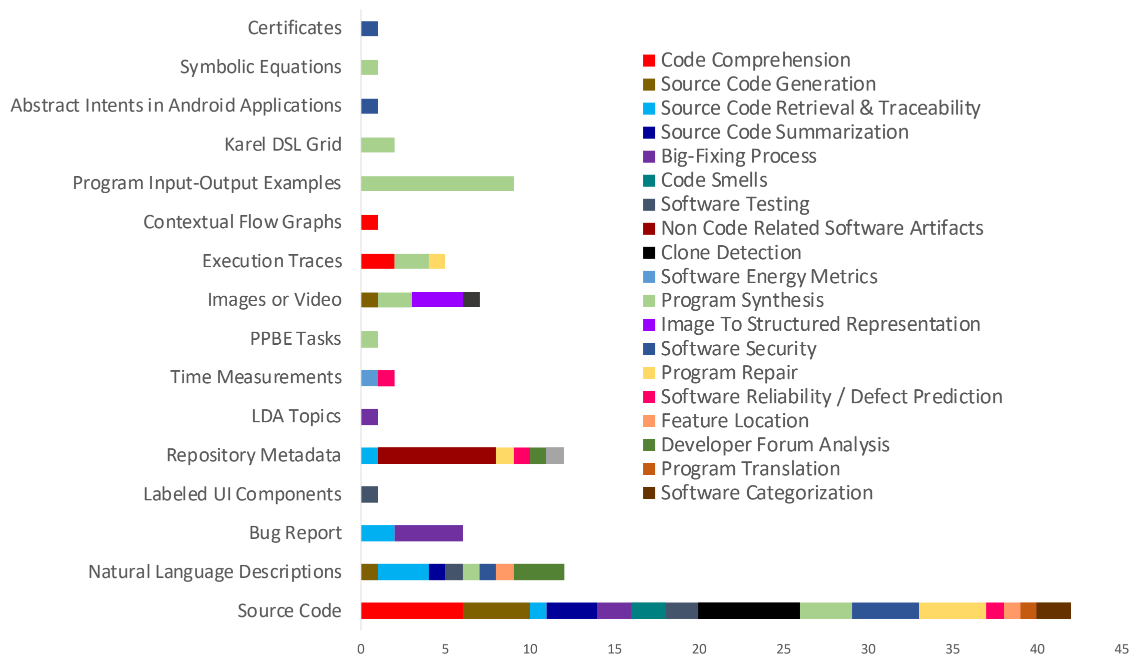


Figure 3.4: SE Tasks Per Year

2015. The first SE tasks to use a DL technique was that of Code Comprehension, Source Code Generation, Source Code Retrieval & Traceability, Bug-Fixing Process, and Feature Location. Each of these tasks uses source code as their primary form of data, which is due to its large abundance in online repositories. This is a result of the Software Engineering community's shift toward becoming more open sourced, allowing for access to millions of software projects through popular hubs such as GitHub, Gitlab, and Bitbucket. Access to a large amount of data and a well understood task is important for DL4SE, since in order for DL to have an effective application two main components are needed: i) a plethora of data to support the training of multi-parameter models capable of extracting complex representations and ii) a task that can be addressed with some type of predictable target. One of the major benefits of DL implementations is the ability for automatic feature extraction. However, this requires the ability to find data that has a distinct relationship with the predicted target variable.

It wasn't until 2017 that DL was being used extensively in solving SE tasks as shown in Figure 3.4, with a large increase in the number of papers, more than doubling over the previous 2 years. During this period, there was also a more diverse set of SE tasks attempting to be solved, such as Code Smells, Software Security, and Software Categorization. However, there are three main SE tasks that have remained active across the years: Code Comprehension, Source Code Retrieval & Traceability, and Program Synthesis[1]. The most popular of the three being Program Synthesis, composing a total of 20 papers out of the 84 we collected. We suspect that a variety of reasons contribute to the multiple applications of DL in program synthesis. First and foremost, is that the accessibility to data is more prevalent. Program synthesis is trained using a set of input-output examples. This makes for high quality training data, since one can train the DL model to generate a program, given the set of formal specifications. The second largest reason is having a clear and established relationship between the training examples and the target program. These relationships are what deep learning methods thrive on as they can adequately model this relationship without the need for hand-crafted features.

We display the full taxonomy in Table 3.2, which associates the cited primary study paired with its respective SE task. Interestingly, we encountered a few tasks that may not initially seem as though they are SE related. For example, tasks such as image to structured representation, software security and software energy metrics do not initially appear directly related to SE. However, upon further analysis we found these tasks to be highly related to SE and therefore were included in our taxonomy.

When performing our exploratory data analysis, we found the the SE task was the most informative feature we extracted, meaning that it gave us the greatest insight into what other features may have been associated. In particular, we found that SE tasks had strong correlations to data preprocessing (1.80), the loss function used (1.45) and the architecture employed (1.40). Therefore, we can predict that there are DL framework components that are better suited to address specific SE tasks. Throughout the remaining RQs, we look to

[1]We recognize that Program Synthesis does not appear in Y19, however since we did not collect papers for all of Y19, we are confident that Program Synthesis papers would appear.

Table 3.2: SE Task Taxonomy

SE Task	Papers
Code Comprehension	[167, 43, 219, 122, 161, 18, 21, 162]
Souce Code Generation	[142, 123, 282, 287, 110, 54]
Source Code Retrieval & Traceability	[58, 112, 108, 76, 159, 52]
Source Code Summarization	[58, 271, 23]
Big-Fixing Process	[166, 202, 48, 76, 159]
Code Smells	[179]
Mobile Testing	[183]
Traditional Software Testing	[102, 71, 242]
Non Code Related Software Artifacts	[136, 184, 237, 65, 67, 66]
Clone Detection	[170, 285, 232, 99, 176, 261]
Software Energy Metrics	[230]
Program Synthesis	[100, 50, 294, 78, 250, 203, 227, 178, 36, 79, 49, 270, 61, 125, 28, 300, 241, 88, 89, 173]
Image To Structured Representation	[75, 55, 200]
Software Security	[119, 297, 53, 99, 120, 72]
Program Repair	[46, 274, 120, 264, 181]
Software Reliability / Defect Prediction	[116, 277, 281]
Feature Location	[70]
Developer Forum Analysis	[56, 289, 175, 191]
Program Translation	[60]
Software Categorization	[27]
Code Location Within Media	[211]
Developer Intention Mining	[129]

expand upon the associations we find to better assist software engineers in choosing the most appropriate DL components to build their approach.

Although the applications of DL based approaches to SE related tasks is apparent, there are many research areas of interest in the SE community as shown in ICSE'20's topics of interest[2] that DL has not been used for. Yet, many of these topics have no applicability for a DL based solution, such as Cloud computing, human and social aspects of software engineering, parallelization, distributed and concurrent systems, etc. These topics are either out of scope for DL based automation or require more stringent methods that are

[2]https://conf.researchr.org/track/icse-2020/icse-2020-papers#Call-for-Papers

not stochastic. Still, some of the interesting topics that we believe can see benefit from DL approaches have yet to be explored by the research community or are underrepresented. Topics of this unexplored nature include refactoring and program analysis, whereas topics which are underrepresented include software testing, both traditional systems and mobile, software documentation, feature location, and defect prediction. Some possible reasons why certain SE tasks have yet to gain traction in using DL could be due to the following:

- There is a lack of available, clean data in order to support such DL techniques,
- The problem itself is not well defined, such that a DL model would struggle to be effectively trained and optimized
- No current architectures are adequately fit to be used on the available data.

3.4.2 RQ2: How are software artifacts being extracted, prepared and used in DL based approaches for SE tasks?

In this research question, we analyzed the type of data that is modeled by the DL approaches in SE. We wanted to address a few aspects of how SE data is used, how the data is extracted or preprocessed, the scale of the data being modeled and the type of learning based on the data available. All four of these points are a crucial step in understanding how the approach is able to model specific software artifacts. These points also open up a discussion about future research toward the improvement of data manipulation and preprocessing steps to increase accuracy of the DL approaches. In many cases, data must be cleaned or filtered in such a way that limits the ability for the approach to model a desired relationship. Since DL models are frequently seen as a black-box in terms of explainability and interoperability, it is important that the relationships between input and output are accurately reflected within the dataset.

3.4.2.1 What types of data are being used?

When analyzing the types of data that were being used in DL based approaches, we provided a high level classification, along with some intuition, as to why some types of data were used for particular tasks. We found that overwhelmingly the most common type of data being used is source code at different granularities. Within this SLR, we found source code being used at the binary level, code snippet level, method level, class level and project level. The high volume of approaches that learn from source code do so for a number of reasons. The first is that source code is plentiful and can be found from a number of different online repositories. This availability aids in the fact that DL models are extremely data hungry in order to learn a representative target function. The second reason is that a majority of SE tasks revolve around the generation, understanding, debugging and documentation of source code, which makes the desire to automate these tasks for developers extremely high.

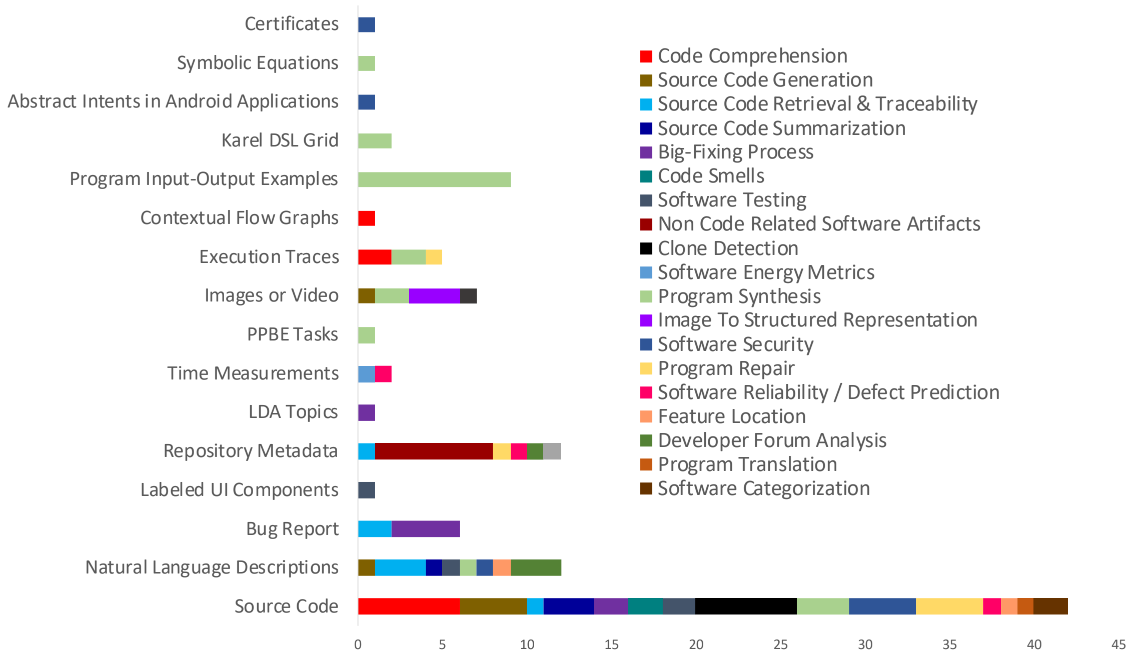


Figure 3.5: Data Types With SE Tasks

We find that out of the 111 uses of data in DL4SE papers, 49 of them are attributed to source code. The reason that there are 111 uses of data, which is larger than the 84 papers we analyze, is because many primary studies use multiple different data types in conjunction to learn from. Although source code is the primary type of data that DL models attempt to learn from, we found that the type of data used is heavily dependent on the SE task. Thus, the SE tasks that focus on the comprehension, generation, summarization and testing of source code will frequently use some granularity of source code to comprise their dataset. However, there are many SE tasks that address problems where source code may not be the most representative type of data to learn from. As an example, the SE tasks of program synthesis primarily uses input and output examples to comprise their dataset. This type of data incorporates attributes, which more strongly correlates to the relationship the engineer desires the model to learn. This pattern continues for SE tasks that can learn from textual data, software artifacts and repository metadata. Our exploratory data analysis also highlighted these points. Our analysis demonstrated that the main data employed in DL4SE papers are input-output (I/O) examples, source code, natural language, repository metadata and visual data. In fact, these categories of data types comprised 78.57% of the distribution we found in this SLR. In conjunction with this finding, execution traces and bug reports represent $\sim 10\%$ of the distribution of data. The remaining $\sim 12\%$ is comprised of a variety of different data types. The full breakdown of data types can be viewed in figure 3.5.

Although we find a variety of different data types used, the data must be accessible in large quantities in order to extract and learn the relevant target hypothesis. With an increase in the number of online repositories, the ability to access these repositories has increased over time. This can partially explain the dramatic increase in DL papers addressing SE tasks, which was a trend discussed in 3.4.1. Interestingly, the growth of online repositories does not only increase the amount of accessible source code but also other types of software artifacts. For example, analyzing SE tasks such as software security, program translation, etc. have only recently been addressed, in part due to the increasing amount of accessible online repository data/metadata. We hope to see this trend continue with new benchmarks and datasets that will be released with future publications.

When exploring online repositories, one attribute of the data that can be problematic for the use of some DL approaches and should be considered is that the data is frequently found unlabeled, meaning that there is no inherent target to associate with this data. For unsupervised learning techniques this is not problematic, however, for supervised learning techniques this type of data is not usable without first establishing a target label for the data. The use of unlabeled data from online software repositories was foreshadowed in previous work [282]. Our findings corroborate this study, as many labeled and unlabeled datasets are used within DL based approaches to address SE tasks. The use of labeled versus unlabeled data can be ascertained based on the type of DL model chosen. However, we find that researchers mined unlabeled data and label it themselves in order to use a particular DL architecture. With the increase in the number of online repositories and the variety of unlabeled data within those repositories, we expected the number of DL based approaches analyzing software artifacts to increase.

Throughout our study, we noticed a few SE artifacts that have not yet been analyzed through a DL based approach. Specifically, we noticed that software requirements, software dependencies and software licensing have not been mined or studied using a DL architecture. We emplore the SE community to continue to mine and analyze different SE artifacts for the automatization of SE tasks. The continuation of this process will lead to more datasets and benchmarks for future DL model comparisons.

3.4.2.2 How is data being extracted and pre-processed into a format that the DL model can appropriately learn from?

In 3.4.2.1, we analyzed what types of data were being used to model complex relationships between software artifacts and the target output function as it relates to a specific SE task. In this RQ, we looked for the mechanism behind the extraction and preprocessing of this data. DL models extract complex, hierarchical features based on the input data. This data is subjected to a number of hidden layers and hidden units, which are responsible for the feature engineering process. The learning associated with these models is the process of tuning the weight matrices between the layers. This tuning enables the extraction of the necessary features to appropriately model the target hypothesis [104]. In many instances, the preprocessing of data can be handled by tools designed to format input to DL models. These tools are necessary because DL models cannot accept raw data mined from the online repositories. Rather, the data is subjected to a preparation and formatting process before given to the DL model. This is an extremely important step for those applying DL to SE, since the process can dramatically affect the performance of the model and the resulting target hypothesis. For example, some primary studies represent source code in an abstracted format. This abstraction will inherently remove some of the complex features within the dataset. This process has both pros and cons. Examples of pros are that abstraction addresses some of the learning problems, such as the open vocabulary problem and generalizability, that can arise when applying a DL model to a SE task. However, a limitation of the abstraction process is that it removes some complex, hierarchical features from the dataset, which can limit the model's ability to accurately predict the target in some cases.

The pros and cons of any preprocessing step must be carefully considered before being implemented. The authors should understand what inherent limitations the preprocessing step can impose on the model. Just as importantly, the preprocessing process must be documented and explained in order for a study that uses this technique to be replicated or used by developers. If one were to use a DL model without also replicating the preprocessing steps, the results of the experiment would be inconclusive.

In light of the importance of preprocessing in DL based approaches, we looked to generate a taxonomy of preprocessing and formatting techniques. We also looked to analyze the relationships between the SE tasks, types of data and the preprocessing steps taken. Unsurprisingly, the preprocessing and formatting techniques of data can be highly specific. This is in part due to the features that the authors look to ascertain from their dataset.

The desired features should help drive the decision on what preprocessing and formatting techniques to use. A preprocessing pipeline that removes or overshadows the desired features from the dataset will ultimately lead to a poor performing model. A breakdown of the preprocessing and formatting techniques used in the primary studies we analyzed can be found in figure 3.6.

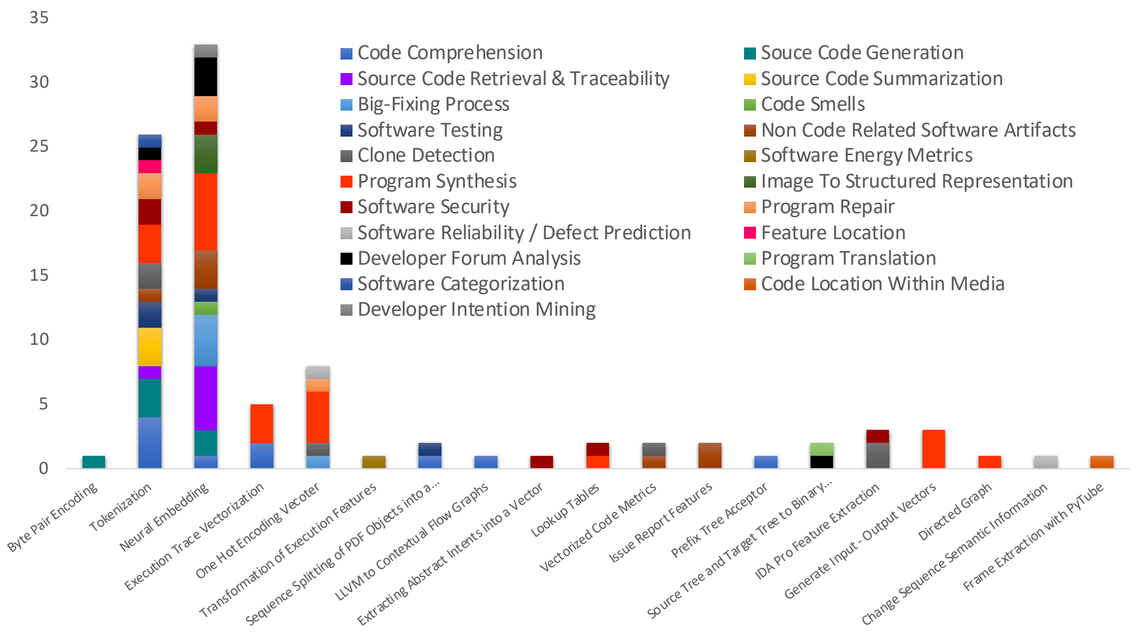


Figure 3.6: Preprocessing Techniques by SE Task

It is important to understand that figure 3.6 shows generally how the preprocessing and formatting techniques break down by SE task. However, within the same preprocessing technique, there can be small nuances across different studies. There is rarely a standard method for the ability to preprocess the data in such a way that will fit the desired DL model. Therefore, we suggest looking into the primary studies for more details pertaining to a specific preprocessing or formatting technique of interest. However, there are some dominant techniques researchers use in order to prepare their data to be analyzed by a DL model. Specifically, the use of tokenization and neural embeddings are popular for a variety of different tasks. This was expected, since we knew that source code was the primary data type used for DL models in SE. Tokenization of that source code is an effective process for preparing different granularities of code to be fed into a DL architecture.

Even more popular than the use of tokenization was the use of neural embeddings (57.14% of the distribution). This technique can use canonical machine learning techniques or other DL architectures to process the data, meaning that the output from these other models are then used as input to an additional DL architecture. We found that Word2Vec and recurrent neural networks (RNNs) were the most popular types of models for the preprocessing of data. Again, this relates to the high amount of sequential data that is used in SE that DL models can take advantage of. Source code, natural language and other repository metadata can have a sequential nature, which can be captured by these techniques. In both cases, the outputs of these models are a series of vectorized numerical values, which capture features about the data they represent. These vectors do not require much additional manipulation before they can be used again as input data to another DL model.

Although tokenization and neural embeddings are the most popular type of preprocessing techniques, there are many more that are required for a different types of data or SE tasks. The ability to accurately represent the data to the DL model is what provides training data for the algorithm to learn from. Any alteration to this process can result in the learning algorithm focusing on different features, leading to an entirely different final hypothesis.

Our exploratory data analysis discovered that the steps taken in preprocessing were strongly dependent on the DL architecture used (0.97). Intuitively, this should make sense since the data must be transformed into a vector that the DL architecture can accept. If the preprocessing steps and architecture are disjointed, then the model will lack the ability to accurately and adequately represent the final hypothesis. Therefore, we encourage future researchers to analyze the preprocessing steps taken by previous authors given the DL architecture they employed. Vice versa, if certain preprocessing steps need to be taken, future researchers should analyze what DL architectures coincide with the resulting vectorized format of the data.

3.4.3 What type of exploratory data analysis is conducted to help inform model design and training?

DL models are incredibly efficient at finding patterns in a large set of data without the intervention of the DL engineer. However, these models have one significant flaw that is frequently overlooked in the SE community: the quality of the data significantly impacts the quality of the learned target function. The analysis and quality of data analyzed has been shown to have adverse effects on machine learning models in the task of code clones [16], however, the issues discussed in this work transcend the particular SE task. Given these limitations, it is the responsibility of the researcher to analyze and evaluate the data, such that it appropriately represents the relationship between input and output. We refer to problems within the dataset as confounding variables, which can explain the results of the learned final hypothesis. In this RQ, we analyzed our primary studies to determine if precautions were taken to limit the number of confounding variables. Primarily, we were interested to see if there exists any sampling bias or data snooping when applying these DL models to SE tasks.

First, we wanted to describe the issues involving sampling bias and data snooping. We found that when DL is applied to SE, there were many instances where there were no methods to protect against these confounding variables. This leads to doubt surrounding the results of the primary study and can severely limit the conclusions that can be drawn from the learned target hypothesis. Sampling bias is a result of collecting data from a distribution incorrectly or in such a way that it does not accurately represent the distribution due to a non-random selection process [216]. The repercussions of sampling bias normally results in a target hypothesis that is only useful for a small subset of the actual distribution of the data. Authors can unknowingly make claims about their predicted target function, which overestimate the actual capabilities of their model, when sampling bias is not appropriately considered. The best mechanism for reducing sampling bias is to limit the amount of criteria and filtering the data is subjected to. Additionally, extracting as much useful data as possible will help to effectively train the model and will reduce sampling bias.

Data snooping is the process of using data more than once for the purpose of training and testing the model. This means that the data the model was trained on is also incorporated into the testing and evaluation of the model's success. Since the model has already seen the data before, it is unclear whether the model has actually learned the target hypothesis or simply tuned itself to effectively reproduce the results for previously seen data points (overfitting). The overall result of data snooping is an overinflated accuracy from the model. Therefore, it is important that software engineers apply methods to reduce data snooping in order to protect the integrity of their results. This can be done through a simple exploratory data analysis or the removal of duplicate data values within the training and testing sets.

To evaluate the level at which methods to mitigate sampling bias and data snooping were already being done, we noted instances within the primary studies where the authors did some type of exploratory data analysis before training. Ideally, the authors performed some type of exploratory data analysis in order to identify if the dataset that has been extracted is a good representation of the target distribution they are attempting to model. Additionally, data exploratory analysis should provide insight into data snooping of the model by determining if the training dataset is mutually exclusive from the validation and testing datasets.

There was a noticeable number of primary studies (over 13) that did not mention or implement any methods to mitigate sampling bias or data snooping. However, this number is a bit misleading as we only analyzed primary studies for any sign of exploratory analysis on their dataset. This exploratory analysis ranged from basic statistical analysis to an in-depth study about the dataset that looked to remove sampling bias and data snooping. Our primary objective within this RQ was to bring attention to the oversight of confounding problems within the data. DL models are entirely dependent on the data to properly model the target hypothesis, thus we wanted to encourage future software engineers to heavily analyze their dataset and their methods for extraction. We also note that as time has progressed, the primary studies we analyzed generally included more details about their data exploration process. We hope to see this trend continue as more DL implementations are used to address SE tasks.

3.4.4 RQ3: What deep learning models are used to support SE tasks?

In 3.4.4 we looked to investigate the uses and applications of DL models to represent SE artifacts. We primarily focused on two key aspects of a deep learning framework: the architecture and the learning algorithm. Analyzing a DL architecture can grant a lot of insight into the types of features the authors anticipate will be extracted from the dataset. Additionally, we wanted to empirically determine if certain architectures pair with specific SE tasks. Additionally, we wanted to know what types of diversities existed within the DL architectures and why those idiosyncrasies were important when considering the specific SE task at hand. However, we not only considered the DL architectures, which provided insight to the automated feature engineering inherent to DL, but we also analyzed the learning algorithms used in conjunction with those architectures. Specifically, we wanted to create a taxonomy of different learning algorithms and determine if there was a correlation between the DL architectures, the learning algorithms and the SE tasks.

3.4.4.1 What types of model architectures are used to perform automated feature engineering of the data related to various SE tasks?

In 3.4.4.1, we wanted to discuss the kinds of DL models software engineers are using to address SE tasks. The DL architectural choice is heavily dependent on the type of data being analyzed and what types of features the researcher desires to automatically extract. For example, the use of a recurrent neural network (RNN) would automatically extract features related to the sequentialness of the data being analyzed. Likewise, the use of a convolutional neural network would look for capturing the spatial relationship between pixels of visual data. Thus, knowing the type of architecture allows us to infer the types of features that will be automatically extracted.

Our main objective in 3.4.4.1 was to see the variety of different architectures applied to certain SE tasks. We wanted to know if there are some architectures that are more widely used and of those models implemented, which have shown to have success in modeling the data. From this analysis, we can determine if there are other DL architectures that have not been applied to a specific SE tasks, thus providing the SE community with potential future research directions. However, we first take note of all DL architectures found in the primary studies, which can be seen in the taxonomy created in figure 3.7.

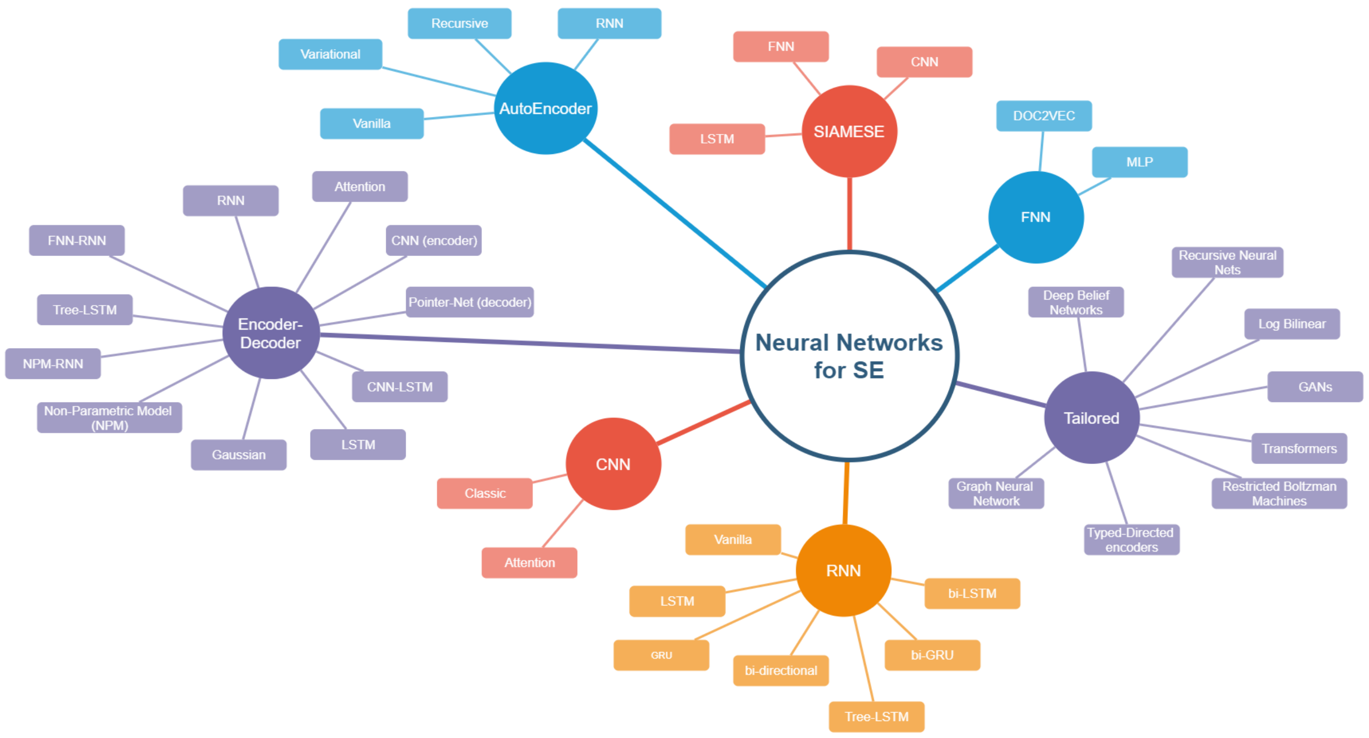


Figure 3.7: DL Model Taxonomy

Looking into the initial breakdown of the taxonomy, we found that the use of recurrent neural networks (RNNs) was highly prevalent. RNNs excel in modeling sequential data since the architecture is formulated such that a weight matrix is responsible for representing the features between the sequence of inputs [104]. In particular, source code is highly sequential and the prediction of the next source code token can be highly reliant on the surrounding tokens. Since the most used type of data we found in 3.4.2.1 was source code, it made sense to see many of our primary studies implementing a RNN to model their data. Another architecture we saw frequently applied to SE tasks was the encoder-decoder architecture. In this architecture, a datum is translated into a feature rich, latent representation, which is then translated back into altered raw data. Encoder-decoder networks are used for a variety of tasks, but they excel at understanding sequential data. Importantly, RNNs are normally a vital component of an encoder-decoder model, however, these models can also incorporate feed forward neural networks, convolutional neural networks and more. The popularity of this architecture aligns with the findings in 3.4.2.2, where neural embeddings were a popular form of preparing the data for the DL model. The encoder's responsibility is to translate the raw data into a latent representation that the decoder is capable of understanding and decoding into the target. We frequently found that neural networks serve as the encoders for this architecture that take raw data and generate a neural embedding for the decoder. Therefore, since neural embeddings were such a popular preprocessing technique for data formatting and preparation, it aligns with the high prevalence of the encoder-decoder DL architecture. In our exploratory data analysis, we found that SE tasks greatly influences the architecture adopted in an approach. The mutual information value between the features of a SE task and a DL architecture is 1.40. We also note that the SE research landscape has primarily focused on textual based problem, which includes source code. This explains why RNNs are used in 40.48% of the papers analyzed in this SLR. The encoder-decoder architecture was also seen frequently (19% of papers), where they take advantage of using RNNs for both the encoder and decoder portion of the model.

We can glean some of the popular types of architectures through our basic taxonomy, however, due to the close relationship between the automated features extracted and the DL architecture, we wanted to explore what architectures are popular within specific SE tasks. We found a large variety of different networks that have been applied to different SE tasks as seen in figure 3.8. As expected, most of the SE tasks having to do with source code generation, analysis, synthesis, traceability and repair, make use of RNNs and encoder-decoder models. This is because source code has many features that are embedded in its sequential nature and these features can be easily captured by RNNs and encoder-decoder models. Likewise, SE tasks involving the use of images or media data have convolutional neural networks commonly applied to them.

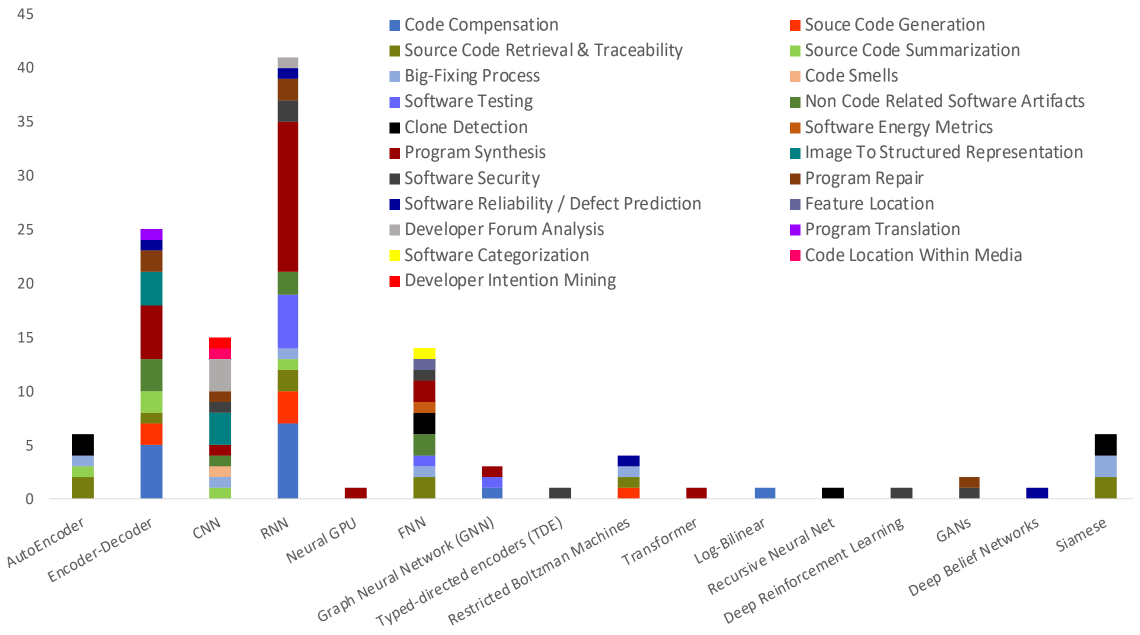


Figure 3.8: DL Architectures by Task

We wanted to bring attention to some of the less popular types of DL architectures, such as: siamese networks, deep belief networks, graphical neural networks and autoencoders. Many of these architectures have only been applied to a few tasks but it is important to note that these architectures have only recently gained any interest. We believe that these networks have a lot of potential when applied to SE tasks and could lead to new research directions. We also felt it important to mention the lack of deep reinforcement learning within the SE community. Deep reinforcement learning excels at modeling decision-making tasks. One could argue that deep reinforcement learning is perfect for SE since SE is comprised of decisions about code structure and algorithm implementation. This is a fairly open area of DL in SE that has not really been explored. Only the SE task of software security had a DL framework that involved the use of deep reinforcement learning.

In addition to the discussion around the DL architectures and their relations to particular SE tasks, we wanted to briefly discuss some trends in the explicit and implicit features extracted from these models. As we discussed in 3.4.2.2, it is common for data to be fed into DL models only after being subjected to certain preprocessing steps. However, in supervised learning, once that data has been preprocessed, the DL model automatically extracts implicit features from the preprocessed data in order to associate those features with a label or classification. In unsupervised learning, the model extracts implicit features from the preprocessed data and groups similar datum together as a form of classification. We refer to the preprocessing steps as highlighting certain explicit features, since these steps frequently perform dimensionality reduction while maintaining important features. In our analysis we found the most common techniques for highlighting explicit features to be tokenization, abstraction, creating a neural embedding or vectorizing a latent representation. These techniques attempt to highlight explicit features that are uniquely tailored to the data being analyzed. Once the data is fed into the model itself, the model is responsible for extracting implicit features to learn a relationship between the input data and target hypothesis. These features are what the weights of a neural network are intended to represent and are used to model the target hypothesis. For example, an LSTM is going to find temporal dependencies of tokenized data such that it can use those features to understand the sequential nature of the data and use that to predict the next token in a sequence. A Tree-LSTM can learn the temporal dependencies of tree based structures, such as a program dependency graph (PDG). This allows the model to understand how the nodes of the tree relate to one another, thus allowing the model to predict additional nodes or similar tree-like structures. The extraction of explicit and implicit features are the reasons that the model is capable of modeling a target function, which can be used to predict the targets of unseen data points.

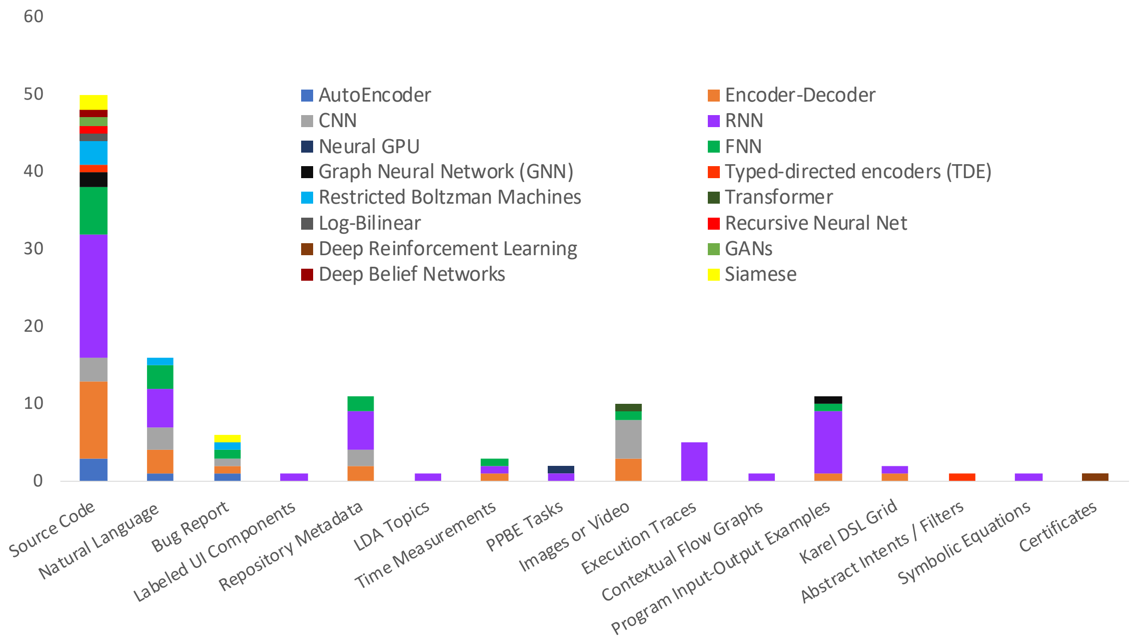


Figure 3.9: DL Architectures by Data Type

Figure 3.9 shows a breakdown of DL architectures by the type of data they are learning from. This relationship between data and architecture is important since the architectures is partially responsible for the type of implicit feature being extracted. For example, images and other visual data are commonly represented with a CNN. This is because CNNs are especially proficient at modeling the spatial relationships of pixel data. There are many other trends seen when associating data types and DL architectures. We hope that future researchers can perform exploratory data analysis and discover the architectures that are optimal when extracting implicit features from the data. These features can then be used to model the target function of interest.

3.4.4.2 What learning algorithms and training processes are used in order to optimize the models?

In addition to the variety of DL models that can be used within a DL based approach, the way in which the model is trained is very reliant on the learning algorithm chosen. The taxonomy associated with the learning algorithm is extremely diverse and can be classified in three primary ways: by the weights associated with the model and how they are adjusted, by the way the overall error associated with prediction is calculated, and by the optimization algorithm, which can systematically adjust the parameters of the learning algorithm as training progresses. Learning algorithms for a DL approach in SE are primarily used without any alteration or adjustment and are based in mathematical principals that adjust the weights of the model. Since researchers in SE are primarily interested in the use of DL models, rather than adjusting the algorithms that power them, a large majority of the studies we analyzed share in the use of out-of-the-box learning algorithms to power their DL framework. The largest commonality among the studies was the incorporation of the gradient descent algorithm as a way to adjust the weights within the deep neural network. The breakdown of learning algorithms throughout our SLR can be found in the following sentences. We found 65/84 of the primary studies used some version of gradient descent to train their DL model. The remaining studies used gradient ascent (2/84), VAE (1/84), policy based learning (3/84), a context free grammar learning algorithm (1/84), and many studies did not explicitly specify their learning algorithm in the paper (12/84). Our exploratory data analysis revealed that papers published in recent years (2018 and 2019) have begun to employ learning algorithms that differ from gradient descent. Instead, they incorporate a reward policy or gradient ascent.

In conjunction with analyzing how the DL approaches adjust their weights, we also quantified how these approaches calculate the loss of the model's prediction when compared to the actual target. We found that there are a variety of ways that DL based implementations calculate error. However, we did find that a majority of the papers we analyzed used cross entropy as their loss function (19/84), which was only paired with gradient descent algorithms. Other common loss functions that were used with gradient descent algorithms were negative log likelihood, maximum log likelihood, and cosine loss. There were a number of papers which did not provide any indication about the loss function within their learning algorithm. We did find that when the primary study was not using gradient descent as a way to adjust the weights associated with the DL model, the error functions used became a lot more diverse. For example, the work done by Ellis *et al.* learned to infer graphical programs from deep learning hand-drawn images. They used gradient ascent rather than descent as their learning algorithm and also used surrogate likelihood function as a way to calculate the error of the model [89]. We found that the loss function influences the learning algorithm with a mutual dependence of 0.72 bit. However, the SE community omits this parameter in 27.38% of the papers we analyzed. We found that approaches that implement reinforcement algorithms are based on a developed policy, which calculates the error associated with the action taken by the model and adjusts the weights to either reinforce or punish that action.

Lastly, we looked at the optimization algorithms to determine if there was any noteworthy pattern or intuition as to why certain optimization algorithms were used. What we discovered is that the choice of optimization algorithm is somewhat agnostic to the model, the weight adjustment algorithm and the error function. In many cases, the optimization algorithm was not reported within the primary study (54.76% of the time). Unfortunately, we were not able to ascertain whether optimization methods were never implemented or were just omited from the paper for space purposes. However, we did analyze the papers that did provide this information and found that there are four major optimization algorithms: AdaGrad, AdaDelta, RMSProp and Adam. Below, we briefly address each optimization algorithm in order to point out potential situations in which they should be used.

Adagrad is an algorithm that adapts the learning rate based on the impact that the parameters have on classification. When a particular parameter is frequently involved in classification across multiple inputs, the amount of adjustment to those parameters is lower. Likewise, when the parameter is only associated with infrequent features, then the adjustment to that parameter is relatively high [87]. A benefit of AdaGrad is that it removes the need for manual adjustment of the learning rates. However, the way that AdaGrad calculates how much to adjust the parameters is by accumulating the sum of the squared gradients. This can lead to summations of the gradient that are too large, which requires the learning rate to become extremely small.

AdaDelta was formulated out of AdaGrad in order to combat this limitation. Rather than consider all the sums of the past squared gradients, AdaDelta only considers the sum of the past squared gradients limited to a fixed size. Additionally, this optimization algorithm does not require a default learning rate as it is defined by an exponentially decaying average of the calculated squared gradients up to a fixed size [293].

RMSprop is the next optimization algorithm, however, this algorithm has never been published or subjected to peer review. This algorithm was developed by Geoff Hinton and follows the similar logic of AdaDelta. The way in which RMSprop battles the diminishing learning rates that AdaGrad generates is by dividing the learning rate by the recent average of the squared gradients. The only difference is that AdaDelta uses the root means squared error in the numerator as a factor that contributes to the adjustment of the learning rate where RMSprop does not.

Adam, the last of our optimization algorithms discussed, also calculates and uses the exponentially decaying average of past squared gradients similar to AdaDelta and RMSprop. However, the optimization algorithm also calculates the exponentially decaying average of the past gradients. Keeping this average depending on gradients rather than just the squared gradients allows Adam to introduce a term which mimics the momentum of how the learning rate is moving. It can increase the rate at which the learning rate is optimized [148]. We see all four types in DL approaches applied to SE tasks, and did not identify any particular patterns or associations worth noting.

3.4.4.3 What methods are employed to combat overfitting and underfitting of the models?

The two major problems associated with using any type of learning based approach, whether that be canonical machine learning or deep learning, is overfitting and underfitting. Both overfitting and underfitting deal with the idea of generalization, *i.e.*, how well does a trained ML/DL model perform on unseen data. Overfitting is the process of a model learning to fit the noise in the training data extremely well, yet not being able to generalize to unseen data. The model has learned such a proficient approximation for this noisy data that it has become ungeneralizable to unseen data. However, this is not a good approximation of the true target function that the model was intending to learn. Underfitting involves having a learning model that is unable to approximate the seen training data in addition to unseen data points. This can occur when the model lacks the necessary complexity, is overly constrained, or has not had the sufficient training time to appropriately approximate the target hypothesis. Figure 3.10[3] gives an overview of some general ways to combat overfitting and underfitting. The figure also addresses what parts of an ML/DL approach is affected by these techniques. In this RQ, we were interested in the specific type of ways researchers are combating these two problems in the context of SE tasks.

Outlined in [13], the use of a validation set is a commonly used method for detecting if a model is overfitting or underfitting to the data, which is why it is very common to split data into training, validation and evaluation sets. The splitting of data helps to ensure that the model is capable of classifying unseen data points. This can be done while performing training, to ensure that overfitting is not occurring. We see this done in almost every paper analyzed in our SLR. However, more pertinent and powerful techniques to prevent overfitting were seen less frequently. Specifically, we were looking for any regularization

[3]Generated in part from multiple blog posts: https://elitedatascience.com/overfitting-in-machine-learning, https://hackernoon.com/memorizing-is-not-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42, https://towardsdatascience.com/dont-overfit-how-to-prevent-overfitting-in-your-deep-learning-models-63274e552323, https://elitedatascience.com/bias-variance-tradeoff

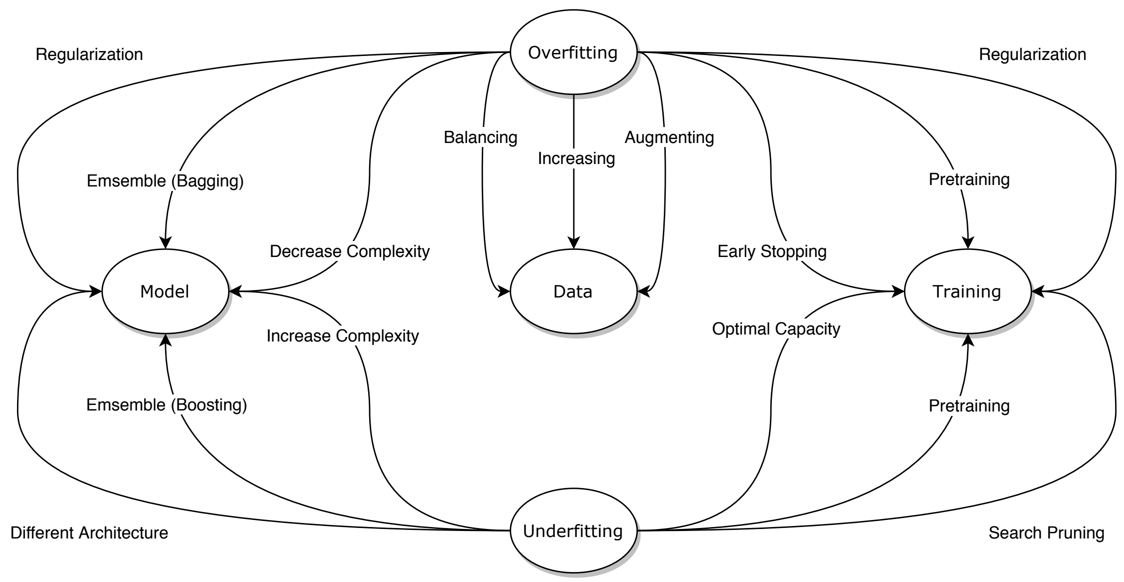


Figure 3.10: Overfitting and Underfitting Overview

techniques or additional methods the authors implemented to prevent the model from overfitting to the training data. As shown in our overview figure, there were three main types of regularization. The first regularizes the model, which includes things such as adding Dropout layers [248] or Batch Normalization [131]. The second regularizes the data itself, either through adding more data or cleaning the data already extracted. The third type of regularization is applied to the training process, which modifies the loss function with L1 regularization, L2 regularization or incorporates early stop training.

As we've done with previous components of DL approaches, we wanted to determine if the SE task had any affect on the techniques to combat overfitting. Figure 3.11 analyzes the relationship between DL approaches and the techniques that combat overfitting. This figure shows that there are some techniques that are much more commonly applied to SE tasks than others. For example, dropout was the most commonly used regularization technique and is used in a variety of DL approaches that address different SE tasks. This trend holds true for early stop training, data cleaning and L1/L2 regularization. However, the trend is somewhat broken in custom methods implemented to combat overfitting. It is unclear exactly what these custom methods are, but we found their highest prevalence in the SE task of program synthesis. We suspected that this is due to the highly specialized programs that are generated from input-output examples. The specialized data may require a specific technique that is not considered one of the common regularization methods. In addition to the aforementioned techniques, we found some unique approaches to combating overfitting including the use of deep reinforcement learning instead of supervised learning [61], gradient clipping, lifelong learning [100], changing the loss function [49], pretraining [272, 242], and ensemble models [136].

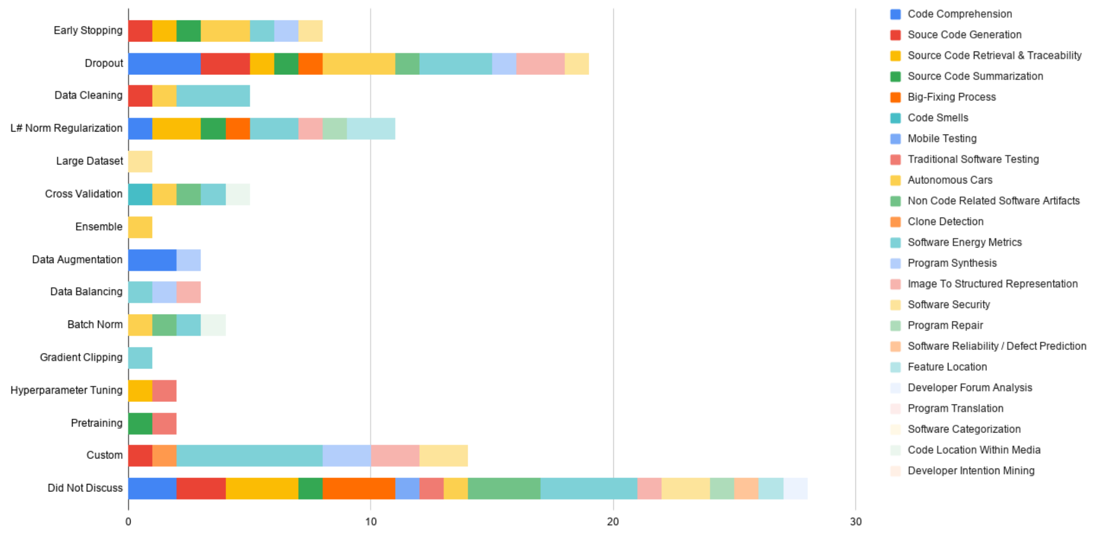


Figure 3.11: Overfitting Techniques per Task Type

After analyzing overfitting techniques in relation to SE tasks, we decided to also analyze the relationship between overfitting techniques and data type. We found a similar pattern in that there are a variety of techniques to combat overfitting regardless of the data type. The only exception to this pattern was seen when analyzing natural language, where L1/L2 regularization was predominately the most used regularization technique. The message from both figure 3.11 and 3.12, is that the techniques to combat overfitting do not have a strong association with either of these two features. Therefore, we can assume that these techniques can work in tandem with a variety of SE tasks and data types to prevent the model from overfitting. The most concerning trend we took away from these two figures is the *Did Not Discuss* category. Given the importance of combating overfitting when applying a DL approach, it is troublesome that so many primary studies did not make mention of using these techniques. Although we cannot be sure whether this is a lack of implementation or merely reporting the use of these techniques, we hope to see this trend dissipate as DL in SE continues to gain popularity.

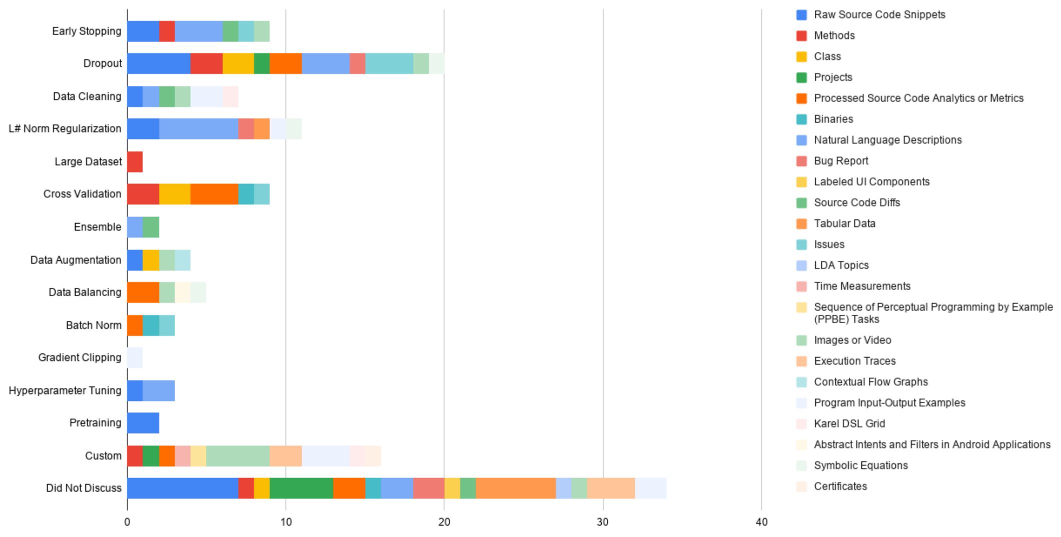


Figure 3.12: Overfitting Techniques per Data Type

For the prevention of underfitting, there are not a lot of current techniques that can be easily applied to the model and training process. However, underfitting greatly depends on the target function and the amount of time needed to appropriately model the target hypothesis. Underfitting normally requires finding the model's optimal capacity. The optimal capacity is the inflection point where the model starts to overfit to the training data and performs worse on the unseen validation set. While there are not many techniques to combat this issue, there are a few key insights to prevent underfitting the data. The first is to maximize training time without losing the generalizability of the model. As mentioned previously, this happens when the training process shows increased performance on the training data, but stagnant or decreased performance on the validation set. Other techniques include the use of a more complex model or a model more suited for the target function, which can be determined by varying the number of neurons, varying the number of layers, trying different DL architectures, pretraining the model, and search pruning the search space. From our SLR, the most commonly used underfitting techniques applied were search pruning the search space of the model [270, 61] and pretraining [272, 242].

Surprisingly, more than 25% of papers did not discuss any techniques used to combat overfitting or underfitting of their models. Combating this issue is a delicate balancing act, as attempting to prevent one can begin to cause the other if not done correctly. For example, having a heavily regularized learning model to prevent overfitting to a noisy dataset can lead to an inability to learn the target function, thus causing underfitting of the model. This is also possible while attempting to prevent underfitting. An increase in the number of parameters within the architecture to increase the complexity of the model can cause the model to learn a target function that is too specific to the noise of the training data. Therefore, the incorporation of techniques to address overfitting and underfitting is crucial to the generalizability of the DL approach. Additionally, reporting on techniques that combat overfitting and underfitting is important toward the goal of creating replicable and reproducible DL approaches in SE. Due to the span of learning components that these techniques affect, knowing which techniques were implemented is an important aspect to report on.

3.4.5 RQ4: How well do DL tasks perform in supporting various SE tasks?

In this RQ, we explored the impact that DL has had in SE. It is difficult to quantify and verify the results of the primary studies we analyzed, however, we can infer the impact of these approaches based on the metrics used to measure the performance of the models and the increased automation the approach brings to the SE task. In some regards, this RQ is task specific because performance is based on the difficulties and progress made within the SE task being addressed. We wanted to provide researchers with some intuition about how the current limits of DL in SE can be extended for particular SE tasks. In many instances, an increase in performance can have drastic implications on the situations in which the DL model can be applicable. Likewise, it is important to know what aspects of a SE task have already been automated by DL models.

We also wanted to provide a comprehensive listing of available baselines and benchmarks used for the evaluation of deep learning models in SE. By making future researchers aware of the baselines they can compare to, we hope to foster the promotion of comparative evaluations when looking at different DL approaches addressing similar SE tasks. Additionally, it is helpful to know what benchmarks exists and are made available for particular SE tasks. Benchmarks allow for an objective comparison between models looking at the same data. The use of benchmarks also helps to mitigate problems corresponding with sampling bias and data snooping. Creating a known listing of available benchmarks should result in a twofold benefit: the increased analysis and scrutiny of those benchmarks to ensure high quality data for comparison, and to promote future benchmarks in SE tasks where they do not exist.

3.4.5.1 What "baseline" techniques are used to evaluate DL models and what benchmarks are used for these comparisons?

Within this RQ, we were interested in evaluating the applicability and effectiveness of DL models applied to SE tasks. Looking at the baseline techniques used for comparison, as well as the evaluation metrics, we can establish a sense of how well a particular approach compares to another within the same task. Specifically, we found that there are certain baselines and metrics that are common to use for comparison within specific SE tasks. A failure to compare against these common baselines can call into question the impact and superiority of a DL approach over its predecessors. Additionally, we noticed that there are not many well documented resources for researchers to find common baselines and benchmarks used in previous approaches. Thus, we highlighted the need for the SE community to compile and make available these common baselines and benchmarks.

We looked at our primary studies in DL4SE and noted that baseline approaches are extremely individualized, even within the same SE task. Some DL4SE papers do not compare against any baseline approaches while others compare against 3-4 different models. Therefore, we included the listing of baselines that each paper compared against in our supplemental material. We found that many of the baseline approaches were canonical machine learning models or very simple neural networks. We suspected the reason for this is in part due to DL4SE being a relatively new field, meaning that there were not many available DL based approaches to compare against. Additionally, the use of canonical machine learning models as baselines demonstrates the need and benefit of applying DL to the specific SE task. However, as DL4SE begins to mature, we hope to see the trend transition to evaluations that include comparisons against previously implemented DL approaches.

Additionally, we find a reoccurring trend of model implementations being unavailable to the public. This, in part, explains why there are so many highly individualized, baseline approaches. Since researchers do not have access to common baselines used for comparison, they are forced to implement their own version of a baseline. It is possible that baseline approaches are implemented incorrectly or configured in such a way that gives them a disadvantage over the newly created approach. Thus, it can be difficult to fully rely on the results when compared to baseline approaches since many papers did not include any information about the baselines themselves. Additionally, a unique implementation of the same baselines could lead to confounding results when looking at improvement over those baselines. We want to encourage future DL4SE studies to include the details surrounding the implementation of baseline approaches or to include the implementation in an online repository.

The use and proper implementation of baseline approaches is essential to a quality evaluation of a DL model. However, the data used when comparing two different approaches is equally important. As we discussed in 3.4.2.2, DL models are very dependent on a high quality dataset free of sampling bias and data snooping. There is a need within the SE

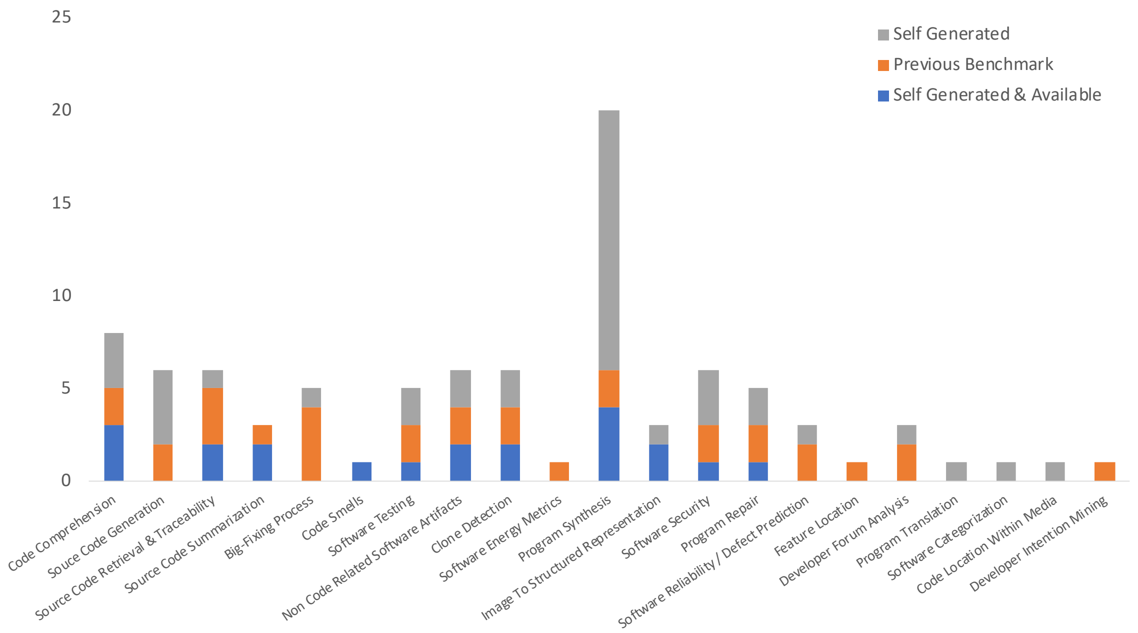


Figure 3.13: Benchmark Usage DL in SE

community to objectively evaluate these common benchmarks to ensure that comparisons between approaches are not skewed due to underlying attributes of the data. Likewise, if a new dataset or benchmark is introduced for evaluation, it is important that the authors can verify the quality of the dataset, such that reviewers can determine if it is fit to be used for comparison with other approaches. As new benchmarks are introduced for specific SE tasks, we can reduce the amount of exploratory data analysis, since we will have more data vetted by reviewers. Since DL models themselves are unbiased mathematical models, any bias or discrimination demonstrated by the DL approach can be attributed to the preprocessing techniques or the dataset itself. This emphasizes the importance of reviewers vetting benchmarks that are used in DL4SE studies.

Within our online repository, we included a list of all the benchmarks and baselines used for each paper within our SLR [280]. The reason we did not include the full list here is due to a number of unique benchmarks synthesized specifically to evaluate a particular approach. To solidify this point, we recorded the number of primary studies that used a previously curated benchmark as opposed to ones that curated their own benchmark. We noted that there is an overwhelming amount of self generated benchmarks. Additionally, we classified self generated benchmarks into those that are publicly available and those that are not. Unfortunately, we found a large majority of self generated benchmarks may not be available for public use. The full breakdown of benchmarks used in the primary studies can be seen in figure 3.13. This trend within DL4SE is worrying as there are few instances where DL approaches can appropriately compare against one another with available benchmarks. We hope that this SLR helps future researchers by providing them with an understanding about which benchmarks are available for an evaluation of their approach within a specific SE task. Additionally, we urge future researchers to make self generated benchmarks publicly available, which will provide a much needed resource not only for comparisons between approaches, but also for available data that a DL framework can use to train on.

Although the use of previously established benchmarks is not the norm, we did want to highlight a few benchmarks we saw used multiple times within the primary studies. For the SE task of clone detection, we found that the dataset BigCloneBench was used frequently to test the quality of the DL frameworks. Also, for the task of defect prediction, we saw uses of the PROMISE dataset as a way to compare previous DL approaches that addressed defect prediction in software. We believe that these two benchmarks are a great example of how benchmarks can be used in the future. These are popular, well document, and tested benchmarks that have been used to compare DL approaches within the same SE task.

The aspect of vetted benchmarks from the primary studies in DL4SE was the least transparent. In particular, details involving the baselines used in the evaluation were scarce or nonexistent. Without the implementation details of the baselines and a detailed analysis of the benchmarks, it is very difficult to objectively compare DL models in their ability to address certain SE tasks.

Understanding baseline approaches and accessing appropriate benchmarks is a crucial part of evaluating any DL based approach in SE. However, we also explored the common

Table 3.3: Metrics Used for Evaluation

Metrics	Studies
MMR	[142, 58, 123, 159, 70, 107, 52]
BLEU Score	[59, 136, 272, 110, 54, 120]
METEOR Score	[58, 271]
Precision	[263, 179, 170, 119, 159, 18, 285, 289, 276, 112, 76, 232]
	[122, 65, 67, 28, 107, 162, 175, 72, 129, 281, 261, 211, 199]
Recall	[180, 170, 119, 18, 289, 276, 112, 76, 232, 122, 65, 67]
	[28, 48, 162, 175, 176, 72, 129, 281, 52, 261, 211]
F1-Measure	[180, 119, 297, 18, 289, 23, 277, 65, 67, 161, 162, 72]
	[129, 281, 261]
Accuracy	[166, 202, 123, 170, 100, 75, 167, 119, 50, 43, 294, 78]
	[250, 203, 46, 227, 219, 56, 289, 178, 79, 76, 49, 270]
	[61, 125, 232, 122, 28, 55, 162, 175, 60, 99, 274, 21]
	[300, 241, 89, 173, 66, 129, 211, 27, 120]
CIDER	[272, 300]
ROC	[297, 232, 99, 281]
Root Mean Squared Error	[116, 237]
Model Perplexity	[282, 287]
Timing	[285, 36, 88, 89, 191]
AUC	[232, 65, 67, 99, 72, 281]
Macro-averaged Mean Absolute Error (MMAE)	[65, 67]
Mean Absolute Error	[65, 66]
Top K Generalization	[241, 200]
Top K Model-Guided Search Accuracy	[241, 181]
Solved Tasks	[242, 88, 281]
Other	[136, 182, 230, 102, 272, 203, 297, 287, 125, 67, 65, 107]
	[53, 71, 274, 66, 261, 200]

metrics used to measure the performance of DL models applied to their respective SE tasks. In many instances, the metrics chosen to analyze the model are common to the type of learning. Therefore, many of the supervised learning methods have metrics that analyze the resulting hypothesis, such as the accuracy, true positives, F1 measure, precision and recall. These metrics are used to compare the supervised learning algorithms with the outputs representing the target hypothesis. Intuitively, the type of metric chosen to evaluate the DL based approach is dependent upon the data type and architecture employed by the approach. Figure 3.3 describes the distribution of metrics found in this SLR. We included the top used metrics to give researchers and indication about which metrics are popular for evaluation. The other category is comprised of the less popular metrics including: likert scale, screen coverage, total energy consumption, coverage of edges, rouge-L, CIDER, jicard distance, minimum absolute difference, Kruskal's υ, cross entropy, macro-averaged mean cost-error, matthews correlation coefficient, f-rank, top-k generalization, top-k model-guided search accuracy, median absolute error, spearmans rank and confusion matrix. We included these for completeness, but it is important to be aware that these metrics can be specialized toward evaluating the model's ability to predict the specific target outputs. Interestingly, we found that many supervised DL approaches omit the ROC or AUC evaluations with a condience level of 0.87. This metric should be included in the comparison with baseline approaches to demonstrate the improvement of the current approach.

3.4.5.2 How is the impact or automatization of DL approaches measured and in what way do these models promote generalizability?

The impact of a particular approach in research is somewhat subjective and can be susceptible to bias. Therefore, we did not look to validate or invalidate any of the research presented in this SLR. We attempted to objectively look at the claimed impacts that the authors of these primary studies have made as a way to determine what value the application of a DL model had for the SE task addressed. In many cases, DL approaches contribute a variety of advantages both in correctness and efficiency of addressing a SE problem. Therefore, we give a brief overview of the potential implications that the primary studies included within this SLR can have. These implications are primarily pulled from the claims of the authors about the impacts of their approach.

We classified each primary study into seven categories, which represents the major contribution of the work. The result of this inquiry can be seen in figure 3.14. We provided this analysis to show future researchers that DL can contribute to the advancement of the SE field in multiple ways. However, the most common contribution of a DL4SE study is the ability to increase automation or efficiency. Interestingly, SE tasks that had multiple DL implementations frequently claimed another novelty such as an advancement in the architecture or increased performance over their predecessor.

Our exploratory data analysis of this RQ revealed that the automation impact is dependent on the SE task deduced from a high mutual information of 0.976 bits. This means that there is a strong association between the SE task and the automation impact of the approach. We found three primary objectives that the implementation of a DL model is meant to address: (i) in 75% of SE tasks observed, a DL approach was implemented with the main goal of increasing automation efficiency; (ii) in 45.8% of the SE tasks observed, a DL approach was implemented with the main goal of advancing or introducing a novel architecture; (iii) in 33% of the SE tasks observed, a DL approach was implemented with the main goal of replacing human expertise. We discuss this taxonomy so that future researchers can think about the main objective they intend their DL approach to address. This objective can be used to help shape the evaluation and the metrics used to quantify effectiveness.

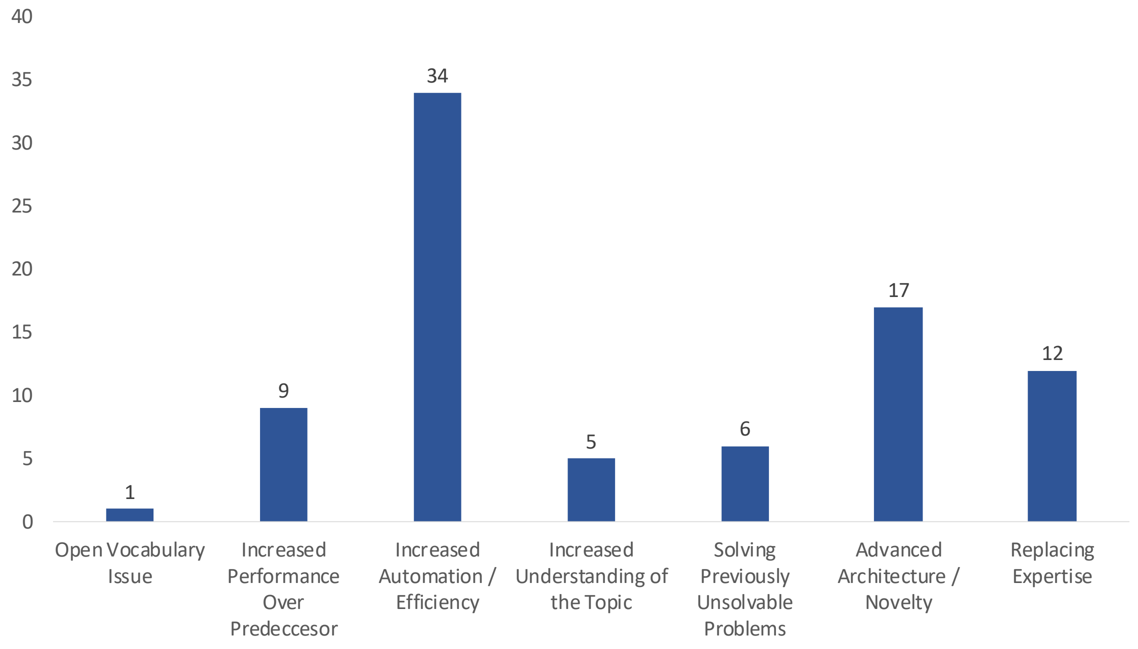


Figure 3.14: Impact of DL4SE

Within our SLR, we saw many DL implementations for a variety of tasks. Yet, we noticed that the primary studies did not put as strong an emphasis on the generalization of a model as opposed to the other aspects of the DL approach. The generalization of a model is very important as it outlines the conditions in which the model can be applied to the task. If a model lacks generalizability, then it may not be applied to data outside of the training and testing set. This can make the DL approaches in SE too specific for developers working on production software. Thus, in this RQ, we analyzed how these primary studies ensured the generalizability of their model. To do this, we looked for evidence that the authors considered Occam's Razor, as well as the efficiency of their model.

Before continuing into our analysis, we wanted to address Occam's Razor and how it relates to DL approaches in SE. Occam's Razor was originally translated to "Entities should not be multiplied beyond necessity" and is quoted from William of Occam in the late Middle Ages [235]. In the current age, this concept has been applied to the field of machine learning to address the over engineering of learning models resulting in less generalizability. Currently, Occam's Razor is separated into two Razors: 1) "Given two models with the same generalization error, the simpler one should be preferred because simplicity is desirable", 2) "Given two models with the same training-set error, the simpler one should be preferred because it is likely to have lower generalization error" [85]. When specifically applied to deep learning, we were interested in determining if the SE task could have been solved with a less complex model. Unfortunately, it was very difficult to prove Occam's Razor in the context of a primary study focused on a SE task. Therefore, we looked for evidence that Occam's Razor was considered by the authors but did not explicitly analyze how the complexity of the model was fully evaluated. In figure 3.15 we break the primary studies into three groups: 1) those that compare against less complex models and analyze the results; 2) those that manipulate the complexity of their own model by testing a variety of layers or nodes per layer; 3) those that did not have any Occam's Razor consideration.

Although a majority of the primary studies do consider Occam's Razor, there are still almost 25% of DL4SE studies that do not consider Occam's Razor. It is possible that the studies we analyzed did not include details about how they considered the complexity of their model when applied to the specific SE task. However, these details are vital to include within the primary study, since they provide evidence that DL is appropriate to apply to the SE task. Without this consideration, it is possible that a canonical machine learning

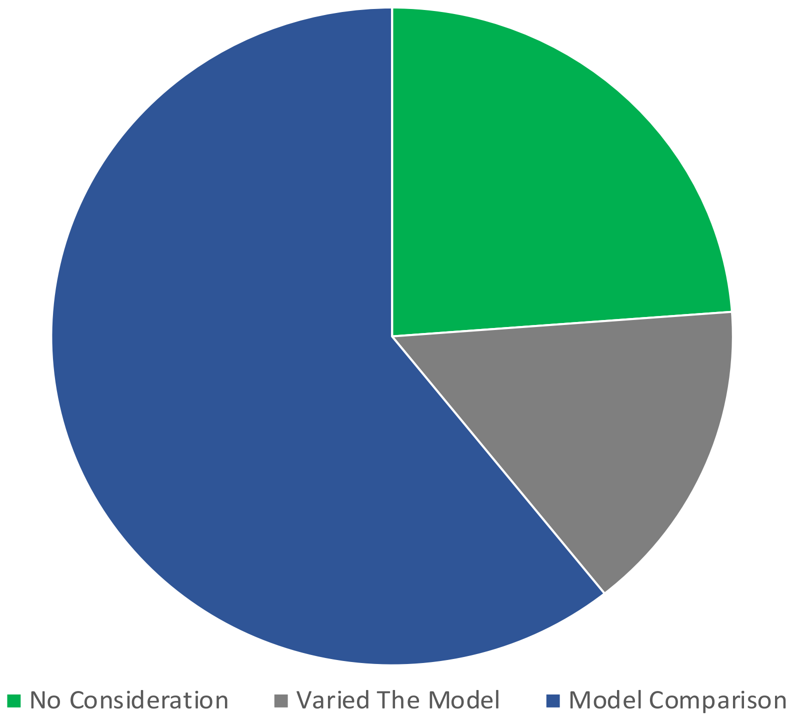


Figure 3.15: Evidence of Occam's Razor

model or a simple statistical based approach may be the optimal solution. Interestingly, in about 15% of the primary studies, the author's considered Occam's Razor by adjusting the complexity of the model being evaluated. This is done by varying the number of layers, the number of nodes, the size of embeddings, etc., which all contribute to the complexity of the approach. The downside to this method is that there is no way to determine if the extraction of complex hierarchical features is more complex than what is necessary to address the SE task. The only way to properly answer this question is to compare against baseline approaches that are not as complex. In our DL4SE studies, this often looked like a comparison with a canonical ML approach. This further bolsters our point made in 3.4.5.1, that baseline approaches are essential to establish the validity of a DL based approach.

3.4.6 RQ5: What common factors contribute to the difficulty reproducing DL studies in SE?

This SLR relies on the details given by the authors of DL approaches applied to SE tasks. As we progressed through our review, we wanted to highlight the importance of reporting key components for the purpose of reproducibility. Reproducibility is of great importance to the research community and as DL becomes a more popular tool to model software artifacts, it is important that papers include the necessary details so that reproducing the study is possible. DL models can also be extremely complex and highly customizable to fit the needs of the particular SE task. We wanted to emphasize this point since even small, nuance changes can lead to drastic affects in the approach's performance. These slight alterations do not necessarily refer to the DL model. They also apply to the extraction and preprocessing of the data, the learning algorithm, the training process and the hyperparameters, which all have a significant contribution to the overall success of a DL approach. In this RQ, we surveyed the work of our SLR to determine the important concepts and details that need to be reported when implementing a DL based approach.

The first and easiest analysis to perform was determining the ability to replicate the DL4SE approaches. To replicate an approach means to re-implement the approach exactly as the authors originally did. Due to the variability that each approach can have, the data and source code should be publicly available in order to deem the work replicable. We found that out of the 84 primary studies we analyzed, only 28 provided enough information through an online repository to fully replicate the approach. This means that the vast majority of DL4SE studies either did not provide the implementation of the approach or did not provide the dataset to train and test the approach. We also analyzed the reproducibility of DL4SE studies. We define reproducibility as the ability for someone within the field of DL4SE to read the study and implement the approach with slight differences either in the implementation or data representation. In this case, small details can be missing but the steps to reproduce the approach, it its entirety, should be present.

We found that 43 out of the 84 studies we analyzed were capable of being reproduced. Out of the 43 studies that can be reproduced, only 18 of those studies were also replicable. Our goal here was to further analyze what factors were responsible for the lack of reproducibility amongst the remaining 41 primary studies. We found that there were ten major factors that contributed to the lack of reproducibility. The breakdown of this analysis for the primary studies are show in in figure 3.16.

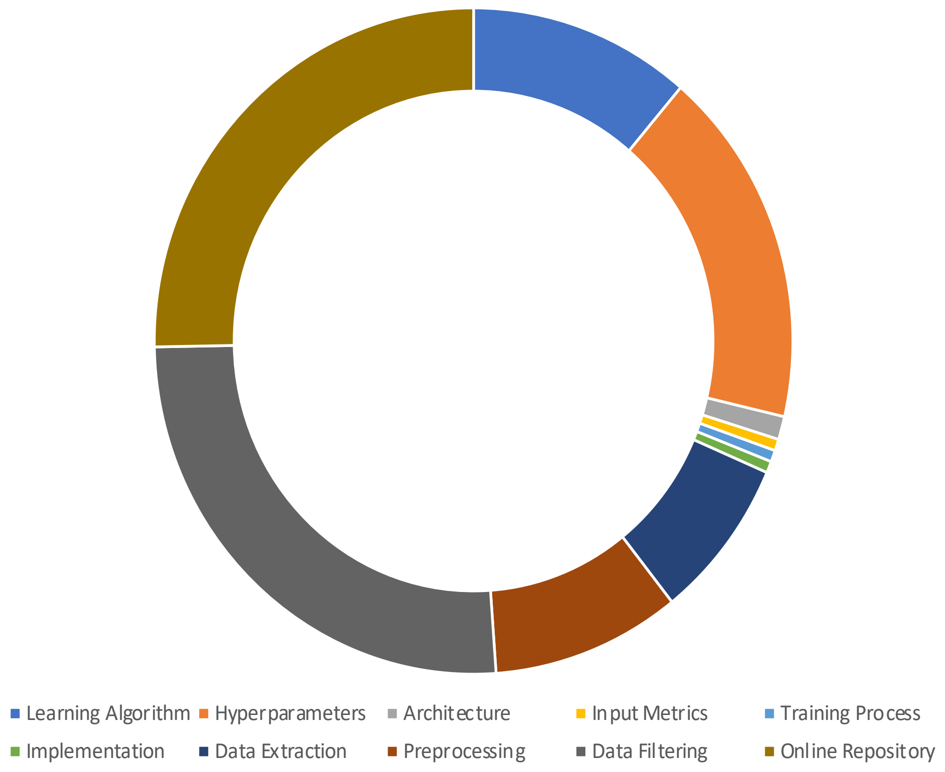


Figure 3.16: Contributing Factors to Reproducibility

In figure 3.16 we show areas where DL approaches in SE may be lacking the necessary details to reproduce or reimplement the approach. The first two areas that contribute to the difficult of reproducibility pertain to the learning algorithm and the hyperparameters. The learning algorithm is the way in which the parameters of the model are tuned to appropriately estimate the target function. We found that there were missing details pertaining to either the method of tuning the weights, the way in which the error was calculated or the optimization algorithm. All three aspects of the learning algorithm can make reimplementation of the approach difficult and prone to errors. Without the proper learning algorithm, it is likely that any attempt to reproduce the study will vary from the originally reported results. This is because there are a vast amount of combinations of these three aspects to the learning algorithm, but all contribute significantly to the reported results of the study. In addition to the learning algorithm, the hyperparameters are also crucial as they can have an affect on the learning algorithm and the model architecture. Consequently, if the hyperparameters are not reported, then it is impossible to know how many parameters contributed to the estimation of the target function, since hyperparameters control the number of layers and the number of nodes per layer. Additionally, the way that the learning rate is adjusted is ultimately what controls the parameters that estimate the target function. An incorrect learning rate can lead to incorrect parameters, which in turn leads to a completely different target function that is modeled.

Although the details pertaining to the learning algorithm and the hyperparameters is important, they are not the only details that a DL approach would need to report in order to be replicated or reproduced. Particularly, the data and the way it was extracted, filtered and formatted would need to be known in order for a study to be reproduced. DL models are data driven, meaning that they extract features from the data without any human intervention. In order for the study to be reproducible, three pieces of information need to be accessible. The first is the extraction details. In order for a study to be reproduced, the data must be accessible which means either the dataset must be readily available or the details about how the data was extracted need to be reported. The second piece of information that is needed is the preprocessing details. Once the data is extracted, it needs to be formatted into a representation that the DL model can accept as input. Additionally, the way that the data is represented to the model, in some part, controls the features that are able to be extracted. So if the data is represented contrary to the way it was done in the original study, then the results of the reproduced study would be invalid. The last attribute of the data that needed to be known was the filtering details. Data can be inherently noisy and authors will frequently need to filter out noisy data in order for the

DL model to learn an effective relationship between the input and the target. This process can involve the removal of certain data points or an entire section of the original dataset based upon a criteria that the authors set. We discovered that 44/84 primary studies are missing crucial details about the filtering of their data. Reporting these details related to the filtering of data should not only include the filtering criteria, but should also explain the steps and methodology taken to remove this data from the dataset.

We performed exploratory data analysis pertaining to the reproducibility of the studies contained within our SLR. Here, we attempted to establish patterns for why papers in SE struggle with irreproducibility. We discovered that SE papers not using visual data, repository metadata, natural language data, or I/O examples have a strong association with irreproducibility (0.70) with a relatively high confidence of 0.82. We also found that not using visual data, repository metadata, or I/O example is strongly associated with lacking extractions details (0.71) with a high confidence of 0.82. Lastly, we found that not using vision data or repository metadata was associated with a lack of data preprocessing details (0.74) with a confidence of 0.81. We describe these associations to bring awareness to the issues with reproducibility. From these facts found within our exploratory data analysis, we created guidelines so that future researchers know what information to include when reporting on their DL based approach in SE.

Lastly, we wanted to promote the idea of open sourced models and datasets. This is not only the best mechanism for reproducibility, but also grants access to datasets and DL approaches to future SE researchers. We have mentioned before that DL models are extremely data hungry. However, there are not many publicly available datasets containing software artifacts to train DL models on. This not only inhibits the use of DL in SE, but it also makes comparative studies with different DL models difficult. The lack of data hurts the evaluation of DL based approaches because there are very few baselines to compare a newly synthesized model to. This can lead to claims regarding the effectiveness of a DL implementation that can be difficult to refute or verify. Therefore, we encourage future researchers to make the datasets and DL models publicly available. We believe that this will drive a greater quality of research and allow for verifiable comparisons of DL based approaches in SE.

3.5 Guidelines for DL in SE

In this section, we provide guidelines for the future use of DL within SE. We first generated a flow diagram that provides a logical pathway through the entire process of applying DL in SE. The diagram, seen in figure 3.17, provides the steps that need to be taken, but we refer back to the sections of our SLR for the specifics behind addressing each component of learning. In addition to the flow diagram we also compiled additional guidelines, which can be treated as a sort of checklist, containing certain pitfalls or points to consider when working through the components of learning. This can be seen in figure 3.18. We do not claim this as a comprehensive guide, as each implementation of these complex models have their own nuances. However, we do cover many of the essential aspects of applying DL to SE and make future researchers aware of the context in which DL can be a useful tool.

After the implementation of the DL model, the reporting process should be complete and thorough. We want to reiterate what we found in figure 3.16 to help guide future researchers through the reporting phase of their research. The reporting process should begin with data extraction. Here, the researchers should note any potential limitations or biases within the data extraction process. DL models are limited in scope by the data they analyze, therefore, any restriction on data extraction or additional data filtering should be well documented. The next phase of the reporting process should address the preprocessing steps taken to prepare the data for the DL model. Steps within this process should be explicitly explained to the point where replication is possible. This would include any abstractions, filters, vocabulary limitations or normalizations that the data is subjected to. In addition, the researchers should report how they vectorize the resulting data, which includes the format of the vector or matrix subjected to the DL model.

After reporting on the data extraction, the data filtering, and its preprocessing steps to prepare for the DL model, the authors should begin addressing the DL framework. This includes information about the learning algorithm, DL architecture, hyperparameterization, optimization methods, and training process. There are many common learning algorithms used, with the most prevalent being variants of gradient descent. It is important that the learning algorithm be discussed, as this can significantly affect the results of the model. In addition to the learning algorithm, the optimization algorithm should also be mentioned. The optimization algorithm describes the process of adjusting the parameters of the model which then results in learning. Researchers should also fully elaborate on the DL architecture used in their approach. This elaboration should include things such as the number of nodes, number of layers, type of layers, activation function, etc. The approach cannot be replicated without all necessary information pertaining to the architecture and should be available to the reader. Lastly, authors should report the hyperparameters and training process. The hyperparameters directly affect the results of the model as their values directly control the process of the model's learning. Hyperparameters are adjusted and set by the author, therefore the process of finding the optimal hyperparameterization, as well as their final values should be explicitly stated. The reporting process should also include details pertaining to the training process of the DL model. This can include the number of training iterations, the time taken to train, and any technique that would help to prevent overfitting of the model.

In addition to the patterns discovered in this work, our findings corroborate many of the issues discussed in the study by Humbatova et al. [130] This study analyzed real faults discovered in DL systems from GitHub. They found three major areas of error when implementing DL systems: errors in the input, the model and the training process. Interestingly, they interviewed developers to determine the severity and effort required to address many of these pitfalls. They found that developers thought the most severe errors related to the proper implementation of the optimizer, entering in the correct input data and correctly implementing the models with the right number of layers. However, they also learned that developers find these three severe errors took a relatively low amount of effort to fix [130].

Our research demonstrates the components of learning within DL4SE work that have been incorrectly implemented or not implemented at all. When taking into consideration the findings of Humbatova et al., we hypothesize that the amount of effort needed to fix these problems when applying the DL framework could be the reason it is not performed. For example, the adjustment and appropriate values of hyperparameters can be an impactful problem when applying a DL model. However, developers rated this issue to be the third highest in the amount of effort needed to address it. Likewise, we see getting the training data and training process correct are ranked number two and number one, respectively, for the most amount of effort required to address the issue. We believe that following these guidelines and looking at previous implementations of DL in SE will aid future researchers in knowing how to quickly and effectively address the issues that arise when applying these complex models.

Our intention with these guidelines is to bring attention to crucial details about applying DL in SE. The figures and explanation included here is not intended to be a comprehensive list of all information to include, rather, it is intended to serve as a basic checklist. Every DL based approach has its own idiosyncrasies that may or may not be important to include when reporting on a DL approach in SE. However, all details mentioned in our guidelines should be readily available in order to encourage reproduction and replication of the experiments done. We also encourage authors to promote transparency of their approach by making all datasets, models and implementation scripts available via an online repository. This will lead to increased quality of future DL research in SE and allow for more meaningful comparisons of different DL approaches addressing similar SE tasks.

3.6 SE4DL

When searching for papers that could be classified as DL4SE, we ran across some primary studies that give way to the new rising field of software engineering for deep learning (SE4DL). This field has seemingly come to fruition from the issues discovered with autonomous vehicles. Since DL models extract complex, hierarchical features automatically through a sequence of nonlinear transformations, developers of DL models are not exactly sure what features are being valued to make the appropriate classification or prediction. This is why DL models are frequently referred to as a black-box. Data is fed into them, mathematical concepts drive the learning and training of these models and they are able to perform classification with a high level of accuracy. This dramatically changes the traditional paradigm of creating software, which is based on the software life cycle. Previous software engineering analyzes a problem, designs a program which attempts to solve that problem, the program is then implemented, tested and maintained. The introduction of deep learning renders this paradigm of SE useless because the model is what is responsible for generating the program that will solve the task at hand. The developer of the DL based approach is responsible for giving the algorithm the input and its associated target or output. The model then learns the appropriate target function, meaning that it essentially generates the program to model the input and output examples. Since the developer is not privy to the way in which the DL algorithm generated the program, it is difficult to test and maintain the program using the traditional software engineering life cycle paradigm. Specifically, the area of testing software is particularly challenging. The current methodology for testing DL models is to feed the model data and evaluate the accuracy on its ability to predict the appropriate target. However, software which has very minimal room for error, such as that for an autonomous vehicle, cannot be confident in a method which can only be tested in an ad hoc manner. There is a need for a systematic methodology that instills confidence in a model, which provides assurance that the model correctly estimates a target function represented by the data.

Currently, this type of systematic methodology does not exist. However, software engineers have developed tools and ideologies that can be used to test these black-box models. Thus, a new field has arisen in SE4DL where the concepts learned from software testing and maintenance are being applied to the development of software based on DL algorithms. The process of our SLR captured some of the primary studies that have worked in applying SE ideals to the models generated by DL algorithms. These works focus on problems such as concolic testing for deep neural networks, addressing the lack of interoperability of DL systems by developing multi-granularity testing criteria, generating unique testing data through the use of generative adversarial networks, detecting bugs in the implementations of these DL models, seeded mutation of input data to create new testing data, specifically detecting bugs when using certain ML frameworks such as TensorFlow and detecting erroneous behaviors in autonomous vehicles powered by deep neural networks [251, 186, 295, 91, 188, 292, 299, 113, 90, 147, 198, 296, 258]. These pieces of work are only the beginning of a growing field that will attempt to understand and interpret the inner-workings of DL models.

3.7 Additional Classifications

In the midst of performing the SLR, we encountered studies which passed our initial inclusion criteria, but were eventually excluded based on their lack of a DL implementation. However, there has been debate about what type of implementation qualifies as a DL based approach. This SLR maintains the definition that deep learning models must automatically extract complex, hierarchical features from the data it is fed. This implies that the data must be subjected to multiple, nonlinear transformations by passing it through multiple hidden layers within a neural network. This type of model would exclude algorithms that we would consider to belong to one of the following categories: artificial intelligence, canonical machine learning and representational learning. This hierarchy and definition of deep learning was taken from Goodfellow *et al.* [104]. Despite the limitations we set on our SLR, we wanted to make the community aware that there are a variety of learning based approaches that have been applied to the field of SE.

There were two common types of papers we encountered when performing our SLR that we felt deserved a brief introduction and explanation as to why they were not included. The first is primary studies which use Word2Vec or some variation of Word2Vec in order to embed some type of sequential data. While we agree that Word2Vec is an unsupervised learning algorithm, it fails to extract complex hierarchical features of the input data. This model is a two layered neural network that excels at creating neural embeddings for words or other sequential data. This type of model learns the proper embeddings to accurately represent a set of words as a set of feature vectors, which are in a numerical format that a DL model could understand. We frequently see this type of embedding technique used in conjunction with DL models but do not consider it a DL model on its own. The other type of learning approach we encountered was the use of canonical machine learning algorithms. These types of algorithms require manual feature engineering and therefore, do not rely on nonlinear transformations to learn complex hierarchical features automatically. Although these types of algorithms do require a more substantial manual effort, they perform relatively well when data is extremely scarce or the developer wants to control what specific features the model considers when being fed data. Canonical ML methods are frequently used in SE and are applied to a variety of tasks, but the models do not fit the criteria to be considered deep learning and are excluded from the study

3.8 Conclusions

This work performed a SLR on the primary studies related to DL4SE from the top software engineering research venues. Our work heavily relied on the guidelines laid out by Kitchenham *et al.* for performing systematic literature reviews in software engineering. We began by establishing a set of research questions that we wanted to answer pertaining to applications of DL models to SE tasks. We then empirically developed a search string to extract the relevant primary studies to the research questions we wanted to answer. We supplemented our searching process with snowballing and manual additions of papers that were not captured by our systematic approach but were relevant to our study. We then classified the relevant pieces of work using a set of agreed upon inclusion and exclusion criteria. After distilling out a set of relevant papers, we extracted the necessary information from those papers to answer our research questions. Through the extraction process and the nature of our research questions, we inherently generated a taxonomy which pertains to different aspects of applying a DL based approach to a SE task.

Through the process of our SLR, we discovered the presence of papers using a Word2Vec like algorithm as well as implementations of canonical ML algorithms. In addition, we discovered evidence of an emerging field in software engineering: software engineering for deep learning (SE4DL). SE4DL applies fundamental concepts from software testing and maintenance to redefine the paradigm of developing software using DL methodologies rather than the traditional software life cycle. Our hope is that this SLR provides future SE researchers with the necessary information and intuitions for applying DL in new and interesting ways within the field of SE. The concepts and relationships described in this review should aid researchers and developers in understanding where DL can be applied and what the necessary considerations are when applying these complex models.

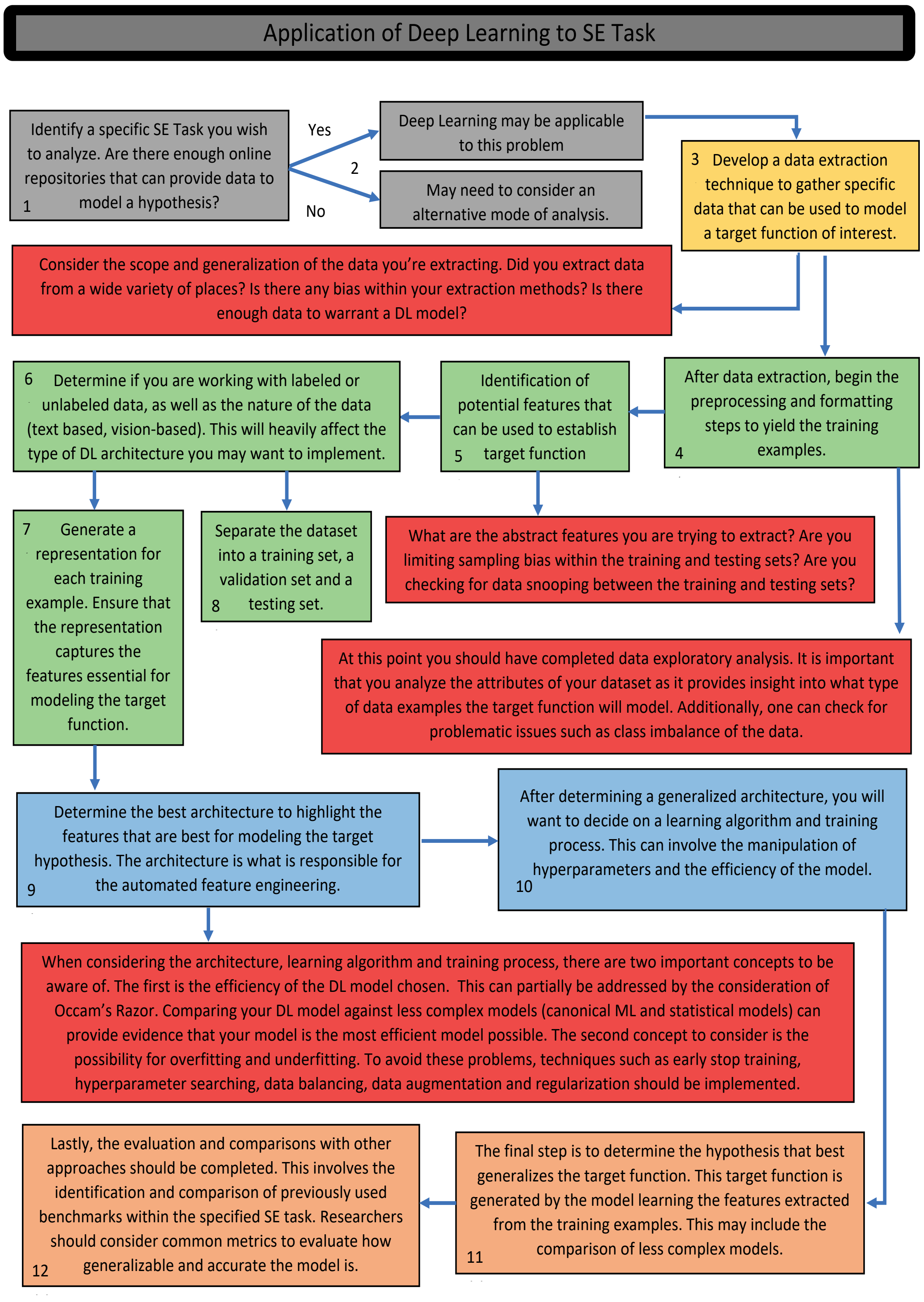
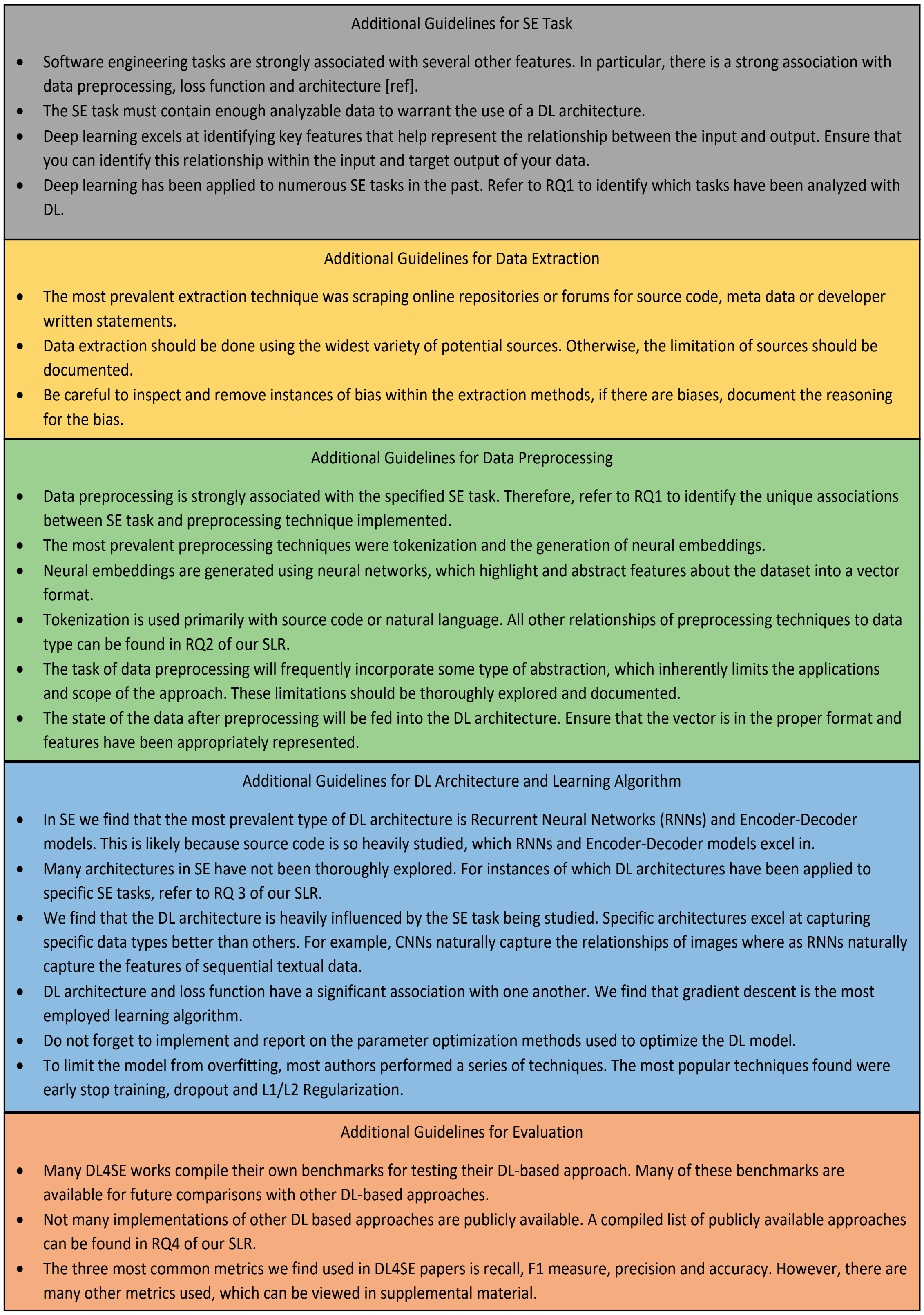

Application of Deep Learning to SE Task
1 Identify a specific SE Task you wish to analyze. Are there enough online repositories that can provide data to model a hypothesis?
Yes
2
No
Deep Learning may be applicable to this problem
May need to consider an alternative mode of analysis.
3 Develop a data extraction technique to gather specific data that can be used to model a target function of interest.
Consider the scope and generalization of the data you're extracting. Did you extract data from a wide variety of places? Is there any bias within your extraction methods? Is there enough data to warrant a DL model?
4 After data extraction, begin the preprocessing and formatting steps to yield the training examples.
5 Identification of potential features that can be used to establish target function
6 Determine if you are working with labeled or unlabeled data, as well as the nature of the data (text based, vision-based). This will heavily affect the type of DL architecture you may want to implement.
7 Generate a representation for each training example. Ensure that the representation captures the features essential for modeling the target function.
8 Separate the dataset into a training set, a validation set and a testing set.
What are the abstract features you are trying to extract? Are you limiting sampling bias within the training and testing sets? Are you checking for data snooping between the training and testing sets?
At this point you should have completed data exploratory analysis. It is important that you analyze the attributes of your dataset as it provides insight into what type of data examples the target function will model. Additionally, one can check for problematic issues such as class imbalance of the data.
9 Determine the best architecture to highlight the features that are best for modeling the target hypothesis. The architecture is what is responsible for the automated feature engineering.
10 After determining a generalized architecture, you will want to decide on a learning algorithm and training process. This can involve the manipulation of hyperparameters and the efficiency of the model.
When considering the architecture, learning algorithm and training process, there are two important concepts to be aware of. The first is the efficiency of the DL model chosen. This can partially be addressed by the consideration of Occam's Razor. Comparing your DL model against less complex models (canonical ML and statistical models) can provide evidence that your model is the most efficient model possible. The second concept to consider is the possibility for overfitting and underfitting. To avoid these problems, techniques such as early stop training, hyperparameter searching, data balancing, data augmentation and regularization should be implemented.
11 The final step is to determine the hypothesis that best generalizes the target function. This target function is generated by the model learning the features extracted from the training examples. This may include the comparison of less complex models.
12 Lastly, the evaluation and comparisons with other approaches should be completed. This involves the identification and comparison of previously used benchmarks within the specified SE task. Researchers should consider common metrics to evaluate how generalizable and accurate the model is.

Figure 3.17: Guidelines



Additional Guidelines for SE Task

- Software engineering tasks are strongly associated with several other features. In particular, there is a strong association with data preprocessing, loss function and architecture [ref].
- The SE task must contain enough analyzable data to warrant the use of a DL architecture.
- Deep learning excels at identifying key features that help represent the relationship between the input and output. Ensure that you can identify this relationship within the input and target output of your data.
- Deep learning has been applied to numerous SE tasks in the past. Refer to RQ1 to identify which tasks have been analyzed with DL.

Additional Guidelines for Data Extraction

- The most prevalent extraction technique was scraping online repositories or forums for source code, meta data or developer written statements.
- Data extraction should be done using the widest variety of potential sources. Otherwise, the limitation of sources should be documented.
- Be careful to inspect and remove instances of bias within the extraction methods, if there are biases, document the reasoning for the bias.

Additional Guidelines for Data Preprocessing

- Data preprocessing is strongly associated with the specified SE task. Therefore, refer to RQ1 to identify the unique associations between SE task and preprocessing technique implemented.
- The most prevalent preprocessing techniques were tokenization and the generation of neural embeddings.
- Neural embeddings are generated using neural networks, which highlight and abstract features about the dataset into a vector format.
- Tokenization is used primarily with source code or natural language. All other relationships of preprocessing techniques to data type can be found in RQ2 of our SLR.
- The task of data preprocessing will frequently incorporate some type of abstraction, which inherently limits the applications and scope of the approach. These limitations should be thoroughly explored and documented.
- The state of the data after preprocessing will be fed into the DL architecture. Ensure that the vector is in the proper format and features have been appropriately represented.

Additional Guidelines for DL Architecture and Learning Algorithm

- In SE we find that the most prevalent type of DL architecture is Recurrent Neural Networks (RNNs) and Encoder-Decoder models. This is likely because source code is so heavily studied, which RNNs and Encoder-Decoder models excel in.
- Many architectures in SE have not been thoroughly explored. For instances of which DL architectures have been applied to specific SE tasks, refer to RQ 3 of our SLR.
- We find that the DL architecture is heavily influenced by the SE task being studied. Specific architectures excel at capturing specific data types better than others. For example, CNNs naturally capture the relationships of images where as RNNs naturally capture the features of sequential textual data.
- DL architecture and loss function have a significant association with one another. We find that gradient descent is the most employed learning algorithm.
- Do not forget to implement and report on the parameter optimization methods used to optimize the DL model.
- To limit the model from overfitting, most authors performed a series of techniques. The most popular techniques found were early stop training, dropout and L1/L2 Regularization.

Additional Guidelines for Evaluation

- Many DL4SE works compile their own benchmarks for testing their DL-based approach. Many of these benchmarks are available for future comparisons with other DL-based approaches.
- Not many implementations of other DL based approaches are publicly available. A compiled list of publicly available approaches can be found in RQ4 of our SLR.
- The three most common metrics we find used in DL4SE papers is recall, F1 measure, precision and accuracy. However, there are many other metrics used, which can be viewed in supplemental material.

Figure 3.18: Additional Guidelines

Chapter 4

Learning Code Similarities

4.1 Introduction

Source code analysis is a rock-solid foundation of every software engineering (SE) task. Source code analysis methods depend on a number of different code representations (or models), which include, source code identifiers and comments, Abstract Syntax Trees (ASTs), Control-Flow Graphs (CFGs), data flow, bytecode, *etc.* These distinct representations of code provide different levels of abstraction, which create explicit and implicit relationships among source code elements. The importance of a specific code representation is dependent upon the SE task. For example, identifiers and comments encode domain semantics and developers' design rationale, making them useful in the context of feature location [81, 228, 82, 84] and software (re)modularization [40, 39, 38]. Additionally, programs that have a well-defined syntax can be represented by ASTs, which, in turn, can be successfully used to capture programming patterns and similarities [205, 209, 25]. Since there are numerous SE tasks, such as static and dynamic program analysis, change history analysis, automated testing, verification, program transformation, clone detection *etc.*, it is important to rely on different available code representations so that different source code relationships can be properly identified. Yet, many existing solutions to these SE tasks are based on "hardcoded" algorithms rooted in the underlying properties of the specific code representation they use, which in order to work properly, also need to be adequately configured [214, 278, 215].

While SE researchers regularly use machine learning (ML) or Information Retrieval (IR) techniques to solve important SE tasks, many of these techniques rely on manually defined or hand-crafted features. These features then allow for ML-based SE applications. For example, distinct identifiers are typically used as features in concept location approaches [83], APIs (a reserved subset of identifiers) are used as features in approaches for identifying similar applications [193, 269], and AST pattern similarities are used to enable clone detection and refactoring [197, 97, 134]. However, it should be noted that all the features are selected via "*an art of intuitive feature engineering*" by SE researchers or domain (task) experts. While there are many successful and widely adopted ML-based approaches that use different code representations to support SE tasks, their performance varies depending on the underlying datasets [214, 215].

Improvements in both computational power and the amount of memory in modern computer architectures have enabled the development of new approaches to canonical ML-based tasks. The rise of deep learning (DL) has ushered tremendous advances in different fields [165, 195]. These new approaches use representation learning [44] — a significant departure from traditional approaches — to automatically extract useful information from the disproportionate amount of unlabeled data in software. The value in circumventing conventional feature engineering when designing ML-based approaches to SE tasks is two-fold. Firstly, learning transformations of the data reduces the cost of modeling code, since software (and various code representations) stores a lot of data to improve the effectiveness of learning. Secondly, performance is increased because generally, learning algorithms are more efficient than humans at discovering correlations in high dimensional spaces.

While the number of applications of DL to SE tasks is growing [124, 96, 160, 206, 24, 15, 160, 157, 42, 106, 275], one recent example by White *et al.* [283] shows that DL can effectively replace manual feature engineering for the task of clone detection. Existing clone detection approaches leverage algorithms working on manually specified sets of features, and include, for example, text-based [137, 139, 138, 86, 231] string-based [34, 33, 35], token-based [141, 172, 233, 234], AST-based [290, 41, 154, 134], graph-based [152, 155, 177, 97, 57], or lexicon-based [192] approaches. Instead, DL-based clone detection does not require any manual specification of features, and is comparable to existing clone detection approaches [283]. While White *et al.* did not manually specify any features, they did use a stream of identifiers and constant code tokens as a way to represent the code.

In this chapter, we posit a fundamental question of whether an underlying code representation can be successfully used to automatically learn code similarities. In our approach, we employ the representations of identifiers, ASTs, CFGs, and Bytecode and use DL algorithms to automatically learn necessary features (rather than "hardcoding" algorithms or crafting features in ML-based approaches), which in turn, can be used to support SE tasks. Moreover, we also study whether combinations of the models trained on diverse code representations can yield more accurate and robust support for the SE task at hand. Our conjecture is that each code representation can provide a different, orthogonal view of the same code fragment, thus allowing a more reliable detection of similar fragments. Being able to learn similarities from diverse code representations can also be helpful in many practical settings, where some representations are simply not available (*e.g.*, when only compiled code is available or ASTs cannot be built due to compilation errors) or when some of the representations are compromised (*e.g.*, code obfuscation would prevent using identifier-based approaches). Many real-world SE tasks, such as third-party library detection [189, 31] and code provenance [73] often deal with code having limited representations available that needs to be analyzed.

While we conduct our experiments on a specific SE task — clone detection — our goal is not to develop an ultimate clone-detection approach, but rather show that effective DL-based solutions can be assembled from diverse representations of source code. Specifically, the noteworthy contributions of this work are as follows:

1. *Deep Learning from different code representations* - We demonstrate that it is indeed possible to automatically learn code similarities from different representations, such as streams of identifiers, AST nodes, bytecode mnemonic opcodes, as well as CFGs. We evaluate how these different representations can be used in a DL-based approach for clone detection. We argue that our results should be applicable to any SE approach that relies on analyzing code similarities.

2. *Assembling a Combined model* - We demonstrate that combined models can be automatically assembled to consider multiple representations for SE tasks (in our example, code clone detection). In fact, we show that the combined model achieves overall better results when compared to stand-alone code representations.

3. *Inter-project similarities* - We also demonstrate that the proposed models can be effectively used to compute similarities, not only in the context of a single project, but also to analyze code similarities among different (diverse) software projects (*e.g.*, detecting clones or libraries across multiple projects).

4. *Model reusability and transfer learning* - We demonstrate that we can learn multi-representation models on available software systems and then effectively apply these models for detecting code similarities on previously unseen systems.

5. *Open science* - We release all the data used in our studies [7]. In particular, we include all the datasets, inputs/outputs, logs, settings, analysis results, and manually validated data.

4.2 Approach

Our approach can be summarized as follows: First, code fragments are selected at the different granularities we wish to analyze (*e.g.*, classes, methods). Next, for each selected fragment, we extract its four different representations (*i.e.*, identifiers, AST, bytecode, and CFG). Code fragments are embedded as continuous valued vectors which are learned for each representation. In order to detect similar code fragments, distances are computed among the embeddings. Finally, we demonstrate how we effectively combine different representations for the task of code clone detection and classification.

4.2.1 Code Fragments Selection

Given a compiled code base formed by source code files and compiled classes, code fragments are selected at the desired level of granularity: classes or methods. We start by listing all the `.java` files in the code base. For each Java file, we build the AST rooted at the `CompilationUnit` of the file. To do this, we rely on the Eclipse JDT APIs. We use the Visitor design pattern to traverse the AST and select all the classes and methods corresponding to `TypeDeclaration` and `MethodDeclaration` nodes. We discard interfaces and abstract classes since their methods do not have a bytecode and CFG representation. While it is possible to extract the other two representations (identifiers and ASTs), we chose to discard them so that we only learned similarities from code which exhibits all four representations. In addition, we filter out fragments (*i.e.*, classes or methods) smaller than ten statements. Similar thresholds have been used in previous work for minimal clone size [279]. Furthermore, smaller repetitive snippets are defined as micro-clones. Syntactic repetitiveness below this threshold has simply been considered uninteresting because it lacks sufficient content [37]. For each code fragment $c_i \in \{Classes \cup Methods\}$ we extract its AST node n_i and its fully qualified name (signature) s_i. Identifier and AST representations are extracted from an AST node n_i, while the bytecode and CFG representations are queried using the fully qualified name s_i.

4.2.2 Code Representation Extraction

We used the following representations of the source code: (i) identifiers; (ii) ASTs; (iii) bytecode; and (iv) CFGs. We then performed extraction and normalization as our preprossessing steps for each representation. In this section we describe these two steps for each selected representation.

4.2.2.1 Identifiers

In this representation, a code fragment is expressed as a stream (sentence) of identifiers and constants from the code. Similar representation is used by White *et al.* [283].

Extraction. Given the code fragment c_i and its corresponding AST node n_i, we consider the sub-tree rooted at the node n_i. To extract the representation, we select the terminal nodes (leaves) of the sub-tree and print their values. The leaf nodes mostly correspond to the identifiers and constants used in the code.

Normalization. Given the stream of printed leaf nodes, we normalize the representation by replacing the constant values with their type ($<$ `int` $>$, $<$ `float` $>$, $<$ `char` $>$, $<$ `string` $>$).

4.2.2.2 AST

In this representation, a code fragment is expressed as a stream (sentence) of the node types that compose its AST.

Extraction. Similarly to what was described above, the sub-tree rooted at the node n_i is selected for a given code fragment c_i. Next, we perform a pre-order visit of the sub-tree printing, for each node encountered, its node type.

Normalization. We remove two AST node types: `SimpleName` and `QualifiedName`. These nodes refer to identifiers in the code, and were removed because: (i) they represent low-level nodes, which are less informative than high-level program construct nodes in the AST (*e.g.*, `VariableDeclarationFragment`, `MethodInvocation`); (ii) they account for ∼46% of the AST nodes leading to a very large yet repetitive corpus; (iii) we target an AST representation able to capture orthogonal information as compared to the identifier representation. The latter is formed for ∼77% of terms belonging to `SimpleName`/`QualifiedName` nodes.

4.2.2.3 CFG

In this representation, a code fragment is expressed as it's CFG.

Extraction. To extract the CFG representation, we rely on Soot [8], a popular framework used by researchers and practitioners for Java code analysis. First, we extract the fully qualified name of the class from the signature s_i of the code fragment c_i. We use it to load the compiled class in Soot. For each method in the class, the CFG $G = (V, E)$ is extracted, where V is the set of vertices (*i.e.*, statements in the method) and E the set of directed edges (*i.e.*, the control flow between statements). In particular, the node represents the numerical ID of the statement as it appears in the method. Since the CFG is an intra-procedural representation, we see the method-level representation as a graph, while the class-level representation is a forest of graphs (CFGs of its methods).

Normalization. The CFG represents code fragments at a high-level of abstraction, therefore, no normalization is performed.

4.2.2.4 Bytecode

In this representation, a code fragment is expressed as a stream (sentence) of bytecode mnemonic opcodes (*e.g.*, `iload`, `invokevirtual`) forming the compiled code.

Extraction. Let c_i be the code fragment and s_i its signature. If c_i is a method, we extract the fully qualified name of its class from s_i, otherwise s_i already represents the name of the class. Then, the bytecode representation is extracted using the command $\texttt{javap} - \texttt{c} - \texttt{private} < \texttt{classname} >$ passing the fully qualified name of the compiled class. The output is parsed, allowing for the extraction of the class- or method-level representation.

Normalization. In the normalization step we remove the references to constants, keeping only their opcodes. For example, the instruction $\texttt{putfield\#2}$ is normalized as `putfield`. We also separate the opcodes stream of each method with the special tag $< \texttt{M} >$.

4.2.3 Embedding Learning

For each code fragment $c \in \{Classes \cup Methods\}$, we extract its representations: r_{ident}, r_{AST}, r_{byte}, and r_{CFG}. We learn a single embedding (*i.e.*, vector) for each representation, obtaining: e_{ident}, e_{AST}, e_{byte}, and e_{CFG}. An embedding represents a code fragment in a multidimensional space where similarities among code fragments can be computed as distances.

We use two strategies to learn embeddings for the aforementioned representations. For identifier, AST and bytecode representations, we use a DL-based approach that relies on recursive autoencoders [244, 245]. For the CFG representation we use a graph embedding technique [212].

4.2.3.1 DL Strategy

Let C be the set of all code fragments and R the corpus comprised by the representations of the code fragments in a representation (r_{ident}, r_{AST}, r_{byte}). For each code fragment $c \in C$, its representation $r \in R$ is a sentence of the corpus R. The sentence r is a stream of words $r = w_1, w_2, \ldots, w_j$, where w_i is a term of the particular representation (*i.e.*, an identifier, AST node type or bytecode opcode). The corpus is associated with a vocabulary V containing the unique terms in the corpus.

To learn an embedding for a sentence r, we perform two steps. In the first stage, we learn an embedding for each term w_i (*i.e.*, word embeddings), which comprises the sentence. In the second stage, we recursively combine the word embeddings to learn an encoding for the entire sentence r. We now describe these two stages in details.

In the first stage we train a Recurrent Neural Network (RtNN) on the corpus R, where the size of the hidden units is set to n, which corresponds to the embedding size [196]. The model, trained on the corpus R, generates a continuous valued vector, called an embedding, for each word $w_i \in V$.

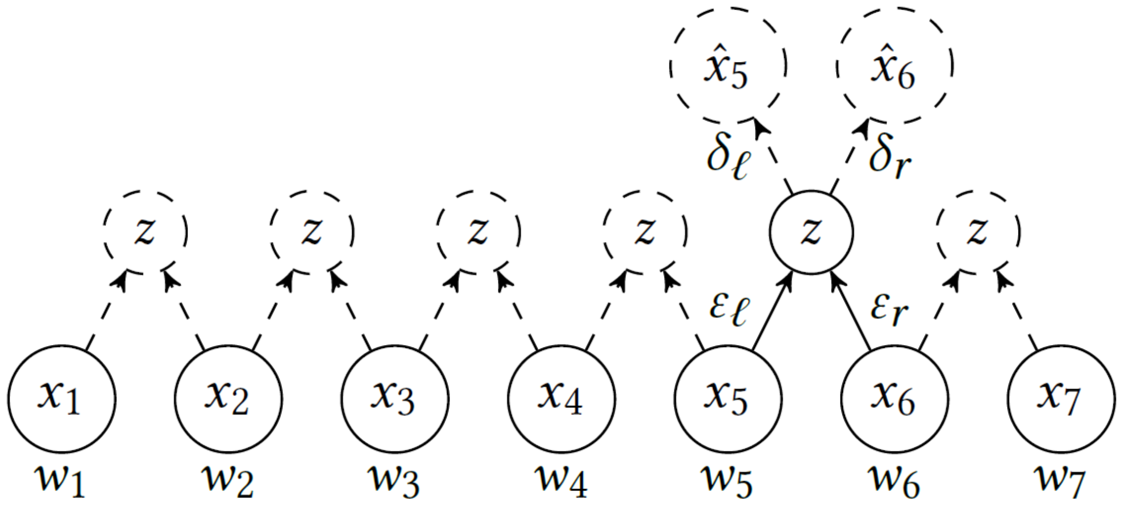


Figure 4.1: First iteration to encode a stream of tokens.

The second stage involves training a Recursive Autoencoder [244, 245] to encode arbitrarily long streams of embeddings. Fig. 4.1 shows the recursive learning procedure. Consider the sentence $r \in R$ formed by seven terms $\{w_1, \ldots, w_7\}$. The first step maps the stream of terms to a stream of $n-$dimensional embeddings $\{x_1, \ldots, x_7\}$. In the example in Fig. 4.1, there are six pairs of adjacent terms (*i.e.*, $[x_i; x_{i+1}]$). Each pair of adjacent terms $[x_\ell; x_r]$ is encoded by performing the following steps: (i) the two n-dimensional embeddings corresponding to the two terms, are concatenated into a single $2n$-dimensional vector $x = [x_\ell; x_r] \in \mathbb{R}^{2n}$; (ii) x is multiplied by a matrix $\varepsilon = [\varepsilon_\ell, \varepsilon_r] \in \mathbb{R}^{n \times 2n}$; (iii) a βias vector $\beta_z \in \mathbb{R}^n$ is added to the result of the multiplication; (iv) the result is passed to a nonlinear vector $f$unction f: $z = f(\varepsilon x + \beta_z)$.

The result z is an n-dimensional embedding that represents an encoding for the stream of two terms, corresponding to x. In the example in Fig. 4.1, x_ℓ and x_r correspond to the embeddings x_5 and x_6 respectively, which in turn, correspond to the terms w_5 and w_6. In this steps the autoencoder performs dimensionality reduction. In order to assess how good z encodes the pair $[x_\ell; x_r]$, the autoencoder tries to reconstruct the original input x from z in the δecoding phase. 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 $\beta_y \in \mathbb{R}^{2n}$: $y = \delta z + \beta_y$.

The output $y = [\hat{x}_\ell; \hat{x}_r] \in \mathbb{R}^{2n}$ is referred to as the model's *reconstruction* of the input. This model $\theta = \{\varepsilon, \delta, \beta_z, \beta_y\}$ is called an *autoencoder*, and training the model involves measuring the Error between the original input vector x and the reconstruction y:

$$E(x;\theta) = ||x_\ell - \hat{x}_\ell||_2^2 + ||x_r - \hat{x}_r||_2^2 \tag{4.1}$$

The model is trained by minimizing Eq. (4.1). Training the model to encode streams with more than two terms requires recursively applying the autoencoder. The recursion can be performed following predefined recursion trees or by using optimization techniques, such as the greedy procedure defined by Socher *et al.* [243]. The procedure works as follows: in the first iteration, each pair of adjacent terms are encoded (Fig. 4.1). The pair whose encoding yields the lowest reconstruction error (Eq. (4.1)) is the pair selected for encoding at the current iteration. For example, in Fig. 4.1, each pair of adjacent terms are encoded (*e.g.*, dashed lines) and the pair of terms w_5 and w_6 is selected to be encoded first. As a result, in the next iteration, x_5 and x_6 are replaced by z and the procedure repeats. Upon deriving an encoding for the entire stream, the backpropagation through structure algorithm [103] computes partial derivatives of the (global) error function w.r.t. θ. Then, the error signal is optimized using standard methods.

4.2.3.2 Graph Embedding Strategy

To generate the embeddings for CFG representations we employ the graph embedding technique HOPE [212] (High-Order Proximity preserved Embedding). We rely on this technique for two main reasons: (i) HOPE has been shown to achieve good results in graph reconstruction [105]; (ii) unlike other techniques such as *e.g.*, SDNe, HOPE embeds directed graphs (as needed in the case of CFGs).

Given a graph $G = (V, E)$, HOPE generates an embedding for each node in the graph. Next, a single embedding is generated for the whole graph performing mean pooling on the node's embeddings. HOPE works by observing a critical property of directed graphs known as asymmetric transitivity. This property helps to preserve the structure of directed graphs by identifying correlations of directed edges. For example, assume three distinct, directed paths from v_1 to v_5 . Each of these paths increases the correlation probability that there exists a directed edge from v_1 to v_5 , however, the lack of directed paths from v_5 to v_1 decreases the probability of there being a direct edge from v_5 to v_1.

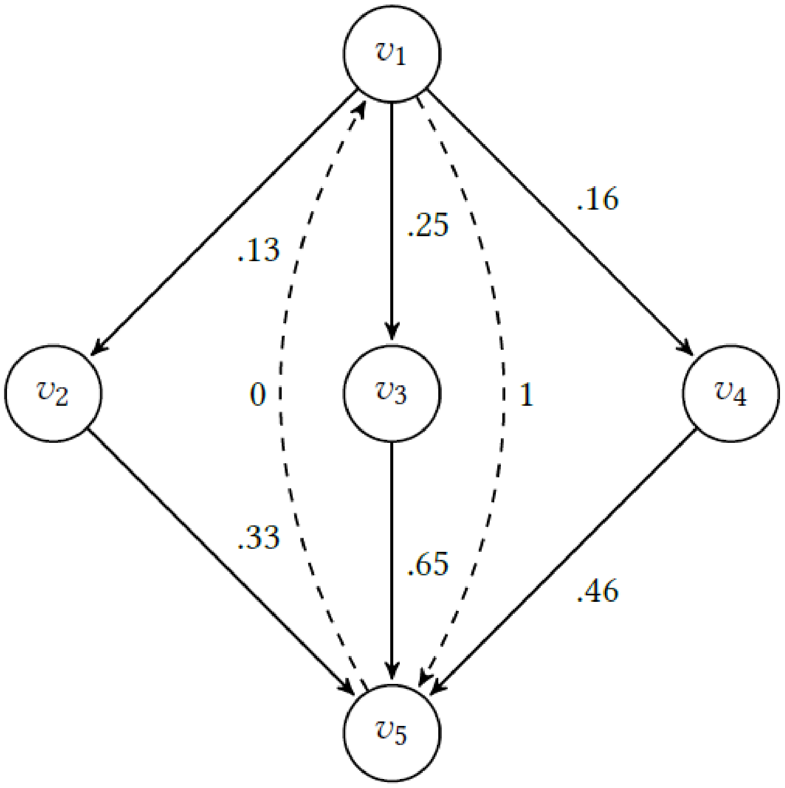


Figure 4.2: HOPE Embedding Technique

HOPE looks to preserve asymmetric transitivity by implementing the highly correlated metric known as the Katz proximity. This metric gives weights to the asymmetric transitivity pathways, such that they can be captured through the embedding vectors. HOPE learns two embedding vectors; the source vector and the target vector for each vertex. These vectors are then assigned values based upon the weights of the edges and their nature (source or target). For example, consider the graph in Fig. 4.2. Each of the solid lined edges are directed edges and the dotted edges capture the asymmetric transitivity of the graph. The vector created for v_1 as the target vector will be 0 since no paths lead to v_1 as their target. However, the value assigned to v_5 target vector will be much higher since many paths end with v_5 as their target. HOPE creates and learns the embeddings of the graph by using Singular Value Decomposition for each vertex. Then, through mean pooling on the nodes embeddings, a single embedding for the entire graph can be generated.

4.2.4 Detecting Similarities

Let E be the set of embeddings learned for all code fragments by a particular representation (*i.e.*, E_{ident}, E_{AST}, E_{byte}, E_{CFG}). We compute pairwise Euclidean distances between each and every pair of embeddings $e_i, e_j \in E$. The set of distances D are normalized between $[0, 1]$ and a threshold t is applied to the distances in order to detect similar code fragments.

4.2.5 Combined Models

Each of the four models we built are trained on a single representation and identifies a specific set of similar code fragments. Such models can be combined using *Ensemble Learning*, an ML paradigm where multiple learners are trained to solve the same problem. In contrast to ordinary machine learning approaches, which try to learn *one* hypothesis from training data, ensemble methods try to construct a *set* of hypotheses and combine them [298].

A simple class of Ensemble Learning techniques are the algebraic combiners, where distances computed by several, single-representation models, are combined through an algebraic expression, such as minimum, maximum, sum, mean, product, median, *etc.*. For example, a weighted average sum of the distances computed from each representation can be computed. Formally, given two code fragments a and b, and their n representations, one can compute a dissimilarity score ds as follows:

$$ds(a, b) = \frac{1}{n} \sum_{i=1}^{n} w_i d_i(a, b)$$

Where $d_i(a, b)$ is the distance between a and b computed using the i-th representation, and w_i is the weight assigned to the i-th representation. Weights can be set based on the importance of each representation and the types of similarities we wish to detect.

Single-representation models can also treated as experts and combined using voting-based methods. Each single-representation model expresses its own vote about the similarity of two code fragments (*i.e.*, are similar *vs* are not similar) and these decisions are combined in a single label. Different strategies can be used depending on the goal: (i) (weighted) majority of voting; (ii) at least one vote (max recall); (iii) all votes needed (max precision).

The aforementioned strategies are non-trainable combiners. However, given a set of instances for which the oracle is available, ensemble learning techniques can be employed to perform training. Random Forest is one of the most effective ensemble learning method for classification, which relies on decision tree learning and bagging [47]. Bagging (*i.e.*, bootstrap aggregating), is a technique that generates bootstrapped replicas of the training data. In particular, different training data subsets are randomly drawn with replacement from the entire training dataset. Random Forest, in addition to classic bagging on the training instances, also selects a random subset of the features (feature bagging). Each training data subset is used to train a different deep decision tree. For a new instance, Random Forest averages the predictions by each decision tree, effectively reducing the overfitting on the training sets.

In the following, we show the effectiveness of combining similarities learned from different representations training two models in the context of clone detection (*i.e.*, a model classifying clone candidates as *true* or *false* positives) and classification (*i.e.*, a model that other than discerning clone candidates in *true* or *false* positives, also classifies the true positives in their own clone type).

Similarities from different code representations can be useful not only to detect clones, but also to classify them into types. For example, clone pairs with very high AST similarity, but low identifiers similarity are likely to be Type-II clones (possibly parameterized clones, where all the identifiers have been systematically renamed).

4.3 Experimental Design

Table 4.1: Project Statistics

Project	#Classes	#Methods	Total LOC
ant-1.8.2	1,608	12,641	127,507
antlr-3.4	381	3,863	47,443
argouml-0.34	1,408	8,465	105,806
hadoop-1.1.2	3,968	21,527	319,868
hibernate-4.2.0	7,119	46,054	431,693
jhotdraw-7.5.1	765	6,564	79,672
lucene-4.2.0	4,629	23,769	412,996
maven-3.0.5	837	5,765	65,685
pmd-4.2.5	872	5,490	60,739
tomcat-7.0.2	1,875	15,275	181,023

The *goal* of this study is to investigate whether source code similarity can be learned from different kinds of program representations, with the *purpose* of understanding whether one can combine different representations to obtain better results, or use alternative representations when some are not available. More specifically, the study aimed to address the following research questions (RQ):

RQ$_1$ How effective are different representations in detecting similar code fragments?

RQ$_2$ What is the complementarity of different representations?

RQ$_3$ How effective are combined multi-representation models?

RQ$_4$ Are DL-based models applicable for detecting clones among different projects?

RQ$_5$ Can trained DL-based models be reused on different, previously unseen projects?

4.3.1 Datasets Selection

We make use of two datasets: (i) *Projects*: a dataset of ten Java projects; (ii) *Libraries*: a dataset comprised of 46 Java libraries.

4.3.1.1 Projects

This dataset is comprised of ten compiled Java projects extracted from the *Qualitas.class Corpus* [256]. We rely on this dataset because it is publicly available and the projects have been already compiled. This (i) avoids any potential problem/inconsistency in compiling projects, and (ii) ensures reproducibility. The selection of ten projects aimed at obtaining a diverse dataset, in terms of application domain and code size. Table 4.1 reports statistics of the dataset that we use in all our research questions except RQ_4.

4.3.1.2 Libraries

This dataset is comprised of 46 different Apache commons libraries [6]. We selected all the Apache commons libraries, for which we were able to identify both binaries and source code of the latest available release. We downloaded the compressed files for binaries and source code. Within the binaries, we located the `jar` file, which represents the library, and extracted the `.class` files. The compressed source code files were simply decompressed. The list of considered libraries is available in our replication package. We use this dataset in RQ_4 for inter-project clone detection.

4.3.2 Experimental Procedure and Data Analysis

We discuss the experimental procedure and data analysis we followed to answer each research question.

4.3.2.1 RQ$_1$:How effective are different representations in detecting similar code fragments?

Given the projects dataset $P = \{P_1, \ldots, P_{10}\}$ and the four code representations $R = \{R_1, R_2, R_3, R_4\}$, we extract for each code artifact $c \in \{Classes \cup Methods\}$ its four representations r_1, r_2, r_3, r_4. Then, for each project P_i we train the representation-specific model on the code representations at class-level. With the trained models, we subsequently generate the embeddings for both classes and methods in the project. Therefore, the code artifact represented as r_1, r_2, r_3, r_4 will be embedded as e_1, e_2, e_3, e_4, where e_i is the embedding of the i-th representation. We set the embedding size of Identifiers, AST and Bytecode to 300, while the CFG embedding size to four. The latter is significantly smaller than the former because (i) CFGs are abstract representations which do not require large embeddings; (ii) in order to converge towards an embedding representation, HOPE (and internally SVD) needs the embedding size to be smaller than the minimum number of nodes or edges in the graph. Pairwise Euclidean distances are computed for each pair of classes and methods of each system P_i. The smaller the distance, the more similar the two code artifacts.

In the next step, for each code representation R_i, we query the clone candidates from the dataset P at class and method level. To query the clone candidates, we apply a threshold on the distances to discern what qualifies as clones. We use the same two thresholds at class- and method-level ($T_{class} = 1.00e - 08$ and $T_{methods} = 1.00e - 16$, similar to [283]) for each representation. While, ideally, these thresholds should be tuned for each project and representation, we chose the same thresholds to facilitate the comparison among the four representations. Once we obtained the two sets of candidates $CandClasses_i$ and $CandMethods_i$ for each representation R_i, we performed the union of the candidate sets of the same granularity among all the representations: $CandClasses = \bigcup_{i=1}^{4} CandClasses_i$ (the same applies for $CandMethods$).

For each candidate $c \in CandClasses$ (or $CandMethods$) we generate a tuple $t_c = \{b_1, b_2, b_3, b_4\}$ where $b_i = Trueiffc \in CandClass_i$ (*i.e.*, if c is identified as a clone by R_i) and $b_i = False$ otherwise. t_c can assume $2^4 = 16$ possible values (*i.e.*, $t_c = \{FFFF$, $FFFT$, $\ldots$, $TTTT\}$). However, the combination $FFFF$ does not appear in our dataset since $CandClasses$ and $CandMethods$ are sets containing the union of the clones identified by all representations, thus ensuring the presence of at least one True value. Therefore, there are 15 unique classes of values for t_c. We use these sets to partition the candidates in $CandClasses$ and $CandMethods$. Next, from each candidate clones partition, we randomly draw a statistically significant sample with 95% confidence level and $\pm 15\%$ confidence interval. The rationale is that we want to evaluate candidates belonging to different levels of agreement among the representations.

Three authors independently evaluated the clone candidate samples. The evaluators decided whether the candidates were true or false positives and, in the former case, the clone type. To support consistent evaluations, we adapted the taxonomy of editing scenarios designed by Svajlenko *et al.* [255] to model clone creation and to be general enough to apply to any granularity level. Given the manually labeled dataset from each of the three evaluators, we computed the two-judges agreement to obtain the final dataset (*e.g.*, a candidate clone was marked as a true positive if at least two of the three evaluators classified it as such). In order to statistically assess the evaluators agreement, we compute the Fleiss' kappa [93]. In particular, we compute the agreement for True and False positive as well as the agreement in terms of Clone Types. On the final dataset, precision and recall were computed for each candidate clones partition (*e.g.*, $TFFF$, $FTFF$, *etc.*) and, overall, for each representation in isolation. We quantitatively and qualitatively discuss TP/FP pairs for each representation, as well as the distribution of clone types.

4.3.2.2 RQ$_2$:What is the complementarity of different representations?

To answer RQ$_2$ we further analyze the data obtained in RQ$_1$ to investigate the complementarity of the four representations. First, we compute the intersection and the difference of the true positive sets identified with each representation. Precisely, the intersection and difference of two representations R_i and R_j are defined as follows:

$$R_i \cap R_j = \frac{|TP_{R_i} \cap TP_{R_j}|}{|TP_{R_i} \cup TP_{R_j}|}\% \quad \text{and} \quad R_i \setminus R_j = \frac{|TP_{R_i} \setminus TP_{R_j}|}{|TP_{R_i} \cup TP_{R_j}|}\%$$

where TP_{R_i} represents true positive candidates identified by R$_i$. We compute the percentage of candidates exclusively identified by a single representation and missed by all the others:

$$EXC(R_i) = \frac{|TP_{R_i} \setminus \bigcup_{j \neq i} TP_{R_j}|}{|\bigcup_j TP_{R_j}|}\%$$

Then, to understand whether these code representations are orthogonal to each other, we collect all the distance values of each representation computed for each pair of code fragments (classes or methods) in the dataset. Given these distance values, we compute the Spearman Rank Correlation [249] between each pair of representations, to investigate the extent to which distances computed from different representations on the same pairs of artifacts correlate.

4.3.2.3 RQ$_3$:How effective are combined multi-representation models?

To answer RQ$_3$ we evaluate the effectiveness of two combined models: *CloneDetector*, which classifies candidate clones as true/false positives, and *CloneClassifier*, which provides information about the type of detected clone (Type-1, 2, 3 or 4). While we are aware that is its possible to determine a clone type by comparing the sequence of tokens, the purpose of this classification is to show the potential of the proposed approach to provide complete information about a clone pair. The two models are trained using the manually labeled dataset obtained in RQ$_1$, with the distances computed by each representation used as features and the manual label used as target for the prediction. To train the two models we rely on Random Forest employing the commonly used 10-fold cross-validation technique to partition training and test sets. We evaluate the effectiveness of the two models computing

Precision, Recall and F-Measure for each class to predict (*e.g.*, clone *vs* not-clone for the *CloneDetector*).

4.3.2.4 RQ$_4$:Are DL-based models applicable for detecting clones among different projects?

The goal of RQ$_4$ is two-fold. On the one hand, we want to instantiate our approach in a realistic usage scenario in which only one code representation is available. On the other hand, we also want to show that the DL based model can be used to identify inter-project clones. Indeed, the latter is one of the major limitations of the work by White *et al.* [283], where given the potentially large vocabulary of identified-based corpora, the approach was evaluated only to detect intra-project clones.

We instantiate two usage scenarios both relying on the same *Library* dataset, and on training performed on the binaries (bytecode from `.class` files). In the first scenario, a software maintainer has to analyze the amount of duplicated code across projects belonging to their organization. We use the entire *Library* dataset and filter for inter-project clone candidates only (*i.e.*, candidates belonging to different projects). Clone candidates are evaluated by inspecting the corresponding Java files from the downloaded code.

In the second scenario, a software developer is using a jar file (*i.e.*, compiled library) in their project. The developer has no information about other libraries that could be redistributed with the jar file, since only compiled code is present. For provenance and/or licensing issues, the developer needs to address whether the jar file j imports/shadows any of the libraries in a given dataset L.

To perform this study we select `weaver-1.3` as the jar j and the remaining 45 libraries in the *Library* dataset as L. We identify similar classes between j and L. Then, to assess whether the identified classes have actually been imported in j from library $x \in L$, we downloaded the j's source code and analyzed the building files (`weaver-1.3` relies on Maven, thus we investigated the `.pom` files) to check whether the library x is imported as a dependency in j.

4.3.2.5 RQ$_5$:Can trained DL-based models be reused on different, previously unseen projects?

One of the major drawbacks of DL-based models is their long training time. This training time could be amortized if these models we able to be reused across different projects belonging to different domains. The major factor that hinders the reusability of such models is the possible variability in the vocabulary for new, unseen projects. For example, a language model and a recursive autoencoder trained on a given vocabulary would not be able to provide adequate results on a vocabulary containing terms not previously seen during the training phase. Such a vocabulary should be cleaned, either by stripping off the unknown terms, replacing them with special words (*e.g.*, $<$ `unkw` $>$) or using smoothing techniques. These solutions negatively impact the performance of the models.

While in principle, with enough training data and available time, any of the aforementioned representation-specific models could be reused on a different dataset (*i.e.*, unseen project), in practice some representation-specific models are more suitable than others for reuse. In particular, models trained on representations with a limited or fixed vocabulary are easier to be reused on different projects. In our study, AST and Bytecode representations both have a fixed vocabulary, limited respectively by the number of different AST node types and bytecode opcodes.

To answer RQ$_5$ we perform a study aimed at evaluating the effectiveness of reusing AST models. We evaluate the effectiveness, showing that a reused model identifies a similar list of clone candidates as compared to the original model trained on the set projects. We select the AST representation which was trained on one of the biggest project in the dataset, RQ$_1$ (*i.e.*, `lucene`). We use this AST model to generate the embeddings for the remaining nine projects in the dataset. Using the generated embeddings we compute the distances and query the clone candidates using the same class- and method-level thresholds used in RQ$_1$. Then, let L_R and L_O be the lists of candidates returned from the reused model and from the original model, we compute the percentage of candidates in L_R that are in L_O and *vice versa*.

We also show that the combined model *CloneDetector*, trained on clone candidates belonging to a single project (`hibernate`), can be effectively used to detect clones in the remaining nine systems of the *Projects* dataset.

4.4 Results

RQ$_1$: How effective are different representations in detecting similar code fragments? Table 4.2 shows the results in terms of precision for different candidate clone partitions, where each clone partition includes clones detected only by a given combination of representations. For example, the first partition $FFFT$ represents the clone candidates identified by the Bytecode representation, but not by the other three representations. Note that for the partition with $ID = 10$ ($TFTF$) no class-level clones have been identified.

Table 4.2: Performances for different clone partitions

ID	Iden	AST	CFG	Byte	Precision % Methods	Precision % Classes
1	F	F	F	**T**	5	49
2	F	F	**T**	F	9	58
3	F	F	**T**	**T**	88	73
4	F	**T**	F	F	79	63
5	F	**T**	F	**T**	95	93
6	F	**T**	**T**	F	100	100
7	F	**T**	**T**	**T**	100	100
8	**T**	F	F	F	95	100
9	**T**	F	F	**T**	100	100
10	**T**	F	**T**	F	100	-
11	**T**	F	**T**	**T**	100	100
12	**T**	**T**	F	F	100	100
13	**T**	**T**	F	**T**	100	100
14	**T**	**T**	**T**	F	100	100
15	**T**	**T**	**T**	**T**	100	100

Table 4.2 shows that most of the partitions exhibit a good precision, with peaks of 100%. The notable exceptions are the partitions with ID 1 and 2, referring to clone candidates detected only by the Bytecode and by the CFG representations, respectively. We investigated such false positives and qualitatively discuss them later.

Table 4.3 shows the overall results for method- and class-level aggregated by representation. The table contains the raw count of True Positives (TP), False Positives (FP) and clone types identified by each representation. Estimated precision and recall is also reported in the last two columns of the tables. Note that the results of the single representation R_i (*e.g.*, Identifier) reported in 4.3 refer to aggregated candidates of table 4.2 where the representation R_i is the True (*e.g.*, clone partitions ID 8-15).

Table 4.3: Performances for different representations

Methods								
Representation	FP	TP	Type I	Type II	Type III	Type IV	Precision	Recall
Iden	1	201	151	15	35	0	100%	52%
AST	11	292	138	132	19	3	96%	75%
CFG	43	178	69	81	19	9	81%	46%
Byte	46	222	89	77	49	7	83%	57%
Classes								
Representation	FP	TP	Type I	Type II	Type III	Type IV	Precision	Recall
Iden	0	120	23	51	46	0	100%	40%
AST	18	188	18	121	44	5	91%	63%
CFG	24	120	7	65	41	7	83%	40%
Byte	34	217	23	115	77	2	86%	73%

The Identifier-based models achieve the best precision, both when working at method- and class-level. AST-based models provide the best balance between precision and recall. In terms of types of clones, Identifier-based models detect the highest number of Type I, AST-based models identify the highest number of Type II, Bytecode models identify the highest number of Type III, and CFG models detect the highest number of Type IV. This suggests that the four representations are complementary, and will be addressed further in RQ_2.

The data presented in this section is based on the agreement of three evaluators. The Fleiss' kappa shows substantial agreement in terms of TP/FP (93% for methods and 65% for classes) and a moderate/substantial agreement in terms of clone types classification (75% for methods and 57% for classes). In the following breakdown, we discuss the results achieved by models using different representations.

Identifiers. The identifiers-based model achieves the best precision both in method- and class-level results. However its estimated recall is respectively 52% and 40%, meaning that the model fails to detect a significant percentage of clones. While the recall could be increased by appropriately raising the threshold (*i.e.*, by adopting a more permissive threshold) at the expense of precision, we found that this would still not be enough to detect most of the clones detected by other representations. Indeed, we manually investigated several TP clone candidates identified by other representations and missed by the Identifier-based model. We found that the distances computed by the Identifier-based model for such clones were several orders of magnitude higher than a reasonable threshold (*e.g.*, the ones we use or the ones used by White *et al.* [283]). Moreover, a close inspection at the vocabulary of the candidates showed that they share a small percentage of identifiers, which in turn, makes them hard to detect with such a representation. Clearly, clone candidates where pervasive renaming or obfuscation has been performed, would be very difficult to detect with this model.

Bytecode & CFG. From Table 4.2, we can notice that candidates detected only by Bytecode or only by CFG models (partitions with ID 1 and 2) tend to be false positives. The precision for such partitions is 5% and 9% at method-level and 49% and 58% at class-level. However, when both Bytecode and CFG models detect clones not detected by the AST and Identifier-based models (partition with ID 3), they achieve reasonable levels of precision (88% and 73%).

Bytecode and CFG representations have a very high degree of abstraction. Within the CFG, a statement is simply represented by a node. Similarly, the bytecode is formed by low level instructions, which do not retain the lexical information present in the code (*e.g.*, a method call is represented as a `invokevirtual` or `invokestatic` opcode). Such a level of abstraction appears to be imprecise for fine granularities such as methods, where the code fragments might have similar structure but, at the same time, perform very different tasks. Better results are achieved at class-level, where false positives are mainly due to Java Beans, thus classes having a very similar structure, *i.e.*, class attributes with getters and setters acting on them performing similar low-level operations (storing/loading a variable, creating an object, *etc.*). While these classes are false positives in terms of code clones, they still represent a successful classification in terms of learning structural and low-level similarities from the code. The general lesson learned is that the use bytecode and CFG in a combined model yield an acceptable precision. Indeed, if bytecode is available (compiled code) CFGs can be extracted as well.

Table 4.3 also shows that Bytecode and CFG are the representations that detect the most Type IV clones in our dataset. Again, this is likely due to their high level of abstraction in representing the code.

AST. AST-based models seem to have the best overall balance between precision and recall. These models can identify similar code fragments, even when their set of identifiers (*i.e.*, vocabulary) is significantly different and share almost no lexical tokens. This has been confirmed by our manual investigation of the candidates. We report several examples of candidates identified exclusively by the AST model in our online appendix.

RQ$_2$: What is the complementarity of different representations? RQ2: What is the complementarity of different representations? Table 4.4 reports the complementarity metrics between the different representations. In particular, on the left side of table 4.4 we report the intersection of sets of true positive candidates detected by the four representations. For example, 40% of the true positives are detected by both the AST and Identifier models. The relatively small overlap among the candidate sets, at both method and class granularity (no more than 51%), suggests that these representations complement each other.

Table 4.4: Complementarity Metrics

Methods											
Intersection %					Difference %					Exclusive %	
$R_1 \cap R_2$	Iden	AST	CFG	Byte	$R_1 \setminus R_2$	Iden	AST	CFG	Byte	R_i	$EXC(R_i)$
Iden		40	21	36	Iden		17	43	29	Iden	5% (21)
AST			42	44	AST	43		46	38	AST	9% (33)
CFG				36	CFG	36	12		24	CFG	1% (4)
Byte					Byte	35	18	39		Byte	1% (2)
Classes											
Intersection %					Difference %					Exclusive %	
$R_1 \cap R_2$	Iden	AST	CFG	Byte	$R_1 \setminus R_2$	Iden	AST	CFG	Byte	R_i	$EXC(R_i)$
Iden		33	14	42	Iden		19	43	8	Iden	3% (8)
AST			31	51	AST	48		49	19	AST	9% (26)
CFG				34	CFG	43	20		14	CFG	7% (21)
Byte					Byte	49	30	52		Byte	7% (21)

The middle section of table 4.4 shows the difference in the sets of true positive candidates detected by the four representations. The reported values show the percentage of true positive clones detected by a certain representation and missed by the other. For example, 48% of the true positive clones identified by the AST model at class level are not identified by the Identifier-based model.

Finally, the left section of table 4.4 shows the percentage and number of instances (in parenthesis) of true positive candidates identified by each representation and missed by all the others. From these results, we note that there are candidates exclusively identified by a single representation. Note that the number of instances and percentages are computed only on the manually validated sample.

Table 4.5 shows the Spearman's Rank correlation coefficient (ρ) for the different representations. While all the computed correlations are statistically significant, their low value suggests that distances computed with different representations do not correlate and provide different perspectives about the similarity of code pairs. The strongest correlation

Table 4.5: Spearman's rank correlation

$\rho(R_1, R_2)$	Ident	AST	CFG	Byte
Ident		0.094	0.120	0.069
AST			0.157	0.031
CFG				0.046
Byte				

we observe is between the AST and the CFG representations, which is still limited to 0.157 (basically, no correlation).

For example, the classes `MessageDestination` and `ContextEjb` were detected only by the AST representation. They both extend `ResourceBase` and offer the same functionalities. These classes share only a few identifiers (therefore not detected by the Identifier model) and differ in some if-structures, making them more difficult to detect by CFG and Bytecode representations. This and other examples are available in our online appendix [7].

RQ$_3$:How effective are combined multi-representation models? Table 4.6 reports the results of the *CloneDetector* at class- and method-levels. When identifying clone instances, the model achieves 90% precision and 93% recall at class, and 98% precision 97% recall at method level. The lower performance at class-level could be due to the smaller dataset available (*i.e.*, 483 methods *vs* 362 classes). The overall classification performance (*i.e.*, the weighted average of the Clone/Not Clone categories) is also lower for the class-level (~85% F-Measure for classes, and ~96% for methods).

We also trained the model by considering a subset of the representations. In our online appendix, we provide results considering all possible subsets of features. We found that single-representation models tend to have worse performance than the combined model, which was trained on all representations. However, the combinations with Identifiers+AST+{CFG *or* Bytecode} obtain results similar to the overall model trained on all representations.

Table 4.7 shows the results of the *CloneClassifier* at class- and method-levels. At class level, the *CloneClassifier* has overall F-Measure of 68%, with average precision of 67% and recall of 68%. At method level, the model obtains an overall F-Measure of 84%, with an average precision and recall of 84% across the different categories (*i.e.*, Not Clone and the

Table 4.6: Performance of the *CloneDetector*

	Methods			Classes		
	Precision %	Recall %	F-Measure %	Precision %	Recall %	F-Measure %
Clone	98	97	98	90	93	91
Not Clone	90	91	90	61	52	56
Weighted Avg.	96	96	96	85	86	85

four types of detected clones). As expected, the category with the lowest precision and recall is the Type IV clones for both method and class level, while other clone types achieve a precision $\geq 74\%$ and a recall $\geq 75\%$. Single-representation models obtain significantly worse results in the classification task (with a bigger gap with respect to what observed for the *CloneDetector*). All the results are available in our online appendix.

Table 4.7: Performance of the *CloneClassifier*

	Methods			Classes		
	Precision %	Recall %	F-Measure %	Precision %	Recall %	F-Measure %
Not Clone	89	94	91	59	61	60
Type I	89	88	88	86	78	82
Type II	82	84	83	81	85	83
Type III	74	75	75	61	59	60
Type IV	67	18	29	00	00	00
Weighted Avg.	84	84	84	67	68	68

RQ$_4$:Are DL-based models applicable for detecting clones among different projects? We identified several groups of duplicate code across different Apache commons libraries in the dataset. The largest group of clones is between the libraries `lang3-3.6` and `text-1.1`. The duplicated code involves classes operating on Strings, for example: StringMatcher, StrBuilderWriter, StrTokenizer, StrSubstitutor *etc.*, for a total of 21 shared similar classes identified. We also identified another group of similar classes between `text-1.1` (package `diff`) and `collections4-4.1` (package `sequence`). In particular, the class StringsComparator in `text-1.1` appears to be a Type III clone of the class SequencesComparator in `collections4-4.1`. An example of Type II clone within these two libraries is instead the pair EditScript and ReplacementsFinder classes. These clones are classified as Type II, since only package information has been changed.

The libraries `math3-3.6.1` and `rng-1.0` contain two shared classes: ISAACRandom and Gamma/InternalGamma. ISACCRandom is a pseudo-random number generator. The classes share many similarities across the two libraries, however, they differ on a statement and structural level. As an example, both classes implement a method which sets a seed. In one class, the seed is set by an integer passed through an argument, while the other class sets the seed by using a combination of the current time and system hash code of the instance. Gamma and InternalGamma, named respectively to the library they belong to, are clones since parts of the Gamma class were used to develop InternalGamma. The developers only took the needed functionalities out of the Gamma class, stripped it of unnecessary code, and built InternalGamma. Therefore, many of the methods in Gamma either do not appear or are significantly smaller in InternalGamma. Despite InternalGamma being significantly smaller than Gamma, our tool was still able to detect similarity between the two classes.

The libraries `codec-1.9` and `net-3.6` share the same implementation for the class Base64, providing encoding/decoding features.

Note that the classes mentioned in this study do not refer to imported libraries but to actual Java duplicated code (*i.e.*, the source code files are in both libraries).

We also identified false positives in this study, mainly due to small inner classes and enumerators. Enumerators have a very similar bytecode structure even if containing different constants. They are uninteresting with respect to the goal of this scenario.

Imported and shaded classes. We identified a large list of shared classes between the library j (`weaver-1.3`) and the following libraries in the dataset L: `collections4-4.1` (373 classes), `lang3-3.6` (79), and `io-2.5` (13). A closer inspection of the building files of `weaver-1.3` showed that the aforementioned libraries have been imported and shaded. That is, the dependencies have been included and relocated in a different package name in order to create a private copy that `weaver-1.3` bundles alongside its own code.

RQ$_5$:Can trained DL-based models be reused on different, previously unseen projects? Table 4.8 shows the percentage of candidates in L_R that are also in L_O and vice versa, both at method- and class-level. Generally, the list of candidates identified by the reused model and the original models tend to be similar. At method-level, we can see that 97% of the candidates identified by the reused model were also identified by the original model. Similarly, 93% of the candidates returned by the original model are identified by the reused model. At class-level we notice smaller percentages. This is mostly due to the fact that fewer clones are identified at the class-level. For example, for `antlr-3.4` the reused model identifies three candidates while the original model only identifies one. For `maven-3.0.5`, two candidates are identified by the reused model and only one by the original model. Still, 90% of the class-level candidates identified by the original models are detected by the reused model.

Table 4.8: Model Reusability

Project	Methods %		Classes %	
	$L_R \in L_O$	$L_O \in L_R$	$L_R \in L_O$	$L_O \in L_R$
ant-1.8.2	99	88	73	31
antlr-3.4	100	100	33	100
argouml-0.34	99	96	97	73
hadoop-1.1.2	99	95	95	74
hibernate-4.2.0	89	82	30	84
jhotdraw-7.5.1	99	98	82	77
maven-3.0.5	97	84	50	100
pmd-4.2.5	97	99	99	99
tomcat-7.0.2	98	97	87	69
Overall	97	93	58	90

We also show that combined models can be reused on different systems. The *CloneDetector* model has been trained only on the data available for one project (`hibernate`) and tested on all the instances of the remaining projects. It achieved 98% precision and 92% recall at method-level and 99% precision and 95% recall at class-level.

4.5 Threats To Validity

Construct validity. The main threat is related to how we assess the complementarity of the code representations. We mitigate this claim performing different analyses: (i) complementarity metrics; and (ii) correlation tests.

Internal validity. This is related to possible subjectiveness when evaluating similarities of code fragments. To mitigate such threat, we employed three evaluators who independently checked the candidates. Then, we computed two-judge agreement on the evaluated candidates. We also qualitatively discuss false positives and borderline cases. Also, all of our evaluations are publicly available [7].

External validity. The results obtained in our study using the selected datasets might not generalize to other projects. To mitigate this threat, we applied our approach in different contexts and used two different datasets; *Projects* and *Libraries*. For *Projects*, which is a subset of systems from the Qualitas.class corpus, we selected diverse systems in terms of size and domain, focusing on popular ones. All the *Libraries* studied in this work, which are comprised of all the *Apache commons* libraries, are publicly available to ensure the replicability of the study. We did not utilize other clone-focused datasets (*e.g.*, such as BigCloneBench [254]) because compiled code is required in order to extract the representations of CFGs and Bytecode. Another threat in this category is related to the fact that we apply our approach on Java code only. While the representation extraction steps are implemented for Java, all the subsequent steps are completely language-agnostic because they rely on a corpus of sentences with arbitrary tokens. In fact, Recursive Autoencoders have been used in several contexts with different inputs such as natural language, source code, images *etc.* We do not compare our approach against code clone detection techniques since the focus of this paper is to show a general technique on how to learn similarities from multiple code representations, rather than building a code clone detector tool. Last, but not least, we focused on four code representations, but there may be others that are worthwhile to investigate (*e.g.*, data-flow or program dependency graphs).

4.6 Conclusion

In this chapter, we show that code similarities can be learned from diverse representations of the code, such as Identifiers, ASTs, CFGs and bytecode. We evaluated the performance of each representation for detecting code clones and show that such representations are orthogonal and complement each other. We also show that our model is reusable, therefore, eliminating the need for retraining the DL approach so that it is project specific. This eliminates a large timesink, native to DL approaches, and broadens the applicability of our approach.

Moreover, combined models relying on multiple representations can be effective in code clone detection and classification. Additionally, we show that Bytecode and CFG representations can be used for library provenance and code maintainability. These findings speak to the vast amount of SE tasks which benefit from analyzing multiple representations of the code. We instantiated our approach in different use case scenarios and datasets.

Our approach inherently highlights the benefits of not only single code representations, but the combinations of these representations. This work also begins to emphasize the attributes that different code representations can accentuate. This allows for a more targeted choice by SE researchers of a code representation when applying representation-DL algorithms to a SE tasks. We believe that learning similarities from different representations of the code, without manually specifying features, has broad implications in SE tasks, and is not limited solely to clone detection.

Chapter 5

Deep Learning Assert Statements

5.1 Introduction

Writing high-quality software tests is a difficult and time-consuming task. To help tame the complexity of testing, ideally, development teams should follow the prescriptions of the test automation pyramid [68], which suggests first writing *unit tests* that evaluate small, functionally discrete portions of code to spot specific implementation issues and quickly identify regressions during software evolution. Despite their usefulness, prior work has illustrated that once a project reaches a certain complexity, incorporating unit tests requires a substantial effort in traceability, decreasing the likelihood of unit test additions [150]. Further challenges exist for updating existing unit tests during software evolution and maintenance [150].

To help address these issues the software testing research community has responded with a wealth of research that aims to help developers by automatically generating tests [95, 213]. However, recent work has pointed to several limitations of these automation tools and questioned their ability to adequately meet the software testing needs of industrial developers [26, 239]. For example, it has been found that the *assert statements* generated by state-of-the-art approaches are often incomplete or lacking the necessary complexity to capture a designated fault. **The generation of meaningful assert statements is one**

of the key challenges in automatic test case generation. Assert statements provide crucial logic checks in a test case to ensure that the program is functioning properly and producing expected results. However, writing or generating effective assert statements is a complex problem that requires knowledge pertaining to the purpose of a particular unit test and the functionality of the related production code. Thus, an effective technique for the generation of assert statements requires predicting both the type and logical nature of the required check, using source and test code as contextual clues for prediction.

To help advance techniques that aid developers in writing or generating unit tests, we designed ATLAS, an approach for automatically generating syntactically and semantically correct unit test assert statements using Neural Machine Translation (NMT). ATLAS generates models trained on large-scale datasets of source code to accurately predict assert statements within test methods. We take advantage of the deep learning strategy of NMT, which has become an important tool for supporting software-related tasks such as bug-fixing [262, 63, 194, 121], code changes [260], code migration [208, 207], code summarization [163, 185, 291], pseudo-code generation [210], code deobfuscation [268, 133] and mutation analysis [265]. To the best of our knowledge, this is the first empirical step toward evaluating an NMT-based approach for the automatic generation of assert statements. Specifically, we embed a test method along with the context of its focal method [222] (*i.e.*, a declared method, within the production code, whose functionality is tested by a particular assert statement) and we "translate" this input into an appropriate assert statement. Since our model only requires the test method and the focal method, we are able to aid developers in automatic assert generation even if the project suffers from a lack of initial testing infrastructure. Note that our approach is not an alternative to automatic test case generation techniques [95, 213], but rather, a complementary technique that can be combined with them to improve their effectiveness. In other words, the automatic test case generation tools can be used to create the test method and our approach can help in defining a meaningful assert statement for it.

To train Atlas, we mined GitHub for every Java project making use of the JUnit assert class. In total we analyzed over 9k projects to extract 2,502,623 examples of developer-written assert statements being used within test methods. This data was used to give the NMT model the ability to generate assert statements that closely resemble those created by developers. Therefore, not only do we enable efficiency within the software testing phase but we also facilitate accuracy and naturalness by learning from manually written assert statements. After we extracted the pertinent test methods containing assert statements from the Java projects, we automatically identified the focal method for each assert statement based on the intuition from Qusef *et al.* [222]. We hypothesize that combining the test method and the focal method should provide the model with enough context to automatically generate meaningful asserts.

We then *quantitatively* and *qualitatively* evaluated our NMT model to validate its usefulness for developers. For our *quantitative* analysis, we compared the models generated assert statements with the oracle assert statements manually written by developers. We considered the model successful if it was able to predict an assert statement which is identical to the developer-written one. Our results indicate that Atlas is able to automatically generate asserts that are identical to the ones manually written by developers in 31.42% of cases (4,968 perfectly predicted assert statements) when only considering the top-1 predicted assert. When looking at the top-5 recommendations, this percentage rises to 49.69% (7,857).

For our *qualitative* analysis we analyzed “imperfect predictions” (*i.e.*, predictions which can differ semantically or syntactically as compared to the assert manually written by developers) to understand whether they could be considered an acceptable alternative to the original assert. We found this to be true in the 10% of cases we analyzed. Finally, we computed the edit distance between the imperfect predictions and original asserts in order to assess the effort required for a developer to adapt a recommended assert statement into one she would use. We show that slight changes to the “imperfect” asserts can easily convert them into a useful recommendation. To summarize, this paper provides the following contributions:

- We introduce Atlas, a NMT-based approach for automatically generating assert statements. We provide details pertaining to the mining, synthesizing, and pre-processing techniques to extract test methods from the wild, and to train and test Atlas;
- An empirical analysis of Atlas and its ability to use NMT to accurately generate a semantically and syntactically correct assert statement for a given test method;
- A quantitative evaluation of the model, and a detailed comparison between modeling raw and abstracted test methods;
- A publicly available replication package [9] containing the source code, model, tools and datasets discussed in this paper.

5.2 Approach

We provide an overview of the Atlas workflow for learning assert statements via NMT in Fig. 5.1. Our approach begins with the mining and extraction of test methods from Java projects. To do this, we mine GitHub projects that use the JUnit testing framework (Sec. 5.2.1). From those projects, we mine all test methods denoted with the `@Test` annotation as well as every declared method within the project (Sec. 5.2.2). We then filter this data, identify the appropriate focal method context, and generate pairs containing the contextual test method (*i.e.*, the test method augmented with information about the focal method it tests) and the relevant assert statement (Sec. 5.2.3 & Sec. 5.2.4). We refer to these pairs as Test-Assert Pairs (TAPs). Next, we generate two datasets of TAPs: (i) Raw Source Code, where TAPs are simply tokenized; (ii) Abstract Code, where we abstract the TAPs through our abstraction process (Sec. 5.2.5). Finally, we train two RNN encoder-decoder

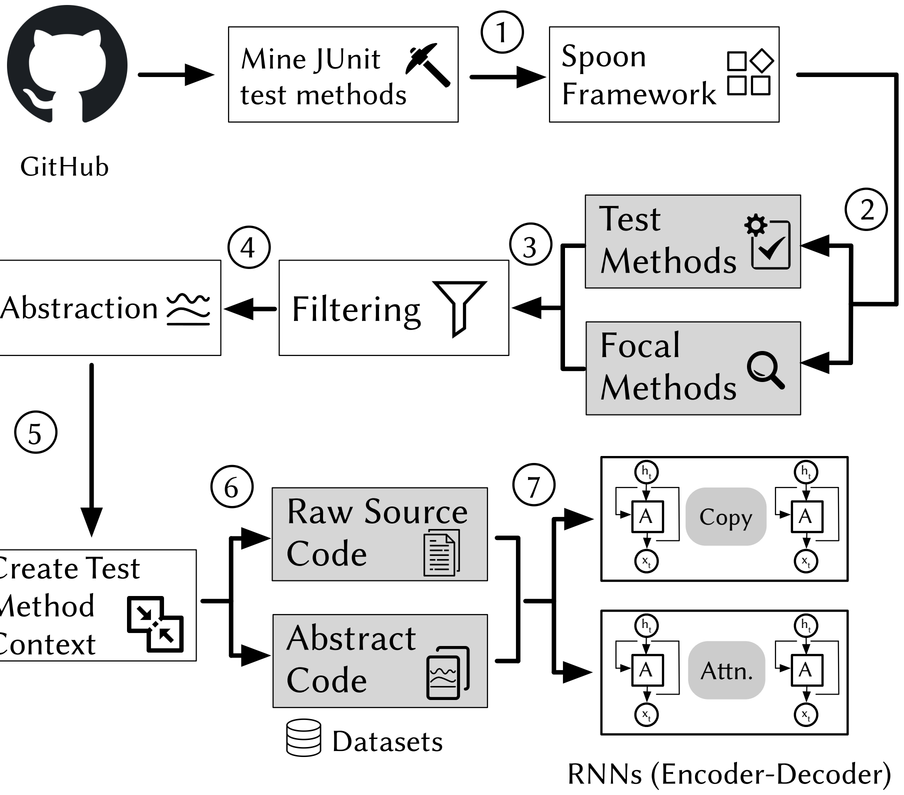


Figure 5.1: Overview of ATLAS Workflow

models; one using the copy mechanism trained on the Raw Source Code TAPs, and another using only the attention mechanism trained on the Abstract Code TAPs (Sec. 5.2.6).

5.2.1 GitHub Mining

Our main motivation toward studying Java projects that use the JUnit framework is *applicability*. As of August 2019 the TIOBE Programming Community Index indicated Java as the most popular programming language [3]. In addition, a study done by Oracle in 2018, found that JUnit was the most popular Java library [221]. Hence, curating a dataset of projects that use Java and JUnit lends to the potential for impact on real-world software development.

We identified GitHub projects using the JUnit testing framework. Since client projects can use JUnit by declaring a dependency through Apache Maven [4], we started by using the GitHub search API to identify all Java projects in GitHub having at least one `pom` file needed to declare dependencies toward Maven libraries. This resulted in the identification of 17,659 client projects, using 118,626 `pom` files and declaring ∼1.1M dependencies in total. We downloaded all the identified `pom` files and mined them to identify all client projects declaring a dependency toward JUnit version 4 and all its minor releases. These dependencies can be easily identified by looking in the `pom` file for artifacts having the `junit` *groupId*, `junit` *artifactId*, and a version starting with "4.". Using this process, we collected a total of 9,275 projects. Note that we decided to focus on JUnit v.4 since, in the mined dataset of `pom` files, we found that the majority of them had a dependency towards this version.

5.2.2 Method Extraction

After mining these projects from GitHub, we downloaded the source code and extracted the relevant test methods using Spoon [218]. This framework allows for source code analysis through the creation of a meta-model where the user can access program elements. To access relevant methods, we extract methods beginning with the `@Test` annotation, which is inherent to the JUnit framework. After extracting the test methods, we extract every method declared within the project, excluding methods from third party libraries. The extracted methods comprise a pool, from which we can determine the focal method of interest for a particular test method. The reason we only consider methods declared within the project is two-fold. First, most assert statements are evaluating the internal information of the project itself rather than information taken from third party libraries or external packages. Second, it would require a substantial effort to retrieve the method bodies and signatures from all the third party libraries and external packages. Since our goal is to learn appropriate assert statements for a given test method and its context, any test method without an assert statement has been discarded. Also, since this is the first work in the literature applying NMT to automatically generate assert statements, we decided to focus on test methods having a single assert statement and, thus, we exclude those implementing multiple asserts. While we acknowledge that this is a simplification of the problem we tackle, we preferred to first investigate the "potential" usefulness of NMT in a well-defined scenario in which, for example, it is safe to assume that the whole test method provides contextual information for the unique assert statement it contains. This assumption is not valid in the case of multiple asserts, and instead requires the development of techniques which are able to link parts of the test method body to the different asserts in order to understand the relevant context for each of them. This is part of our future work. Overall, we collected 188,154 test methods with a single assert statement.

5.2.3 Identifying Focal Methods

Our next task is to identify the focal method that the assert statement, within the test method, is testing. To accomplish this we implement a heuristic inspired by Qusef *et al.* [222]. We begin by extracting every method called within the test method. The list of invoked methods is then queried against the previously extracted list of methods defined inside the project, considering the complete method signature. We then assume that the last method call before the assert is the focal method of the assert statement [222]. In some instances, the assert statement contains the method call within its parameters. In these cases, we consider the method call within the assertion parameters as the focal method. It may appear problematic that we use the line we attempt to predict in order to extract the focal method (since, in theory, the line to generate should not exist). However, in a real usage scenario, we assume that the developer can provide the focal method to our model (*i.e.*, she knows the method she wants to test). Since identifying the focal method manually for a large number of assert statements is unreasonable, we used the heuristic previously described in place of manual identification of the focal method. We note this as a limitation but find it reasonable that either a developer or an automated test generation strategy would provide this information to our approach.

5.2.4 Filtering

In this work, we are attempting to generate semantically and syntactically correct assert statements from the test method and focal method context. Thus, we are creating a model which must learn relationships from source code. Modeling this type of data presents certain challenges, such as the open vocabulary problem and the "length of the input" problem [151]. Usually, these problems are tackled by limiting the vocabulary and the input length so that the model can adequately learn [252]. We employ similar solutions when training our model. We filter the data in three distinct ways: i) excluding test methods longer than 1,000 tokens; ii) filtering test methods that contain an assert statement which requires the synthesis of unknown tokens; and iii) removing duplicate examples within the dataset. Every filtering step helps to address NMT-related challenges and have been used in previous approaches that take advantage of this deep learning based strategy [151, 63, 262].

Our first filtering step is fairly straightforward: we remove all test methods that exceed 1,000 tokens. The second filtering step removes test methods in which the appropriate assert statement requires the synthesis of one or more unknown tokens. This means that the syntactically and semantically correct assert statement requires a token that cannot be found in the vocabulary or in the contextual method (*i.e.*, test method + focal method). Indeed, there is no way to synthesize these tokens when the model attempts to generate a prediction. We further explain this problem as well as our developed solution in section 5.2.4.1. Lastly, our third filtering step aims at removing duplicated instances, ensuring that every contextual method and assert statement pair in our dataset is unique.

5.2.4.1 Vocabulary

We have alluded to the open vocabulary problem which is an inherent limitation of NMT. This issue arises because developers are not limited in the number of unique tokens they can use. They are not limited to, for example, English vocabulary, but also create "new" tokens by combining existing words (*e.g.*, by using the CamelCase notation) or inventing new words to comprise identifiers (*e.g.*, V0_new). For this reason, the source code vocabulary frequently needs to be artificially truncated. To deal with this problem, we studied the tokens distribution in our dataset, observing that it follows Zipf's law, as also found by previous work analyzing open source software lexicons [220]. This means that the dataset's tokens follow a power-law like distribution with a long tail, and that many of the TAPs in our dataset can be successfully represented by only considering a small subset of its 695,433 unique tokens. Based on analyzing the data and on previous findings in the literature [62], we decided to limit our vocabulary to the 1,000 most frequent tokens. This allows us to successfully represent all tokens for 41.5% of the TAPs in our dataset (*i.e.*, 204,317 out of 491,649).

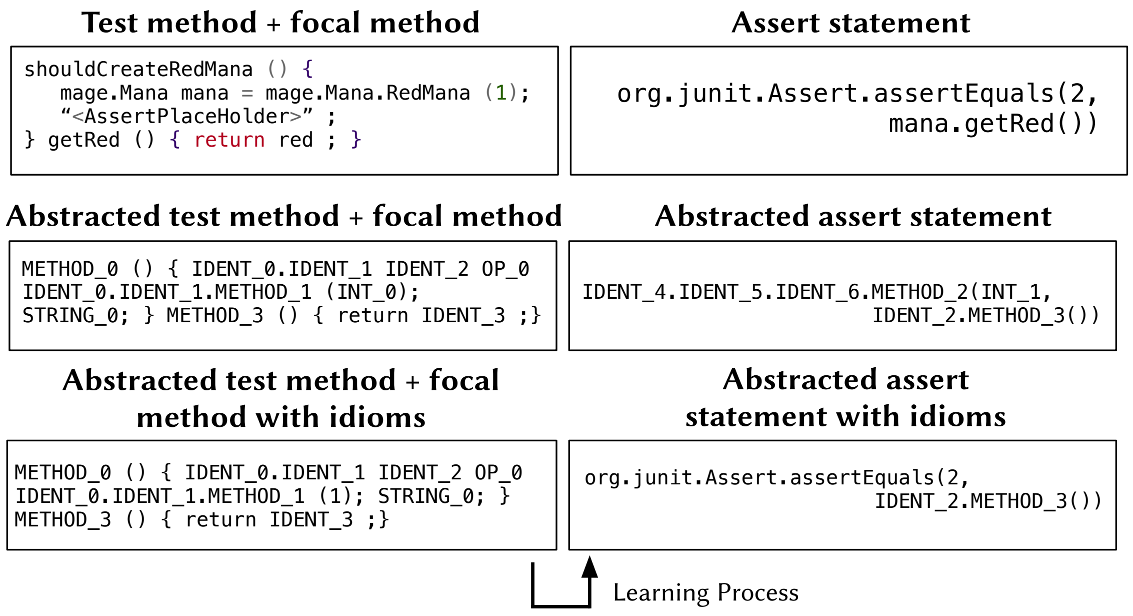
This filtering step aimed at removing instances for which the model would need to generate an unknown token. This can be formally defined as follows: Given a contextual method (*i.e.*, the test method tm and the corresponding focal method fm), we remove TAPs for which the anticipated assert to generate contains a token t such that $t \notin \{ V_{global} \cup V_{tm} \cup V_{fm} \}$, where V represents the vocabulary. We refer to $V_{tm} \cup V_{fm}$ as the contextual method. Each filtering step was due to concrete limitations of state-of-the-art NMT models. In numbers, starting from ~ 750 thousand test methods having a single assert, ~ 2.5 thousand tests are removed due to their excessive length, and ~ 280 thousand due to unknown tokens. Thus, $\sim 37\%$ of the single assert test methods are removed.

5.2.5 Test-Assert Pairs and Abstraction

The next step of our approach consists of preparing the data in such a manner that it can be provided as input to Atlas's NMT model. This process involves: i) the concatenation of the focal method to the test method in order to create the Test-Assert Pair (TAP); ii) the tokenization of the TAP; and iii) the abstraction of the TAP. Starting from the first point, it is important to note that not every test method will inherently contain a focal method being tested. However, for the test methods that do posses a focal method, we append its signature and body to the end of the test method. While comments and whitespaces are ignored, we do not remove any other token from either the test or focal method. We then proceed to remove the entire assert statement from the test method, replacing it with the unique token `"AssertPlaceHolder"`. Therefore, the first part of a TAP consists of the concatenated test and focal method, and the second part consists of the assert statement to generate.

We generate two separate datasets of TAPs. The first uses the raw source code to represent the test and the focal method. The second dataset consists of abstracted TAPs, in which the source code tokens are abstracted to limit the vocabulary and to increase the possibility of observing recurring patterns in the data. As previous studies have shown [262], this can lead to an increase in performance without losing the ability to automatically map back the abstracted tokens to the raw source code.

Our abstraction process tokenizes the input, determines the type for each individual token, replaces the raw token with its abstraction and finally creates a map from the abstracted tokens back to the raw ones. Each TAP is abstracted in isolation and has no affect on the way other TAPs are abstracted. We start by using the `javalang` tool to tokenize the input, which allows us to analyze the type of token. This tool transforms the Java method into a stream of tokens, which is then parsed by the same tool to determine their type (*e.g.*, whether a token represents an identifier, a method declaration, *etc.*) [257]. We use these types in order to create an expressive representation that maintains the structure of the code. In total, there are 13 different types of tokens that we use as part of our abstraction process (complete list in our replication package [9]). When we encounter one of these types, we replace the raw source code token with an abstraction term that indicates the type and number of occurrences of that type of token up to that moment. In other words, we use a numerical value to differentiate tokens of the same type. For example, suppose we encounter a token that is determined to be a method call. This token is replaced with the term METHOD_0, since this is the first type of that token we have encountered. The next time we encounter a new method call in the same stream of tokens, it would be assigned the term METHOD_1. In the event where the same token appears multiple times within the TAP, it is given the same abstraction term and numerical value. This means that if the same method is invoked twice in the test or in the focal method and it is represented with the term METHOD_0, the abstraction will contain "METHOD_0" twice. Since all TAPs are abstracted in isolation, we can reuse these terms when abstracting a new TAP, thus limiting the vocabulary size for our NMT model.


Test method + focal method
Assert statement
Abstracted test method + focal method
Abstracted assert statement
Abstracted test method + focal method with idioms
Abstracted assert statement with idioms
Learning Process

```
shouldCreateRedMana () {
   mage.Mana mana = mage.Mana.RedMana (1);
   "<AssertPlaceHolder>" ;
} getRed () { return red ; }
```

```
org.junit.Assert.assertEquals(2,
                    mana.getRed())
```

```
METHOD_0 () { IDENT_0.IDENT_1 IDENT_2 OP_0
IDENT_0.IDENT_1.METHOD_1 (INT_0);
STRING_0; } METHOD_3 () { return IDENT_3 ;}
```

```
IDENT_4.IDENT_5.IDENT_6.METHOD_2(INT_1,
                        IDENT_2.METHOD_3())
```

```
METHOD_0 () { IDENT_0.IDENT_1 IDENT_2 OP_0
IDENT_0.IDENT_1.METHOD_1 (1); STRING_0; }
METHOD_3 () { return IDENT_3 ;}
```

```
org.junit.Assert.assertEquals(2,
                        IDENT_2.METHOD_3())
```

Figure 5.2: Overview of the abstraction process

In addition to the abstraction terms that replace the raw source code tokens, we take advantage of idioms in order to create an abstraction that captures more semantic information. The inclusion of idioms can also help contextualize surrounding tokens, since some abstracted tokens may be more likely to appear around a particular idiom. In addition, idioms help to prevent the exclusion of a TAP due to the synthesis of an unknown token. For instance, consider the example of abstraction shown in figure 5.2. In the middle of the figure it is possible to see that INT_1, IDENT_4, IDENT_5 and IDENT_6 only appear in the abstracted assert statement, but do not appear in the abstracted test and focal method. If we only relied on the abstraction, we would be unable to resolve these tokens that are unique to the predicted assert statement. Therefore, we keep the raw value of common idioms (*i.e.*, the top 1,000 tokens in our dataset in terms of frequency) in our abstracted representation, as shown in the bottom part of figure 5.2.

Overall, the vocabulary of the Abstract TAPs comprises 1,000 idioms plus ∼100 typified IDs, while the vocabulary of the Raw Source Code TAPs contains 1,000 tokens.

5.2.6 Sequence to Sequence Learning

Our approach applies sequence-to-sequence learning through a recurrent neural network (RNN) encoder-decoder model to automatically learn assert statements within test methods. This model is inspired by Chen *et al.* [63], which attempts to predict a single line of code that has a predetermined place holder within the method. The goal of this deep learning strategy is to learn a conditional distribution of a variable length sequence conditioned on a completely separate variable length sequence $P(y_1, y_2, \ldots, y_m | x_1, x_2, \ldots, x_n)$. Where n and m may differ. During training, the model's encoder is fed the tokenized input sequence of the test method plus the context of the focal method as a single stream of tokens $(x_1, \ldots, x_n)$. The assert statement, which is our target output sequence, has been removed and replaced with a specialized token. The decoder attempts to accurately predict the assert $(y_1, \ldots, y_m)$ by minimizing the error between the decoder's generated assert and the oracle assert statement. This is accomplished by using the negative log likelihood of the target tokens using stochastic gradient descent [144]. An overview of an RNN encoder-decoder can be seen in figure 5.1.

5.2.7 Encoder

The encoder is a single layer bi-directional RNN, which is comprised of two distinct LSTM RNNs. This bidirectionality allows the encoder to consider both the tokens that come before and the tokens that come after as context for the token of interest. The encoder takes a variable length sequence of source code tokens $X = (x_1, x_2, \ldots, x_n)$ as input. From these tokens, the encoder produces a sequence of hidden states $(h_1, h_2, \ldots, h_n)$ generated from LSTM RNN cells. These cells perform a series of operations to propagate relevant

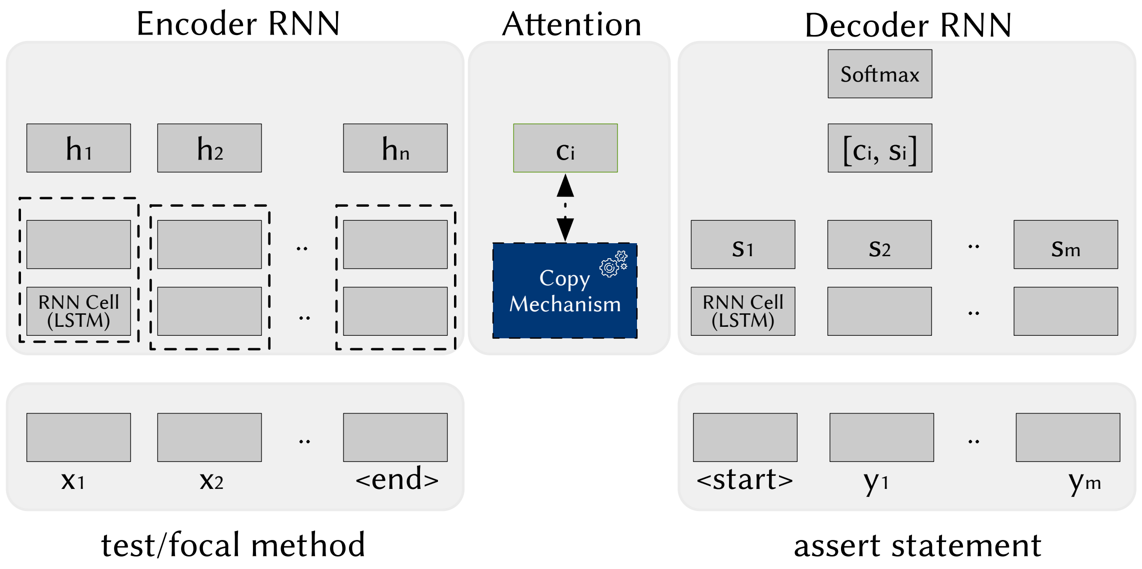


Figure 5.3: The RNN bidirectional encoder decoder model

information, including previous hidden states, to the next cell. Due to the bidirectionality of the encoder, there exists a sequence of hidden states when considering the token sequence from left to right $\overrightarrow{h_i} = f(x_i, h_{i-1})$ and right to left $\overleftarrow{h_i} = f(x_{i_0}, h_{i_0+1})$. For our model, each hidden state can be formally described as the non-linear activation of the current sequence token and the previously synthesized hidden state. Once the hidden state for each directional pass is found, they are concatenated to derive the finalized hidden state h_i. The sequence of resulting hidden states is propagated through the model as the context vector. The encoder also applies the regularization technique of dropout at a rate of 0.2.

5.2.8 Attention Mechanism

The context vector $\mathcal{C}$, commonly referred to as an attention mechanism, is computed as a weighted average of the hidden states from the encoder $\mathcal{C} = \sum_{i=1}^{n} \alpha_i h_i$. In this equation α represents a vector of weights used to denote the influence of different parts of the input sequence. In this manner, the model can pay greater *attention* to particular tokens of the input sequence when attempting to predict the output token y_i. The weights which influence the attention mechanism are trained over time to help identify the patterns of contribution from different input tokens.

5.2.9 Decoder and Copy Mechanism

The decoder is a double layer LSTM RNN that learns to take a fixed length context vector and translates it into a variable length sequence of output tokens. Given a previous hidden state $h_{\hat{i}-1}$, the previous predicted token $y_{\hat{i}-1}$ and the context vector $\mathcal{C}$ the decoder generates a new hidden state that can be used to predict the next output token. As done for the encoder, we apply regularization to the decoder by using dropout at a rate of 0.20, and the Adam optimizer for learning with a starting learning rate of 0.0001:

$$h_{\hat{i}} = f(h_{\hat{i}-1}, y_{\hat{i}-1}, \mathcal{C})$$

The decoder generates a new hidden state each time it predicts the next token in the sequence until a special stop token is reached. The hidden states are generated using the equation above. However, these hidden states are also used by the copy mechanism to help predict the appropriate output token. In particular, the copy mechanism works to calculate two separate probabilities. The first, is the probability that the next predicted token in the output sequence should be taken from the vocabulary. The second, is the probability that the next predicted token in the output sequence should be *copied* from the input sequence. With the ability to consider copying tokens from the input sequence to the output sequence, we can artificially extend the vocabulary of the encoder-decorder RNN model for the raw dataset. Each sequence inferred from the model now has the ability to consider any predetermined vocabulary token, in addition to any token from the input sequence. The downside is that the copy mechanism is a trainable extension of the model, that learns which input tokens should be copied over. The benefit is that the model can better deal with rare tokens that would otherwise not appear in the vocabulary.

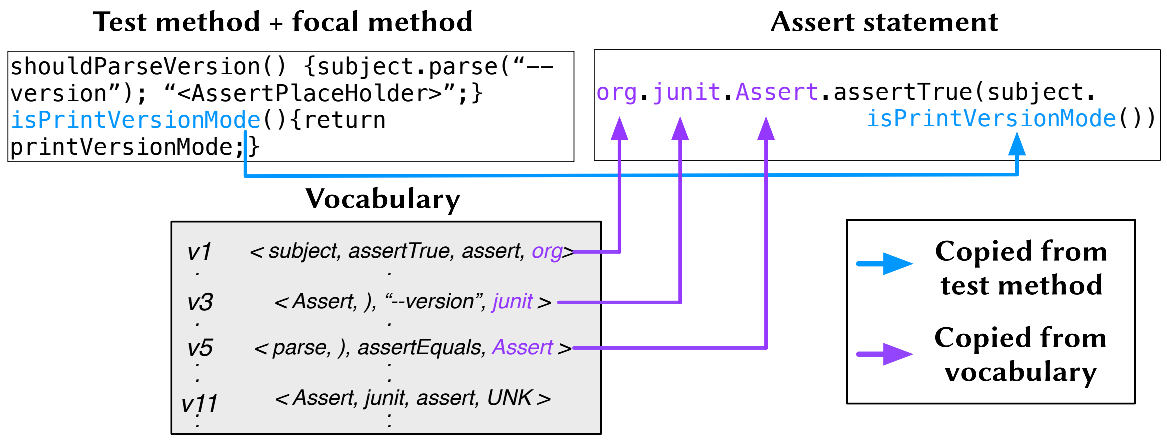

Test method + focal method
shouldParseVersion() {subject.parse("--version"); "<AssertPlaceHolder>";} isPrintVersionMode(){return printVersionMode;}
Assert statement
org.junit.Assert.assertTrue(subject.isPrintVersionMode())
Vocabulary
v1 < subject, assertTrue, assert, org>
v3 < Assert,), "--version", junit >
v5 < parse,), assertEquals, Assert >
v11 < Assert, junit, assert, UNK >
Copied from test method
Copied from vocabulary

Figure 5.4: Copy Mechanism Example

To further demonstrate how the copy mechanism works, consider figure 5.4. In this example, we see the vocabulary tokens predicted for each time step v_t. For the purpose of this example we do not consider the translation of the separator tokens. However, in a real scenario, our model would predict these tokens in identical fashion. The first predicted token for the output sequence is the `org` token, which is copied from the vocabulary. This process is repeated for the `junit` token, the `Assert` token and would continue for all further tokens but the last one (*i.e.*, `isPrintVersionMode`). The last token is not found anywhere in the vocabulary. At time step v_{11} the model recommends the `UNK` token to indicate that the highest probability token does not exist within the defined vocabulary. In this case, the copy mechanism is used to determine which input token has the highest probability to be the predicted token. In the case shown in the example, the context appended from the focal method contains this token and it is copied to the output sequence. Without the copy mechanism, there would have been no way to resolve this example and it would have been discarded. It is important to note that the copy mechanism is trained along with the network and requires no effort on the end users.

Also, note that the copy mechanism is only applied to the raw dataset of source code tokens. The reasoning is that after abstracting the source code, all tokens are available within the vocabulary. Therefore, the probability that a predicted token would be outside of the vocabulary, and within the input sequence, is 0%, rendering the copy mechanism useless. Rather, our abstraction process serves as a pseudo copy mechanism of typified IDs. Consider the previous example in figure 5.2 where `isPrintVersionMode` was not found anywhere in the vocabulary. In our abstraction process, this token is abstracted into METHOD_1 and since METHOD_1 is a term contained within our vocabulary the model has no problem in predicting this token in the output assert statement. Then, when we map the abstracted assert statement back to raw source code, we replace METHOD_1 with the token `isPrintVersionMode` without relying on the copy mechanism.

5.3 Experimental Design

The *goal* of our study is to determine if ATLAS can generate meaningful, syntactically and semantically correct assert statements for a given test method and focal method context. Additionally, it is important for our approach to be lightweight in order to require as little overhead as possible if, for example, it is combined with approaches for the automatic generation of test cases.

The *context* of our study is represented by a dataset of 158,096 TAPs for the abstracted dataset and 188,154 TAPs for the raw dataset. These datasets are further broken down into 126,477 TAPs for training, 15,809 TAPs for validation, and 15,810 TAPs for testing in the abstract dataset. Likewise, we had 150,523 TAPs for training, 18,816 TAPs for validation and 18,815 TAPs for testing in our raw dataset. The differences in number of examples between the two datasets is due exclusively to the removal of duplicates. Since the abstracted model reuses typified IDs, there is a greater chance for the duplication of TAPs within the abstracted dataset. Our evaluation aims at answering the research questions described in the following paragraphs.

RQ$_1$: Is ATLAS able to generate assert statements resembling those manually written by developers? We assess whether ATLAS is a viable solution for generating semantically and syntactically correct assert statements. Therefore, we perform experiments on real-world test methods and determine if our model can predict the appropriate assert statement. We use both datasets (*i.e.*, raw source code and abstracted code) to train the encoder-decoder recurrent neural network model. During training, we use our validation set to determine the optimal parameterization of the NMT model (complete list of used parameters available in [9]). We then evaluate the trained model on the test set, which contains examples previously unseen in both the training set and the validation set.

We begin training our model on TAPs, feeding the model the test method and associated focal method context. We train our model until the evaluation on the validation set shows that the models parameterization has reached a (near-)optimal state (*i.e.*, the model is no longer improving the calculated loss for data points outside the training set). This is a common practice to prevent the effects of overfitting to the training data. Our training phase results in two separate models, one for predicting raw source code assert statements and the other for predicting abstracted asserts. Remember that when working with raw source code, we also implement the copy mechanism, which is not used for abstracted code. In total, the abstract model trained for 34 hours while the raw model trained for 38 hours. The difference in training time can be attributed to the use and training of the copy mechanism in conjunction with the lack of abstraction.

Once the model is trained, inference is performed using beam search [226]. The main intuition behind beam search decoding is that rather than predicting at each time step the token with the best probability, the decoding process keeps track of k hypotheses (with k being the beam size). Thus, for a given input (*i.e.*, test method + focal method), the model will produce k examples of assert statements. We experiment with beam sizes going from $k = 1$ to $k = 50$ at steps of 5.

Given the assert statements predicted by our approach, we consider a prediction as correct if it is identical to the one manually written by developers for the test method provided as input. We refer to these asserts as "perfect predictions". When experimenting with different beam sizes, we check whether a perfect prediction exists within the k generated solutions. We report the raw counts and percentages associated with the number of perfect predictions.

Note that, while the perfect predictions certainly represent cases of success for our approach, this does not imply that the "imperfect predictions" all represent failure cases (*i.e.*, the generated asserts are not meaningful). Indeed, for the same test/focal method, different assert statements could represent a valid solution. Therefore, we sample 100 "imperfect predictions" and manually analyze them to understand whether, while different from the original assert statements written by the developer, they still represent a meaningful prediction for the given test method. In particular, we split the "imperfect predictions" into four sets based on their BLEU-4 score [217] value. The BLEU score is a well-known metric for assessing the quality of text automatically translated from one language to another [217]. In our case, the two "languages" are represented by i) the test method and the focal method, and ii) the generated assert statement. We use the BLEU-4 variant, meaning that the BLEU score is computed by considering the 4-grams in the generated text, as previously done in other software-related tasks in the literature [109, 135]. The BLEU score ranges between 0% and 100%, with 100% indicating, in our case, that the generated assert is identical to the reference one (*i.e.*, the one manually written by developers). We use the BLEU score ranges 0-24, 25-49, 50-74 and 75-99 to split the imperfect predictions. Then, we randomly selected 25 instances from each set and the first author manually evaluated them to determine if the generated assert statement is meaningful in the context of the related test/focal methods. To avoid subjectiveness issues, the 100 instances were also randomly assigned to four other authors (25 each) who acted as second evaluator for each instance. Conflicts (*i.e.*, cases in which one of the two evaluators classified the assert statement as meaningful while the other did not) arose in a single case that was solved through an open discussion. We report the number of meaningful assert statements we found in the manually analyzed sample as empirical evidence that imperfect assert statements could still be useful in certain cases. Note that, while there might be authors' bias in assessing the meaningfulness of the "imperfect" assert statements (*i.e.*, authors may tend to be too positive in evaluating the meaningfulness of the asserts), we make our evaluation publicly available in the replication package [9], to allow the reader to analyze the performed classification.

RQ$_2$: Which types of assert statements is ATLAS capable of generating? After obtaining the perfect predictions from RQ$_1$, we analyze the types of assert statements our model can generate. In particular, we analyze the taxonomy of assert statements generated by our approach in the context of perfect predictions to determine the types of assert statements the model is able to correctly predict. We then report the raw counts and the percentages for each type of assert statement the model can perfectly predict.

Note for this evaluation, we only report results for $k = 1$, since we found that already with this beam size the model was able to generate all types of assert statements in the used dataset.

RQ$_3$: Does the abstraction process aid in the prediction of meaningful assert statements? Remember that while our abstraction model generates abstracted asserts, we can map them back to the raw source code at no cost to the developer. Thus, we can check whether the generated assert is a perfect prediction, as we do for the raw source code. Besides comparing the performance of the two models, we also analyze whether they produce complementary results by computing the following overlap metrics:

$$pp_{R \cap A} = \frac{|pp_R \cap pp_A|}{|pp_R \cup pp_A|} \quad pp_{R \setminus A} = \frac{|pp_R \setminus pp_A|}{|pp_R \cup pp_A|} \quad pp_{A \setminus R} = \frac{|pp_A \setminus pp_R|}{|pp_R \cup pp_A|}$$

The formulas above use the following metrics: pp_R (pp_A) represents the set of perfect predictions generated using the raw (abstracted) source code dataset; $pp_{R \cap A}$ measures the overlap between the set of perfect predictions generated by using the two datasets; $pp_{R \setminus A}$ measures the perfect predictions generated on the raw source code but not when using abstraction (*vice versa* for $pp_{A \setminus R}$).

RQ$_4$: What is the effect of using the copy mechanism in our model? With the addition of the copy mechanism we can perfectly predict assert statements which contain tokens only found in the input sequence and not in the vocabulary. Here we want to quantify the effect of the copy mechanism and see how many assert predictions we would be capable of producing without its usage. Therefore, we analyze the perfect prediction set from the raw source code model. If the perfect prediction contains a token not found in the vocabulary, then we know that its generation was possible due to the usage of the copy mechanism. We report the raw counts and percentages of the number of assert statements that were resolved thanks to the use of the copy mechanism.

RQ_5: Does ATLAS outperform a baseline, frequency-based approach? As output of RQ_2 we defined a taxonomy of assert statements that our approach is able to correctly generate in the perfect predictions. We noted that there are eight types of assert statements that the model is capable of generating. However, the model also predicts the variables and method calls contained within the assert statement. Therefore, we want to determine if the most frequently used assert statements found within our dataset (*e.g.*, *assert(true)*), could be used to create a frequency-based approach that outperforms our learning-based approach.

We analyze the duplicated assert statements generated in our perfect prediction set. This means that the model generated the same (correct) assert statement for different test methods provided as input. It is important to highlight here that while the same assert statement can be used in different test methods, this does not imply that we have duplications in our datasets. Indeed, while the same assert statement can be used in different TAPs, the test/focal methods are guaranteed to be unique.

The developed frequency-based approach takes the most commonly found assert statements and applies them as a solution for a given test method. In particular, we take the top k most frequent assert statements and test if any of them represents a viable solution for each test method in the test set. In this evaluation we set the same k for both approaches (*i.e.*, the frequency-based and the learning-based ones). For example, assuming $k = 5$, this means that for the frequency-based approach we use the five most frequent assert statements as predictions, while for the learning-based approach we set beam size to five. We report the raw counts of the frequency-based approach as compared to our NMT-based approach. We compare the two approaches at $k = 1, 5, 10$.

RQ$_6$: What is the inference time of the model? Our last research question speaks to the applicability and ease of use of our model. When developing test cases within a software project, it is unreasonable to expect the developer to spend a considerable amount of time to set up and run an inference of the model. Therefore, we performed a timing analysis to assess the time needed to generate assert statements for a variety of beam sizes. In particular, we used $k = 1$ to $k = 50$ with an increment of 5 to test the trade off between the timing of the model's inference and the increased prediction results of the model.

Table 5.1: Prediction Classification

Beam Size	Raw Model		Abstract Model	
	Peferct Prediction Percentage	Perfect Prediction Counts	Perfect Prediction Percentage	Perfect Prediction Counts
1	17.66%	3323	31.42%	4968
5	23.33%	4390	49.69%	7857
10	24.73%	4654	55.73%	8812
15	25.53%	4805	58.76%	9291
20	25.88%	4871	60.43%	9554
25	26.19%	4929	61.75%	9764
30	26.43%	4973	62.73%	9918
35	26.63%	5012	63.68%	10068
40	26.81%	5045	64.38%	10179
45	26.91%	5064	64.81%	10247
50	27.01%	5083	65.31%	10327

We record the results in number of seconds and map the increased performance against the increased time. We do not consider the time it may take for a developer to look at all resulting predictions of assert statements for different beam sizes. Note that we do not consider the training time since this is a one time cost that does not affect the usability of the approach.

5.4 Results

RQ_1 & RQ_3 & RQ_4: Ability to generate meaningful assert statements, comparison between raw and abstracted dataset, and usefulness of copy mechanism. Table 5.1 shows the perfect prediction rate for both the "raw model" and the "abstract model" for beam sizes 1 and 5-50 at increments of 5 (RQ_1 & RQ_3). As expected, the perfect prediction rate increases using larger beam sizes, with a plateau reached at beam size 20 (*i.e.*, only minor increases in performance are observed for larger beam sizes).

When using beam size equal to 1 for the model trained/tested with the raw dataset, our approach generates 17.66% perfect predictions, resulting in over 3.3k correctly generated assert statements. The average BLEU score for the asserts predicted when only considering the top recommendation (*i.e.*, beam size = 1) is 61.85. We also found that the copy mechanism helps the raw model in generating perfect predictions (RQ_4). Indeed, we determined how many perfect predictions require the use of the copy mechanism, finding that, when using beam size equals 1, our approach is able to perfectly predict 3323 examples by using the copy mechanism, and 2439 when only relying on the vocabulary. This means that the copy mechanism is responsible for resolving 884 perfect predictions, which constitutes 4.69% of the perfect prediction rate.

For the abstract model, the percentage of perfect predictions goes up to 31.42% ($\sim$5k correctly generated asserts). Here the average BLEU score is 68.01. Note that we mention the BLEU scores for completeness, but do not perform a full evaluation using this metric.

Of particular note is the substantial bump in perfect predictions that we obtain as result of increasing the beam size to five for both models, especially for the abstracted one. For the latter, in 49.69% of test/focal methods provided as input, one of the generated asserts is identical to the one manually written by the developer (for a total of 7.9k perfectly predicted asserts). This indicates the potential of our approach in an automatic "code completion" scenario, in which the developer could easily pick one of the five recommended asserts.

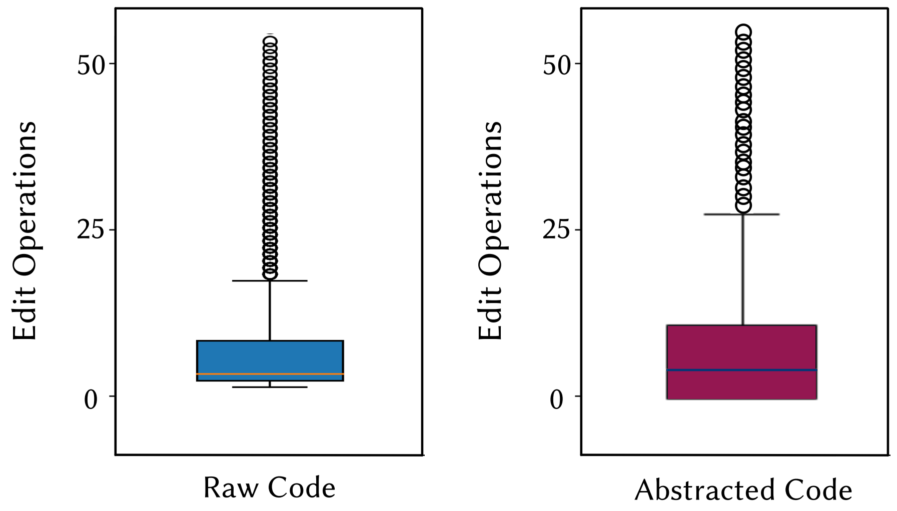
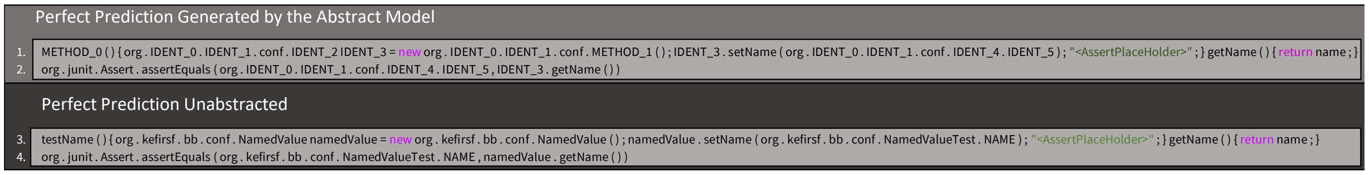


Figure 5.5: Edit Distance between Imperfect Predictions and Ground Truth (Truncated tail of higher edit distances)

As mentioned in section 5.3, we also analyze the complementarity of the perfect predictions generated by the raw and the abstracted models when using beam size equals 1. In terms of overlap (*i.e.*, $pp_{R \cap A}$) we found that only 117 examples were captured and perfectly predicted by both models. Also, 3206 (39.2%) of the perfect predictions are only generated by the raw model ($pp_{R \setminus A}$), while 4851 (59.3%) can only be obtained by using the abstracted model ($pp_{A \setminus R}$). This shows a surprising complementarity of the two models, that are able to learn different patterns in the data. The combination of these two different representations, for example by using a model with two encoders (one per representation) and a single decoder, may be an interesting direction for future work.

However, the overall higher performance in both BLEU score and perfect prediction rate of the abstract model shows that it is better able to *translate* a meaningful assert statement from the test and focal method context. This is likely due to the different way in which the two models see the out-of-vocabulary tokens. In the raw dataset they are all represented as unknown tokens, while in the abstracted dataset they are represented with the associated type (*e.g.*, METHOD, VAR, INT, *etc.*). When dealing with out-of-vocabulary tokens, this helps the abstracted model in learning patterns such as: a "METHOD" token is likely to


Perfect Prediction Generated by the Abstract Model

```
1. METHOD_0 ( ) { org . IDENT_0 . IDENT_1 . conf . IDENT_2 IDENT_3 = new org . IDENT_0 . IDENT_1 . conf . METHOD_1 ( ) ; IDENT_3 . setName ( org . IDENT_0 . IDENT_1 . conf . IDENT_4 . IDENT_5 ) ; "<AssertPlaceHolder>" ; } getName ( ) { return name ; }
2. org . junit . Assert . assertEquals ( org . IDENT_0 . IDENT_1 . conf . IDENT_4 . IDENT_5 , IDENT_3 . getName ( ) )
```

Perfect Prediction Unabstracted

```
3. testName ( ) { org . kefirsf . bb . conf . NamedValue namedValue = new org . kefirsf . bb . conf . NamedValue ( ) ; namedValue . setName ( org . kefirsf . bb . conf . NamedValueTest . NAME ) ; "<AssertPlaceHolder>" ; } getName ( ) { return name ; }
4. org . junit . Assert . assertEquals ( org . kefirsf . bb . conf . NamedValueTest . NAME , namedValue . getName ( ) )
```

Figure 5.6: Example of a Perfect Prediction Made by the Abstract Model, and its unabstracted mapping to actual code

follow a "VAR" and a "." token. For the raw model, this only results in observing "UNK . UNK", hindering its ability to learn meaningful patterns in these cases. Due to the better performance ensured by the abstract model, the subsequent analyses are performed using this model.

Here, we discuss an interesting example generated by our abstracted model. Figure 5.6 presents the abstracted contextual method in line 1 and the abstracted assert statement in line 2. Lines 3 and 4 are the result of the unabstraction process, where we map back the abstracted tokens into raw source code. In this example, the test method is creating a new object *namedValue* and setting the *NAME* attribute of this object. The test method contains the method call *getName*, which our heuristic identifies as the focal method and is appended to the end of the test method. The model then generates an assert statement that compares the *NAME* attribute of the new object with the results from the *getName* method call. Indeed, the model learns the relationship between the test and focal method in order to generate a meaningful assert statement which appropriate tests the method's logic. This assert statement is a nontrivial example of the model determining what type of assert statement to generate, as well as the appropriate parameters the assert statement should contain.

Concerning the manual inspection of the 100 "imperfect predictions", we found that 10% of them can represent a valuable assert statement for the provided contextual method, despite them being different from the asserts manually written by developers. Note that, while a 10% might look like a "negative" result, it is important to understand that these results are in addition to the already perfectly predicted assert statements. The full evaluation of the 100 cases is available in our replication package [9]. Here we discuss a representative example, for a provided test/focal method, where our approach generated the assert statement `assertSame(input,result)` while the assert manually written by the developer was `assertEquals("Blablabla",result)`. The `input` object is present in the test method, and contains indeed the value "Blablabla". Basically, our model generated an "easier to maintain" assert, since changes to the value assigned to the input objects do not require changes to the assert statement. Concerning the difference between `assertSame` and `assertEquals`, the former compares two objects using the `==` operator, while the latter uses the `equals` method that, if not overridden, does also perform the comparison using the `==` operator.

Although we have shown that our model can produce meaningful results outside of the perfect predictions, we wanted to understand "how far" are the imperfect predictions from the manually written assert statements. Therefore, for each imperfect prediction generated by the abstract model with $k = 1$, we computed the number of tokens one needs to change, add, or delete to convert it into the manually written assert. We found that by only changing one token it is possible to convert 23.62% of the imperfect assert statements (*i.e.*, 3660 instances) into perfect predictions. Also, the median of 3 indicates that in half of the cases, changing only three tokens would be sufficient. Note that the average number of tokens in the generated assert statements is 17.1. Figure 5.5 shows the distribution of edit changes needed for the imperfect predictions to become perfect predictions.

Summary for RQ$_1$, RQ$_3$, & RQ$_4$. ATLAS's abstracted model is able to perfectly predict assert statements (according to a developer written ground truth) 31.42% and 49.69% of the time for top-1 and top-5 predictions respectively. Conversely, the model operating on raw source code with the copy mechanism achieved a perfect prediction rate of 17.66% and 23.33% for top-1 and top-5 predictions respectively. This indicates the abstracted model performs better overall. However, we also found that models had a relatively high degree of orthogonality, with 39.2% of all perfect predictions generated by the raw model, and 59.3% generated by the abstracted model, illustrating that the copy-mechanism allowed for the prediction of a unique set of assert statements.

RQ$_2$: Which types of assert statements is ATLAS capable of generating? We also analyzed the types of JUnit4 assert statements that were perfectly predicted. Here we use the 4,968 perfect predictions generated using the abstract model and beam size equals one. Table 5.2 shows that our approach was able to correctly predict eight different types of assert statements, with `assertEquals` and `assertTrue` being the most commonly predicted type of assert statement. Note that some JUnit4 assert types (*e.g.*, `assertNotSame` and `assertFail`) are not generated by our model because they were not present in our dataset.

Table 5.2: Types of Predicted Assert Statements

Assert Type	Count	Total in Dataset
assertEquals	2518	7923
assertTrue	973	2817
assertNotNull	606	1175
assertThat	250	1449
assertNull	238	802
assertFalse	232	1017
assertArrayEquals	102	311
assertSame	49	314

As it can be seen in table 5.2, the distribution of assert statements we are able to predict is similar to the one of the assert statements in the entire dataset. This result mitigates the possible threat that our approach is only successful in generating a specific type of assert

Table 5.3: Learning Based vs. Frequency Based

Beam Size	**Number of Perfect Predictions**	
	Frequency Model	**Abstract Model**
1	455	4968
5	970	7857
10	1299	8812

statement. Indeed, as shown in table 5.2, the lack of a uniform distribution of data points in the dataset from which Atlas is learning seems to be the main reason for the skewed distribution of correctly predicted asserts. The main exception to this trend is represented by the `assertThat` statements. We hypothesize that `assertThat` statements are more difficult to predict due to the nature of the assert itself. These types of asserts compare a value with a matcher statement, which can be quite complex, since matcher statements can be negated, combined, or customized. Despite the complexities of *assertThat* statements, the model is still able to perfectly predict 17.2% of the ones seen in the testing set.

Summary for RQ$_2$. We found that `assertEquals` was the most common type of assert generated, matching the distribution we were learning from. Our analysis showed that Atlas is capable of learning every type of assert statement found in developer written test cases.

RQ$_5$: Does ATLAS outperform a baseline, frequency-based approach? In this research question we explore whether our abstract model outperforms a baseline, frequency-based approach (see section 5.3). Table 5.3 shows the results of this comparison. We note that our DL-based approach outperforms the frequency based approach at each experimented beam size. The difference in terms of performance is substantial, resulting in 6.8 (beam size=10) to 10.9 (beam size=1) times more perfect predictions generated by our approach. For example, when only considering the top candidate assert statement for both techniques, Atlas correctly predicts 4968 assert statements, as compared to the 455 of the frequency-based model. The achieved results indicate that Atlas is in fact learning relationships based on hierarchical features and not being overwhelmed by repetitive assert statements. We also want to note that Atlas encompassed a majority of the assert statements found by the frequency based baseline. Therefore, we do not combine a frequency based approach with ATLAS and believe our implementation to be superior on its own.

Summary for RQ$_5$. ATLAS is able to significantly outperform a frequency-based baseline prediction technique by a factor of 10.

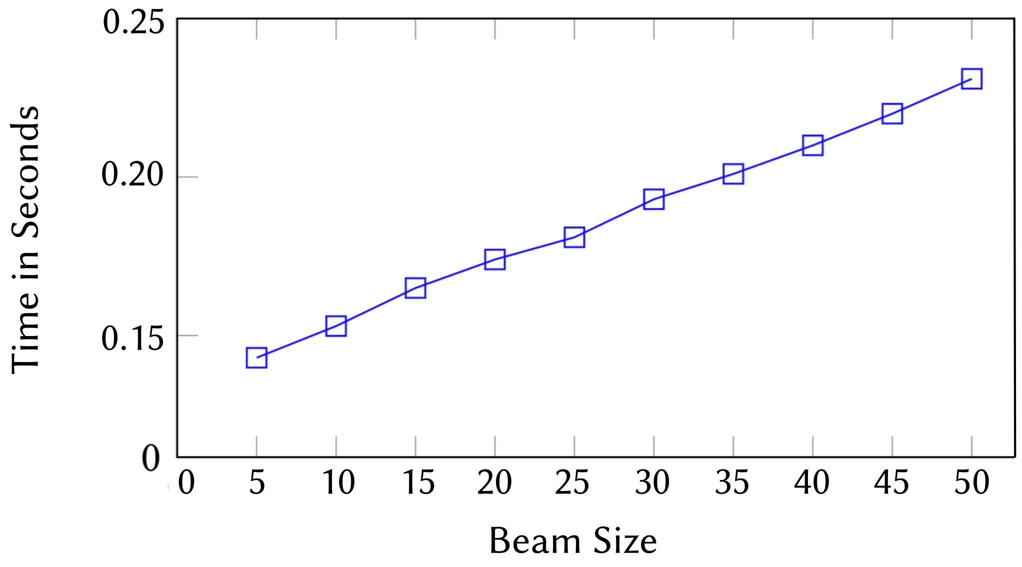


Figure 5.7: Seconds per Assert Generation

RQ$_6$: What is the inference time of the model? Our last research question assesses the time required by our approach to generate meaningful assert statements. Given the previously discussed performance of the experimented models, we computed the generation time on the abstract model. It is important to note the reported time does not include the code abstraction nor the mapping of the prediction back to source code, although these operations are typically more efficient than inference. Figure 5.7 shows the increase in time based on the number of solutions the model generates. These timings concern the generation of assert statements for 15,810 test/focal methods provided as input. The reported time is in seconds per provided input, and includes the generation of all assert statements in the given beam size. For example, assuming a beam size of 5, it would take around 0.14 seconds to generate all 5 predictions for a particular test/focal method. These results were computed using a single consumer-grade NVIDIA RTX GPU.

Summary for RQ$_6$. We found that with a beam size of 5, we can generate all predictions in 0.14 seconds per test method + focal method pairing. When increasing beam size, we find that the time needed to generate assert statements appears to scale linearly.

5.5 Threats

Construct validity threats concern the relationship between theory and observation, and are mainly related to sources of imprecision in our analyses. We automatically mined and parsed the TAPs used in our study. In this process, the main source of noise is represented by the heuristic we used to identify the focal method for a given assert statement. As said, in a real usage scenario, this information could be provided by the developer who wrote the test or by the automatic test case generation tool. Despite the presence of introduced noise, our approach was still able to generate meaningful asserts, confirming the robustness of our NMT model.

Internal validity threats concern factors internal to our study that could influence our results. The performance of our approach depends on the hyperparameter configuration, that we report in our online replication package [9]. However, given how computationally-expensive the hyperparameter search was, we did not investigate the impact of the copy mechanism across all configurations.

External validity threats concern the generalizability of our findings. We did not compare ATLAS with state-of-the-art test case generation techniques using the heuristics described in section 5.2 to define appropriate asserts. This comparison would require a manual evaluation of the correctness of the asserts generated by ATLAS and by the competitive techniques for automatically generated tests, likely on software of which we have little knowledge (assuming Open-Source projects). Furthermore, we would have to make the subject systems executable (e.g., as required by EvoSuite) which is known to be difficult. Hence, this would not allow us to scale such an experiment to the size of our current evaluation. Indeed, our main goal was to empirically investigate the feasibility of a learning-based approach for assert statement generation. Comparing/combining the two families of approaches is part of our future work. Finally, we only focused on Java programs and the JUnit framework. However, ATLAS's learning process is language-independent and its NMT infrastructure can easily be ported to other programming languages.

5.6 Conclusion

In this work, we mined over 9k GitHub projects to identify test methods with their related assert statements. We then used a heuristic to identify the focal method associated with an assert statement and generated a dataset of Test-Assert Pairs (TAPs), composed by i) an input sequence of the test and focal method, and ii) a target sequence reporting the appropriate assert statement for the input sequence. We used these TAPs to create a NMT-based approach, called ATLAS, capable of learning semantically and syntactically correct asserts given the context of the test and focal methods.

We found that the model was capable of successfully predicting over 31% of the assert statements developers wrote by only using the top-ranked prediction. When looking at the top-5 predictions the percentage of correctly generated assert statements grew up to ~50%. We also showed that among the "imperfect predictions", meaning the scenario in which ATLAS generates assert statements different from the ones manually written by developers, there are assert statements that either represent a plausible alternative the the one written by the developer or can be converted into the latter by just modifying a few tokens (≤ 3 in 50% of cases). Finally, some of the limitations of our approach, as well as the extensive empirical study we conducted, provide us with a number of lessons learned that can drive future research in the field:

Raw code vs. Abstracted code: Our results show that through the abstraction mechanism, applications of NMT on code can ensure better performance, as already observed in previous studies [260, 265, 262]. More interestingly, we found that the two code representations are quite complementary, and allow our approach to generate different sets of "perfect predictions". This points to the possibility of combining the two representations into a single model that could benefit from an architecture having two encoders (one per representation) and a single decoder.

On the possibility of generating multiple assert statements: We investigated this research direction while working on ATLAS. The main problem we faced was the automatic identification of the part of test method body that it is relevant for a given assert. Indeed, while with a single assert we can assume that the whole method body is relevant, this is not the case when dealing with multiple asserts. Here, a possible heuristic could be to consider the statements preceding the assert statement a_1, but coming after the previous assert a_0, as relevant for a_1. However, it is unlikely that the statements coming before a_0 are totally irrelevant for a_1. Another possibility we considered was to apply backward slicing on each assert statement but, unfortunately, this resulted in scalability issues and in (well-known) problems related to the automatic compilation of open source projects [259]. Approaching this problem is a compelling direction for future work.

Integrating a learning-based approach in tools for automatic test case generation: As discussed earlier, we foresee two possible usages for ATLAS. First, it can be used as a code completion mechanism when manually writing unit tests in the IDE. Second, it could be combined with existing tools for automatic test case generation [94, 213, 5]. We currently lack empirical evidence to substantiate any claim on the effectiveness of ATLAS in improving automatically generated tests, which would be an important first step prior to integration. Additionally, before combining ATLAS with existing tools, it is necessary to deeply understand the cases in which the two families of approaches (*i.e.*, the ones integrated in the test case generation tool and the learning-based one) succeed and fail. In this way, learning-based approaches could be used only when needed (*i.e.*, when the standard approach implemented in the test case generation tools is likely to fail), thus increasing the effectiveness of the generated tests. Studying and comparing the strengths and weakness of the two families of techniques is part of our future research agenda.

The automatic generation of meaningful assert statements is a compelling problem within the software engineering community. We showed that a learning-based approach could help aid in this problem, and opened a complementary research direction to the one already adopted in automatic test case generation tools [94, 213, 5].

Chapter 6

Conclusions and Future Work

In this dissertation, we presented two distinct DL based approaches for the task of detecting similar pieces of source code and the automatic generation of assert statements for test methods. We then presented a systematic literature review, which explores the vast array of DL solutions for a variety of SE tasks. The goal of this work is to show the practical applications of DL in SE, but also to consider the future of DL applications and the potential of applying SE methodologies to these complex models.

We begin this dissertation by focusing on the summarization of the use of DL within SE. We follow the guidelines set for SLRs by Kitchenham *et al.* and systematically gather research that implements a DL based approach for a SE task. We then build a taxonomy of the research gathered for the SE tasks addressed, the SE artifacts mined and processed for DL, and the DL architectures used in the approaches. Throughout our review, we find a number of papers related to the idea of DL but do not fall within the scope of our study. Although our taxonomy focuses on papers that implement a DL based approach, we take note of the interesting and emerging field of SE for DL. In many ways, these papers examine the faulty nature of using DL based algorithms without fully understanding how they work. As our survey showed, a number of important SE tasks have DL based solutions applied to them. As a community, we have subjected ourselves to results based analysis of these models and overlook their shortcomings when applying them to SE problems.

Therefore, we provide guidelines and instructions for important components of learning to include and report on. We also provide a clear road map for future directions in DL4SE research. In the future, we hope the SE community not only looks to evolve and optimize the use of DL solutions to SE problems, but also reevaluate the way in which we apply these powerful models. SE research has created methods for developing, optimizing and testing traditional software projects, however, DL development calls for a shift in the paradigm of the software development life cycle. This survey finds research papers that indicate the beginnings of that paradigm shift.

Our next chapter begins by using a recursive autoencoder to find similar pieces of source code through the use of multiple code representations. We mine 10 pertinent Java projects and extract their classes and methods. Our next step encodes these code fragments into four separate representations: identifiers, bytecode, AST and CFG. We used a recurrent neural network to create a continuous value vector for each word within the code fragment. We then implemented a recursive autoencoder to encode the arbitrarily long vector of encoded words. The resulting embedding from the recursive autoencoder maps to a multidimensional space and the similarity to other embedded code fragments can be calculated based on distance. After we measured this similarity, we combined these four models via ensemble learning where we demonstrated that considering multiple representations allowed us to detect similar code fragments that no single model could entirely capture. To show the value of considering multiple representations of source code, we applied our methodology to the process of detecting clones among 10 different Java projects. We found that we achieve a relatively high precision, recall and F-measure when detecting clones among these 10 Java projects. More importantly, through PCA, we found that our ability to detect clones relied on the orthogonal contributions of each representation.

Our last chapter involves the DL strategy of NMT, which encoded a sequence of tokens into a hidden state that is then *translated* into different tokens by a decoder. We applied this methodology to the task of assert statement generation for test methods. We mined thousands of Java projects from GitHub that used the Junit package. We used the tool spoon to extract the test methods and all potential focal methods that the assert statements evaluate. Next, we used a heuristic to identify the focal method pertaining to the assert statement we wanted to predict. We then extracted the focal method signature and body as context, which helped the model to predict an accurate assert statement. We performed a series of filtering and data preparation steps. We then created two distinct models, one for raw source code tokens and the other for abstracted source code tokens. These NMT models took the test method and focal method context as input, and output a syntactically and semantically correct assert statement. Our evaluation determined how many of the predicted assert statements were identical to the assert statement written by developers. We also evaluated the quality of the potential assert statement predictions to determine if our model could generate asserts tantamount to those written by developers.

For future work, we consider the taxonomies highlighted in the SLR and the SE issues that are not addressed with DL based solutions. We also see a lot of growth and development in applying SE methodologies to DL software. There is a lot of potential for researchers to examine Tensorflow, PyTorch, Theano and others for software bugs, implementation bugs and overarching testing strategies. It is the opinion of this researcher that DL based approaches will continue to be applied to issues within SE as well as other domains. Therefore, there will be a large opportunity for the development of a new software development life cycle pertaining to DL.

Bibliography

[1] 2015 impact factor released for cochrane database of systematic reviews.

[2] How many words are there in the english language.

[3] Tiobe index for august 2019.

[4] `https://maven.apache.org/`.

[5] Utilizing fast testing to transform java development into an agile, quick release, low risk process.

[6] Apache commons project distributions: `https://archive.apache.org/dist/commons/`, 2017.

[7] Online appendix (anonymized): `https://sites.google.com/view/learningcodesimilarities`, 2017.

[8] Soot: `https://github.com/Sable/soot`, 2017.

[9] Atlas anonymous online appendix: `https://sites.google.com/view/atlas-nmt/home`, 2019.

[10] Best data science machine learning platform, Feb 2020.

[11] B A. Kitchenham. *Kitchenham, B.: Guidelines for performing Systematic Literature Reviews in software engineering. EBSE Technical Report EBSE-2007-01.* 01 2007.

[12] Yaser S Abu-Mastafa. The learning problem.

[13] Yaser S Abu-Mastafa. The learning problem.

[14] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012.

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

[16] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. *CoRR*, abs/1812.06469, 2018.

[17] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM.

[18] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM.

[19] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37, July 2018.

[20] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. A survey of machine learning for big code and naturalness. *CoRR*, abs/1709.06182, 2017.

[21] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[22] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. *CoRR*, abs/1602.03001, 2016.

[23] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. *CoRR*, abs/1602.03001, 2016.

[24] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2091–2100, 2016.

[25] Miltiadis Allamanis and Charles A. Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 472–483, 2014.

[26] M. Almasi et al. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *ICSE-C'17*.

[27] Anh Tuan Nguyen and T. N. Nguyen. Automatic categorization with deep neural network for open-source java projects. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 164–166, May 2017.

[28] Forough Arabshahi, Sameer Singh, and Animashree Anandkumar. Combining symbolic expressions and black-box function evaluations in neural programs. *CoRR*, abs/1801.04342, 2018.

[29] Andrea Arcuri and Gordon Fraser. Java enterprise edition support in search-based junit test generation. In *International Symposium on Search Based Software Engineering*, pages 3–17. Springer, 2016.

[30] M. I. Babar, M. Ghazali, and D. N. A. Jawawi. Systematic reviews in requirements engineering: A systematic review. In *2014 8th. Malaysian Software Engineering Conference (MySEC)*, pages 43–48, Sep. 2014.

[31] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 356–367, 2016.

[32] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2015.

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

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

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

[36] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

[37] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 306–317, New York, NY, USA, 2014. ACM.

[38] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. An empirical study on the developers&#039; perception of software coupling. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 692–701, Piscataway, NJ, USA, 2013. IEEE Press.

[39] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Trans. Softw. Eng. Methodol.*, 23(1):4:1–4:33, February 2014.

[40] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Trans. Softw. Eng.*, 40(7):671–694, July 2014.

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

[42] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 63–74, New York, NY, USA, 2016. ACM.

[43] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *CoRR*, abs/1806.07336, 2018.

[44] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828, August 2013.

[45] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[46] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 60–70, New York, NY, USA, 2018. ACM.

[47] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[48] Amar Budhiraja, Kartik Dutta, Raghu Reddy, and Manish Shrivastava. Dwen: Deep word embedding network for duplicate bug report detection in software repositories. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 193–194, New York, NY, USA, 2018. ACM.

[49] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *CoRR*, abs/1805.04276, 2018.

[50] Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. *CoRR*, abs/1704.06611, 2017.

[51] C. Calero, M. F. Bertoa, and M. Á. Moraga. A systematic literature review for software sustainability measures. In *2013 2nd International Workshop on Green and Sustainable Software (GREENS)*, pages 46–53, May 2013.

[52] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[53] Chao Chen, Wenrui Diao, Yingpei Zeng, Shanqing Guo, and Chengyu Hu. DRLGENCERT: deep learning-based automated testing of certificate verification in SSL/TLS implementations. *CoRR*, abs/1808.05444, 2018.

[54] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 665–676, New York, NY, USA, 2018. ACM.

[55] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 665–676, New York, NY, USA, 2018. ACM.

[56] G. Chen, C. Chen, Z. Xing, and B. Xu. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 744–755, Sep. 2016.

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

[58] Qingying Chen and Minghui Zhou. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 826–831, New York, NY, USA, 2018. ACM.

[59] Qingying Chen and Minghui Zhou. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 826–831, New York, NY, USA, 2018. ACM.

[60] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, pages 2547–2557. Curran Associates, Inc., 2018.

[61] Xinyun Chen, Chang Liu, and Dawn Xiaodong Song. Towards synthesizing complex programs from input-output examples. *arXiv: Learning*, 2018.

[62] Zimin Chen et al. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *CoRR*, abs/1901.01808, 2019.

[63] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *CoRR*, abs/1901.01808, 2019.

[64] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[65] M. Choetkiertikul, H. K. Dam, T. Tran, A. Ghose, and J. Grundy. Predicting delivery capability in iterative software development. *IEEE Transactions on Software Engineering*, 44(6):551–573, June 2018.

[66] M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, A. Ghose, and T. Menzies. A deep learning model for estimating story points. *IEEE Transactions on Software Engineering*, 45(7):637–656, July 2019.

[67] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. Predicting the delay of issues with due dates in software projects. *Empirical Software Engineering*, 22(3):1223–1263, Jun 2017.

[68] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum.* Addison-Wesley Professional, 1st edition, 2009.

[69] C. S. Corley, K. Damevski, and N. A. Kraft. Exploring the use of deep learning for feature location. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 556–560, Sep. 2015.

[70] C. S. Corley, K. Damevski, and N. A. Kraft. Exploring the use of deep learning for feature location. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 556–560, Sep. 2015.

[71] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 95–105, New York, NY, USA, 2018. ACM.

[72] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[73] Julius Davies, Daniel M. Germán, Michael W. Godfrey, and Abram Hindle. Software bertillonage - determining the provenance of software development artifacts. *Empirical Software Engineering*, 18(6):1195–1237, 2013.

[74] Rina Dechter. Learning while searching in constraint-satisfaction-problems. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI'86, pages 178–183. AAAI Press, 1986.

[75] Yuntian Deng, Anssi Kanervisto, Jeffrey Ling, and Alexander M. Rush. Image-to-markup generation with coarse-to-fine attention. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML17, page 980989. JMLR.org, 2017.

[76] J. Deshmukh, A. K. M, S. Podder, S. Sengupta, and N. Dubash. Towards accurate duplicate bug retrieval using deep learning techniques. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 115–124, Sep. 2017.

[77] P. Devanbu. New initiative: The naturalness of software. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 543–546, May 2015.

[78] Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew J. Hausknecht, and Pushmeet Kohli. Neural program meta-induction. *CoRR*, abs/1710.04157, 2017.

[79] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. *CoRR*, abs/1703.07469, 2017.

[80] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: A systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pages 31–36, New York, NY, USA, 2007. ACM.

[81] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, and Giuliano Antoniol. Can better identifier splitting techniques help feature location? In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 11–20, Washington, DC, USA, 2011. IEEE Computer Society.

[82] Bogdan Dit, Evan Moritz, Mario Linares-Vásquez, Denys Poshyvanyk, and Jane Cleland-Huang. Supporting and accelerating reproducible empirical research in software evolution and maintenance using tracelab component library. *Empirical Softw. Engg.*, 20(5):1198–1236, October 2015.

[83] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[84] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Softw. Engg.*, 18(2):277–309, April 2013.

[85] Pedro Domingos. Occams two razors: The sharp and the blunt. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD98, page 3743. AAAI Press, 1998.

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

[87] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[88] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided bayesian program induction. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, pages 7805–7815. Curran Associates, Inc., 2018.

[89] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, pages 6059–6068. Curran Associates, Inc., 2018.

[90] F. Falcini, G. Lami, and A. Mitidieri. Yet another challenge for the automotive software: Deep learning. *IEEE Software*, pages 1–1, 2017.

[91] Fabio Falcini, Giuseppe Lami, and Alessandra Costanza. Deep learning in automotive software. *IEEE Software*, 34:56–63, 05 2017.

[92] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37, Mar. 1996.

[93] J.L. Fleiss et al. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[94] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, Szeged, Hungary, 2011. ACM.

[95] Gordon Fraser et al. Evosuite: automatic test suite generation for object-oriented software. In *ESEC/FSE'11*, pages 416–419.

[96] Wei Fu and Tim Menzies. Easy over hard: a case study on deep learning. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 49–60, 2017.

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

[98] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou. Variability in software systems—a systematic literature review. *IEEE Transactions on Software Engineering*, 40(3):282–306, March 2014.

[99] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 896–899, New York, NY, USA, 2018. ACM.

[100] Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *Proceedings of the 34th International Conference on Machine Learning*, Doina Precup and Yee Whye Teh, editors, volume 70 of *Proceedings of Machine Learning Research*, pages 1213–1222, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[101] R. L. Glass. The naturalness of object orientation: beating a dead horse? *IEEE Software*, 19(3):104–103, May 2002.

[102] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&#38;fuzz: Machine learning for input fuzzing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 50–59, Piscataway, NJ, USA, 2017. IEEE Press.

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

[104] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[105] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *CoRR*, abs/1705.02801, 2017.

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

[107] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 933–944, New York, NY, USA, 2018. ACM.

[108] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 933–944, New York, NY, USA, 2018. ACM.

[109] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, New York, NY, USA, 2016. ACM.

[110] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, New York, NY, USA, 2016. ACM.

[111] J. Guo, J. Cheng, and J. Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 3–14, May 2017.

[112] J. Guo, J. Cheng, and J. Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 3–14, May 2017.

[113] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: Differential fuzzing testing of deep learning systems. *CoRR*, abs/1808.09413, 2018.

[114] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 3–14, Piscataway, NJ, USA, 2017. IEEE Press.

[115] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 1345–1351. AAAI Press, 2017.

[116] Kamill Gusmanov. On the adoption of neural networks in modeling software reliability. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 962–964, New York, NY, USA, 2018. ACM.

[117] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, Nov 2012.

[118] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. pages 1–12. ACM Press, 2000.

[119] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng. Learning to predict severity of software vulnerability using only vulnerability description. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 125–136, Sep. 2017.

[120] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, and peter chin. Learning to repair software vulnerabilities with generative adversarial networks. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, pages 7933–7943. Curran Associates, Inc., 2018.

[121] Hideaki Hata, Emad Shihab, and Graham Neubig. Learning to generate corrective patches using neural machine translation. *CoRR*, abs/1812.07170, 2018.

[122] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 152–162, New York, NY, USA, 2018. ACM.

[123] Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 763–773, New York, NY, USA, 2017. ACM.

[124] Vincent J. Hellendoorn and Premkumar T. Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 763–773, 2017.

[125] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Mohammad Amin Alipour. On the naturalness of proofs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 724–728, New York, NY, USA, 2018. ACM.

[126] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847, June 2012.

[127] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[128] S. Hosseini, B. Turhan, and D. Gunarathna. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, 45(2):111–147, Feb 2019.

[129] Q. Huang, X. Xia, D. Lo, and G. C. Murphy. Automating intention mining. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[130] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems, 2019.

[131] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[132] A.G. Ivakhnenko and V.G. Lapa. *Cybernetic Predicting Devices*. JPRS 37, 803. Purdue University School of Electrical Engineering, 1966.

[133] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 20–30, New York, NY, USA, 2018. ACM.

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

[135] S. Jiang, A. Armaly, and C. McMillan. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE'17, pages 135–146, October 2017. ISSN:.

[136] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 135–146, Piscataway, NJ, USA, 2017. IEEE Press.

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

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

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

[140] Nal Kalchbrenner and Phil Blunsom. Recurrent continuous models. In *EMNLP*, 2013.

[141] 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.

[142] Rafael-Michael Karampatsis and Charles A. Sutton. Maybe deep neural networks are the best choice for modeling source code. *CoRR*, abs/1903.05734, 2019.

[143] Andrej Karpathy and Andrej Karpathy. Software 2.0, Nov 2017.

[144] Jeannette Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. 1952.

[145] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page to appear, New York, NY, USA, 2010. ACM.

[146] Junhwi Kim, Minhyuk Kwon, and Shin Yoo. Generating test input with deep reinforcement learning. In *Proceedings of the 11th International Workshop on Search-Based Software Testing*, SBST '18, pages 51–58, New York, NY, USA, 2018. ACM.

[147] Pieter-Jan Kindermans, Kristof T. Schtt, Maximilian Alber, Klaus-Robert Mller, Dumitru Erhan, Been Kim, and Sven De. Learning how to explain neural networks: Patternnet and patternattribution, 2017.

[148] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[149] B. Kitchenham and S Charters. Guidelines for performing systematic literature reviews in software engineering, 2007.

[150] C. Klammer et al. Writing unit tests: It's now or never! In *ICSTW'15*, pages 1–4, April 2015.

[151] Philipp Koehn and Rebecca Knowles. Six challenges for neural machine translation. *CoRR*, abs/1706.03872, 2017.

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

[153] B. Korel and A. M. Al-Yami. Assertion-oriented automated test data generation. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 71–80, March 1996.

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

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

[156] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, pages 1097–1105. Curran Associates, Inc., 2012.

[157] A. Lam, A. Nguyen, H. Nguyen, and T. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports. ASE'15.

[158] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 476–481, Nov 2015.

[159] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In

2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 476–481, Nov 2015.

[160] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 218–229, 2017.

[161] Tien-Duy B. Le, Lingfeng Bao, and David Lo. Dsm: A specification mining tool using recurrent neural network based language model. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 896–899, New York, NY, USA, 2018. ACM.

[162] Tien-Duy B. Le and David Lo. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 106–117, New York, NY, USA, 2018. ACM.

[163] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 795–806, Piscataway, NJ, USA, 2019. IEEE Press.

[164] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[165] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. Convolutional networks and applications in vision. In *International Symposium on Circuits and Systems (ISCAS 2010), May 30 - June 2, 2010, Paris, France*, pages 253–256, 2010.

[166] Sun-Ro Lee, Min-Jae Heo, Chan-Gun Lee, Milhan Kim, and Gaeul Jeong. Applying deep learning based automatic bug triager to industrial projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 926–931, New York, NY, USA, 2017. ACM.

[167] Dor Levy and Lior Wolf. Learning to align the source code to the compiled object code. In *Proceedings of the 34th International Conference on Machine Learning*, Doina Precup and Yee Whye Teh, editors, volume 70 of *Proceedings of Machine Learning Research*, pages 2043–2051, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[168] D. Li, Z. Wang, and Y. Xue. Fine-grained android malware detection based on deep learning. In *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–2, May 2018.

[169] D. Li, Z. Wang, and Y. Xue. Fine-grained android malware detection based on deep learning. In *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–2, May 2018.

[170] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara G. Ryder. Cclearner: A deep learning-based clone detection approach. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, 2017.

[171] Xiangang Li and Xihong Wu. Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition. *CoRR*, abs/1410.4281, 2014.

[172] 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.

[173] Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. Memory augmented policy optimization for program synthesis and seman-

tic parsing. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, pages 9994–10006. Curran Associates, Inc., 2018.

[174] B. Lin, C. Nagy, G. Bavota, and M. Lanza. On the impact of refactoring operations on code naturalness. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 594–598, Feb 2019.

[175] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto. Sentiment analysis for software engineering: How far can we go? In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 94–104, May 2018.

[176] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: Cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 667–678, New York, NY, USA, 2018. ACM.

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

[178] Chang Liu, Xinyun Chen, Richard Shin, Mingcheng Chen, and Dawn Song. Latent attention for if-then program synthesis. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, pages 4574–4582. Curran Associates, Inc., 2016.

[179] Hui Liu, Zhifeng Xu, and Yanzhen Zou. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 385–396, New York, NY, USA, 2018. ACM.

[180] Hui Liu, Zhifeng Xu, and Yanzhen Zou. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 385–396, New York, NY, USA, 2018. ACM.

[181] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[182] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng. Automatic text input generation for mobile testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 643–653, May 2017.

[183] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. Automatic text input generation for mobile testing. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 643–653, Piscataway, NJ, USA, 2017. IEEE Press.

[184] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: How far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 373–384, New York, NY, USA, 2018. ACM.

[185] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: How far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 373–384, New York, NY, USA, 2018. ACM.

[186] Lei Ma, Felix Juefei-Xu, Jiyuan Sun, Chunyang Chen, Ting Su, Fuyuan Zhang, Minhui Xue, Bo Li, Li Li, Yang Liu, Jianjun Zhao, and Yadong

Wang. Deepgauge: Comprehensive and multi-granularity testing criteria for gauging the robustness of deep learning systems. *CoRR*, abs/1803.07519, 2018.

[187] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 120–131, New York, NY, USA, 2018. ACM.

[188] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. Mode: Automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 175–186, New York, NY, USA, 2018. ACM.

[189] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 653–656, 2016.

[190] David J. C. MacKay. *Information Theory, Inference Learning Algorithms.* Cambridge University Press, USA, 2002.

[191] Suvodeep Majumder, Nikhila Balaji, Katie Brey, Wei Fu, and Tim Menzies. 500+ times faster than deep learning: A case study exploring faster methods for text mining stackoverflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 554–563, New York, NY, USA, 2018. ACM.

[192] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *16th IEEE International Conference on Automated Software*

Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA, pages 107–114, 2001.

[193] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 364–374, Piscataway, NJ, USA, 2012. IEEE Press.

[194] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: Learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 925–936, New York, NY, USA, 2019. ACM.

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

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

[197] Narcisa Andreea Milea, Lingxiao Jiang, and Siau-Cheng Khoo. Vector abstraction and concretization for scalable detection of refactorings. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 86–97, New York, NY, USA, 2014. ACM.

[198] Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable abstract interpretation for provably robust neural networks. In *Proceedings of the 35th International Conference on Machine Learning*, Jennifer Dy and Andreas Krause, editors, volume 80 of *Proceedings of Machine Learning Research*, pages 3578–3586, Stockholmsman, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[199] K. P. Moran, C. Bernal-Cenas, M. Curcio, R. Bonett, and D. Poshyvanyk. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[200] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. Machine learning-based prototyping of graphical user interfaces for mobile apps. *CoRR*, abs/1802.02312, 2018.

[201] Kevin Moran, Cody Watson, John Hoskins, George Purnell, and Denys Poshyvanyk. Detecting and summarizing gui changes in evolving mobile apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 543–553, New York, NY, USA, 2018. ACM.

[202] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding api usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 151–162, New York, NY, USA, 2017. ACM.

[203] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018.

[204] Graham Neubig. Neural machine translation and sequence-to-sequence models: A tutorial. *CoRR*, abs/1703.01619, 2017.

[205] Anh Tuan Nguyen and Tien N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 858–868, Piscataway, NJ, USA, 2015. IEEE Press.

[206] Anh Tuan Nguyen and Tien N. Nguyen. Automatic categorization with deep neural network for open-source java projects. In *Proceedings of the 39th International*

Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume, pages 164–166, 2017.

[207] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 651–654, New York, NY, USA, 2013. ACM.

[208] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 544–547, New York, NY, USA, 2014. ACM.

[209] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[210] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 574–584, Piscataway, NJ, USA, 2015. IEEE Press.

[211] J. Ott, A. Atchison, P. Harnack, A. Bergh, and E. Linstead. A deep learning approach to identifying source code in images and video. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 376–386, May 2018.

[212] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1105–1114, New York, NY, USA, 2016. ACM.

[213] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *OOPSLA'07*, pages 815–816, 01 2007.

[214] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 522–531, 2013.

[215] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. Parameterizing and assembling ir-based solutions for SE tasks using genetic algorithms. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 314–325, 2016.

[216] Stefano Panzeri, Cesare Magri, and Ludovico Carraro. Sampling bias.

[217] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, 2002.

[218] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Softw. Pract. Exper.*, 46(9):1155–1179, September 2016.

[219] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on Machine Learning*, Francis Bach and David Blei, editors, volume 37 of *Proceedings of Machine Learning Research*, pages 1093–1102, Lille, France, 07–09 Jul 2015. PMLR.

[220] D. Pierret and D. Poshyvanyk. An empirical exploration of regularities in open-source software lexicons. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 228–232, May 2009.

[221] Yolande Poirier. What are the most popular libraries java developers use? based on github's top projects.

[222] Abdallah Qusef, Rocco Oliveto, and Andrea De Lucia. Recovering traceability links between unit tests and classes under test: An improved method. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10, 2010.

[223] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics. *Inf. Softw. Technol.*, 55(8):1397–1418, August 2013.

[224] Carl Edward Rasmussen and Zoubin Ghahramani. Occam's razor. In *Advances in Neural Information Processing Systems 13*, T. K. Leen, T. G. Dietterich, and V. Tresp, editors, pages 294–300. MIT Press, 2001.

[225] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 428–439, May 2016.

[226] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM.

[227] Scott Reed and Nando Freitas. Neural programmer-interpreters. 11 2015.

[228] Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 14–23, Washington, DC, USA, 2010. IEEE Computer Society.

[229] Daniel Rodriguez, Javier Dolado, and Javier Tuya. Bayesian concepts in software testing: An initial review. In *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*, A-TEST 2015, pages 41–46, New York, NY, USA, 2015. ACM.

[230] S. Romansky, N. C. Borle, S. Chowdhury, A. Hindle, and R. Greiner. Deep green: Modelling time-series of software energy consumption. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 273–283, Sep. 2017.

[231] Chanchal Kumar Roy and James R. Cordy. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, pages 172–181, 2008.

[232] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: Detection of clones in the twilight zone. *CoRR*, abs/1806.05837, 2018.

[233] Hitesh Sajnani and Cristina Lopes. A parallel and efficient approach to large scale clone detection. In *Proceedings of the 7th International Workshop on Software Clones*, IWSC '13, pages 46–52, Piscataway, NJ, USA, 2013. IEEE Press.

[234] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM.

[235] Jonathan Schaffer. What not to multiply without necessity. *Australasian Journal of Philosophy*, 93(4):644–664, 2015.

[236] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[237] Jan Schroeder, Christian Berger, Alessia Knauss, Harri Preenja, Mohammad Ali, Miroslaw Staron, and Thomas Herpel. Predicting and evaluating software model growth in the automotive industry. *CoRR*, abs/1708.02884, 2017.

[238] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, pages 2503–2511. Curran Associates, Inc., 2015.

[239] Sina Shamshiri. Automated Unit Test Generation for Evolving Software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, FSE'15, pages 1038–1041, Bergamo, Italy, 2015. ACM.

[240] P. Sharma and J. Singh. Systematic literature review on software effort estimation using machine learning approaches. In *2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS)*, pages 43–47, Dec 2017.

[241] Richard Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, pages 8917–8926. Curran Associates, Inc., 2018.

[242] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, pages 7751–7762. Curran Associates, Inc., 2018.

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

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

[245] 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.

[246] Richard Socher, Jeffrey Pennington, Eric H. Huang, Andrew Y. Ng, and Christopher D. Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 151–161, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[247] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep

models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.

[248] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[249] Student. An experimental determination of the probable error of dr spearman's correlation coefficients. *Biometrika*, 13(2/3):263–282, 1921.

[250] Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *Proceedings of the 35th International Conference on Machine Learning*, Jennifer Dy and Andreas Krause, editors, volume 80 of *Proceedings of Machine Learning Research*, pages 4790–4799, Stockholmsman, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[251] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 109–119, New York, NY, USA, 2018. ACM.

[252] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[253] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, pages 3104–3112. Curran Associates, Inc., 2014.

[254] J. Svajlenko and C. Roy. Evaluating clone detection tools with bigclonebench. ICSME'15, 2015.

[255] Jeffrey Svajlenko, Chanchal K. Roy, and James R. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *Proceeding of the 7th International Workshop on Software Clones, IWSC 2013, San Francisco, CA, USA, May 19, 2013*, pages 8–9, 2013.

[256] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013.

[257] Chris Thunes. c2nes/javalang, Feb 2019.

[258] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 303–314, New York, NY, USA, 2018. ACM.

[259] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838.

[260] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. *CoRR*, abs/1901.09102, 2019.

[261] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. In *Proceedings of the 15th International*

Conference on Mining Software Repositories, MSR '18, pages 542–553, New York, NY, USA, 2018. ACM.

[262] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 832–837, New York, NY, USA, 2018. ACM.

[263] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 832–837, New York, NY, USA, 2018. ACM.

[264] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *CoRR*, abs/1812.08693, 2018.

[265] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. *CoRR*, abs/1812.10772, 2018.

[266] Anne-Wil Harzing Sat 6 Feb 2016 16:10 (updated Thu 5 Sep 2019 10:36). Publish or perish.

[267] Muhammad Usman, Emilia Mendes, Francila Weidt, and Ricardo Britto. Effort estimation in agile software development: A systematic literature review. In *Proceedings of the 10th International Conference on Predictive Models in Software Engineering*, PROMISE '14, pages 82–91, New York, NY, USA, 2014. ACM.

[268] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 683–693, New York, NY, USA, 2017. ACM.

[269] Mario Linares Vásquez, Andrew Holtzhauer, and Denys Poshyvanyk. On automatically detecting similar android apps. In *24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016*, pages 1–10, 2016.

[270] Ashwin J. Vijayakumar, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *CoRR*, abs/1804.01186, 2018.

[271] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 397–407, New York, NY, USA, 2018. ACM.

[272] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 397–407, New York, NY, USA, 2018. ACM.

[273] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang Wang, and Peixin Zhang. Adversarial sample detection for deep neural network through model mutation testing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page to appear, New York, NY, USA, 2019. ACM.

[274] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. *CoRR*, abs/1711.07163, 2017.

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

[276] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308, May 2016.

[277] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 297–308, New York, NY, USA, 2016. ACM.

[278] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 455–465, 2013.

[279] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 455–465, New York, NY, USA, 2013. ACM.

[280] Cody Watson, David Palacio, Nathan Cooper, Kevin Moran, and Denys Poshyvanyk. Data analysis for the systematic literature review of dl4se.

[281] M. Wen, R. Wu, and S. C. Cheung. How well do change sequences predict defects? sequence learning from software changes. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[282] M. White. Deep representations for software engineering. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 781–783, May 2015.

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

[284] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, Sep. 2016.

[285] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, Sep. 2016.

[286] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press.

[287] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press.

[288] H. Wu, L. Shi, C. Chen, Q. Wang, and B. Boehm. Maintenance effort estimation for open source software: A systematic literature review. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 32–43, Oct 2016.

[289] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In

2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 51–62, Sep. 2016.

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

[291] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine parallel natural language/source code corpora from stack overflow. pages 388–389, 05 2018.

[292] Tom Zahavy, Nir Ben-Zrihem, and Shie Mannor. Graying the black box: Understanding dqns. *CoRR*, abs/1602.02658, 2016.

[293] Matthew D. Zeiler. Adadelta: An adaptive learning rate method, 2012.

[294] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. Neural guided constraint logic programming for program synthesis. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, pages 1737–1746. Curran Associates, Inc., 2018.

[295] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 132–142, New York, NY, USA, 2018. ACM.

[296] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 129–140, New York, NY, USA, 2018. ACM.

[297] Jinman Zhao, Aws Albarghouthi, Vaibhav Rastogi, Somesh Jha, and Damien Octeau. Neural-augmented static analysis of android communication. *CoRR*, abs/1809.04059, 2018.

[298] Zhi-Hua Zhou. *Ensemble Learning*, pages 270–273. Springer US, Boston, MA, 2009.

[299] Luisa M. Zintgraf, Taco S. Cohen, Tameem Adel, and Max Welling. Visualizing deep neural network decisions: Prediction difference analysis. *CoRR*, abs/1702.04595, 2017.

[300] Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, pages 2094–2103. Curran Associates, Inc., 2018.