Intelligent Software Tooling for Improving Software Development

Nathan Allen Cooper

Encinitas, California, USA

Bachelor of Science, University of West Florida, 2018

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

Department of Computer Science

William & Mary
May 2023

APPROVAL PAGE

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

Doctor of Philosophy

Nathan Cooper

Approved by the Committee, May 2023

Committee Chair
Professor Denys Poshyvanyk, Computer Science
William & Mary

Assistant Professor Oscar Chaparro, Computer Science
William & Mary

Assistant Professor Adwait Nadkarni, Computer Science
William & Mary

Assistant Professor Huajie Shao, Computer Science
William & Mary

Professor Robert Michael Lewis, Computer Science
William & Mary

Professor Adrian Marcus, Computer Science
The University of Texas at Dallas

COMPLIANCE PAGE

Research approved by

Protection of Human Subjects Committee

Protocol number(s):	PHSC-2019-01-22-13374
	PHSC-2020-04-27-14262
Date(s) of approval:	01/22/2019
	04/27/2020

ABSTRACT

Software has eaten the world with many of the necessities and quality of life services people use requiring software. Therefore, tools that improve the software development experience can have a significant impact on the world. Specifically, automating many of the software processes such as detecting and flagging potential error, compilation chains of complex systems, and even the coding process can significantly speed up the development and improve the quality of software. The success of Deep Learning over the past decade has shown huge advancements in automation across many domains, including Software Development processes. One of the main reasons behind this success, besides scientific and engineering advancements in Deep Learning model development and training, is the availability of large datasets such as open-source code available through GitHub or image datasets of mobile Graphical User Interfaces (GUIs) with RICO [109] and ReDRAW [257] to be trained on. With this in mind, the central research question my dissertation explores is: *In what ways can the software development process be improved through leveraging deep learning techniques on the vast amounts of unstructured software engineering artifacts?*

We coin the approaches that leverage Deep Learning to automate or augment various software development task as *Intelligent Software Tools.* To guide our research of these intelligent software tools, we performed a systematic literature review to understand the current landscape of research of applying Deep Learning techniques to software tasks and any gaps that exist. This involved processing 128 papers manually to extract information such as the software engineering task and deep learning model used. From this literature review, we found generative tasks such as code generation and comprehension to be the overwhelming majority with other types of tasks and artifacts such as information retrieval (*e.g.*, code search or impact analysis) or tasks involving images and videos to be understudied. With these results in mind, we set out to explore the application of Deep Learning to these understudied tasks and artifacts. Specifically, we developed a tool for automatically detecting duplicate mobile bug reports from user submitted videos. This is done by first training a large deep learning model, we used the popular Convolutional Neural Network (CNN), to learn important features from a large collection of mobile screenshots. This model is then used to extract the important features from the frames in a user submitted video-based bug report. The features are then compared between a newly submitted video and a corpus of existing ones from previous bug reports to determine their similarity and produce a ranked list of duplicate candidates that a developer can review for duplicates. Next, we explored impact analysis, a critical software maintenance task that identifies the effects of a given set of code changes on a larger software project with the intention of avoiding potential adverse effects. However, many of the previous approaches that perform impact analysis treat code in isolation. This is unrealistic as a

majority of code that developers will interact with will be part of a larger software system. Therefore, we introduced, Athena, a novel approach to impact analysis that integrates knowledge of a software system through its call-graph along with high-level representations of the code inside the system to improve information retrieval performance. Concretely, we explore extending state-of-the-art approaches in code representation that leverage the popular Transformer architecture [343] by injecting the call-graph information using a embedding propagation strategy inspired by Graph Convolutional Networks (GCNs) [201]. Lastly, we explored the task of code completion, which has seen heavy interest from industry and academia [171, 335, 167, 290, 60, 189, 185, 38, 165, 176, 268, 298, 299, 325, 326, 375, 85, 95, 142]. Specifically, we explored techniques to improve the efficiency of training these models since training Deep Learning models can incur a heavy cost due to the necessity of large models and datasets. In this work, we explored various methods that modify the positional encoding scheme of the Transformer architecture for allowing these models to incorporate longer sequences of tokens when predicting completions than seen during their training. Having a model that can be trained on shorter sequences and then later tested on longer sequences without a huge impact to performance has a number of advantages, namely in being able to decrease the cost of training since the standard self-attention mechanism the Transformer utilizes has a quadratic memory consumption based on the sequence length. Concretely, we explored whether the lessons learned from natural language processing could be extended to code.

Through numerous empirical evaluations, of which included user studies and specially designed benchmarks, we show the usefulness, practicality, and performance of these works.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

This dissertation was not performed in a bubble. It was the culmination of constant learning from and support by friends, family, and coworkers. First I would like to generously thank my Ph.D. advisor Denys Poshyvanyk for his guidance and support. He was especially supportive during personal strife and challenges during my Ph.D. Additionally, Denys consistently pushed me to do my best research and offered advise and guidance when needed. This dissertation would not be possible without his invaluable feedback. Thank you, Denys!

Additionally, I would like to thank my lab members, David Nader Palacio, Kevin Moran, Amit Ami, Carlos Bernal-Cardenas, Cody Watson, Michele Tufano, and Yanfu Yan, for their friendship and invaluable knowledge they shared with me. I would also like to thank my Ph.D. committee members, Dr. Oscar Chaparro, Dr. Adwait Nadkarni, Dr. Huajie Shao, and Dr. Robert Lewis for their extremely valuable feedback.

I would also like to extend my thanks to my collaborators Antonio Mastropaolo, Rosalia Tufano, Emad Aghajani, Dr. Gabriele Bavota, Dr. Massimiliano Di Penta, Dr. Andrian Marcus, and many others. It is a pleasure to collaborate with each of them and I hope to continue collaboration.

Lastly, I would like to thank my family and friends for the continued support, especially my mom who always believed in me.

To my mom, without whom I would not be here and to whom I owe everything.

From you I have learned the kindness, respect, and love that the world needs.

I love you mom.

LIST OF TABLES

LIST OF FIGURES

Intelligent Software Tooling for Improving Software Development

Chapter 1

Introduction

We're making this analogy that AI is the new electricity. Electricity transformed industries: agriculture, transportation, communication, manufacturing.

Andrew Ng (Baidu)

The 21st century has seen the fruition of the computation revolution in the 20th through the ubiquity of computers in the world from large datacenters down to wearables. The heart of this fast adoption across all sectors of life I believe to be due to the software that has unlocked the usefulness of these computers. With a many of the top businesses in the world being software first companies [1], *i.e.*, companies whose main product is software, and the projected growth of software developer jobs to grow over the 2020s by 25% [2], it is safe to say that improving the construction, reliability, maintenance, and security of software can have a huge impact on the world.

The computation revolution has also brought the storage and access to large amounts of data that is extremely diverse. Such access has lead to a huge success of one specific branch of Artificial Intelligence called Machine Learning, where statistical methods are used to learn a model on data for a specific task such as classifying a movie review as positive or negative. With the growing access to software artifacts on sites such as GitHub and

[1] `https://www.forbes.com/lists/global2000`

[2] `https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm`

GitLab and the processing of such data in the field of Mining Software Repositories (MSR), Software Development tools have begun to explore using Deep Learning architectures to improve or automate tasks. These tasks include code search [183], program repair [339], code generation [171, 335, 167, 290, 60, 189, 185, 38, 165, 176, 268, 298, 299, 325, 326, 375, 85, 95], bug detection [111, 281, 231], and GUI sketch generation [47, 257] to name a few. With this in mind, this dissertation focuses on exploring the following research question: *In what ways can the software development process be improved through leveraging deep learning techniques on the vast amounts of unstructured software engineering artifacts?* This research question gives rise to the following hypothesis:

> Deep learning allows for learning novel automations from software engineering artifacts that can help facilitate the software development process.

To answer this question and verify this hypothesis we both expanded on the set of tasks as well as improving the performance of existing tasks

1.1 Contributions

The first work I will discuss is a survey we performed that canvases the application of deep learning to software engineering tasks. It gives insights into the progress that has been made in the community and where future work is needed and is instrumental in guiding this dissertation's work. The survey collected and analyzed a total of 128 papers across a 10 year period, which included 23 software engineering tasks.

The second work I will discuss will be our work on leveraging advances in computer vision that uses troves of android screenshots to learn high level representations that are useful in determining if videos of bugs in mobile applications are duplicates of each other. This work uses a popular self-supervised learning approach, *i.e.*, no labeled data is required making it extremely scalable, called SimCLR [87] to learn a representation of frames in videos that are then used to compute a global video representation of video-based bug reports that can then be used to compare against other video representations in a corpus of previously submitted bug reports to determine if a new video is a duplicate or not.

The third work investigates improving upon information retrieval techniques in impact analysis (IA) by incorporating knowledge about a system through its call-graph. We introduce a novel IA approach, called ATHENA, that combines a software system's call graph information with a conceptual coupling approach that uses advances in deep representation learning for code without the need for change histories. Our approach is unsupervised and therefore does not require any labeled IA data. Previous IA benchmarks are small, containing less than 10 software projects, and suffer from tangled commits, making it difficult to measure accurate results. Therefore, we constructed a large-scale IA benchmark, from 25 open source software projects, that utilizes fine-grained commit information from bug fixes.

The fourth work explores the generalizability of current state of the art code completion models. Specifically, a recent trend in NLP research has been this idea of generalizing to longer sequences than seen during training [105, 283, 366]. Press *et al.* [283] introduced a simple and efficient change to the standard decoder-only Transformer positional encoding scheme that showed interesting results in ability to generalize to longer sequences than trained on compared to other approaches such as xPOS embeddings [317, 321] or T5 bias [285]. we present a large empirical study evaluating this generalization property of these and two other encoding schemes proposed in the literature, namely Sinusoidal, xPOS, ALiBi, and T5, as they have been at the heart of this generalization debate in NLP. We found that none of these solutions successfully generalize to unseen lengths and that the only safe solution is to ensure the representativeness in the training set of all lengths likely to be encountered at inference time.

In addition to the discussion of the works discussed in this dissertation, the author has worked and collaborated on a wide array of other works, which included the following topics: multi-task learning for software engineering [254], program repair [98], soundness in cryptographic misuse tools [30], and reproduction of mobile bugs [50]. Lastly, all work

discussed in this dissertation was not done in isolation and is the product of a large collaborative effort among my talented colleagues, collaborators, and mentors.

Chapter 2

Background & Related Work

In this section we will provide the necessary background on software development and maintenance in general and specific to mobile. Additionally, we will provide a landscape of the related work that exist in this area.

2.1 Software Maintenance Tasks and Tooling

2.1.1 Duplicate Bug Report Detection Tooling

In this subsection, we outline work related to our research, TANGO, that helps developers find duplicate video-based bug reports in an issue tracking system. This research is specifically related to work in near duplicate video retrieval, analysis of graphical software artifacts, and duplicate detection of textual bug reports.

Near Duplicate Video Retrieval. Extensive research has been done outside SE in near-duplicate video retrieval, which is the identification of similar videos to a query video (*e.g.*, exact copies [118, 209, 187, 158] or similar events [88, 188, 295, 205]).

The closest work to ours with TANGO is by Kordopatis-Zilos *et al.* [205], who addressed the problem of retrieving videos of incidents (*e.g.*, accidents). In their work, they explored using handcrafted-based [246, 46, 365, 393, 190, 191] (*e.g.*, SURF or HSV histograms) and DL-based [79, 206, 220, 207, 334, 73] (*e.g.*, CNNs) visual feature extraction techniques and ways of combining the extracted visual features [193, 295, 315, 188, 70, 207] (*e.g.*, VLAD). While we do make use of the best performing model (CNN+BoVW) from this work [205], we did not use the proposed handcrafted approaches, as these were designed for scenes about real-world incidents, rather than for mobile bug reporting. We also further modified and extended this approach given our different domain, through the combination of visual and textual information modalities, and adjustments to the CCN+BoVW model, including the layer configuration and training objective. We discuss Tango in-depth in Sec. 4

Analysis of Graphical Software Artifacts. The analysis of graphical software artifacts to support software engineering tasks has been common in recent years. Such tasks include mobile app testing [192, 259, 177, 50], developer/user behavior modelling [67, 43, 132], GUI reverse engineering and code generation [116, 269, 48, 82, 257, 84], analysis of programming videos [249, 213, 374, 278, 23, 389], and GUI understanding and verification [74, 390]. None of these works deal with finding duplicate video-based bug reports, which is our focus.

Detection of Duplicate Textual Bug Reports. Many research projects have focused on detecting duplicate textual bug reports [53, 57, 76, 77, 162, 170, 172, 204, 214, 223, 240, 267, 273, 287, 286, 300, 304, 319, 318, 323, 330, 331, 356, 348, 349, 397]. Similar to Tango, most of the proposed techniques return a ranked list of duplicate candidates [196, 76]. The work most closely related to Tango is by Wang *et al.* [348], who leveraged attached mobile app images to better detect duplicate textual reports. Visual features are extracted from the images (*e.g.*, representative colors), using computer vision, which are combined with textual features extracted from the text to obtain an aggregate similarity score. While this is similar to our work, Tango is intended to be applied to videos rather than single images and focuses on video-based bug reports alone, without any extra information such as bug descriptions.

2.1.2 Impact Analysis

The task of IA is concerned with determining the rippling effects a change to a software system may incur. It is composed of two parts, the first is the change entities of which there are several types, including feature requests, bug reports, or refactorings. The second being the impact set that contains all the entities that would be impacted by the initial change [222]. In this paper, we structure our dataset such that, for each IA task, there is a single change entity which is the initial method modified by a given change and the impact set is the set of affected methods.

Lehnert [222] constructed a taxonomy of different IA approaches illustrating the different underlying techniques that have been utilized for IA, ranging from program slicing [333, 135, 395, 208, 346, 308] and call graph construction [294, 305, 367, 39] to information retrieval [34, 71, 72, 344, 280, 355] and history mining [134, 371, 372, 380, 398]. Most of these approaches can be classified as coupling approaches, where a coupling metric is used as a proxy for how likely a change to one entity will impact another. For example, Briand *et al.* [59] introduced structural coupling metrics such as Coupling Between Objects (CBOs) where usage of methods or attributes between classes classified them as coupled or information-flow-based coupling where the number of method invocations (weighted by their parameter counts) between classes are used for IA. Dynamic coupling metrics take into account dynamic analysis, such as dynamic binding and polymorphism [36, 160]. Evolutionary and logical coupling [134, 398] use co-change occurrences to determine coupling. Poshyvanyk *et al.* [280] introduced the use of information retrieval on source code to determine a conceptual coupling metric using Latent Semantic Indexing (LSI) [108]. Huang *et al.* [180] are among the only to leverage Neural Networks to tackle the problem of IA. However, unlike our proposed technique, they rely on handcrafted features that are then used to train a neural network.

The most similar work to our own is work by Wang *et al.* [355] wherein they combine an LSI model with doc2vec [215], a canonical neural network that takes a bag of words and generates a rich vector representation. However, they focus on using change requests to find impacted methods whereas we strictly use method information for both the change and the impact set. Additionally, we leverage information from a software system's call graph to improve the performance of our technique.

As for the evaluation, most previous IA techniques for estimating impact sets have been evaluated on fewer than ten open-source projects. Our paper constructs an evaluation benchmark based on well-curated accurate impact sets sourced via manual labeling of changed lines in bug fixing commits from 25 software systems [169].

2.1.3 Neural Code Representation

Neural representation of text has been common place in natural language processing (NLP) since the advent of word2vec [255], with recent Transformer [343] based methods such as BERT [113] achieving state-of-the-art results across a multitude of tasks. Unlike natural language, programming languages can be represented in multiple ways. For instance, one useful representation that has been commonly used is the Abstract Syntax Tree (AST). White *et al.* [362] introduced a DL method that utilized source code and ASTs to perform code clone detection. Similarly, Alon *et al.* [29] introduced code2vec, which learned a dense vector representation of code snippets that were useful for tasks such as method renaming and retrieval. Code2vec relied on AST representations to generate these dense vector representations. Feng *et al.* [128] adapted the successful BERT model to function with bimodal code information, namely, code and the code's corresponding documentation. Using a masked token prediction [113] and replaced token detection [96] pretraining task followed by a downstream finetuning scheme, the authors were able to achieve high performance on many code-related benchmarks including code search. Similar to Alon *et al.*, Guo *et al.* [149] integrated code specific information, namely Data Flow Graphs (DFGs), into the training of a BERT-like model by introducing two additional pretraining objectives aside from the masked token prediction task. These two objectives involve predicting edge information of the DFGs and aligning the representations of the code and its DFG.

Guo *et al.* [148] introduced UnixCoder, a unified model for code understanding and generation leveraging different attention masks to accomplish several pretraining objectives, *i.e.*, masked language model, causal language modeling, and denoising. The authors also incorporated AST information, which through ablations, was illustrated to improve performance across multiple tasks including code search. Finally, they also define two tasks, including multi-modal contrastive learning and cross-modal generation, to utilize the multi-modal inputs and align code representations among programming languages based on code comments.

Although these transformer-based code representation models achieved state-of-the-art performance across multiple code-related tasks, they have not been applied to the task of IA. Our work is the first to evaluate three representative neural models, (*i.e.*, CodeBERT, GraphCodeBERT, and UniXCoder), on IA with/without incorporating additional contextual information related to the call graph of the entire software system. Next, we discuss the necessary background of these models to aid in the understanding of our approach description (Section 5.1), by focusing our discussion on the overarching architecture upon which they all rely – the Transformer [343].

2.1.4 Transformer Architecture

Originally, the Transformer architecture was proposed for the task of machine translation leveraging an encoder-decoder architecture [343]. However, the main advantage of the Transformer architecture is the usage of attention [40] without any recurrent mechanism, as was used in previous architectures for machine translation, such as Long-Short Term Memory networks [174]. More specifically, the Transformer's attention mechanism uses three vectors that represent keys, values, and queries, which are different token representations akin to an information retrieval setting.

In this setting, the query vector represents a word that is fed into the model, and the key and value vectors represent the "memory" of the model, *i.e.*, all the words that have been processed or generated previously. Key vectors that have a high dot product with the query vector have a stronger "memory" signal that focuses the attention of "head" with the keys' matching value vectors. Transformers leverage multiple types of attention, referred to as multi-head attention, wherein each head can specialize its attention for different things (*e.g.*, co-referencing, parts of speech, *etc.*). In essence, the attention mechanism can be thought of as a fuzzy dictionary look up with an input word being transformed into a query vector, and searches for keys in the memory (*i.e.*, other words in the sequence) and then a softmax function is used to "select" which previous word values to "pay attention" to and how much to weight them. This can be represented mathematically by:

$$A_{head} = softmax(\alpha Q_{head} \cdot K_{head}) \cdot V \tag{2.1}$$

where A_{head} is the attention head, and K, Q and V are the query, key and value vectors. The main benefit of this formulation is that it is completely differentiable allowing for gradient optimization to be applied. The information that is routed between the layers is represented by the V vector in the above equation. For CodeBERT and GraphCodeBERT, only the encoder portion of the Transformer is used whereas for UnixCoder, both encoder and decoder portions are used. However, during the embedding phase, only the encoder is used in UnixCoder.

2.1.5 Code Completion

We discuss the literature related to (i) code completion, (ii) techniques aimed at improving the generalizability of Transformers, and (iii) NLP studies aimed at investigating the extent to which Transformers can generalize to instances different from those seen during training.

2.1.6 Code Completion

Code completion has been studied extensively for several years in SE [171, 335, 167, 290, 60, 189, 185, 38, 165, 176, 268, 298, 299, 325, 326, 375, 85, 95, 37, 131, 270, 24]. It has seen many iterations going from techniques able to generate simple predictions such as method calls [251] to recent DL models able to predict multiple code statements [85, 95, 131, 271, 24].

Our work can be thought of as a research thrust continuation to that one of Ciniselli *et al.* [95]. In their work, they explored the applicability of Transformer models for the task of code completion, especially as the number of tokens to complete grows. They found a T5 architecture to perform fairly well on this task, reporting however a performance degradation as the number of tokens to predict grew. While Ciniselli *et al.* [95] study how a T5 model trained on a specific dataset can work with prediction tasks of different complexity, we study how models trained on different datasets featuring instances characterized by different lengths generalize to unseen lengths.

Hendrycks *et al.* [168] and Chen *et al.* [85] proposed a systematic evaluation for code generation tools using functional-correctness. However, their focus was on generating complete programs rather than completing existing code.

Fried *et al.* [131] investigated a novel infilling pretraining scheme for decoder-only Transformer architectures that allow them to use bidirectional context to complete code. They found this infilling scheme allows for models that achieve a higher code completion rate than left context only models for single and multi-line code completions. Our study is complementary to these since we focus on encoder-decoder models and on the generalization to code completion length, rather than general performance.

2.1.7 Methods to Improve Length Generalization

There has been a plethora of literature on different techniques to improve generalization of inference length of Transformer models. Neishi and Yoshinaga [264] demonstrated that replacing the positional encoding scheme in Transformers with a Recurrent Neural Network (RNN) improves machine translation performance on sentences longer than those seen during training. In a similar vein, Dai *et al.* [105] take inspiration from RNNs by adding a segment-level recurrence mechanism to improve performance on long sequences. Dubois *et al.* [119] introduced a separate location and content based attention to improve generalization to longer sequences than those seen during training. Newman *et al.* [265] showed that training models to predict the end of sequence (EOS) token significantly degraded a model's ability to generalize to longer sequences than those seen during training. Specifically, they found that the hidden states of models trained on predicting EOS tokens lead to stratification of the hidden state manifold by length and get clustered into *length attractors*, which are areas where the EOS token is given the highest-probability prediction. This causes a failure to generalize to longer sequences that are not present when omitting the prediction of the EOS token. Lastly, several works [203, 233, 283, 317, 321] introduced various modifications to the positional encoding schemes of Transformers to improve generalization to longer sequences not seen during training. Among those, we considered in our study the four described in Section 6.1 due their good representativeness of the different types of encoding schemas, namely Absolute Positional Encoding (APE) and Relative Positional Encoding (RPE). In addition, we considered the T5 model since Press *et al.* [283] also showed it had an ability to slightly generalize to longer sequence lengths than it had seen during training.

2.1.8 Evaluations of Length Generalization

The most similar work to ours (from the NLP field) is Rosendahl *et al.* [302]'s study on analyzing a variety of positional encoding schemes and their ability to generalize to longer machine translation sentences than those seen during training. Similar to other works [264, 283], they found that relative positional encodings are superior to absolute positional encodings for generalizing to longer sequences. Lake and Baroni [211] and Hupkes *et al.* [182]'s work focused on measuring the composability of language models. One type of composition was specific to generalization of sequence length and they both found current language models to be unable to perform well on such tasks.

Chapter 3

A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research

Software engineering (SE) research investigates questions pertaining to the design, development, maintenance, testing, and evolution of software systems. As software continues to pervade a wide range of industries, both open- and closed-source code repositories have grown to become unprecedentedly large and complex. This has resulted 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 canonical machine learning (ML) techniques to identify patterns and unique relationships within this data to automate or enhance many tasks typically performed manually by developers. Unfortunately, the process of implementing ML techniques can be a tedious exercise in careful feature engineering, wherein researchers experiment with identifying salient attributes of data that can be leveraged to help solve a given problem or automate a given task.

However, 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 set of training data. Given the immense amount of structured and unstructured data in software repositories that are likely to contain hidden patterns, DL techniques have ushered in advancements across a range of tasks in software engineering research including automatic program repair [339], code suggestion [145], defect prediction [353], malware detection [227], feature location [101], among many others [248, 347, 238, 362, 373, 150, 332, 242]. A recent report from the 2019 NSF Workshop on Deep Leaning & Software Engineering has referred to this area of work as Deep Learning for Software Engineering (DL4SE) [282].

The applications of DL to improve and automate SE tasks 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. In an effort to map and guide research at the intersection of DL and SE, we conducted a systematic literature review (SLR) to identify and systematically enumerate the synergies between the two research fields. As a result of the analysis performed in our SLR, we synthesize a detailed *research roadmap* of past work on DL techniques applied to SE tasks[1] (*i.e.*, DL4SE), complete with identified open challenges and best practices

[1]It should be noted that another area, known as Software Engineering for Deep Learning (SE4DL), which explores improvements to engineering processes for DL-based systems, was also identified at the 2019 NSF workshop. However, the number of papers we identified on this topic was small, and mostly centered around emerging testing techniques for DL models. Therefore, we reserve a survey on this line of research for future work.

for applying DL techniques to SE-related tasks and data. Additionally, we analyzed the impacts of these DL-based approaches and discuss some observed concerns related to the potential reproducibility and replicability of our studied body of literature.

We organize our work around five major 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 Abu-Mostafa [22], to aid in grounding the creation of our research roadmap and exploration of the DL4SE topic. Our overarching interest is to identify best practices and promising research directions for applying DL frameworks to SE contexts. Clarity in these respective areas will provide researchers with the tools necessary to effectively apply DL models to SE tasks. To answer our RQs, we created a taxonomy of our selected research papers that highlights important concepts and methodologies characterized by 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 details that are critical to the components of learning that are often omitted. Therefore, in addition to our taxonomy that describes how the components of learning have been addressed, we provide insight into components that are often omitted, alongside strategies for avoiding such omissions. As a result, this paper provides the SE community with important guidelines for applying DL models that address issues such as sampling bias, data snooping, and over- and under-fitting of models. Finally, we provide an online appendix with all of our data and results to facilitate reproducability and encourage contributions from the community to continue to taxonomize DL4SE research[2] [359, 357].

3.1 Research Question Synthesis

[2]http://wm-semeru.github.io/dl4se/

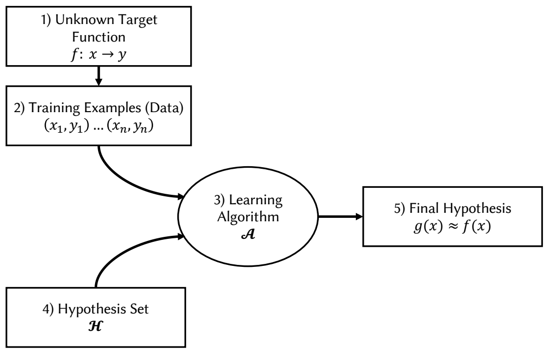

1) Unknown Target Function $f: x \to y$
2) Training Examples (Data) $(x_1, y_1) \dots (x_n, y_n)$
3) Learning Algorithm $\mathcal{A}$
4) Hypothesis Set $\mathcal{H}$
5) Final Hypothesis $g(x) \approx f(x)$

Figure 3.1: The Components of Learning

The synthesis and generation of research questions (RQs) is an essential step to any systematic literature review (SLR). In order to study the intersection of DL & SE, our intention was to formulate RQs that would naturally result in the derivation of a taxonomy of the surveyed research, establish coherent guidelines for applying DL to SE tasks, and address common pitfalls when implementing these complex models. Therefore, in order to properly accomplish these tasks and frame our review, we centered the synthesis of our RQs on the *components of learning* [22], which are illustrated in Figure 3.1. The components of learning are a formalization introduced by Abu-Mostafa [22] in an effort to enumerate the conditions for computational learning. By framing our top-level research questions according to these components, we can ensure that that analysis component of our literature review effectively captures the essential elements that any research project applying a *deep* learning-based solution should discuss, allowing for a thorough taxonomic inspection. Given that these components represent essential elements that should be described in any application of computational learning, framing our research questions in this manner allows for the extrapolation of observed trends related to those elements that are commonly included or omitted from the surveyed literature. This, in turn, allows us to make informed recommendations to the community related to the reproducibility of our surveyed work. In the remainder of this section, we detail how each of our top-level research questions were derived from the elements of learning. Note that, in order to perform our analysis to a sufficient level of detail, in addition to our top-level RQs, we also define several Sub-RQs that allow for a deeper analysis of some of the more complex elements of learning. We provide the full list of all the research questions at the end of this section.

3.1.1 The First Element of Learning: The Target Function

The first component of learning is an unknown *target function* ($f : x \to y$), which represents the relationship between two observed phenomenon x and y. The target function is typically tightly coupled to the task to which a learning algorithm is applied. By analyzing 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 applicable architectures. To capture the essence of this component of learning we formulated the following research question:

> ***RQ*$_1$*:*** *What types of SE tasks have been addressed by DL-based approaches?*

In understanding what SE tasks have been analyzed, we are 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 as well as the target users of these DL approaches, given the SE task they address.

3.1.2 The Second Element of Learning: The (Training) Data

The second component of learning is defined by the *data* that is presented to a given learning algorithm in order 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 providing further intuition about the potential features and applicable DL architectures that can be used to extract those features. Thus, in capturing this component of learning, we aimed to derive a taxonomy of the data used, how it was extracted and preprocessed, and how these relate to different DL architectures and SE tasks. This taxonomy captures relationships between data and the other elements of learning, illustrating effective (and ineffective) combinations for various SE-related applications. Our intention is that this information can inform researchers of effective combinations and potentially unexplored combinations of data/models/tasks to guide future work. Thus, our second RQ is formulated as follows:

> ***RQ_2:*** *How are software artifacts being extracted, prepared, and used in DL-based approaches for SE tasks?*

Given the multi-faceted nature of selecting, creating, and preprocessing data, we specifically examine three sub-research questions that explore the use of SE data in DL approaches in depth:

- ***RQ_{2a}:*** *What types of SE data are being used?*
- ***RQ_{2b}:*** *How is this data being extracted and pre-processed into formats that are consumable by DL approaches?*
- ***RQ_{2c}:*** *What type of exploratory data analysis is conducted to help inform model design and training?*

RQ_{2a} explores the different types of data that are being used in DL-based approaches. Given the plethora of different software artifacts currently stored in online repositories, it is important to know which of those artifacts are being analyzed and modeled. RQ_{2b} examines how data is being extracted and pre-processed into a format that a DL model can appropriately consume. The results of this RQ will enumerate the potential tools and techniques to mine different types of data for various DL applications within SE. Additionally, the representation of data is often dependent on the DL architecture and its ability to extract features from that representation, which lends importance to the discussion of the relationship between DL architectures and the data they process. RQ_{2c} investigates what type of exploratory data analysis is conducted to help inform model design and training. In order to perform effectively, DL models typically require large-scale datasets, and the quality of the learned hypothesis is a product of the quality of data from which the model learns. Therefore, since the quality of a given DL model is often directly associated with its data, we examined how research performed (or didn't perform) various analyses to avoid common data- related pitfalls recognized by the ML/DL community, including sampling bias and data snooping.

3.1.3 The Third & Fourth Elements of Learning: The Learning Algorithm & Hypothesis Set

Next, we jointly examine both the third and fourth components of learning, the *learning algorithm* and *hypothesis set*, in a single research question due to their highly interconnected nature. The learning algorithm is a mechanism that navigates the hypothesis set in order to best fit a given model to the data. The learning algorithm typically consists of a numeric process that uses a probability distribution over the input data to appropriately approximate the optimal hypothesis from the hypothesis set. The hypothesis set is a set of all hypotheses, based on the learning algorithm, to which the input can be mapped. 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. Taken together the learning algorithm and the hypothesis set are referred to as the learning model, thus, our third RQ is formulated as follows:

> $\boldsymbol{RQ}_3$**:** *What deep learning models are used to support SE tasks?*

Given the various types of DL model architectures and optimization techniques that may be applicable to various SE tasks, we examine RQ_3 through the lens of three sub-RQs, which address the aforementioned attributes of the learning model individually.

- $\boldsymbol{RQ}_{3a}$**:** *What types of model architectures are used to perform automated feature engineering of the data related to various SE tasks?*
- $\boldsymbol{RQ}_{3b}$**:** *What learning algorithms and training processes are used in order to optimize the models?*

- ***RQ*$_{3c}$*:** *What methods are employed to combat over- and under-fitting of the models?*

Firstly, RQ_{3a} explores the different types of model architectures that are used to perform automated feature engineering of different SE artifacts for various SE tasks. As part of the analysis of this RQ we also examine 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, RQ_{3b} examines the different learning algorithms and training processes that are used to optimize various DL models. As part of this analysis, we explore a 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, RQ_{3c} analyses the methods used to combat over- and under-fitting. Our intention with this RQ is to understand the specific methods (or lack thereof) used in SE research to combat over- or under-fitting, and the successes and shortcomings of such techniques.

3.1.4 The Fifth Element of Learning: The Final Hypothesis

Our fourth RQ addresses the component of learning known as the *final hypothesis*, which is the target function learned by the model that is used to predict aspects of previously unseen data points. In essence, in order to investigate this component of learning in the context of SE applications, we examine the *effectiveness* of the learned hypothesis as reported according to a variety of metrics across different SE tasks. Our intention with this analysis is to provide an indication of the advantages of certain data selection and processing pipelines, DL architectures, and training processes that have been successful for certain SE tasks in the past. Thus, our fourth RQ is formulated as follows:

> ***RQ*$_4$*:** *How well do DL tasks perform in supporting various SE tasks?*

Analyzing the effectiveness of DL applied to a wide range of SE tasks can be a difficult undertaking due to the variety of different metrics and evolving evaluation settings used in different contexts. Thus we examined two primary aspects of the literature as sub-RQs in order to provide a holistic illustration of DL effectiveness in SE research:

- ***RQ_{4a}*:** *What "baseline" techniques are used to evaluate DL models and what benchmarks are used for these comparisons?*
- ***RQ_{4b}*:** *How is the impact or automatization of DL approaches measured and in what way do these models promote generalizability?*

Understanding the metrics used to quantify the comparison between DL approaches is important for informing future work regarding methods for best measuring the efficacy of newly proposed techniques. Thus, RQ_{4a} explores trade-offs related to model complexity and accuracy. In essence, we examine applications of DL architectures through the lens of the *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" [289]. We attempted to answer this overarching RQ by first delineating the baseline techniques that are used to evaluate new DL models and identifying 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. RQ_{4b} examines how DL-based approaches are impacting the automatization of SE tasks through measures of their effectiveness and in what ways these models generalize to practical scenarios, as generalizability of DL approaches in SE is vital for their usability. For instance, if a state-of-the-art DL approach is only applicable within a narrowly defined set of circumstances, then there may still be room for improvement.

3.1.5 Analyzing Trends Across RQs

Our last RQ encompasses all of the components of learning by examining the extent to which our analyzed body of literature properly accounts for and describes each element of learning. In essence, such an analysis explores the potential *reproducibility* & *replicability* (or lack thereof) of DL applied to solve or automate various SE tasks. Therefore, our final RQ is formulated as follows:

> ***RQ_5:*** *What common factors contribute to the difficulty when reproducing or replicating DL4SE studies?*

Our goal with this RQ is to identify common DL components which may be absent or underdescribed in our surveyed literature. In particular, we examined both the *reproducibility* and *replicability* of our primary studies as they relate to the sufficient presence or absence of descriptions of the elements of computational learning. Reproducibility is defined as the ability to take the exact same model with the exact same dataset from a primary study and produce the same results [2]. Conversely, replicability is defined as the process of following the methodology described in the primary study such that a similar implementation can be generated and applied in the same or different contexts [2]. The results of this RQ will assist the SE community in understanding what factors are being insufficiently described or omitted from approach descriptions, leading to difficulty in reproducing or replicating a given approach.

Lastly, given the analysis we perform as part of RQ_5 we derive a set of guidelines that both enumerate methods for properly applying DL techniques to SE tasks, and advocate for clear descriptions of the various different elements of learning. 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 synthesized these steps to be flexible and generalizable. In addition, we provide 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 replicability/reproducibility.

3.1.6 Research Questions At-a-Glance

We provide our full set of research questions below:

- ***RQ*$_1$*: What types of SE tasks have been addressed by DL-based approaches?***
- ***RQ*$_2$*: How are software artifacts being extracted, prepared, and used in DL-based approaches for SE tasks?***
  - ***RQ*$_{2a}$*:*** *What types of SE data are being used?*
  - ***RQ*$_{2b}$*:*** *How is this data being extracted and pre-processed into formats that are consumable by DL approaches?*
  - ***RQ*$_{2c}$*:*** *What type of exploratory data analysis is conducted to help inform model design and training?*
- ***RQ*$_3$*: What deep learning models are used to support SE tasks?***
  - ***RQ*$_{3a}$*:*** *What types of model architectures are used to perform automated feature engineering of the data related to various SE tasks?*
  - ***RQ*$_{3b}$*:*** *What learning algorithms and training processes are used in order to optimize the models?*
  - ***RQ*$_{3c}$*:*** *What methods are employed to combat over- and under-fitting of the models?*
- ***RQ*$_4$*: How well do DL tasks perform in supporting various SE tasks?***
  - ***RQ*$_{4a}$*:*** *What "baseline" techniques are used to evaluate DL models and what benchmarks are used for these comparisons?*
  - ***RQ*$_{4b}$*:*** *How is the impact or automatization of DL approaches measured and in what way do these models promote generalizability?*
- ***RQ*$_5$*: What common factors contribute to the difficulty when reproducing or replicating DL studies in SE?***

Note: For this dissertation, I have decided to only include the results from the research questions that apply to this dissertation. Specifically, only RQ$_1$, RQ$_3$, and RQ$_4$.

3.2 RQ$_1$: What types of SE tasks have been addressed by DL-based approaches?

This RQ explores and quantifies the different applications of DL approaches to help improve or automate various SE tasks. Out of the 128papers we analyzed for this SLR, we identified 23 separate SE tasks where a DL-based approach had been applied. Figure 3.2 provides a visual breakdown of how many SE tasks we found within these 128primary studies across a 10 year period. Unsurprisingly, there was very little work done between the years of 2009 and 2014. However, even after the popularization of DL techniques brought about by results achieved by approaches such as AlexNet [210], it took the SE community nearly $\approx$ 3 years to begin exploring the use of DL techniques for SE tasks. This also coincides with the offering and popularization of DL frameworks such as PyTorch and TensorFlow. The first SE tasks to use a DL technique were those of *Source Code Generation*, *Code Comprehension*, *Source Code Retrieval & Traceability*, *Bug-Fixing Processes*, and *Feature Location*. Each of these tasks uses source code as their primary form of data. Source code served as a natural starting point for applications of DL techniques given the interest in large scale mining of open source software repositories in the research community, and relative availability of large-scale code datasets to researchers. Access to a large amount of data and a well-defined task is important for DL4SE, since in order for DL to have an effective application two main components are needed: i) a large-scale dataset 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 data associated with the predicted target variable.

It was not until 2017 that DL was used extensively in solving SE tasks as shown in Figure 3.2, with a large increase in the number of papers, more than doubling over the previous year from 9 to 25. During this period, the set of target SE tasks also grew to

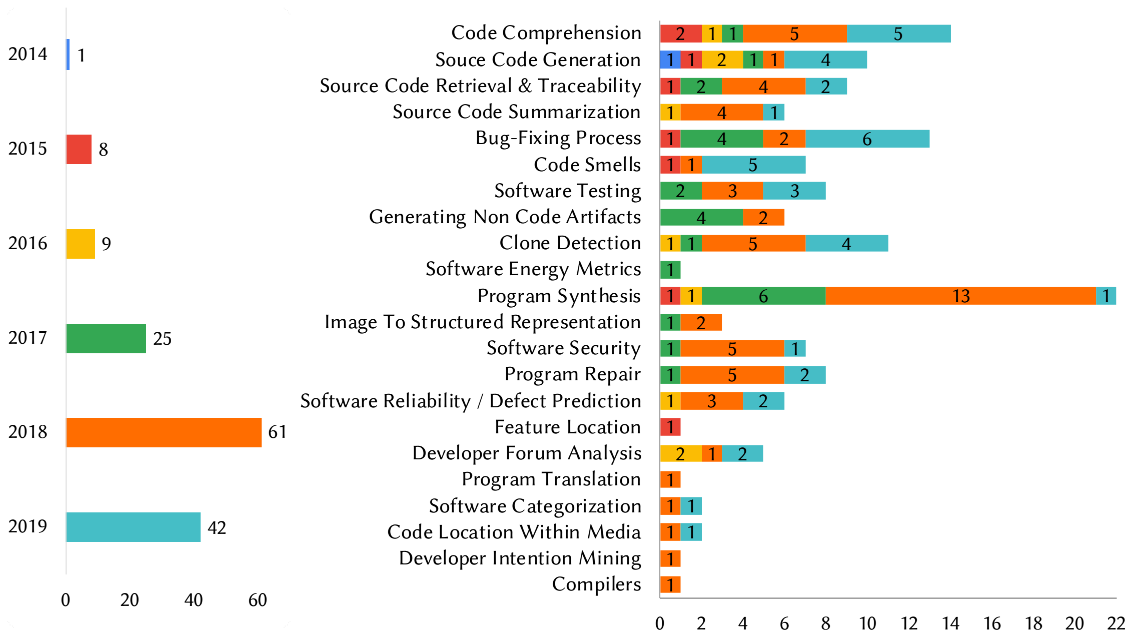


Figure 3.2: Papers published per year according to SE task. Note that a single paper can be associated with multiple SE Tasks.

become more diverse, including tasks such as *Code Smell Detection*, *Software Security*, and *Software Categorization*. However, there are three main SE tasks that have remained the most active across the years: *Code Comprehension*, *Source Code Retrieval & Traceability*, and *Program Synthesis*. The most popular of the three being Program Synthesis, composing a total of 22 papers out of the 128we 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 accessible, high quality training data, since one can train the DL model to generate a program, given a set of existing or curated specifications. The second largest reason is the clear mapping between training examples and a target programs. Given that it has proven difficult to engineer effective features that are capable to predict or infer programs, DL techniques are able to take advantage of the structured nature of this problem and extracting effective hierarchical representations. We display the full taxonomy in Table 3.1, which associates the cited primary study paired with its respective SE task.

Table 3.1: SE Task Taxonomy

SE Task	Papers
Code Comprehension	[225, 49, 277, 164, 216, 27, 26, 312, 260, 379, 256, 385, 178, 147]
Souce Code Generation	[197, 165, 364, 146, 82, 322, 291, 137, 266, 103]
Source Code Retrieval & Traceability	[86, 150, 145, 111, 212, 80, 27, 377, 369]
Source Code Summarization	[86, 347, 28, 337, 377, 178]
Bug-Fixing Process	[221, 261, 111, 212, 151, 153, 379, 239, 337, 281, 231, 181, 388]
Code Smells	[238, 26, 241, 337, 250, 329, 125]
Software Testing	[242, 143, 102, 314, 243, 387, 394]
Non Code Related Software Artifacts	[186, 310, 92, 94, 93, 179, 217]
Clone Detection	[229, 362, 306, 136, 236, 338, 391, 381, 385, 276, 66]
Software Energy Metrics	[301]
Program Synthesis	[138, 69, 386, 112, 320, 262, 292, 237, 41, 115, 65, 195, 89, 166, 35, 399, 313, 121, 122, 232, 275, 44]
Image To Structured Representation	[110, 82, 257]
Software Security	[157, 396, 81, 136, 159, 106, 127]
Program Repair	[54, 350, 159, 341, 239, 151, 153, 363]
Software Reliability / Defect Prediction	[353, 361, 354, 107, 173, 245]
Feature Location	[101]
Developer Forum Analysis	[83, 373, 234, 352, 147]
Program Translation	[90]
Software Categorization	[63, 64]
Compilers	[198]
Code Location Within Media	[272, 389]
Developer Intention Mining	[179]
Software Resource Control	[155, 217]

3.2.1 Results of Exploratory Data Analysis

In performing our exploratory data analysis, we derived two primary findings. First, it is clear that SE researchers apply DL techniques to a diverse set of tasks, as 70% of our derived SE task distribution was comprised of distinct topics that were evenly distributed ($\approx$ 3-5%). Our second finding is that the SE task was the most informative feature we extracted ($\approx 4.04B$), meaning that it provides the highest level of discriminatory power in predicting the other features (*e.g.*, elements of learning) related to a given study. In particular, we found that SE tasks had strong correlations to data (1.51B), the loss function used (1.14B) and the architecture employed (1.11B). This suggests that there are DL framework components that are better suited to address specific SE tasks, as authors clearly chose to implement certain combinations of DL techniques associated with different SE tasks. For example, we found that SE tasks such as program synthesis, source code generation and program repair were highly correlated with the preprocessing technique of tokenization.

Additionally, we discovered that the SE tasks of source code retrieval and source code traceability were highly correlated with the preprocessing technique of neural embeddings. When we analyzed the type of architecture employed, we found that code comprehension, prediction of software repository metadata, and program repair were highly correlated with both recurrent neural networks and encoder-decoder models. When discussing some of the less popular architectures we found that clone detection was highly correlated with siamese deep learning models and security related tasks were highly correlated with deep reinforcement learning models. Throughout the remaining RQs, we look to expand upon the associations we find to better assist software engineers in choosing the most appropriate DL components to build their approach.

3.2.2 Opportunities for Future Work

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[3] that DL has not been used for. Many of these topics have no *readily apparent* applicability for a DL-based solution. Still, some potentially interesting topics that seem well suited or positioned to benefit from DL-based techniques have yet to be explored by the research community or are underrepresented. Topics of this unexplored nature include software performance, program analysis, cloud computing, human aspects of SE, parallel programming, feature location, defect prediction and many others. Some possible reasons certain SE tasks have yet to gain traction in DL-related research is likely 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]https://conf.researchr.org/track/icse-2020/icse-2020-papers#Call-for-Papers

We believe that one possible research interest could be in the application of new DL models toward commonly explored SE tasks. For example, a DL model that is gaining popularity is the use of transformers, such as BERT, to represent sequential data [114]. It is possible that models such as this could be applied to topics related to clone detection and program repair. There is sufficient exploration in the use of DL within these topics to determine if these new architectures would be able to create a more meaningful representation of the data when compared to their predecessors.

Summary of Results for RQ$_1$:

Researchers have applied DL techniques to a diverse set of tasks, wherein *program synthesis*, *code comprehension*, and *source code generation* are the most prevalent. The SE task targeted by a given study is typically a strong indicator of the other details regarding the other components of learning, suggesting that certain SE tasks are better suited to certain combinations of these components. Our associative rule learning analysis showed a strong correlation amongst SE task, data type, preprocessing techniques, loss function used and DL architecture implemented, indicating that the SE task is a strong signifier of what other details about the approach are present. While there has been a recent wealth of work on DL4SE, there are still underrepresented topics that should be considered by the research community, including different topics in software testing and program analysis.

3.3 RQ$_3$: What Deep Learning Models are Used to Support SE Tasks?

In RQ$_2$ we investigated how different types of SE data were used, preprocessed, and analyzed for use in DL techniques. In this section, we shift our focus to the two key components of DL models: the *architecture* and the *learning algorithm*. The type of architecture selected for use in a DL application reveals key aspects of the types of features that researchers hope to model for a given SE task. Thus, we aim to empirically determine if certain architectures pair with specific SE tasks. Additionally, we aim to explore the diversity of the types of architectures used across different SE tasks and whether or not

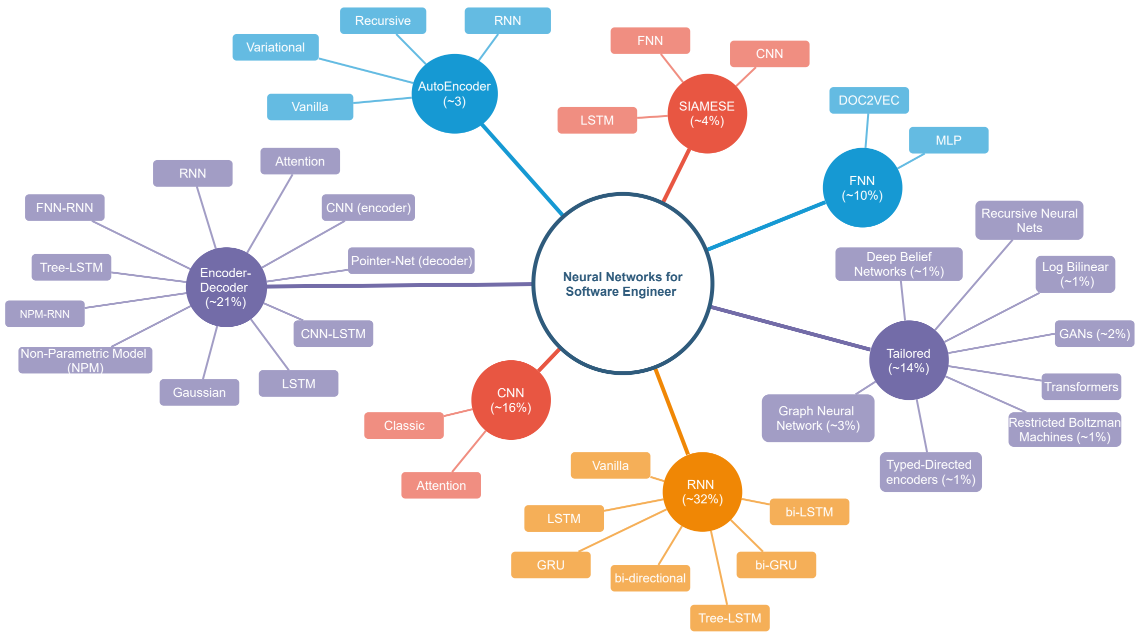


Figure 3.3: DL Model Taxonomy & Type Distribution

idiosyncrasies between architectures might be important when considering the specific SE task at hand. We also examined how various architectures are used in conjunction with different learning or optimization algorithms. Specifically, we aimed 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.

3.3.1 RQ_{3A}: What types of model architectures are used to perform automated feature engineering of the data related to various SE tasks?

In this section, we discuss the different types of DL models software engineers are using to address SE tasks. Figure 3.3 illustrates the various different DL architecture types that we extracted from our selected studies. We observe seven major architecture types: *Recurrent Neural Networks (RNNs)* ($\approx 45\%$), *Encoder-Decoder Models* ($\approx 22\%$), *Convolutional Neural Networks (CNNs)* ($\approx 21\%$), *Feed-Forward Neural Networks (FNNs)* ($\approx 13\%$), *AutoEncoders* ($\approx 8\%$), *Siamese Neural Networks* ($\approx 5\%$), as well as a subset of other custom, highly tailored architectures. We observe an additional level of diversity within each of

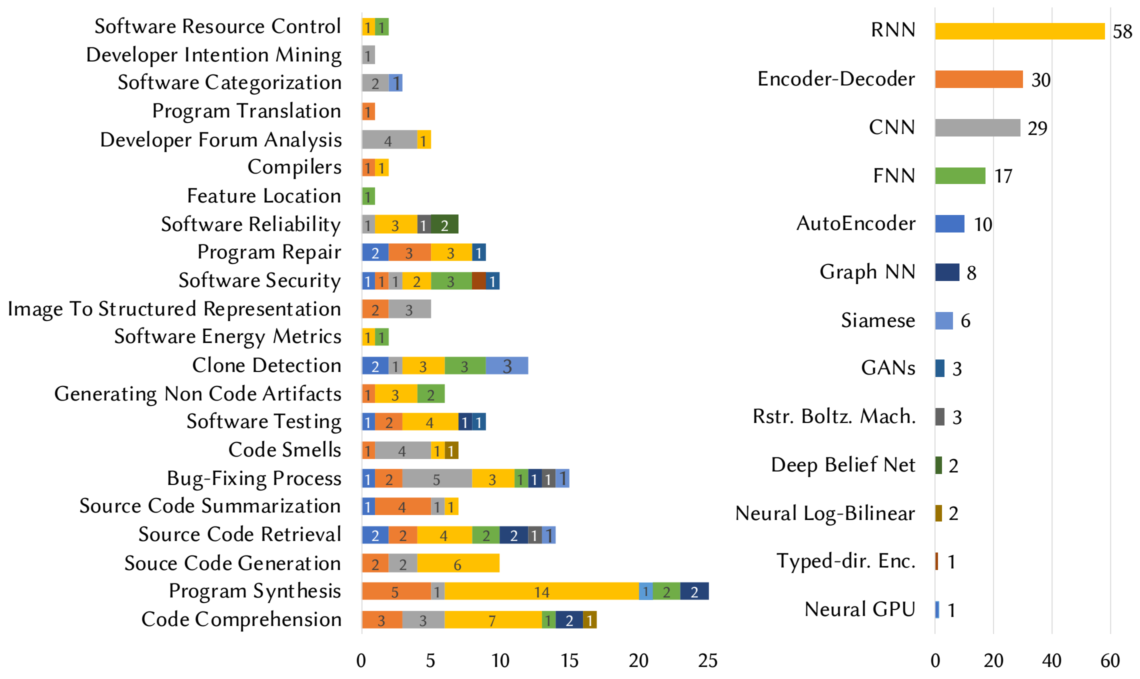


Figure 3.4: DL Architectures by the Task

these different types of architectures with Encoder-Decoder models illustrating the most diversity, followed by RNNs and the tailored techniques. The diversity of Encoder-Decoder models is expected, as this type of model is, in essence, a combination of two distinct model types, and is therefore extensible to a range of different combinations and hence architectural variations. The variance in RNNs is also logical. 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 [144], making them suitable to source code. Given that one of the most popular SE data types is source code which is inherently sequential data, the varied application of RNNS is expected. We also observe a number of architectures, such as Graph Neural Networks, that are specifically tailored for given SE tasks. For instances, graph-based neural networks have been adapted to better model the complex *structural* relationships between code entities.

Figure 3.4 delineates the prevalence of various different types of architectures according to the SE tasks to which they are applied. The popularity of our identified techniques closely mirrors their diversity. Examining this data, we find that RNNs are the most prevalent architectures, followed by Encoder-Decoder models, CNNs, and FNNs. The prevalence of RNNs is not surprising given the prevalence of source code as a utilized data type, as discussed above. The flexibility of Encoder-Decoder models is also expected as they excel at understanding and "translating" between parallel sets of sequential data, which is a common scenario in SE data (*e.g.*, code and natural language). 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. 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. CNNs serve as the most popular architectures for processing visual data, such as images, and hence are popular for visual SE data.

In addition to the prevalence, we observed certain trends between the DL architecture utilized and the corresponding SE task, as illustrated in Figure 3.4. 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. Likewise, SE tasks involving the use of images or media data have CNNs commonly applied to them.

We also observed some pertinent trends related to some of the less popular types of DL architectures, including: siamese networks, deep belief networks, GNNs and auto-encoders. While these architectures have only been applied to a few taszks it is important to note that they have only recently gained prominence and become accessible outside of ML/DL research communities. It is possible that such architectures can highlight orthogonal features of SE data that other architectures may struggle to observe. For example, the use of GNNs may better capture the structure or control flow of code or possibly the transition to different mobile screens within a mobile application. There may also be an opportunity for the use of Siamese networks in software categorization, as they have been shown to classify data into unique classes accurately based only on a few examples [306]. One notable absence from our identified architecture types is *deep reinforcement learning*, signaling its relative lack of adoption within the SE community. Deep reinforcement learning excels at modeling decision-making tasks. One could argue that deep reinforcement learning is highly applicable to a range of SE tasks that can be modeled as decisions frequently made by developers. This is a fairly open area of DL in SE that has not been sufficiently explored. The only type of SE task that had an application of Reinforcement Learning was related to program verification. In this paper the authors propose an approach that constructs the structural external memory representation of a program. They then train the approach to make multi-step decisions with an autoregressive model, querying the external memory using an attention mechanism. Then, the decision at each step generates subparts of the loop invariant [314].

In addition to the discussion around the DL architectures and their relations to particular SE tasks, it is also important to understand trends related to the *explicit* and *implicit* features extracted from these different architectures. As we discuss in Section for (RQ_{2B})[4], 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, neural embeddings and vectorizing latent representations. 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 function. The extraction of explicit and implicit features dramatically impacts a DL model's ability to represent a target function, which can be used to predict unobserved data points.

[4]Section omitted for dissertation

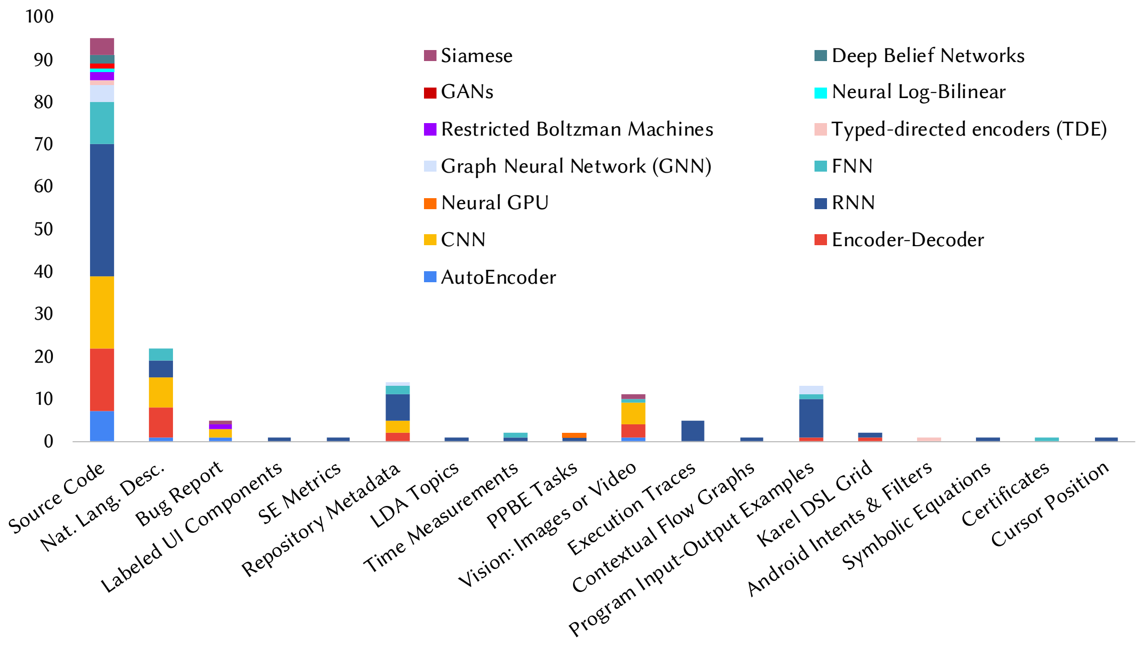


Figure 3.5: DL Architectures by Data Type

Figure 3.5 shows a breakdown of DL architectures by the type of data to which they are applied. This relationship between data and architecture is important since the architecture is partially responsible for the type of implicit features being extracted. For example, images and other visual data are commonly represented with a CNN. This is because CNNs are particularly proficient at modeling the spatial relationships of pixel-based data. We also discovered a strong correlation between RNNs and sequential data such as source code, natural language and program input-output examples. This correlation is expected due to RNNs capturing implicit features relating to the sequential nature of data. The models are able to capture temporal dependencies between text and source code tokens. Another correlation we observed was the use of CNNs for visual data or data which requires dimensionality reduction. This included the data types of images, videos, and even natural language and source code. CNNs have the ability to reduce features within long sequential data which makes them useful for tasks involving sentiment analysis or summarization. We also observed less popular combinations such as the use of deep belief networks (DBNs) for defect prediction [353]. Here, a DBN is used to learn semantic features of token vectors from a program's AST graph to better predict defects. A DBN can be a useful architecture in this situation due to its ability to extract the necessary semantic features from a tokenized vector of source code tokens. Those features are then used within their prediction models to drastically increase performance.

3.3.1.1 Results of Exploratory Data Analysis

In our exploratory data analysis, we found that SE tasks greatly influence the architecture adopted in an approach. The mutual information value between the features of a SE task and a DL architecture is $1.11B$. We also note that the SE research landscape has primarily focused on SE tasks that consist primarily of text-based data, including source code. This helps to explain why RNNs are used in $\approx 45\%$ of the papers analyzed in this SLR. The encoder-decoder architecture was also seen frequently ($\approx 22\%$ of papers), which generally makes use of RNNs.

3.3.1.2 Opportunities for Future Work

We were able to correlate different DL architectures with particular SE tasks and data types, primarily due to the fact that a given architecture is typically suited for a specific type of implicit feature engineering. However, there exists a fundamental problem in the ability of current research to validate and quantify these implicit features the model is extracting. This leads to decreased transparency in DL models, which in turn, can impact their practical applicability and deployment for real problems. Thus, there exists an open research problem related to being able to explain how a given model was capable of predicting the target function, *specifically* as it relates to SE data [383, 376, 32, 141, 368]. While interpretability is a broader issue for the DL community, insights into implicit feature engineering specifically for SE data would be beneficial for DL4SE work. It is necessary for developers to understand what complex hierarchical features are used by the model for this prediction. This could demystify their ability to correctly predict the output for a given input datum.

The ability to increase the interpretability of DL4SE solution also contributes toward the novel field of SE4DL, where SE methods are applied to the creation, sustainability and maintenance of DL software. The ability to interpret DL based solutions could help to create more complete testing suites for DL based software. This paradigm becomes even more important as new and innovative DL architectures are being developed. The SE community could take inspiration from the recent success in the NLP community on developing benchmarks for explainability [297]. Peeling back the "black box" nature of DL models should allow for an analysis on the integrity of the learning algorithms and an ability to better understand and build usable tools around their predictions.

Summary of Results for RQ$_{3A}$:

Our analysis revealed seven major types of DL architectures that have been used in work on DL4SE including: *Recurrent Neural Networks (RNNs)* ($\approx$ 45%), *Encoder-Decoder Models* ($\approx$ 22%), *Convolutional Neural Networks (CNNs)* ($\approx$ 21%), *Feed-Forward Neural Networks (FNNs)* ($\approx$ 13%), *AutoEncoders* ($\approx$ 8%), *Siamese Neural Networks* ($\approx$ 5%), as well as a subset of other custom, highly tailored architectures. RNNs and Encoder-Decoder models were both the most prevalent architecture used in our surveyed studies and the most diverse in terms of their varying configurations. We also discovered strong correlations between particular DL architectures to data types. For example, we found that architectures capable of capturing temporal differences within sequential data are used to study source code, natural language, repository metadata and program input-output examples. Likewise, architectures capable of capturing spatial and structural features from data have been used to study images, bug reports and program structures (ASTs, CFGs, etc.).

3.3.2 *RQ$_{3B}$*: 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 can also vary. To answer RQ$_{3B}$ we aimed to analyze the learning algorithms used in three primary ways: according to (i) the manner in which the weights of the model are updated, (ii) the overall error calculation, and (iii) by the optimization algorithm, which governs the parameters of the learning algorithm as training progresses. Learning algorithms that have been defined in ML/DL research are typically used in an "off-the-shelf" manner, without any alteration or adjustment, in the context of SE research. This is likely a result of researchers in SE being primarily interested in DL applications, rather than the intricacies of learning algorithms.

In terms of the process for adjusting weights, the most prevalent technique employed among our analyzed studies was the incorporation of the gradient descent algorithm. The breakdown of learning algorithms throughout our SLR are as follows: We found $\approx 76\%$ of the primary studies used some version of gradient descent to train their DL model. The remaining studies used gradient ascent $\approx 2\%$, or policy based learning $\approx 2\%$. Other studies did not explicitly specify their learning algorithm in the paper $\approx 18\%$. 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, such as reward policies or gradient ascent.

Our analysis reveled 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 $\approx 20\%$, which was most commonly paired with gradient descent algorithms. Other common loss functions that were used with gradient descent algorithms were negative log likelihood ($\approx 9\%$), maximum log likelihood ($\approx 9\%$), and cosine loss ($\approx 2\%$). There were a number of papers which did not provide any indication about the loss function within their learning algorithm ($\approx 42\%$). 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 [122]. 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.

Lastly, we examined the use of optimization algorithms to determine if there were any relevant patterns. We discovered 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 ($\approx 53\%$ of the time). However, we did analyze the papers that provided this information and identified four major optimization algorithms: Adagrad (3) , AdaDelta (3), RMSprop (11), and Adam (30). 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 [120]. A benefit of AdaGrad is that it removes the need for manual adjustment of the learning rates. However, the technique that AdaGrad calculates the degree by which it should adjust the parameters is using an accumulation the sum of the squared gradients. This can lead to summations of the gradient that are too large, often requiring an extremely small learning rate.

AdaDelta was formulated out of AdaGrad in order to combat the gradient size problem. 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 [384].

RMSprop is the next optimization algorithm, however, this algorithm has not been published or subjected to peer review. This algorithm was developed by Hinton *et al.* 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 dependent 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 [200].

3.3.2.1 Results of Exploratory Data Analysis

We found that the loss function is correlated to the chosen technique to combat overfitting with a mutual dependence of $1.00B$. However, the SE community omits reporting the loss function in $\approx 33\%$ of the papers we analyzed. Additionally, the loss function is correlated to SE task with a mutual dependence of $1.14B$

3.3.2.2 Opportunities for Future Work

A consequential highlight of our analysis of employed learning algorithms was the lack of data available from the primary studies. However, we did find a strong correlation between certain loss functions paired to specific learning algorithms. One aspect we believe could provide vital insight into the DL process is an analysis regarding how learning algorithms affect the parameters of the model for certain types of data. It would not only be important to study the type of data that learning algorithms and loss functions are associated with, but also what preprocessing techniques influence the learning algorithms and loss functions chosen. It is possible that some loss functions and learning algorithms are more efficient when applied to data that has been subjected to a particular preprocessing technique.

Finding the optimal pairing of loss function and learning algorithm for an architecture/data pair remains an open problem.

Summary of Results for RQ$_{3B}$:

Our analysis revealed four different techniques for updating the weights of the DL models, with the large majority making use of gradient descent. We found four major techniques that were utilized for calculating error, including cross entropy $\approx 20\%$, negative log likelihood $\approx 9\%$, maximum log likelihood $\approx 9\%$, and cosine loss $\approx 2\%$– with cross entropy being the most prevalent. Finally, we observed the use of four major optimization algorithms, including Adagrad (3) , AdaDelta (3), RMSprop (11), and Adam (30).

3.3.3 *RQ$_{3C}$*: What methods are employed to combat over- and underfitting?

Two potential problems associated with the use of any type of learning based approach, whether that be canonical machine learning or deep learning, are *overfitting* and *underfitting*. Both of these issues are related to the notion 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 training data extremely well, yet not being able to generalize to unseen data, and hence is a poor approximation of the actual target function to be learned [327]. Underfitting is typically described as the scenario in which a given model incurs a high error rate on a training set. This can occur when the model lacks the necessary complexity, is overly constrained, or has not had the sufficient training iterations to appropriately approximate the target function. For RQ$_{3C}$, we are primarily interested in the specific methods employed by researchers to combat these two problems in the context of SE tasks.

Figure 3.6 provides an overview of some general methods used to combat overfitting and underfitting[5]. The figure also addresses what parts of an ML/DL approach are affected

[5]Generated through an analysis of the following sources:
`https://elitedatascience.com/overfitting-in-machine-learning,`
`https://hackernoon.com/memorizing-is-not-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42,`

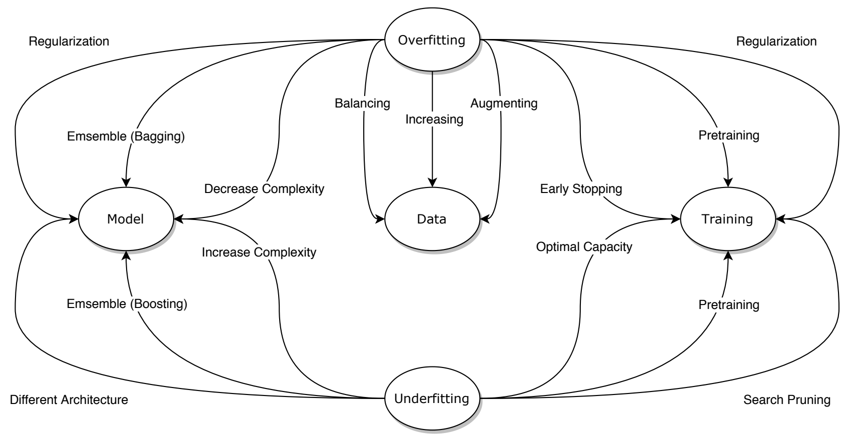


Figure 3.6: Overfitting and Underfitting Overview

by these techniques. As illustrated, there are three main types of regularization. The first regularizes the model, which includes things such as adding Dropout layers [316] or Batch Normalization [184]. 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 outlined in [22], 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 in parallel with a training procedure, to ensure that overfitting is not occurring. We see cross-validation in $\approx 11\%$ papers we analyzed. However, other potentially more effective techniques were seen less frequently.

We aimed to determine if a given SE task had any relationship with the methods employed to prevent over/under-fitting. Figure 3.7 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* ($\approx 32\%$) was the most commonly used regularization technique and

https://towardsdatascience.com/dont-overfit-how-to-prevent-overfitting-in-your-deep-learning-models-63274e552323, https://elitedatascience.com/bias-variance-tradeoff is used in a variety of DL approaches that address different SE tasks, followed by *data cleaning* ($\approx 14\%$), *L1/L2 regularization* ($\approx 15\%$), and *early stopping* ($\approx 13\%$). Dropout is one of the most popular regularization techniques because of its effectiveness and ease of implementation. Dropout randomly blocks signals from a node within a layer of the neural network with a certain probability determined by the researcher. This ensures that a single node doesn't overwhelmingly determine the classification of a given data point. We also observed a number of custom methods that were employed. These methods are configured to address the specific neural network architecture or data type being used. For example, in Sun et al. [320], they encourage diversity in the behavior of generated programs by giving a higher sampling rate to the perception primitives that have higher entropy over K different initial states. In Delvin et al. [112] they perform multiple techniques to combat overfitting which include the even sampling of the dataset during training and ensuring that each I/O grid of every example is unique. In addition to the aforementioned techniques, we found a subset of more unique approaches including the use of deep reinforcement learning instead of supervised learning [347], gradient clipping, lifelong learning [138], modification of the loss function [65], pretraining [347, 314], and ensemble modeling [186].

We also analyzed the relationships between techniques to combat over/under-fitting, and the underlying data type that a given model operated upon. We observed similar patterns 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 used. Figure 3.7 illustrates that the techniques used to combat overfitting do not have a strong association with the SE task. Therefore, we observe that a range of different techniques are applicable across many different contexts.

One of the more concerning trends that we observed is the number of papers categorized into the *Did Not Discuss* ($\approx 19\%$) category. Given the importance of combating overfitting when applying a DL approach, it is troublesome that so many primary studies did not mention these techniques. We hope that our observation of this trend signals the importance of recording such information.

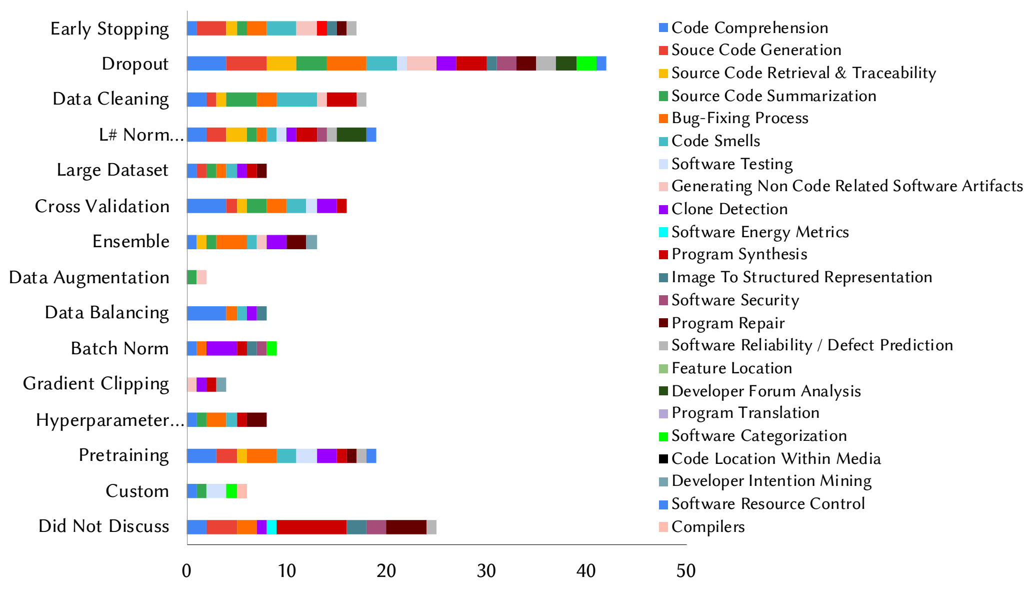


Figure 3.7: Overfitting Techniques per Task Type

Combating underfitting is a more complex process, as there aren't a well-defined set of standard techniques that are typically applied. One method that can be used to combat underfitting is searching for the optimal capacity of a given model. 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. One technique for achieving this optimal capacity include maximizing training time while monitoring performance on validation data. Other techniques include the use of a more complex model or a model better suited for the target function, which can be determined by varying the number of neurons, varying the number of layers, using different DL architectures, pretraining the model, and pruning the search space. From our SLR, the most commonly used underfitting techniques applied were pruning the search space of the model [195, 89], curriculum training [386, 292, 89] and pretraining [347, 314]. We found that only 6/128primary studies explicitly stated the implementation of an underfitting technique. This is a stark contrast to the number of studies implementing an overfitting technique, 97/128.

Surprisingly, more than $\approx 19\%$ of our studied papers did not discuss any techniques used to combat overfitting or underfitting. Combating this issue is a delicate balancing act, as attempting to prevent one can begin to cause the other if the processes are not carefully considered. 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 over- and under-fitting is crucial to the generalizability of the DL approach.

3.3.3.1 Opportunities for Future Research

Given the relative lack of discussion of techniques to combat the over- and under-fitting observed in our studies, it is clear that additional work is needed in order to better understand different mitigation techniques in the context of SE tasks and datasets, culminating in a set of shared guidelines for the DL4SE community. In addition, more work needs to be done to analyze and understand specialized techniques for SE tasks, data types, and architectures. Similar to preprocessing data, the implementation of over- and underfitting techniques are subject to a set of variables or parameters that define how they work. An in-depth analysis on how these details and parameters change depending on the type of SE task, architecture or data, is beyond the scope of this review. However, it would be useful to the SE community to provide some intuition about what combination of over- and underfitting techniques to apply and what parameters inherent to those techniques will likely lead to beneficial results.

Summary of Results for RQ$_{3C}$:

Our analysis shows that *dropout* ($\approx$ 32%) was the most commonly used method to combat over/under-fitting, followed by *data cleaning* ($\approx$ 14%), *L1/L2 regularization* ($\approx$ 15%), and *early stopping* ($\approx$ 13%). Nearly 1/4 of papers did not discuss such techniques.

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

In this RQ, we aim to explore the impact that DL4SE research has had through an examination of the effectiveness of the techniques proposed in our selected studies. we primarily analyze metrics on a per task basis and summarize the current state of benchmarks and baselines in DL4SE research.

3.4.1 RQ_{4A}: What "baseline" techniques are used to evaluate DL models and what benchmarks are used for these comparisons?

For RQ_{4A}, we examine the baseline techniques and evaluation metrics used for comparison in DL4SE work. In general, while we did observe the presence of some common benchmarks for specific SE tasks, we also found that a majority of papers self-generated their own benchmarks. We observed 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 [359, 357]. We found that many of the baseline approaches were canonical machine learning models or very simple neural networks. We suspect 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. As the field of DL4SE begins to mature, we expect to see a transition to evaluations that include comparisons against previously implemented DL approaches.

One somewhat troubling pattern that we observed is that many model implementations do not include a publicly available implementation of a DL approach. 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. The robustness of the results of such papers may suffer from

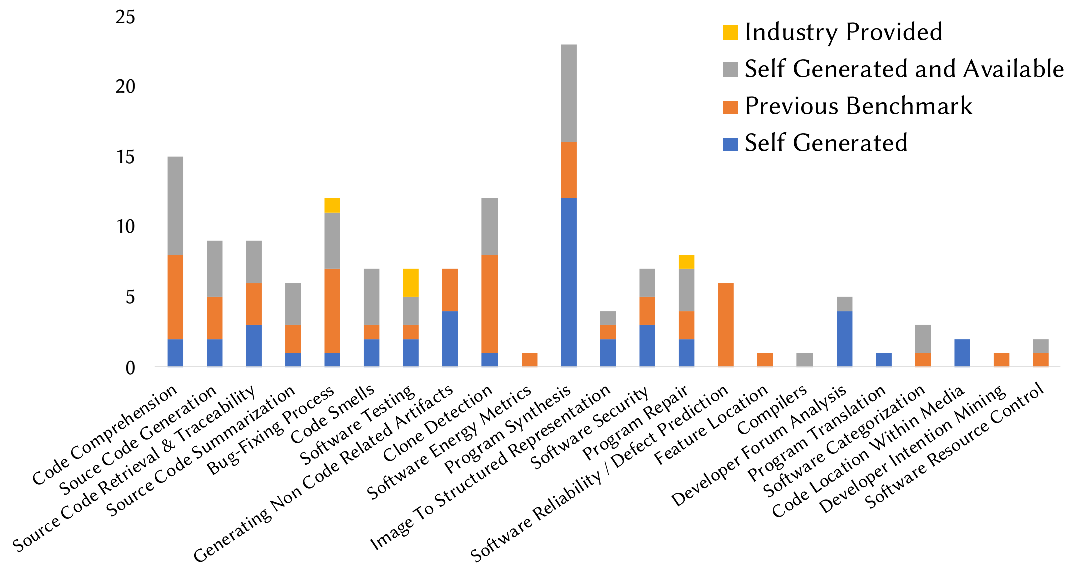


Figure 3.8: Benchmark Usage DL in SE

the fact that 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 attempting to examine purported improvements. While we expect that the set of existing, publicly available baselines will continue to improve over time, we also acknowledge the need for well-documented and publicly available baselines, and guidelines that dictate their proper dissemination.

Our online appendix [359, 357] includes a list of all the benchmarks and baselines used for each paper within our SLR. The diversity and size of this list of benchmarks prohibited its inclusion to the text of this manuscript. However, 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 number 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 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 3.8. 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 our online repository aids 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 applicable to DL techniques.

Although the use of previously established benchmarks was not common among our studies, we did observe a subset of benchmarks that were used multiple times within our primary studies. For the SE task of clone detection, we found that the dataset BigCloneBench [324] 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 [309] as a way to compare previous DL approaches that addressed defect prediction in software.

3.4.1.1 Opportunities for Future Research

The use of baselines and benchmarks in DL4SE studies, for the purpose of evaluation, is developing into a standard practice. However, there exists a need for replicable, standardized, baseline approaches that can be used for comparison when applying a new DL approach to a given SE task. The baseline implementations should be optimized for the benchmark used as data for a non-biased evaluation. This requires a thorough and detailed analysis of each published benchmark, within a specific SE task, for high quality data that does not suffer from sampling bias, class imbalance, etc. Many of the primary studies used a comparative approach for their evaluation, however, with a lack of standardized baselines the evaluations are dependent on how optimized the baseline is for a particular dataset. This can lead to confusing or conflicting results across SE tasks. We have started to see recent progress in the derivation and sharing of large-scale datasets with efforts such as the CodeXGlue dataset from Microsoft [247].

Summary of Results for RQ$_{4A}$:

Our analysis revealed a general lack of well-documented, reusable baselines or benchmarks for work on DL4SE. A majority of the baseline techniques utilized in the evaluations of our studied techniques were self-generated, and many are not publicly available or reusable. While a subset of benchmark datasets do exist for certain tasks, there is a need for well-documented and vetted benchmarks.

3.4.2 *RQ$_{4B}$*: How is the impact or automatization of DL approaches measured and in what way do these models promote generalizability?

Table 3.2 describes the distribution of metrics found in this SLR. In our analysis of utilized metrics within work on DL4SE, we observed that the metrics chosen are often related to the type of learning. Therefore, many of the supervised learning methods have metrics that analyze the resulting hypothesis, such as the accuracy ($\approx$ 46%), precision ($\approx$ 35%), recall ($\approx$ 33%), or F1 measure ($\approx$ 26%). In fact, classification metrics are reported in $\approx$ 74% of the papers. 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. The "other" category illustrated in Figure 3.2 is comprised of less popular metrics including: *likert scale*, *screen coverage*, *total energy consumption*, *coverage of edges*, *ROUGE*, *Jaccard similarity*, *minimum absolute difference*, *cross entropy*, *F-rank*, *top-k generalization*, *top-k model-guided search accuracy*, *Spearman's rank correlation coefficient*, and *confusion matrices.* In addition to the use of these metrics, we found a limited number of statistical tests to support the comparison between two approaches. These statistical tests included: *Kruskal's γ*, *macro-averaged mean cost-error*, *Matthew's correlation coefficient*, and *median absolute error.* Surprisingly, only approximately 5% of papers made use of statistical tests.

Table 3.2: Metrics Used for Evaluation

Measurement Type	Metrics	Studies
Alignment Scores	Rouge-L	[347]
	BLEU Score	[86, 186, 347, 146, 82, 159, 178, 322, 218, 137]
	METEOR Score	[86, 347]
Classification Measures	Precision	[339, 238, 229, 157, 212, 26, 362, 373, 353, 150, 111, 306, 164, 92, 94, 35, 145, 217] [234, 106, 179, 361, 338, 272, 257, 241, 250, 389, 63, 377, 231, 391, 381, 385, 256, 329, 352, 276, 181, 369, 147, 125]
	Recall	[238, 229, 157, 26, 373, 353, 150, 111, 306, 164, 92, 94, 35, 217, 234, 236, 106, 179, 361] [80, 338, 272, 241, 250, 389, 63, 377, 281, 231, 391, 379, 381, 385, 256, 329, 352, 276, 147, 369, 147]
	Confusion Matrix	[257, 281]
	Accuracy	[221, 261, 165, 229, 138, 110, 225, 157, 69, 49, 386, 112, 320, 262, 54, 292, 277, 83, 373, 237, 115, 111, 65, 195, 89] [166, 306, 164, 35, 82, 217, 234, 90, 136, 350, 27, 399, 313, 122, 232, 93, 179, 272, 159, 155, 337, 281, 322, 260, 153, 379, 64, 245, 363, 266, 103]
	ROC/AUC	[396, 306, 361, 92, 94, 136, 106, 361, 377, 312, 107, 173, 245, 66]
	F-Score	[238, 157, 396, 26, 373, 28, 353, 92, 94, 216, 217, 106, 179, 361, 338, 241, 250, 389, 63, 231, 312, 391, 381, 385, 329, 107, 352, 276, 147, 369, 125]
	Matthews Correlation	[92, 329]
	Scott-Knott Test	[245]
	Exam-Metric	[388]
	Clustering-Based	[127]
Coverage & Proportions	Rate or Percentages	[145, 102, 81, 151, 243, 379, 394, 354, 125]
	Coverage-Based	[242, 143, 243, 275, 387, 352]
	Solved Tasks	[314, 121, 361, 44, 151]
	Cost-Effectiveness	[245, 363]
	Total Energy or Memory Consumption	[301]
Distance Based	CIDER	[347, 399]
	Cross Entropy	[166]
	Jaccard Distance	[262]
	Model Perplexity	[364, 198, 107]
	Edit Distance	[198, 137]
	Exact Match	[137]
	Likert Scale	[186]
Approximation Error	Mean Absolute Error	[92, 93]
	Minimum Absolute Difference	[262]
	Macro-averaged Mean Absolute Error	[92, 94]
	Root Mean Squared Error	[310]
	Median Absolute Error	[93]
	Macro-averaged Mean Cost Error	[94]
Ranking	F-Rank	[145]
	Top K - Based	[313, 257, 239, 291, 181]
	Spearmans Rank	[338]
	MRR	[197, 86, 165, 212, 101, 145, 80, 181]
	Kruskal's γ	[396]
Timing	Time	[362, 41, 121, 122, 155]

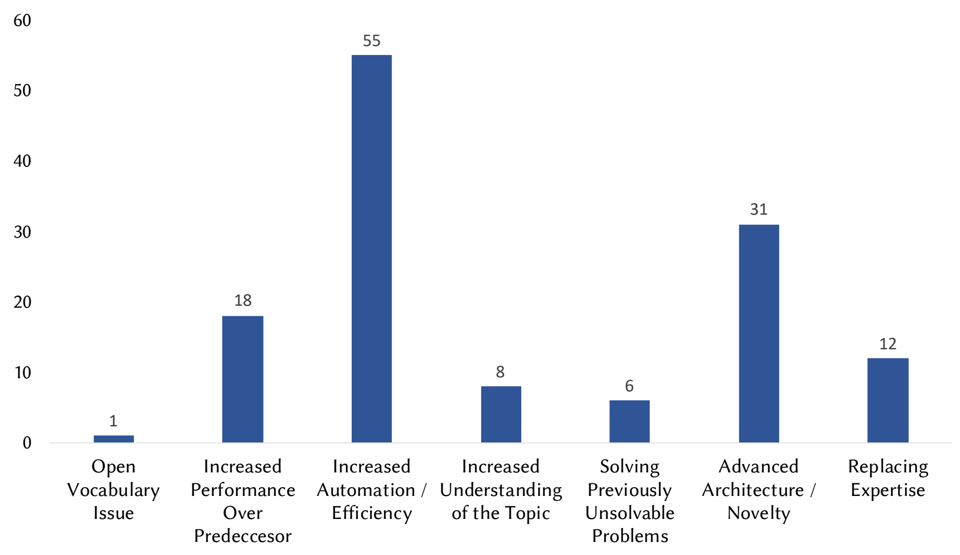


Figure 3.9: Impact of DL4SE

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 3.9. We found three primary objectives that the implementation of a DL model is meant to address: (i) in $\approx$ 43% of papers observed, a DL approach was implemented with the main goal of increasing automation efficiency; (ii) in $\approx$ 24% of the papers observed, a DL approach was implemented with the main goal of advancing or introducing a novel architecture; (iii) in $\approx$ 14% of the papers observed, a DL approach was implemented with the main goal of increasing performance over a prior technique.

In addition to the primary studies major objectives, we also observed that many papers did not analyze the complexity or generalizability of their implemented models. Thus to examine this further, we analyzed our primary studies through the lends of Occam's Razor and model efficiency. A valid question for many proposed DL techniques applied to SE tasks is whether the complexity of the model is worth the gains in effectiveness or automation for a given task, as recent research has illustrated [133]. This concept is captured in a notion known as *Occam's Razor*. Occam's Razor is defined by two major viewpoints: 1) "Given two models with the same generalization error, the simpler one should be preferred because simplicity is desirable" [117], 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" [117]. In the context of our SLR, we aimed to investigate the concept of Occam's Razor through analyzing whether authors considered technically "simpler" baseline techniques in evaluating their approaches. In Figure 3.10 we break the primary studies into four 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 perform both; 4) those that did not have any Occam's Razor consideration. Note that these are overlapping groupings and so the sum of papers exceeds the number of papers in our SLR.

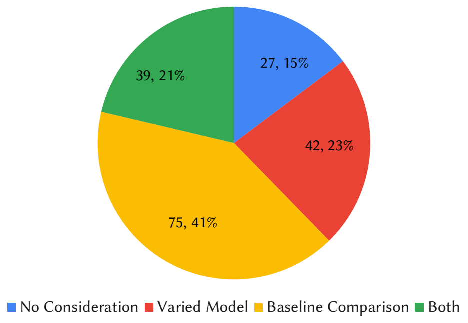


Figure 3.10: Evidence of Occam's Razor

Although a majority of the primary studies do consider Occam's Razor, there are still ≈ 16% of DL4SE studies that do not consider the principle. Without a consideration of Occam's Razor, it is possible that a canonical machine learning model or a simple statistical based approach could yield an optimal solution. This idea coincides with the findings mentioned by Fu et al. [133], who discovered that by applying a simpler optimizer to fine tune an SVM they were able to outperform a DL model applied to the same task. Fu *et al.* warn against the blind use of DL models without a thorough evaluation regarding whether the DL technology is a necessary fit for the problem [133]. Interestingly, in ≈ 23% 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. 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 took the form of a comparison to a canonical ML technique.

3.4.2.1 Results of Exploratory Data Analysis

Our exploratory data analysis revealed papers that combat overfitting, excluding data augmentation, omit ROC or AUC evaluations with a confidence level of ≈ 0.95. This metric is a common means by which comparisons to baseline approaches can be performed. Our exploratory data analysis of this RQ revealed that the automation impact is correlated to the SE task deduced from a mutual information of $0.71B$. This means that there is a subtle association between the SE task and the claimed automation impact of the approach.

3.4.2.2 Opportunities for Future Research

Throughout our analysis regarding the evaluation of DL4SE studies, it became apparent that there is a troubling lack of consistency of analysis, even within a given application to an SE task. Thus, there is an opportunity to develop guidelines and supporting evaluation infrastructure for metrics and approach comparisons. Such work would allow for clearer and more concise evaluations of new approaches, solidifying claims made from the results of a given evaluation. DL models are evaluated on their ability to be generalizable, this is normally accomplished through the use of a testing set, which the model has not been trained on. However, these testing sets can suffer from under representing certain class of data that can be found in the real world. More work is needed on evaluating the quality of testing sets and determining how representative they are when being used for evaluation. Having the limitations of DL approaches well document will create a greater opportunity for these DL solutions to be applied in the context of industry and real software development scenarios. Lastly, it would be advantageous for the research community to develop a methodology that could demonstrate the *need* for the complexity that DL offers when addressing a particular problem.

Summary of Results for RQ$_{4b}$:

Our analysis illustrates that a variety of metrics have been used to evaluate DL4SE techniques, with *accuracy* ($\approx$ 46%), *precision* ($\approx$ 35%), *recall* ($\approx$ 33%), and *F1-measure* ($\approx$ 26%) being the most prominent. In terms of claimed impact of our primary studies, the most claimed was *increased automation or efficiency*, followed by *advancing a DL architecture*, and *replacing human expertise.* We also found that most studies did consider the concept of Occam's Razor and offered a comparison to a conceptually simpler learning model.

3.5 Threats to Validity

Our systematic literature review was conducted according to the guidelines set forth by Kitchenham et al. [202]. However, as with any SLR our review does exhibit certain limitations primarily related to our search methodology and our data extraction process employed to build our paper taxonomy.

3.5.0.1 External Validity

Issues related to external validity typically concern the generalization of the conclusions drawn by a given study. A potential threat to the external validity to our systematic literature review is the search string and filtering process used to identify meaningful DL4SE studies. It is possible that our search string missed studies that should have been included in our review. This could be due to a missed term or combination of terms that may have returned more significant results. We mitigated this threat by testing a variety of DL and SE terms such as:

1. ("Deep Learning" OR "Neural Network")
2. ("Learning") AND ("Neural" OR "Automatic" OR "Autoencoder" OR "Represent")
3. ("Learning") AND ("Supervised" OR "Reinforcement" OR "Unsupervised" OR "Semi-supervised")
4. ("Learning" OR "Deep" OR "Neural" OR "Network")

5. ("Learning" OR "Deep" OR "Neural")
6. ("Artificial Intelligence" OR "Learning" OR "Representational" OR "Neural" OR "Network")

We evaluated these potential search strings through an iterative process as outlined by Kitchenham et al. The utilized search string "Deep" `OR` "Learning" `OR` "Neural" returned the greatest number of DL4SE studies. This search string was also chosen to limit selection bias since it "cast the widest net" in order to bolster completeness and limit potential biases introduced by more restrictive search strings. However, the trade-off was that it required a much more substantial effort to remove studies that were not applicable to DL4SE.

We also face potential selection bias of the studies to be included into our SLR. We attempt to mitigate this threat through the use of inclusion and exclusion criteria, which is predefined before the filtering process begins, and which we have listed in our online appendix [359, 357]. This criteria is also helpful in reducing the manual effort of filtering papers given our broad search string. We also perform snowballing as a means to mitigate selection bias. In this method, we collect all the references from the primary studies that passed our inclusion and exclusion criteria and determine if any of those references should be considered for the SLR.

Additionally, to further illustrate the generalizability of our paper sampling methodology, we perform a probability sampling to determine if we capture a significant proportion of DL4SE papers. We found that our expert sampling strategy captures a statistically significant number of studies, such that we are confident in our taxonomy's representation. Therefore, we feel that the trends highlighted in this review can be generalized to the entire body of DL4SE work.

Another potential threat to our systematic literature review consists of the venues chosen for consideration. For our review, we included the top SE, PL, and AI related conferences and journals. We included venues with at least a C CORE ranking [21], which helped us to determine venue impact. Although it is possible that not considering other conferences and journals caused us to miss some pertinent studies, we wanted to capture trends as seen in top SE, PL, and AI venues. Furthermore, we provide our current taxonomy and list of venues on our website, and welcome contributions from the broader research community. We intend for our online appendix to serve as a "living" document that continues to survey and categorize DL4SE research.

3.5.0.2 Internal Validity

A major contribution of this paper lies in our derived taxonomy that characterizes the field of DL4SE. To mitigate any mistakes in our taxonomy, we followed a process inspired by open coding in constructivist grounded theory [78] where each attribute classification of a primary study within our SLR was reviewed by at least three authors. However, while multiple evaluators limit the potential bias of this process, the classifications are still potentially affected by the collective views and opinions of the authors. Therefore, in effort to provide transparency into our process and bolster the integrity of our taxonomy, we have released all data extraction and classifications in our online repository [359, 357]. In releasing this information, authors of the works included in the SLR can review our classifications.

3.5.0.3 Construct Validity

One point of construct validity is the conclusions we draw at the end of each research question. In order to draw these conclusions, we performed an exploratory data analysis using rule association mining. In this analysis, we mine associations between attributes of DL solutions to SE tasks, which provides inspiration to further research why certain attributes show a strong or weak correlation.

Another threat to construct validity is our methodology for data synthesis and taxonomy derivation. To mitigate this threat we followed a systematic and reproducible process for analyzing the primary studies and creating a resulting taxonomy. To reduce the potential bias of data extraction, the authors developed and agreed upon a data extraction form to apply to each study. For our taxonomy, primary studies were categorized by three authors and refined by one additional authors. Through this process, we limit the number of individual mistakes in extracting the data and synthesizing the taxonomy.

3.6 Bibliographical Notes

The paper supporting the content described in this Chapter was written in collaboration with other members of the SEMERU group at William & Mary and a researcher from George Mason University. I have received permission from the publisher and co-authors to reprint sections of this work:

Watson, C., **Cooper, N.**, Palacio, D. N., Moran, K., & Poshyvanyk, D. (2020). A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. ACM Transactions on Software Engineering and Methodology (TOSEM), 31(2), 1-58.

Chapter 4

Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports

Many modern mobile applications (apps) allow users to report bugs in a graphical form, given the GUI-based nature of mobile apps. For instance, Android and iOS apps can include built-in screen-recording capabilities in order to simplify the reporting of bugs by end-users and crowd-testers [11, 19, 13]. The reporting of visual data is also supported by many crowd-testing and bug reporting services for mobile apps [11, 19, 20, 14, 13, 18, 12, 10, 9, 16], which intend to aid developers in collecting, processing, and understanding the reported bugs [51, 252].

The proliferation of sharing images to convey additional context for understanding bugs, *e.g.*, in Stack Overflow Q&As, has been steadily increasing over the last few years [263]. Given this and the increased integration of screen capture technology into mobile apps, developers are likely to face a growing set of challenges related to processing and managing app screen-recordings in order to triage and resolve bugs — and hence maintain the quality of their apps.

One important challenge that developers will likely face in relation to video-related artifacts is determining whether two videos depict and report the same bug (*i.e.*, detecting duplicate video-based bug reports), as it is currently done for textual bug reports [286, 287, 52]. When video-based bug reports are collected at scale, either via a crowdsourced testing service [11, 19, 20, 14, 13, 18, 12, 10, 9, 16] or by popular apps, the sizable corpus of collected reports will likely lead to significant developer effort dedicated to determining if a new bug report depicts a previously-reported fault, which is necessary to avoid redundant effort in the bug triaging and resolution process [52, 230, 51, 252]. In a user study which we detail later in this paper (Sec. 4.2.5), we investigated the effort required for experienced programmers to identify duplicate video-based bug reports and found that participants reported a range of difficulties for the task (*e.g.*, a single change of one step can result in two very similar looking videos showing entirely different bugs), and spent about 20 seconds of comprehension effort on average per video viewed. If this effort is extrapolated to the large influx of bug reports that could be collected on a daily basis [286, 287, 52, 76], it illustrates the potential for the excessive effort associated with video-based duplicate bug detection. This is further supported by the plans of a large company that supports crowd-sourced bug reporting (name omitted for anonymity), which we contacted as part of eliciting the design goals for this research, who stated that they anticipate increasing developer effort in managing video-based reports and that they are planning to build a feature in their framework to support this process.

To aid developers in determining whether video-based bug reports depict the same bug, this work introduces Tango, a novel approach that analyzes both visual and textual information present in mobile screen-recordings using tailored computer vision (CV) and text retrieval (TR) techniques, with the goal of generating a list of candidate videos (from an issue tracker) similar to a target video-based report. In practice, Tango is triggered upon the submission of a new video-based report to an issue tracker. A developer would then use Tango to retrieve the video-based reports that are most similar (*e.g.*, top-5) to the incoming report for inspection. If duplicate videos are found in the ranked results, the new bug report can be marked as a duplicate in the issue tracker. Otherwise, the developer can continue to inspect the ranked results until she has enough confidence that the newly reported bug was not reported before (*i.e.*, it is not a duplicate).

TANGO operates *purely* upon the graphical information in videos in order to offer flexibility and practicality. These videos may show the *unexpected behavior* of a mobile app (*i.e.*, a crash or other misbehavior) and the *steps to reproduce* such behavior. Two videos are considered to be *duplicates* if they show the same unexpected behavior (*a.k.a.* a bug) regardless of the steps to reproduce the bug. Given the nature of screen-recordings, video-based bug reports are likely to depict unexpected behavior towards the end of the video. TANGO attempts to account for this by leveraging the temporal nature of video frames and weighting the importance of frames towards the end of videos more heavily than other segments.

We conducted two empirical studies to measure: (i) the *effectiveness* of different configurations of TANGO by examining the benefit of combining visual and textual information from videos, as opposed to using only a single information source; and (ii) TANGO's ability to *save developer effort* in identifying duplicate video-based bug reports. To carry out these studies, we collected a set of 180 video-bug reports from six popular apps used in prior research [50, 75, 258, 259], and defined 4,860 duplicate detection tasks that resemble those that developers currently face for textual bug reports – wherein a corpus of potential duplicates must be ranked according to their similarity to an incoming bug report.

The results of these studies illustrate that TANGO's most effective configuration, which selectively combines visual and textual information, achieves 79.8% mRR and 73.2% mAP, an average rank of 1.7, a HIT@1 of 67.7%, and a HIT@2 of 83%. This means that TANGO is able to suggest correct duplicate reports in the top-2 of the ranked candidates for 83% of duplicate detection tasks. The results of the user study we conducted with experienced programmers demonstrate that on a subset of the tasks, TANGO can reduce the time they spend in finding duplicate video-based bug reports by $\approx$ 65%.

In summary, the main contributions of this work are the following:

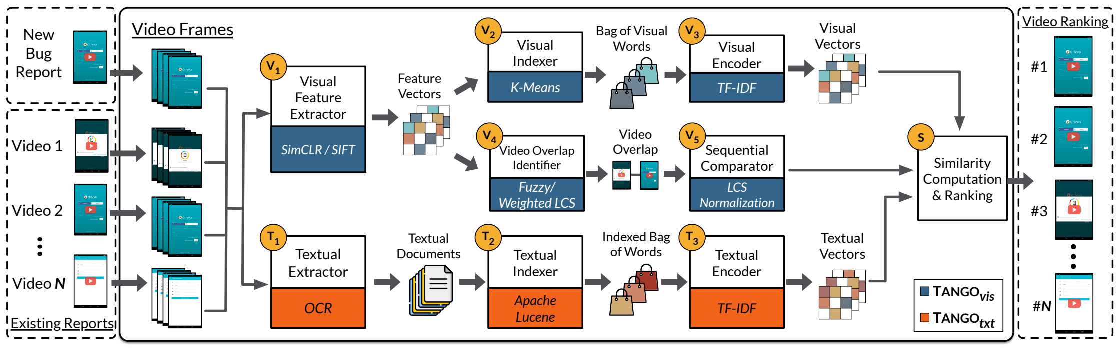


Figure 4.1: The TANGO approach for detecting duplicate video-based bug reports.

1. TANGO, a duplicate detection approach for video-based bug reports of mobile apps which is able to accurately suggest duplicate reports;

2. The results of a comprehensive empirical evaluation that measures the *effectiveness* of TANGO in terms of suggesting candidate duplicate reports;

3. The results of a user study with experienced programmers that illustrates TANGO's practical applicability by measuring its potential for *saving developer effort*; and

4. A benchmark (included in our online appendix [100]) that enables (i) future research on video-based duplicate detection, bug replication, and mobile app testing, and (ii) the replicability of this work. The benchmark contains 180 video-based bug reports with duplicates, source code, trained models, duplicate detection tasks, TANGO's output, and detailed evaluation results.

4.1 Tango's Approach

TANGO (deTecting duplicAte screeN recordinGs of sOftware bugs) is an automated approach based on CV and TR techniques, which leverages visual and textual information to detect duplicate video-based bug reports.

4.1.1 Tango Overview

Tango models duplicate bug report detection as an information retrieval problem. Given a new video-based bug report, Tango computes a similarity score between the new video and videos previously submitted by app users in a bug tracking system. The new video represents the query and the set of existing videos represent the corpus. Tango sorts the corpus of videos in decreasing order by similarity score and returns a ranked list of candidate videos. In the list, those videos which are more likely to show the same bug as the new video are ranked higher than those that show a different bug.

Tango has two major components, which we refer to as Tango$_{vis}$ and Tango$_{txt}$ (Fig. 4.1), that compute video similarity scores independently. Tango$_{vis}$ computes the *visual similarity* and Tango$_{txt}$ computes the *textual similarity* between videos. The resulting similarity scores are linearly combined to obtain a final score that indicates the likelihood of two videos being duplicates. In designing Tango$_{vis}$, we incorporated support for two methods of computing visual similarity — one of which is sensitive to the *sequential order* of visual data, and the other one that is not — and we evaluate the effectiveness of these two techniques in experiments described in Sec. 4.2-4.3.

The first step in Tango's processing pipeline (Fig. 4.1) is to decompose the query video, and videos from the existing corpus, into their constituent frames using a given sampling rate (*i.e.*, 1 and 5 frames per second - fps). Then, the Tango$_{vis}$ and Tango$_{txt}$ components of the approach are executed in parallel. The *un-ordered* Tango$_{vis}$ pipeline is shown at the top of Fig. 4.1, comprising steps V1-V3; the *ordered* Tango$_{vis}$ pipeline is illustrated in the middle of Fig. 4.1, comprising steps V1, V4, and V5; and finally, the Tango$_{txt}$ pipeline is illustrated at the bottom of Fig. 4.1 through steps T1-T3. Any of these three pipelines can be used to compute the video ranking independently or in combination (*i.e.*, combining the two Tango$_{vis}$ together, one Tango$_{vis}$ pipeline with Tango$_{txt}$, which we call Tango$_{comb}$, or all three – see Sec. 4.2.3). Next, we discuss these three pipelines in detail.

4.1.2 TANGO$_{vis}$: Measuring Unordered Visual Video Similarity

The *unordered* version of TANGO$_{vis}$ computes the visual similarity (S_{vis}) of video-based bug reports by extracting visual features from video frames and converting these features into a vector-based representation for a video using a Bag-of-Visual-Words (BoVW) approach [315, 188]. This process is depicted in the top of Fig. 4.1. The visual features are extracted by the *visual feature extractor* model ($\textcircled{V_1}$ in Fig. 4.1). Then, the *visual indexer* $\textcircled{V_2}$ assigns to each frame feature vector a visual word from a visual codebook and produces a BoVW for a video. The *visual encoder* $\textcircled{V_5}$, based on the video BoVW, encodes the videos using a TF-IDF representation that can be used for similarity computation.

4.1.2.1 Visual Feature Extraction

The *visual feature extractor* $\textcircled{V_1}$ can either use the SIFT [246] algorithm to extract features, or SimCLR [87], a recently proposed Deep Learning (DL) model capable of learning visual representations in an unsupervised, contrastive manner. TANGO's implementation of SimCLR is adapted to extract visual features from app videos.

The first method by which TANGO can extract visual features is using the Scale-Invariant Feature Transform (SIFT) [246] algorithm. SIFT is a state-of-the-art model for extracting local features from images that are invariant to scale and orientation. These features can be matched across images for detecting similar objects. This matching ability makes SIFT promising for generating features that can help locate duplicate images (in our case, duplicate video frames) by aggregating the extracted features. TANGO's implementation of SIFT does not resize images and uses the top-10 features that are the most invariant to changes and are based on the local contrast of neighboring pixels, with higher contrast usually meaning more invariant. This is done to reduce the number of SIFT features, which could reach at least three orders of magnitude for a single frame, and make the *visual indexing* $\textcircled{V_2}$ (through k-Means – see Sec. 4.1.2.2) computationally feasible.

The other technique that TANGO can use to extract features is SimCLR. In essence, the goal of this technique is to generate robust visual features that cluster similar images together while maximizing the distance between dissimilar images in an abstract feature space. This is accomplished by (i) generating sets of image pairs (containing one original image and one augmented image) and applying a variety of random augmentations (*i.e.*, image cropping, horizontal flipping, color jittering, and gray-scaling); (ii) encoding this set of image pairs using a base encoder, typically a variation of a convolutional neural network (CNN); and (iii) training a multi-layer-perceptron (MLP) to produce feature vectors that increase the cosine similarity between each pair of image variants and decrease the cosine similarity between negative examples, where negative examples for a given image pair are represented as all other images not in that pair, for a given training batch. TANGO's implementation of SimCLR employs the ResNet50 [163] CNN architecture as the base encoder, where this architecture has been shown to be effective [87].

To ensure that TANGO's *visual feature extractor* is tailored to the domain of mobile app screenshots, we trained this component on the entire RICO dataset [109], which contains over 66k Android screenshots from over 9k of the most popular apps on Google Play. Our implementation of SimCLR was trained using a batch size of $1,792$ and 100 epochs, the same hyperparameters (*e.g.*, learning rate, weight decay, *etc.*) recommended by Chen *et al.* [87] in the original SimCLR paper, and resized images to 224×224 to ensure consistency with our base ResNet50 architecture. The training process was carried out on an Ubuntu 20.04 server with three NVIDIA T4 Tesla 16GB GPUs.

The output of the *feature extractor* for SimCLR is a feature vector (of size 64) for each frame of a given video.

4.1.2.2 Visual Indexing

While the SimCLR or SIFT feature vectors generated by TANGO's *visual feature extractor* $\textcircled{\small V_1}$ could be used to directly compute the similarity between video frames, recent work has suggested that a BoVW approach combined with a TF-IDF similarity measure is more adept to the task of video retrieval [205]. Therefore, to transform the SimCLR or SIFT feature vectors into a BoVW representation, TANGO uses a *visual indexing process* $\textcircled{V_2}$.

This process produces an artifact known as a Codebook that maps SimCLR or SIFT feature vectors to "visual words" — which are discrete representations of a given image, and have been shown to be suitable for image and video recognition tasks [205]. The Codebook derives these visual words by clustering feature vectors and computing the centroids of these clusters, wherein each centroid corresponds to a different visual word.

The Codebook makes use of the k-Means clustering algorithm, where the k represents the diversity of the visual words, and thus can affect the representative power of this indexing process. TANGO's implementation of this process is configurable to 1k, 5k, or 10k for the k number of clusters (*i.e.*, the number of visual words - VW). 1k VW and 10k VW were selected as recommended by Kordopatis-Zilos *et al.* [205] and we included 5k VW as a "middle ground" to better understand how the number of visual words impacts TANGO's performance. A Codebook is generated only once for a given k, however, it must be trained before it can be applied to convert an input feature vector to its corresponding visual word(s). Once trained, a Codebook can then be used to map visual words from frame feature vectors without any further modification. Thus, we trained TANGO's six Codebooks, three for SIFT and three for SimCLR, using features extracted from $15,000$ randomly selected images from the RICO dataset [109]. We did not use the entire RICO dataset due to computational constraints of the k-means algorithm.

After the feature vector for a video frame is passed through the *visual indexing* process, it is mapped to its BoVW representation by a trained Codebook. To do this, the Codebook selects the closest centroid to each visual feature vector, based on Euclidean distance. For SIFT, this process may generate more than one feature vector for a single frame, due to the presence of multiple SIFT feature descriptors. In this case, TANGO assigns multiple visual words to each frame. For SimCLR, TANGO assigns one visual word to each video frame, as SimCLR generates only one vector per frame.

4.1.2.3 Visual Encoding

After the video is represented as a BoVW, the *visual encoder* $\textcircled{V_3}$ computes the final vector representation of the video through a TF-IDF-based approach [307]. The term frequency (TF) is computed as the number of visual words occurrences in the BoVW representation of the video, and the inverse document frequency (IDF) is computed as the number of occurrences of a given visual word in the BoVW representations built from the RICO dataset. Since RICO does not provide videos but individual app screenshots, we consider each RICO image as one document. We opted to use RICO to compute our IDF representation for two reasons: (i) to combat the potentially small number of frames present in a given video recording, and (ii) to bolster the generalizability of our similarity measure across a diverse set of apps.

4.1.2.4 Similarity Computation

Given two videos, Tango$_{vis}$ encodes them into their BoVW representations, and each video is represented as one visual TF-IDF vector. These vectors are compared using cosine similarity, which is taken as the *visual similarity* $\textcircled{S}$ of the videos ($S_{vis} = S_{BoVW}$).

4.1.3 Tango$_{vis}$: Measuring Ordered Visual Video Similarity

The *ordered* version of Tango$_{vis}$ considers the sequence of video frames when comparing two videos and is capable of giving more weight to common frames nearer the end of the videos, as this is likely where buggy behavior manifests. To accomplish this, the *feature vector extractor* $\textcircled{V_1}$ is used to derive descriptive vectors from each video frame using either SimCLR or SIFT. Tango determines how much the two videos overlap using an adapted longest common substring (LCS) algorithm $\textcircled{V_4}$. Finally, during the *sequential comparison process* $\textcircled{V_5}$, Tango calculates the similarity score by normalizing the computed LCS score.

4.1.3.1 Video Overlap Identification

In order to account for the sequential ordering of videos, TANGO employs two different versions of the longest common substring (LCS) algorithm. The first version, which we call fuzzy-LCS (f-LCS), modifies the comparison operator of the LCS algorithm to perform fuzzy matching instead of exact matching between frames in two videos. This fuzzy matching is done differently for SimCLR and SIFT-derived features. For SimCLR, given that each frame is associated with only a single visual word, the standard BoVW vector would be too sparse for a meaningful comparison. Therefore, we compare the feature vectors that SimCLR extracts from the two frames *directly* using cosine similarity. For SIFT, we utilize the BoVW vectors derived by the *visual encoder* $\textcircled{V_3}$, but at a per-frame level.

The second LCS version, which we call weighted-LCS (w-LCS), uses the same fuzzy matching that f-LCS performs. However, the similarity produced in this matching is then weighted depending on where the two frames from each video appeared. Frames that appear later in the video are weighted more heavily, since that is where the buggy behavior is typically occurring in a video-based bug report, and thus should be given more weight for duplicate detection. The exact weighting scheme used is $\frac{i}{m} \times \frac{j}{m}$, where i is the ith frame of video A, m is the # of frames in video A, j is the jth frame of video B, and n is the # of frames in video B.

4.1.3.2 Sequential Comparison

In order to incorporate the LCS overlap measurements into TANGO's overall video similarity calculation, the overlap scores must be normalized between zero and one ($[0, 1]$). To accomplish this, we consider the case where two videos overlap perfectly to be the upper bound of the possible LCS score between two videos, and use this to perform the normalization. For f-LCS, this is done by simply dividing by the # of frames in the smaller video since the *max* possible overlap that could occur is when the smaller video is a subsection in the bigger video, calculated as $overlap/min$ where *overlap* denotes the amount the two videos share in terms of their frames and *min* denotes the # of frames in the smaller of the two videos. For w-LCS, if the videos are different lengths, we align the end of the shorter video to the *end* of the longer video and consider this the upper bound on the LCS score, which is normalized as follows:

$$S_{w-LCS} = \frac{overlap}{\sum_{i=min}^{1} \frac{i}{min} \times \frac{max-i}{max}} \tag{4.1}$$

where S_{w-LCS} is the normalized similarity value produced by w-LCS, *overlap* and *min* are similar to the f-LCS calculation and *max* denotes the # of frames in the longer of the two videos. The denominator in Eq. 4.1 calculates the maximum possible overlap that can occur if the videos were exact matches, summing across the similarity score of each frame pair. Our online appendix contains the detailed f/w-LCS algorithms with examples [100].

4.1.3.3 Similarity Computation

f-LCS and w-LCS output the *visual similarity* Ⓢ score S_{f-LCS} and S_{w-LCS}, respectively. This can be combined with S_{BoVW} to obtain an aggregate visual similarity score: $S_{vis} = (S_{BoVW} + S_{f-LCS})/2$ or $S_{vis} = (S_{BoVW} + S_{w-LCS})/2$. We denote these Tango$_{vis}$ variations as B+f-LCS and B+w-LCS, respectively.

4.1.4 Determining the Textual Similarity between Videos

In order to determine the textual similarity between video-based bug reports, Tango leverages the textual information from labels, titles, messages, *etc.* found in the app GUI components and screens depicted in the videos.

Tango$_{txt}$ adopts a standard text retrieval approach based on Lucene [161] and Optical Character Recognition (OCR) [1, 17] to compute the textual similarity (S_{txt}) between video-based bug reports. First, a textual document is built from each video in the issue tracker ($\textcircled{T_1}$ in Fig. 4.1) using OCR to extract text from the video frames. The textual documents are pre-processed using standard techniques to improve similarity computation, namely tokenization, lemmatization, and removal of punctuation, numbers, special characters, and one- and two-character words. The pre-processed documents are indexed for fast similarity computation $\textcircled{\scriptsize T_2}$. Each document is then represented as a vector using TF-IDF and the index [307] $\textcircled{\scriptsize T_3}$.

In order to build the textual documents from the videos, Tango$_{txt}$ applies OCR on the video frames through the Tesseract engine [1, 17] in the *textual extractor* $\textcircled{\scriptsize T_1}$. We experiment with three strategies to compose the textual documents using the extracted frame text. The first strategy (all-text) concatenates all the text extracted from the frames. The second strategy (unique-frames) concatenates all the text extracted from unique video frames, determined by applying exact text matching (before text pre-processing). The third strategy (unique-words) concatenates the unique words in the frames (after pre-processing).

4.1.4.1 Similarity Computation

Tango computes the *textual similarity* (S_{txt}) in $\textcircled{\scriptsize S}$ using Lucene's scoring function [15] based on cosine similarity and document length normalization.

4.1.5 Combining Visual and Textual Similarities

Tango combines both the visual (S_{vis}) and textual (S_{txt}) similarity scores produced by Tango$_{vis}$ and Tango$_{txt}$, respectively ($\textcircled{\scriptsize S}$ in Fig. 4.1). Tango uses a linear combination approach to produce an aggregate similarity value:

$$S_{comb} = (1 - w) \times S_{vis} + w \times S_{txt} \tag{4.2}$$

where w is a weight for S_{vis} and S_{txt}, and takes a value between zero (0) and one (1). Smaller w values weight S_{vis} more heavily, and larger values weight S_{txt} more heavily. We denote this approach as Tango$_{comb}$.

Based on the combined similarity, TANGO generates a ranked list of the video-based bug reports found in the issue tracker. This list is then inspected by the developer to determine if a new video reports a previously reported bug.

4.2 TANGO's Empirical Evaluation Design

We empirically evaluated TANGO with two goals in mind: (i) determining how effective TANGO is at detecting duplicate video-based bug reports, when considering different configurations of components and parameters, and (ii) estimating the effort that TANGO can save developers during duplicate video bug detection. Based on these goals, we defined the following research questions (RQs):

RQ$_1$: *How effective is* TANGO *when using either visual or textual information alone to retrieve duplicate videos?*

RQ$_2$: *What is the impact of combining frame sequence and visual information on* TANGO*'s detection performance?*

RQ$_3$: *How effective is* TANGO *when combining both visual and textual information for detecting duplicate videos?*

RQ$_4$: *How much effort can* TANGO *save developers in finding duplicate video-based bug reports?*

To answer our **RQs**, we first collected video-based bug reports for six Android apps (Sec. 4.2.1), and based on them, defined a set of duplicate detection tasks (Sec. 4.2.2). We instantiated different configurations of TANGO by combining its components and parameters (Sec. 4.2.3), and executed these configurations on the defined tasks (Sec. 4.2.4). Based on standard metrics, applied on the video rankings that TANGO produces, we measured TANGO's effectiveness (Sec. 4.2.4). We answer **RQ$_1$**, **RQ$_2$**, and **RQ$_3$** based on the collected measurements. To answer **RQ$_4$** (Sec. 4.2.5), we conducted a user study where we measured the time humans take to find duplicates for a subset of the defined tasks, and estimated the time Tango can save for developers. We present and discuss the evaluation results in Sec. 4.3.

4.2.1 Data Collection

We collected video-based bug reports for six open-source Android apps, namely AntennaPod (APOD) [3], Time Tracker (TIME) [7], Token (TOK) [8], GNUCash (GNU) [5], GrowTracker (GROW) [6], and Droid Weight (DROID) [4]. We selected these apps because they have been used in previous studies [50, 75, 258, 259], support different app categories (finance, productivity, *etc.*), and provide features that involve a variety of GUI interactions (taps, long taps, swipes, *etc.*). Additionally, none of these apps are included as part of the RICO dataset used to train Tango's SimCLR model and Codebooks, preventing the possibility of data snooping. Since video-based bug reports are not readily available in these apps' issue trackers, we designed and carried out a systematic procedure for collecting them.

In total, we collected 180 videos that display 60 distinct bugs – 10 bugs for each app and three videos per bug (*i.e.*, three duplicate videos per bug). From the 60 bugs, five bugs (one bug per app except for DROID) are reported in the apps' issue trackers. These five bugs were selected because they were the only ones in the issue trackers that we were able to reproduce based on the provided bug descriptions. During the reproduction process, we discovered five *additional* new bugs in the apps not reported in the issue trackers (one bug each for APOD, GNU, and TOK, and two bugs for TIME) for a total of 10 confirmed real bugs.

The remaining 50 bugs were introduced in the apps through mutation by executing MutAPK [123], a mutation testing tool that injects bugs (*i.e.*, mutants) into Android APK binaries via a set of 35 mutation operators that were derived from a large-scale empirical study on *real* Android application faults. Given the empirically-derived nature of these operators, they were shown to accurately simulate real-world Android faults [123].

We applied MutAPK to the APKs of all six apps. Then, from the mutant list produced by the tool, we randomly selected 7 to 10 bugs for each app, and ensured that they could be reproduced and manifested in the GUI. To diversify the bug pool, we selected the bugs from multiple mutant operators and ensured that they affected multiple app features/screens.

When selecting the 60 bugs, we ensured they manifest graphically and were reproducible by manually replicating them on a specific Android emulator configuration (virtual Nexus 5X with Android 7.0 configured via Android Studio). For all the bugs, we screen-recorded the bug and the reproduction scenario. We also generated a textual bug report (for bugs that did not have one) containing the description of the unexpected and expected app behavior and the steps to reproduce the bug.

To generate the remaining 120 video-based bug reports, we asked two professional software engineers and eight computer science (CS) Ph.D. students to replicate and record the bugs (using the same Android emulator), based only on the textual description of the unexpected and expected app behavior. The participants have between 2 and 10 years of programming experience (median of 6 years).

All the textual bug reports given to the study participants contained *only* a brief description of the observed and expected app behavior, with *no specific reproduction steps.* We opted to perform the collection in this manner to ensure the robustness of our evaluation dataset by maximizing the diversity of video-based reproduction steps, and simulating a real-world scenario where users are aware of the observed (incorrect) and expected app behavior, and must produce the reproduction steps themselves.

We randomly assigned the bugs to the participants in such a way that each bug was reproduced and recorded by two participants, and no participant recorded the same bug twice. Before reproducing the bugs, the participants spent five minutes exploring the apps to become familiar with their functionality. Since some of the participants could not reproduce the bugs after multiple attempts (mainly due to bug misunderstandings) and some of the videos were incorrectly recorded (due to mistakes), we reassigned these bugs among the other participants, who were able to reproduce and record them successfully.

Our bug dataset consists of 35 crashes and 25 non-crashes, and include a total of 470 steps (397 taps, 12 long taps, 14 swipes, among other types), with an average of 7.8 steps per video. The average video length is $\approx$ 28 seconds.

4.2.2 Duplicate Detection Tasks

For each app, we defined a set of tasks that mimic a realistic scenario for duplicate detection. Each duplicate detection task is composed of a new video (*i.e.*, the new bug report, *a.k.a.* the query) and a set of existing videos (*i.e.*, existing bug reports in the issue tracker, *a.k.a.* the corpus). In practice, a developer would determine if the new video is a duplicate by inspecting the corpus of videos in the order given by TANGO (or any other approach). For our task setup, the corpus contains both duplicate and non-duplicate videos. There are two different types of duplicate videos that exist in the corpus: (i) those videos that are a duplicate of the query (the *Same Bug* group), and (ii) those videos which are duplicates of each other, but are not a duplicate of the query (the *Different Bug* group). This second type of duplicate video is represented by bug reports marked as duplicates in the issue tracker and their corresponding master reports [286, 318, 76]. Each non-duplicate video reports a distinct bug.

We constructed the duplicate detection tasks on a *per app* basis, using the 30 video reports collected for each app (*i.e.*, three video reports for each of the 10 bugs, for a total of 30 video reports per app). We first divided all the 30 videos for an app into three groups, each group containing 10 videos (one for each bug) created by one or more participants. Then, we randomly selected a video from one bug as the query and took the other two videos that reproduce the same bug as the *Same Bug* duplicate group (*i.e.*, the ground truth). Then, we selected one of the remaining nine bugs and added its three videos to the *Different Bug* duplicate group. Finally, we selected one video from the remaining eight bugs, and used these as the corpus' *Non-Duplicate* group. This resulted in a total of 14 distinct bug reports per task (two in the *Same Bug* group, three in the *Different Bug* group, eight in the *Non-Duplicate* group, and the query video). After creating tasks based on all the combinations of query and corpus, we generated a total of 810 duplicate detection tasks per app or $4,860$ aggregating across all apps.

We designed the duplicate detection setting described above to mimic a scenario likely to be encountered in crowd-sourced app testing, where duplicates of the query, other duplicates not associated with the query, and other videos reporting unique bugs, exist in a shared corpus for a given app. While there are different potential task settings, we opted not to vary this experimental variable in order to make for a feasible analysis that allowed us to explore more thoroughly the different Tango configurations.

4.2.3 Tango Configurations

We designed Tango$_{vis}$ and Tango$_{txt}$ to have different configurations. Tango$_{vis}$'s configurations are based on different visual feature extractors (SIFT or SimCLR), video sampling rates (1 and 5 fps), # of visual words (1k, 5k, and 10k VW), and approaches to compute video similarity (BoVW, f-LCS, w-LCS, B+f-LCS, and B+w-LCS). Tango$_{txt}$'s configurations are based on the same sampling rates (1 and 5 fps) and the approaches to extract the text from the videos (all-text, unique-frames, and unique-words). Tango$_{comb}$ combines Tango$_{vis}$ and Tango$_{txt}$ as described in Sec. 4.1.5.

4.2.4 Tango's Execution and Effectiveness Measurement

We executed each Tango configuration on the $4,860$ duplicate detection tasks and measured its effectiveness using standard metrics used in prior text-based duplicate bug detection research [286, 318, 76]. For each task, we compare the ranked list of videos produced by Tango and the expected duplicate videos from the ground truth.

We measured the *rank* of the first duplicate video found in the ranked list, which serves as a proxy for how many videos the developer has to watch in order to find a duplicate video. A smaller *rank* means higher duplicate detection effectiveness. Based on the *rank*, we computed the *reciprocal rank* metric: $1/rank$. We also computed the *average precision* of Tango, which is the average of the precision values achieved at all the cutting points k of the ranked list (*i.e.*, precision@k). Precision@k is the proportion of the top-k returned videos that are duplicates according to the ground truth. We also computed *HIT@k* (*a.k.a.* Recall Rate@k [286, 318, 76]), which is the proportion of tasks that are successful for the cut point k of the ranked list. A task is successful if at least one duplicate video is found in the top-k results returned by Tango. We report HIT@k for cut points k = 1-2 in this paper, and 1-10 in our online appendix [100].

Additionally, we computed the average of these metrics over sets of duplicate detection tasks: mean reciprocal rank (mRR), mean average precision (mAP), and mean rank (μ rank or μRk) per app and across all apps. Higher mRR, mAP, and HIT@k values indicate higher duplicate detection effectiveness. These metrics measure the overall performance of a duplicate detector.

We focused on comparing mRR values to decide if one Tango configuration is more effective than another, as we consider that it better reflects the usage scenario of Tango. In practice, the developer would likely stop inspecting the suggested duplicates (given by Tango) when she finds the first correct duplicate. This scenario is captured by mRR, through the *rank* metric, which considers only the first correct duplicate video as opposed to the entire set of duplicate videos to the query (as mAP does).

4.2.5 Investigating Tango's Effort Saving Capabilities

We conducted a user study in order to estimate the effort that developers would spend while manually finding video-based duplicates. This effort is then compared to the effort measurements of the best Tango configuration, based on *μ rank* and *HIT@k*. This study and the data collection procedure were conducted remotely due to COVID-19 constraints.

4.2.5.1 Participants and Tasks

One professional software engineer and four CS Ph.D. students from the data collection procedure described in Sec. 4.2.1 participated in this study. The study focused on APOD, the app that all the participants had in common from the data collection. We randomly selected 20 duplicate detection tasks, covering all 10 APOD bugs.

4.2.5.2 Methodology

Each of the 20 tasks was completed by two participants. Each participant completed four tasks, each task's query video reporting a unique bug. The assignment of the tasks to the participants was done randomly. For each task, the participants had to watch the new video (the query) and then find the videos in the corpus that showed the same bug of the new video (*i.e.*, find the duplicate videos). All the videos were anonymized so that the duplicate videos were unknown to the participants. To do this, we named each video with a number that represents the video order and the suffix "vid" (*e.g.*, "2_vid.mp4").

The corpus videos were given in random order and the participants could watch them in any order. To make the bug of the new video clearer to the participants, we provided them with the description of the unexpected and expected app behavior, taken from the textual bug reports that we generated for the bugs. We consider the randomization of the videos as a reasonable baseline given that other baselines (*e.g.*, video-based duplicate detectors) do not currently exist and the video-based bug reports in our dataset do not have timestamps (which can be used to give a different order to the videos). This is a threat to validity that we discuss in Sec. 4.4.

4.2.5.3 Collected Measurements

Through a survey, we asked each participant to provide the following information for each task: (i) the name of the first video they deemed a duplicate of the query, (ii) the time they spent to find this video, (iii) the number of videos they had to watch until finding the first duplicate (including the duplicate), (iv) the names of other videos they deemed duplicates, and (v) the time they spent to find these additional duplicates. We instructed the participants to perform the tasks without any interruptions in order to minimize inaccuracies in the time measurements.

Table 4.1: Effectiveness for the best Tango configurations that use either visual (SimCLR/SIFT) or textual (OCR&IR) information.

App	**Config.**	**mRR**	**mAP**	**μRk**	**HIT@1**	**HIT@2**
APOD	SIFT	64.6%	51.1%	3.0	47.7%	71.7%
	SimCLR	80.0%	66.8%	1.7	**68.1%**	82.6%
	OCR&IR	**80.8%**	**75.3%**	**1.5**	65.7%	**88.6%**
DROID	SIFT	66.3%	55.0%	2.5	49.1%	69.5%
	SimCLR	64.6%	59.2%	2.6	49.5%	61.7%
	OCR&IR	**67.9%**	**64.7%**	**2.3**	**52.0%**	**69.8%**
GNU	SIFT	66.1%	57.2%	2.2	47.4%	68.4%
	SimCLR	81.8%	75.1%	1.6	70.1%	85.3%
	OCR&IR	**84.5%**	**82.3%**	**1.4**	**72.2%**	**92.0%**
GROW	SIFT	56.0%	49.9%	3.0	36.5%	54.3%
	SimCLR	72.7%	68.8%	2.0	57.4%	75.6%
	OCR&IR	**76.8%**	**69.0%**	**1.9**	**63.6%**	**80.1%**
TIME	SIFT	49.2%	40.7%	3.3	26.7%	46.4%
	SimCLR	**74.8%**	**67.6%**	**2.3**	**63.7%**	**75.9%**
	OCR&IR	47.4%	37.7%	4.0	28.3%	44.4%
TOK	SIFT	39.0%	32.1%	4.4	17.0%	33.7%
	SimCLR	**77.7%**	**69.3%**	**1.6**	**60.6%**	**86.7%**
	OCR&IR	61.3%	53.3%	2.6	42.6%	60.7%
Overall	SIFT	56.9%	47.7%	3.1	37.4%	57.3%
	SimCLR	**75.3%**	**67.8%**	**1.9**	**61.6%**	**78.0%**
	OCR&IR	69.8%	63.7%	2.3	54.1%	72.6%

4.2.5.4 Comparing Tango and Manual Duplicate Detection

The collected measurements from the participants were compared against the effectiveness obtained by executing the best Tango configuration on the 20 tasks, in terms of *μ rank* and *HIT@k*. We compared the avg. number of videos the participants watched to find one duplicate against the avg. number of videos they would have watched had they used Tango.

4.3 Tango's Evaluation Results

4.3.1 RQ1: Using Only Visual or Textual Information

We analyzed the performance of Tango when using only visual or textual information exclusively. In this section, we present the results for Tango's best performing configurations. However, complete results can be found in our online appendix [100]. Table 4.1

shows the results for TANGO_{vis} and TANGO_{txt} when using SimCLR, SIFT, as the *visual feature extractor*, and OCR as the *textual extractor*. For simplicity, we use SimCLR, SIFT, and OCR&IR to refer to SimCLR-based TANGO_{vis}, SIFT-based TANGO_{vis}, and TANGO_{txt}, respectively. The best results for each metric are illustrated in bold on a per app basis. The results provided in Table 4.1 are those for the best parameters of the SimCLR, SIFT, and OCR&IR feature extractors, which are (BoVW, 5 fps, 1k VW), (w-LCS, 1 fps, 10k VW), and (all-text, 5 fps), respectively.

Table 4.1 shows that TANGO_{vis} is more effective when using SimCLR rather than SIFT across all the apps, achieving an overall mRR, mAP, avg. rank, HIT@1, and HIT@2 of 75.3%, 67.8%, 1.9, 61.6%, and 78%, respectively. SimCLR is also superior to OCR&IR overall, whereas SIFT performs least effectively of the three approaches. When analyzing the results per app, we observe that SimCLR is outperformed by OCR&IR (by 0.7% - 4% difference in mRR) for APOD, DROID, GNU and GROW; with OCR&IR being the most effective for these apps. SimCLR outperforms the other two approaches for TIME and TOK by more than 16% difference in mRR. The differences explain the overall performance of SimCLR and OCR&IR. SimCLR is more consistent in its performance compared to OCR&IR and SIFT. Across apps, the mRR standard deviation of SimCLR is 6.2%, which is lower than that for SIFT and OCR&IR: 11.1% and 13.9%, respectively. The trend is similar for mAP and avg. rank.

Since the least consistent approach across apps is TANGO_{txt} in terms of effectiveness, we investigated the root causes for its lower performance on TIME and TOK. After manually watching a subset of the videos for these apps, we found that their textual content was quite similar across bugs. Based on this, we hypothesized that the amount of vocabulary shared between duplicate videos (from the same bugs) and non-duplicate videos (across different bugs) affected the discriminatory power of Lucene-based TANGO_{txt} (see Sec. 4.1.4).

To verify this hypothesis, we measured the shared vocabulary of duplicate and non-duplicate video pairs, similarly to Chaparro *et al.*'s analysis of textual bug reports [76]. We formed unique pairs of duplicate and non-duplicate videos from all the videos collected

Table 4.2: Vocabulary agreement & effectiveness for the best TANGO$_{txt}$.

App	**Vocabulary agreement**			**mRR**	**mAP**
	V_d	V_{nd}	$\lvert V_d - V_{nd} \rvert$		
APOD	70.8%	37.9%	32.9%	80.8%	75.3%
DROID	73.9%	57.0%	16.9%	67.9%	64.7%
GNU	82.2%	58.6%	23.6%	84.5%	82.3%
GROW	67.0%	41.7%	25.4%	76.8%	69.0%
TIME	86.0%	86.3%	0.3%	47.4%	37.7%
TOK	69.6%	61.0%	8.6%	61.3%	53.3%
Overall	74.2%	56.7%	17.5%	69.8%	63.7%

for all six apps. For each app, we formed 30 duplicate and 405 non-duplicate pairs, and we measured the avg. amount of shared vocabulary of all pairs, using the vocabulary agreement metric used by Chaparro *et al.* [76]. Table 4.2 shows the vocabulary agreement of duplicate (V_d) and non-duplicate pairs (V_{nd}) as well as the mRR and mAP values of TANGO$_{txt}$ for each app. The table reveals that the vocabulary agreement of duplicates and non-duplicates is very similar for TIME and TOK, and dissimilar for the other apps. The absolute difference between these measurements (*i.e.*, $|V_d - V_{nd}|$) for TIME and TOK is 0.3% and 8.6%, while for the other apps it is above 16%. We found 0.94 / 0.91 Pearson correlation [130] between these differences and the mRR/mAP values.

The results indicate that, for TIME and TOK, the similar vocabulary between duplicate and non-duplicate videos negatively affects the discriminatory power of TANGO$_{txt}$, which suggests that for some apps, using only textual information may be sub-optimal for duplicate detection.

Answer for RQ$_1$: SimCLR performs the best overall with an mRR and HIT@1 of 75.3% and 61.6%, respectively. For 4 of 6 apps, OCR&IR outperforms SimCLR by a significant margin. However, due to issues with vocabulary overlap, it performs worse overall. SIFT is the worst-performing technique across all the apps.

4.3.2 RQ2: Combining Visual and Frame Sequence Information

To answer $\mathbf{RQ}_2$, we compared the effectiveness of the best configuration of Tango when using visual information alone (SimCLR, BoVW, 5fps, 1k VW) and when combining visual & frame sequence information (*i.e.*, B+f-LCS and B+w-LCS).

The results are shown in Table 4.3. Overall, using Tango with BoVW alone is more effective than combining the approaches; Tango based on BoVW achieves 75.3%, 67.8%, 1.9, 61.6%, and 78% mRR, mAP, avg. rank, HIT@1, and HIT@2, respectively. Using BoVW and w-LCS combined is the least effective approach. BoVW alone and B+f-LCS are comparable in performance. However, BoVW is more consistent in its performance across apps: 6.2% mRR std. deviation vs. 6.6% and 9.2% for B+f-LCS and B+w-LCS.

The per-app results reveal that B+w-LCS consistently is the least effective approach for all apps except for GROW, for which B+w-LCS performs best. After watching the videos for GROW, we found unnecessary steps in the beginning/middle of the duplicate videos, which led to their endings being weighted more heavily by w-LCS, where steps were similar. In contrast, BoVW and B+f-LCS give a lower weight to these cases thus reducing the overall video similarity.

The lower performance of B+f-LCS and B+w-LCS, compared to BoVW, is partially explained by the fact that f-LCS and w-LCS are more restrictive by definition. Since they find the longest common sub-strings of frames between videos, small variations (*e.g.*, extra steps) in the reproduction steps of the bugs may lead to drastic changes in similarity measurement for these approaches. Also, these approaches only find one common substring (*i.e.*, the longest one), which may not be highly discriminative for duplicate detection. In the future, we plan to explore additional approaches for aligning the frames, for example, by using an approach based on longest common sub-sequence algorithms [154] that can help align multiple portions between videos. Another potential reason for these results may lie in the manner that Tango combines visual and sequential similarity scores – weighting both equally. In future work, we plan to explore additional combination techniques.

Table 4.3: Effectiveness for the best TANGO$_{vis}$ configuration using either visual information (BoVW) or a combination of visual and frame sequence information (B+f-LCS and B+w-LCS).

App	Config.	mRR	mAP	μRk	HIT@1	HIT@2
APOD	B+f-LCS	79.3%	**67.8%**	**1.7**	66.2%	82.3%
	B+w-LCS	77.2%	65.5%	1.9	64.2%	80.1%
	BoVW	**80.0%**	66.8%	**1.7**	**68.1%**	**82.6%**
DROID	B+f-LCS	**64.8%**	**60.7%**	**2.6**	**50.2%**	61.6%
	B+w-LCS	63.7%	54.8%	2.7	48.9%	62.3%
	BoVW	64.6%	59.2%	**2.6**	49.5%	**61.7%**
GNU	B+f-LCS	**83.3%**	**75.6%**	**1.6**	**73.2%**	**85.6%**
	B+w-LCS	77.3%	65.7%	1.8	62.3%	83.6%
	BoVW	81.8%	75.1%	**1.6**	70.1%	85.3%
GROW	B+f-LCS	76.0%	70.2%	2.0	64.2%	75.2%
	B+w-LCS	**81.3%**	**75.0%**	**1.7**	**70.9%**	**82.8%**
	BoVW	72.7%	68.8%	2.0	57.4%	75.6%
TIME	B+f-LCS	70.4%	63.4%	**2.3**	54.4%	74.3%
	B+w-LCS	63.8%	58.5%	2.8	48.0%	64.9%
	BoVW	**74.8%**	**67.6%**	**2.3**	**63.7%**	**75.9%**
TOK	B+f-LCS	73.4%	65.6%	1.7	54.0%	82.5%
	B+w-LCS	59.2%	53.7%	2.6	37.9%	60.0%
	BoVW	**77.7%**	**69.3%**	**1.6**	**60.6%**	**86.7%**
Overall	B+f-LCS	74.5%	67.2%	2.0	60.4%	76.9%
	B+w-LCS	70.4%	62.2%	2.2	55.4%	72.3%
	BoVW	**75.3%**	**67.8%**	**1.9**	**61.6%**	**78.0%**

Answer for RQ$_2$: Combining ordered visual information (via f-LCS and w-LCS) with the orderless BoVW improves the results for four of the six apps. However, across all apps, BoVW performs more consistently.

4.3.3 RQ3: Combining Visual and Textual Information

We investigated TANGO's effectiveness when combining visual and textual information. We selected the best configurations of TANGO$_{vis}$ (SimCLR, BoVW, 5 fps, 1k VW) and TANGO$_{txt}$ (all-text, 5 fps) from **RQ$_1$** based on their mRR score and measured its performance overall and per app. We provide the results for the best weight we obtained for TANGO's *similarity computation and ranking* which was $w = 0.2$, *i.e.*, a weight of 0.8 and 0.2 on TANGO$_{vis}$ and TANGO$_{txt}$, respectively. These weights were found by evaluating different w values from zero (0) to one (1) in increments of 0.1 and selecting the one leading

Table 4.4: Effectiveness of the best TANGO_{comb}, TANGO_{vis}, and TANGO_{txt}.

App	Config.	mRR	mAP	μRk	HIT@1	HIT@2
APOD	TANGO_{comb}	**84.4%**	**75.8%**	**1.4**	**73.1%**	87.9%
	TANGO_{vis}	80.0%	66.8%	1.7	68.1%	82.6%
	TANGO_{txt}	80.8%	75.3%	1.5	65.7%	**88.6%**
DROID	TANGO_{comb}	**70.6%**	**66.7%**	**2.2**	**55.9%**	**71.0%**
	TANGO_{vis}	64.6%	59.2%	2.6	49.5%	61.7%
	TANGO_{txt}	67.9%	64.7%	2.3	52.0%	69.8%
GNU	TANGO_{comb}	**89.5%**	**84.7%**	**1.3**	**81.6%**	**94.2%**
	TANGO_{vis}	81.8%	75.1%	1.6	70.1%	85.3%
	TANGO_{txt}	84.5%	82.3%	1.4	72.2%	92.0%
GROW	TANGO_{comb}	**81.7%**	**75.4%**	**1.7**	**71.4%**	**82.5%**
	TANGO_{vis}	72.7%	68.8%	2.0	57.4%	75.6%
	TANGO_{txt}	76.8%	69.0%	1.9	63.6%	80.1%
TIME	TANGO_{comb}	59.6%	51.7%	2.8	40.2%	58.8%
	TANGO_{vis}	**74.8%**	**67.6%**	**2.3**	**63.7%**	**75.9%**
	TANGO_{txt}	47.4%	37.7%	4.0	28.3%	44.4%
TOK	TANGO_{comb}	69.8%	60.8%	2.0	50.9%	76.9%
	TANGO_{vis}	**77.7%**	**69.3%**	**1.6**	**60.6%**	**86.7%**
	TANGO_{txt}	61.3%	53.3%	2.6	42.6%	60.7%
Overall	TANGO_{comb}	**75.9%**	**69.2%**	**1.9**	**62.2%**	**78.5%**
	TANGO_{vis}	75.3%	67.8%	**1.9**	61.6%	78.0%
	TANGO_{txt}	69.8%	63.7%	2.3	54.1%	72.6%

to the highest overall mRR score. Complete results can be found in our online appendix [100].

Table 4.4 shows that the overall effectiveness achieved by TANGO_{comb} is higher than that achieved by TANGO_{txt} and TANGO_{vis}. TANGO_{comb} achieves 75.9%, 69.2%, 1.9, 62.2%, and 78.5% mRR, mAP, avg. rank, HIT@1, and HIT@2, on average. The avg. improvement margin of TANGO_{comb} is substantially higher for TANGO_{txt} (6.2%/5.5% mRR/mAP) than for TANGO_{vis} (0.7%/1.4% mRR/mAP).

Our analysis of the per-app results explains these differences. Table 4.4 reveals that combining visual and textual information substantially increases the performance over just using one of the information types alone, except for the TIME and TOK apps. This is because TANGO_{txt}'s effectiveness is substantially lower for these apps, compared to the visual version (see Table 4.1), due to the aforementioned vocabulary agreement. Thus, incorporating the textual information significantly harms the performance of TANGO_{comb}.

4.3.3.1 A Better Combination of Visual and Textual Information

The results indicate that combining visual and textual information is beneficial for most of our studied apps but harmful for a subset (TIME and TOK). This is because the textual information used alone, for TIME and TOK, leads to low performance. The analysis we made for Tango$_{txt}$ in **RQ**$_1$, revealed that the reason for the low performance of Tango$_{txt}$ lies in the similar amount of vocabulary overlap between duplicate and non-duplicate videos. Fortunately, based on this amount of vocabulary, we can predict the performance of Tango$_{txt}$ for new video-based bug reports as follows [76]. In practice, the issue tracker will contain reports marked as duplicates (reporting the same bugs) from previous submissions of bug reports as well as non-duplicates (reporting unique bugs). This information can be used to compute the vocabulary agreement between duplicates and non-duplicates, which can be used to predict how well Tango$_{txt}$ would perform for new reports.

Based on this, we defined a new approach for Tango, which is based on the vocabulary agreement metric from [76] applied on existing duplicate and non-duplicate reports. This approach dictates that if the difference of vocabulary agreement between existing duplicates and non-duplicates is greater than a certain threshold, then Tango should combine visual and textual information. Otherwise, Tango should only use the visual information because it is likely that the combination would not be better than using the visual information alone.

From the vocabulary agreement measurements shown in Table 4.2, we infer a proper threshold from the new Tango approach. This threshold may be taken as one value from the interval 8.6% - 16.9% (exclusive) because those are the limits that separate the apps for which Tango$_{txt}$ obtains low (TIME and TOK) and high performance (APOD, DROID, GNU, and GROW). For practical reasons, we select the threshold to be the middle value: $8.6 + (16.9 - 8.6)/2 = 12.8\%$. In future work, we plan to further evaluate this threshold on other apps.

We implemented this approach for Tango, using 0.2 as weight, and measured its effectiveness. This approach resulted in a mRR, mAP, avg. rank, HIT@1 and HIT@2 of 79.8%, 73.2%, 1.7, 67.7%, and 83%, respectively. The approach leads to a substantial improvement (*i.e.*, 3.9% / 4.1% higher mRR / mAP) over TANGO$_{comb}$ shown in Table 4.4.

The results mean that the best version of TANGO is able to suggest correct duplicate video-based bug reports in the first or second position of the returned candidate list for 83% of the duplicate detection tasks.

Answer for RQ$_3$: Combining visual and textual information significantly improves results for 4 of 6 apps. However, due to the vocabulary agreement issue, across all apps, this approach is similar in effectiveness to using visual information alone. Accounting for this vocabulary overlap issue through a selective combination of visual and textual information via a threshold, TANGO achieves the highest effectiveness: an mRR, mAP, avg. rank, HIT@1, and HIT@2 of 79.8%, 73.2%, 1.7, 67.7%, and 83%, respectively.

4.3.4 RQ4: Time Saved Discovering Duplicates

As expected, the participants were successful in finding the duplicate videos for all 20 tasks. In only one task, one participant incorrectly flagged a video as duplicate because it was highly similar to the query. Participants found the first duplicate video in 96.4 seconds and watched 4.3 videos on avg. across all tasks to find it. Participants also found all the duplicates in 263.8 seconds on avg. by watching the entire corpus of videos. This means they spent 20.3 seconds in watching one video on average.

We compared these results with the measurements taken from TANGO's best version (*i.e.*, selective TANGO) on the tasks the participants completed. TANGO achieved a 1.5 avg. rank, which means that, by using TANGO, they would only have to watch one or two videos on avg. to find the first duplicate. This would have resulted in $(4.3 - 1.5)/4.3 = 65.1\%$ of the time saved. In other words, instead of spending $20.3 \times 4 = 81.2$ seconds (on avg.) finding a duplicate for a given task, the participants could have spent $20.3 \times 1.5 = 30.5$

seconds. These results indicate the potential of Tango to help developers save time when finding duplicates.

Answer for RQ$_4$: On average, Tango's best-performing configuration can save 65.1% of the time participants spend finding duplicate videos.

4.4 Tango Limitations & Threats to Validity

Limitations. Tango has three main limitations that motivate future work.

The first one stems from the finding that textual information may not be beneficial for some apps. The best Tango version implements an approach for detecting this situation, based on a threshold for the difference in vocabulary overlap between duplicate and non-duplicate videos, which is used for selectively combining visual or textual information. This threshold is based on the collected data and may not generalize to other apps. Second, the visual TF-IDF representation for the videos is based on the mobile app images from the RICO dataset, rather than on the videos found in the tasks' corpus, as it is typically done in text retrieval. Additionally, we considered single images as documents rather than groups of frames that make up a video. These decisions were made to improve the generalization of Tango's visual features and to support projects that have limited training data. Third, differences in themes and languages across video-based bug reports for an application could have an impact in the performance of Tango. We believe that different themes (*i.e.*, dark vs. light modes) will not significantly impact Tango since the SimCLR model is trained to account for such color differences by performing color jittering and gray-scaling augmentations. However, additional experiments are needed to validate this conjecture. For different languages, Tango currently assumes the text in an application to be English when performing OCR and textual similarity. Therefore, its detection effectiveness where the bug reports display different languages (*e.g.*, English vs. French) could be negatively impacted. We will investigate this aspect in our future work.

Internal & Construct Validity. Most of the mobile app bugs in our dataset were introduced by MutAPK [123], and hence potentially may not resemble real bugs. However, MutAPK's mutation operators were empirically derived from a large-scale study of real Android faults, and prior research lends credence of the ability of mutants to resemble real faults [31]. We intentionally selected generated mutants from a range of operators to increase the diversity of our set of bugs and mitigate potential biases. Another potential threat is related to using real bugs from issue trackers that cannot be reproduced or that do not manifest graphically. We mitigated this threat by using a small, carefully vetted subset of real bugs that were analyzed by multiple authors before being used in our dataset. We did not observe major differences in the results between mutants and real bugs.

Another threat to validity is that our approach to construct the duplicate detection tasks does not take into account bug report timestamps, which would be typical in a realistic scenario [286], and timestamps could be used as a baseline ordering of videos for comparing against the ranking given by TANGO. The lack of timestamps stems from the fact that we were not able to collect the video-based bug reports from existing mobile projects. We mitigated this threat in our user study by randomizing the ordering of the corpus videos given to the participants. We consider this as a reasonable baseline for evaluating our approach considering that, to the best of our knowledge, (1) no existing datasets, with timestamps, are available for conducting research on video-based duplicate detection, and (2) no existing duplicate detectors work exclusively on video-based bug reports, as TANGO does.

External Validity. We selected a diverse set of apps that have different functionality, screens, and visual designs, in an attempt to mitigate threats to external validity. Additionally, our selection of bugs also attempted to select diverse bug types (crashes and non-crashes), and the duplicate videos were recorded by different participants. As previously discussed, there is the potential that TANGO's different parameters & thresholds may not generalize to video data from other apps.

4.5 Bibliographical Notes

The paper supporting the content described in this Chapter was written in collaboration with other members of the SEMERU group at William & Mary and a researcher from George Mason University. I have received permission from the publisher and co-authors to reprint sections of this work:

Cooper, N., Bernal-Cárdenas, C., Chaparro, O., Moran, K., & Poshyvanyk, D. (2021, May). It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 957-969). IEEE.

Chapter 5

Athena: Towards Improving Semantic Code Search with Knowledge Graphs

Modern software systems are long-lived, with extensive development and maintenance histories.

Projects experience churn in the developers or teams working on them, and can consist of millions of lines of code. Even understanding the potential cascading impacts of seemingly simple code changes can be a difficult proposition. As such, the premise of impact analysis (IA) is that a given change may result in *undesirable side effects*, such as a fault that leads to an erroneous program state, caused by unintended interactions between the changes and other parts of a software system [194]. Thus, the task of IA involves estimating an impact set of entities, usually classes or methods of a software system, from a given change to an entity, also usually a class or a method [55]. This process can be cognitively challenging for developers, as reasoning about complex interactions of a software system requires careful comprehension of large volumes of code. Given that many important engineering and maintenance tasks – such as bug fixing and refactoring – require code changes, they necessarily require IA as well. This process is typically performed *manually* by developers, but given its complexity, researchers have proposed a range of approaches for automating it.

The most recent approaches which aim to automate IA utilize coupling metrics between parts of a software system as a proxy for how likely a change in one part will affect another [59]. Popular coupling metrics include evolutionary/logical coupling [134, 398] where change histories are used to inform future changes or conceptual coupling [280, 296] where underlying textual similarities among code elements are used to estimate impact sets. This formulation has also been found to be better aligned with how developers perceive coupling [45]. However, making use of evolutionary information requires careful mining of change histories, and past work that measures textual similarity across code entities make use of textual representations that are limited in their ability to accurately assess the semantic similarity of terms and identifiers used in source code. Other techniques have made use of static [333, 135, 395, 208, 346, 308] or dynamic [36, 160] analyses to aid in impact set estimation. However, static analysis techniques tend to suffer from trade-offs in the soundness and completeness of their analyses [56] and running programs to collect dynamic information can be prohibitively expensive, particularly for large projects.

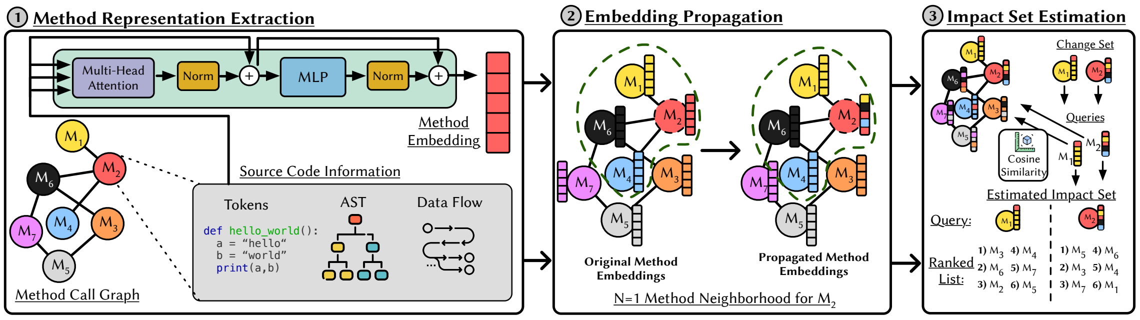
Recently, Deep Learning (DL) has seen great success in code generation and understanding tasks such as code search [183, 128, 149, 148], code completion [95, 85, 37, 131], bug fixing [153, 152, 337, 378], clone detection [362, 228, 236, 62, 276, 306, 338, 382, 385, 392], *etc.*. Generally, these approaches learn code semantics from unimodal code-only data or bimodal (comment, code pair) data and map code snippets to high dimensional vector representations that can be used to automate downstream code-related tasks. However, despite their success, none of these approaches have been applied to the task of IA. Therefore, in this paper, we aim to explore the potential of adapting three representative transformer-based [343] models [128, 149, 148] which utilize their corresponding semantically rich code representations to advance automated approaches for IA.

However, there are at least two issues that complicate the application of neural models for source code to the task of impact analysis. First, we currently lack extremely large-scale, vetted datasets or benchmarks that would allow a neural model to directly learn the task of impact analysis. This is due to the fact that deriving an IA dataset requires substantial human effort, as impact sets cannot be easily mined from software repositories without manual validation. Despite this, neural embeddings could be used to calculate semantic similarities between code entities to estimate conceptual coupling. However, while such an approach would already represent an advancement over past work, it ignores the context the code finds itself in, *i.e.*, how the code is used within a software system. Recent work has shown this information to be important to developers when determining relevance of retrieved code snippets for code search [183], a similar task to IA.

To overcome these limitations, and advance the task of automated impact analysis, we introduce Athena, a DL-based approach to IA that integrates *both* rich neural representations of code snippets and structural information about a given code snippet's relationships with other entities in a project using call graph information. Specifically, Athena builds a software system's call graph, where nodes represent methods and edges represent the call dependencies between methods. This call graph information is then used to aggregate the neural representations of the methods using an Embedding Propagation strategy inspired by work on Graph Convolutional Networks (GCNs) [201] that requires no additional training. In this way, Athena not only utilizes the local code semantics within the methods, but also utilizes the information of global call dependencies for the task of IA. More importantly, our technique does not require a specialized training procedure, and makes use of existing neural language models for code that can be trained in an unsupervised manner on massive datasets of code mined at scale.

Evaluating our proposed approach effectively also presents challenges. Existing IA benchmarks have been found to contain tangled commits resulting in inaccurate impact sets [169]. Additionally, they tend to be small, *i.e.*, less than ten software systems. Therefore, to evaluate Athena for the task of IA, we created a large-scale IA benchmark, called Alexandria, that leverages an existing dataset of fine-grained, manually untangled commit information from bug-fixes [169]. The benchmark consists of 910 commits across 25

Figure 5.1: Overview of the Workflow of the Athena Impact Analysis Approach



open-source Java projects, which we use to construct 4,405 IA tasks – where each task consists of a query method and a set of impacted methods. Using standard information retrieval metrics of mRR, mAP, and HIT@10, we find Athena significantly (based on statistical tests) improve over the neural semantic-only baseline without call graph information by 4.58%-5.32%, 3.85%-4.62%, and 5.67%-7.51%, respectively.

In aggregate, we make the following contributions:

- A new large-scale evaluation benchmark for impact analysis, called Alexandria, composed of 4,405 IA tasks from 910 commits of 25 open source software systems;
- Athena, a novel approach to automated impact analysis that utilizes call graph information as well as neural code semantics to estimate impact sets;
- A thorough set of ablations and qualitative analyses showing the improved performance is attributable to integrating call-graph information;
- A comprehensive online appendix [33] that contains the code for Athena, our IA benchmark Alexandria and our experimental infrastructure to allow for the replication of this work to help foster future work that aims to advance automated IA.

5.1 Athena's Approach

We formulate impact analysis as an information retrieval task where if a developer intends to modify a method (*i.e.*, query method) in a software system, Athena will return a ranked list of other methods being potentially impacted in descending order of likelihood. All methods but the query are used as the search corpus. Formally, for a software system S containing a set of methods $S = \{m_1, m_2, ..., m_n\}$, a change to one of the methods $m_i \in S$ triggers ATHENA to rank all other methods thus estimating the impact set.

Figure 5.1 provides an overview of the ATHENA approach. ATHENA first builds a call graph generator to aid in capturing the call dependencies among all methods across an entire software system in which the nodes represent the methods, and the edges represent the call dependencies between methods. All methods are then converted into method tokens which are processed by a state-of-the-art neural-based models, *e.g.*, CodeBERT, GraphCodeBERT and UniXcoder, fine-tuned on the code search task, to extract the method representations. Next, ATHENA analyzes the global call dependencies and propagates information from the neural embeddings of "neighbor" nodes in a method call graph to a given target method. Therefore, the initial representation vectors are updated based on a propagation strategy inspired by Graph Convolutional Networks (GCNs) [201] so that each method node incorporates the contextual information from its neighboring methods. The cosine similarity between the updated representations of a given query method and each method in the corpus is computed to obtain a final ranked list. We will now discuss each step of ATHENA in detail.

5.1.1 Software System Call Graph Generator

The first step of ATHENA is to build a call graph to capture method caller-callee dependencies across a software system. Call graphs can be generated either statically or dynamically. Given that dynamic approaches may result in incomplete information [370] as they usually analyze only a small set of execution paths (and also require extensive test suites and/or manual construction of scenarios for execution trace collection), ATHENA uses static analysis to build more complete call-graphs in a more efficient manner. Since existing tools such as DOOP [58] and WALA [129] only analyze the call dependencies within a given package but ignore the dependencies across packages, we created our own tool for generating call graphs that covers all call dependencies, even across packages, without requiring tests and only using the code files of a system as input.

Algorithm 1 provides an overview of our call graph generation process. For each `.java` production source file of the repository, Athena identifies all methods, imported packages, and all method invocation statements by using the Tree-Sitter [61] library, which builds a concrete syntax tree for each file and supports searching for various patterns in the tree. The type of the return values and the method arguments are not parsed to save time and ensure scalability of the approach. Next, we resolve each method invocation statement in each file to obtain the index of the caller method and the information needed for finding the callee by analyzing the file in a bottom-up manner using the Tree-sitter query syntax. To handle the inheritance relationship between classes, if the callee is not found in the class based on the given file path with the class name, the method will be further searched in its extended class. We use both the method name and # arguments (rather than the complete signature) to identify each method. Thus, some overloaded methods cannot be uniquely identified. Therefore, the edges are added between the caller and each of the overloaded callee methods if they have the same method name and # arguments. For nested method calls, only the outermost call is resolved and the others are discarded as return value types are not resolved.

By using our tool, a static directed call graph $G = (V, E)$ is constructed, where V is the set of method nodes identified by the method index and its content and E is the set of edges representing the method invocation relationships. Each edge in E is a pair of (caller, callee) indices. The method content is directly attached to each method node instead of using only partial information, such as method signatures as in previous techniques, to help facilitate the process of method representation extraction.

5.1.2 Method Representation Extraction

To extract method representations, all the methods in the software system are first preprocessed to be converted into method tokens. Since the three neural-based models we

Algorithm 1: Call Graph Generation from Repository

```
Input: A software repository $R$
Output: A call graph representing method invocations in the project
1  $V \leftarrow \varnothing, I \leftarrow \varnothing, C \leftarrow \varnothing, E \leftarrow \varnothing$
2  foreach $file \in R$ do
3      if $file \notin$ test files then
           /* Identify all methods, imported classes/packages, and calls in
              the file                                                   */
4          $methods, imports, calls = fileContents(file)$
5          $V = V \cup methods, I = I \cup imports, C = C \cup calls$
6      end
7  end
8  foreach $file \in R$ do
9      foreach $call \in calls$ do
           /* Identify the caller index, and the import module, class name,
              method name, # arguments of callee                         */
10         $caller_{idx}, name_{cls}, name_{mthd}, n_{args}, file_{path} = callInfo(M, I, call, file)$ /* Find the
              callee in the software based on the file path, class name,
              method name, # arguments                                   */
11         $called_{idx} = callInfo(path_{file}, name_{cls}, name_{mthd}, n_{args}, M)$
12         $E = E \cup (caller_{idx}, callee_{idx})$
13     end
14 end
15 return $G = (V, E)$
```

evaluate treat the comment and code in separate ways by using a special $[SEP]$ token, we first remove all the docstrings and comments from a method to obtain only code. The methods are then parsed into an AST using the Tree-Sitter library [61] to obtain the ordered method tokens, and composite identifier names are split into subtokens based on the preprocessing pipeline from CodeSearchNet [183].

Next, the method tokens are passed through a neural-based code model, *e.g.*, CodeBERT [128], GraphCodeBERT [149] or UniXcoder [148], to generate representation vectors, as shown in Figure 5.1-①. We chose CodeBERT as one of our models since it was one of the first bi-modal pre-trained models which captures the semantic connections between natural language (*i.e.*, comment) and programming languages (*i.e.*, code) so that the information from the natural language comments can enhance the model's code understanding. CodeBERT is a representative model that makes full use of the sequence information existing in the comment and code.

In Contrast to CodeBERT, GraphCodeBERT further extracts the DFG in the code to learn the inherent semantic-level structure information by incorporating the relationship of "where-the-value-comes-from" between variables. Finally, UniXcoder is one of the latest open-source code representation models that achieved state-of-the-art performance in code understanding and generation tasks by utilizing the AST instead of the DFG to learn rich syntactic information from code. Both the AST and DFG are mapped into sequence structures to be easily learned by transformers.

CodeBERT, GraphCodeBERT and UniXcoder are pretrained on 2.3 million (comment, code) pairs across six programming languages from the CodeSearchNet dataset. Code representations can be directly obtained from the pre-trained models, but the pre-training objectives (*i.e.*, masked language modeling, replaced token detection, code fragment representation learning, *etc.*) are quite different from IA and these representations are too general to represent one specific Java programming language. Although these models have been further fine-tuned for downstream tasks, none of them have been fine-tuned or evaluated for IA. In the absence of large available training dataset for IA, we fine-tuned the three models based on the code search task. Code search aims to retrieve relevant code snippets given a natural language query, and we choose this task as a proxy for IA because the models implicitly learn the functional goal of each code snippet during the fine-tuning, and it is highly likely that methods with similar functional goals get changed together when performing IA, as shown in the conceptual coupling metric introduced by Poshyvanyk *et al.* [280].

Specifically, we fine-tuned the three models on all $164,923$ (comment, code) pairs of the CodeSearchNet Java split based on the Siamese framework according to similar fine-tuning pipelines shared by Ranasinghe *et al.* [288]. Each code snippet in the paired data is a complete method from a software repository. During the fine-tuning process, the comment and method tokens are first converted into token IDs based on the tokenizer from the three models and then passed into the comment encoder and method encoder, respectively, to obtain the comment and method embeddings. Both encoders share the same configurations with the same model parameters and weights. Then, for each batch of data, the distance between the comment and its corresponding method is minimized in the embedding space by using the standard cross-entropy loss function. We use the AdamW [199] optimizer and the same hyperparameters recommended by the three respective models (*e.g.*, # epochs, learning rate, batch size *etc.*) for fine-tuning, and the whole process was performed on an Ubuntu 20.04 server with an NVIDIA A100 40GB GPU. For GraphCodeBERT, we use the same approach of including the DFG information of the method during the fine-tuning process. CodeBERT and UniXcoder models only use the method information without any additional content. These trained models are then used to generate the embedding representations of the methods for the IA task, *i.e.*, we perform zero-shot learning [328] without any further training for the IA task.

5.1.3 Embedding Propagation

By using any one of the three neural-based models, we obtain the initial method embeddings of all methods in a software system call graph $G = (V, E)$, where $|V| = N$.

However, these embeddings are only equipped with the local method semantics but ignore the global dependencies between methods. Therefore, we utilize an embedding propagation strategy to update each of them based on the embeddings of its neighbor methods to combine the local semantics and structural information, *i.e.*, DFG or AST, with global dependency information from the call graph. We visualize this process in Figure 5.1-②. Formally, this is represented as the following:

$$m_i = f(m_i, m_1^{nebr}, m_2^{nebr}, ..., m_k^{nebr}), \tag{5.1}$$

where m_i is the method being updated through the embedding propagation strategy f with its neighbors $m_j^{nebr}(1 \leq j \leq k)$. Since a change in the callee method can still require a change in the caller method and this dependency is not capture in a directed call graph, we treat all edges as undirected for the propagation. Specifically, our embedding propagation strategy is inspired from the Graph Convolutional Network [201] which adopts layer-wise propagation on the neural networks motivated by a localized first-order approximation of spectral graph convolutions:

$$M' = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}MW), \tag{5.2}$$

where σ represents an activation function and W is a trainable weight matrix. $\tilde{A} = A + I_N$ denotes the adjacency matrix of G with self-connections. I_N is the identity matrix and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. This propagation strategy has been modified using a renormalization method [201] in order to mitigate the effects of numerical instabilities and exploding/vanishing gradients when matrix multiplication operators are repeated during the training of the deep neural network. Since we do not train the call graph in this phase, our embedding propagation strategy is directly derived from the first-order approximation of localized spectral filters on graphs, which can be summarized as follows:

$$M' = (I_N + wD^{-\frac{1}{2}}AD^{-\frac{1}{2}})M. \tag{5.3}$$

$M \in \mathbb{R}^{N\times F}$ represents the matrix of all the method embeddings from G and M' represents the updated matrix by incorporating the information from the neighbor methods. F denotes the dimension of each method embedding (*i.e.*, 768). A is the adjacency matrix of G without self-connections and D is the degree matrix of A so that the adjacency matrix is normalized by D with respect to both the row and the column. w is a constant to balance the information from the original method with contextual information from the neighbor methods. Moreover, in order to evaluate the effect of the distance of neighbor methods used for embedding propagation, neighbor methods in other orders (distances) are also utilized in addition to the direct neighbors:

$$M' = (I_N + w\sum_i D_i^{-\frac{1}{2}}A_iD_i^{-\frac{1}{2}})M, \tag{5.4}$$

where $1 \leq i \leq 3$ since we at most take into account the neighbor methods within three orders due to computational constraints. After the Embedding propagation strategy has

Table 5.1: Dataset statistics of our evaluation benchmark

Settings	# queries	# commits	ground-truth set	corpus
1 - whole	4,405	910	15.14	3,346
2 - inner	3,379	734	3.47	31
3 - outer	2,999	444	17.21	3,440

completed, all of the identified methods in a given project will have an augmented embedding calculated by propagating the *original* method embedding from neighbors (*i.e.*, as generated by one of our three neural models) to the target method, as illustrated at the top of Figure 5.1-③.

5.1.4 Impact Set Estimation

Finally, as illustrated in Figure 5.1-③, ATHENA computes the cosine similarity between the augmented embeddings of a given query method and the augmented embeddings for each method in the search corpus. Based on the cosine similarity scores, our approach returns a ranked list in descending order to help developers find other methods that are possibly affected and likely to be modified.

5.2 Evaluation

To evaluate ATHENA's effectiveness in the IA task, we investigate the following research questions (RQs):

RQ$_1$: *How well do traditional information retrieval and neural-based techniques perform on the task of impact analysis?*

RQ$_2$: *Does augmenting baseline techniques with call-graph information improve the effectiveness on impact analysis?*

RQ$_3$: *What leads to the difference in effectiveness (or lack thereof) between baseline techniques and techniques augmented with call-graph information?*

RQ$_4$: *How do properties of different impact analysis tasks affect our studied techniques?*

5.2.1 Impact Analysis Benchmark: Alexandria

As one of the most common and important types of changes to a software system, we chose to focus our efforts on bug fixes. Many of the existing datasets, such as the one by Tufano *et al.* [341], are quite large, but unfortunately do not contain manually vetted data to be useful for IA. In contrast, existing IA datasets tend to be quite small only containing fewer than ten software projects [244, 355]. Further, there is the potential for *tangling* in software repository commits, wherein a commit which claims to be fixing a bug, both fixes the bug and may include additional unrelated changes, such as refactorings. As a result, the ground-truth impact sets from previous benchmarks may carry with them inaccuracies in their impact sets, as some methods may not actually be affected by a given change.

Recently, Herbold *et al.* [169] introduced a large dataset consisting of 3,498 commits (*i.e.*, changes) from 28 Java projects, with the purpose of studying the tangling that occurs in bug fixing commits. In this dataset, each changed line was annotated with its type of change, whether it was modified to fix a bug, or was a change to tests, whitespace, a refactoring, or a documentation change. The data were annotated by four participants, and consensus was obtained if at least three participants agreed on the annotation to ensure accuracy. This paper also illustrated that many of the changes in the bug-fixing commits were changes to non-production artifacts, such as tests or documentation, rather than bug fixes. Therefore, we constructed our evaluation benchmark at method-level granularity based on this dataset which manually untangles the commits so that we know exactly which methods are changed for addressing one single concern, namely fixing one bug.

Impact Set Construction. To construct the impact set, we systematically mined the dataset from Herbold *et al.* [169]. Specifically, for each changed line in the production code file labeled as *"contributes to the bug fix"*, we added the corresponding method to our benchmark by recording the information of GitHub Diff URL, repository name, commit ID, parent commit ID, file path, method name, line numbers where the method starts and ends corresponding to both current commit and parent commit. Since Herbold *et al.* [169]

does not provide the method-related information, such as method name and line numbers where the method starts and ends, we checked-out the source code of the repository for the parent commit. We then used the srcML library [97] to locate the changed methods based on the labeled changed line number. We use the snapshot of the software system that corresponds to the parent commit of a given as that is the state in which the change will be applied. Then, for each parent commit, we formulate the changed method set based on the concurrently changed methods. Since there is no clear single query method, *i.e.*, which method was changed "first" in the commit, we treat each method in the changed method set as a potential query, whereas the others are the ground truth impact set. Each query to impact set pair is considered a task. From developers' point of view, they usually at least know where the change starts, and intend to know which other methods need to be modified. We further post-process the dataset by removing commits that contain only one changed method. This process of formulating co-changing methods into impact analysis tasks has been widely used by past work to assess IA approaches [222, 194, 140, 139].

Task Definition and Settings. For each changed method set $M = \{m_1, m_2, ..., m_n\}, n \geq 2$, we perform the impact analysis task with the query being $\forall\ m_i \in M$ and the ground-truth impact set being $M - m_i$. The search corpus is all methods except the query in the production files from the corresponding repository (Setting 1 - whole). In practice, the developer would determine whether a method should be modified by inspecting the corpus of methods in the order given by the specific approach. After inspecting our dataset, we found that methods in the same file could potentially be changed together. To mitigate the effect of possible shortcuts taken by approaches which pay more attention to all methods from the same file as the query, we formulate two more specific task settings: (i) The methods in both the ground-truth impact set and the search corpus are from the same file as the query (Setting 2 - inner). (ii) The methods in both the ground-truth impact set and the search corpus are from different files than the query (Setting 3 - outer). For setting 2, we focus on the methods that need to be modified in the same file as the query, while for setting 3, we focus on the methods that need to be modified in other files. The data corresponding to these two settings are further filtered by discarding the impact sets whose size is less than two.

Dataset Statistics. Two software projects in [169] are no longer accessible (*santuario-java* and *wss4j*) and for the software project *eagle*, we could not build any valid impact set,*i.e.*, the size of the impact set less than two. As a result, our benchmark contains 25 open-source software systems and # tasks/queries per software project is shown in Table 5.5. Moreover, for each of the three settings, Table 5.1 shows # tasks/queries, # commits, the average number of methods in the ground-truth impact set and in the search corpus respectively.

Compared to Setting 2 (inner) which retrieves three or four affected methods from 31 methods, the Setting 3 (outer) is far more challenging which retrieves 17 or 18 methods from the corpus with 3,440 methods on average.

5.2.2 Evaluation Metrics

We use standard information retrieval metrics to measure the effectiveness of the proposed approaches, namely mRR (mean Reciprocal Rank), mAP (mean Average Precision) and Hit@k. Specifically, for each task, the ranked list obtained from the proposed approach is compared with the expected impact set from the ground truth. Given one query method, we computed the *rank* of the first actually affected method found in the ranked list, which indicates how many methods developers have to check to find the first one that needs to be modified. Then, we computed the *reciprocal rank* for each task and averaged it across all the tasks to obtain the final mRR score. The mRR score measures the ability of the approach in helping developers find at least one method that needs to be modified. Correspondingly, the AP score for each task is calculated and averaged across tasks to obtain the final mAP score. AP is the average of the precision values which are computed after each ground-truth method in the impact set is retrieved to approximate the area under the uninterpolated *Precision-Recall* curve. mAP scores measure the ability of the approach in helping developers find all the methods that need to be modified. Moreover, we use HIT@k to measure the proportion of successful tasks for the cut point k. A successful task means finding at least one affected method among the top-k (10 in this paper) results returned by the approach.

Most prior IA techniques [244] [68] use *Precision, Recall* and *F-measure* to evaluate their approaches since they consider the IA as a binary classification task by finding the possible affected methods based on dependencies instead of analyzing all methods in the repository. Therefore, what their method produces is not a ranked list, but an unordered estimated impact set, which is then directly compared with the ground truth impact set to obtain an F-score (*i.e.*, the harmonic mean of the *Precision* and *Recall* values). However, Poshyvanyk *et al.* [280] previously formulated impact analysis as an information retrieval task, but adapted prior metrics to the IR setting. We argue that IR metrics provide a more realistic representation of the potential benefit that an automated IA approach may actually provide to a developer in a recommender system setting. Furthermore, mAP score is more accurate than F-measure because it analyzes *Precision-Recall* relationship globally rather than just based on the mean value calculation.

5.2.3 ATHENA Configurations

Three models are generated using our proposed approach, namely Athena (CodeBERT), Athena (GraphCodeBERT) and Athena (UniXcoder), whose initial method representations are extracted based on each of the three neural-based code understanding models and we set $w = 0.5$ for information balancing. To compare with our approach, we also conducted the experiments based on these models without call graph information, using only the initial method representations without embedding propagation to compute the cosine similarity which serve as the corresponding baselines, including CodeBERT, GraphCodeBERT and UniXcoder.

Moreover, we use the traditional *bag-of-words* model (*i.e.*, TF-IDF) to obtain method representations where the term frequency (TF) is computed as the number of times a given code token appears in all tokens of its corresponding method, and the inverse document frequency (IDF) is calculated as the number of occurrences of a given code token in all code tokens built from the search corpus which includes all the production methods in a repository. Since TF-IDF representations are not real meaningful embeddings but frequency numbers without fixed vector length, we directly use a simple similarity weighting strategy for the ATHENA (TF-IDF), in which the cosine distance between the query and each of its neighbor methods is reduced by 50% to obtain the final ranked list.

In addition, instead of using call graphs, we build class graphs to further validate the effectiveness of call graphs when applied to IA, where the edges are added between each pair of methods in the same class resulting in many small strongly connected graphs for a repository. Due to the magnitude of # edges, we still use the similarity weighting strategy to obtain class-graph-based CodeBERT, GraphCodeBERT, and UniXcoder respectively. Note that due to the nature of strongly connected graphs, there is no difference when neighbor methods of different orders are taken into account.

5.3 Results

5.3.1 RQ$_1$: Baseline Performance on IA

Table 5.2 gives an overview of our different models both without (Baseline) and with (Athena) call graph information, which we will discuss in the next subsection. The first observation is that for setting 1 (whole), the traditional TF-IDF and neural-based models CodeBERT, GraphCodeBERT, and UnixCoder all achieve similar mAP scores, 22.89, 23.77, 24.25, and 23.65, respectively.

The second observation is that GraphCodeBERT, regardless of setting or metric, is the best performing model. This is in opposition of work from Guo *et al.* [148], showing UnixCoder to outperform GraphCodeBERT. This might be due to GraphCodeBERT being the only model using additional information through Dataflow Graphs (DFGs) during the fine-tuning phase, thereby being able to fully utilize the semantic and structural information within the method.

Table 5.2: Effectiveness of the baseline models and their ATHENA versions with the call graph information

Type	Models	Setting	mAP	mRR	HIT@10
Baseline	TF-IDF	Whole	22.89	45.36	63.91
		Inner	61.59	71.12	92.51
		Outer	14.97	32.01	46.62
	CodeBERT	Whole	23.77	46.25	67.20
		Inner	62.50	73.00	94.58
		Outer	19.16	37.81	53.55
	GraphCodeBERT	Whole	24.42	46.70	68.79
		Inner	63.97	73.04	94.32
		Outer	19.85	38.13	54.09
	UniXcoder	Whole	23.65	45.96	66.95
		Inner	61.95	72.07	93.58
		Outer	19.19	37.33	52.38
ATHENA	TF-IDF	Whole	23.96	47.41	70.60
		Inner	60.69	69.42	93.40
		Outer	15.79	33.48	50.78
	CodeBERT	Whole	27.38	50.96	73.24
		Inner	63.28	73.32	**95.59**
		Outer	21.60	**41.43**	**59.49**
	GraphCodeBERT	Whole	**28.27**	**51.28**	**74.46**
		Inner	**63.78**	**73.73**	95.12
		Outer	**22.12**	41.40	59.29
	UniXcoder	Whole	27.28	49.91	72.83
		Inner	62.75	72.40	94.58
		Outer	21.35	40.20	57.55

When deconstructing setting 1 (whole) into its constituents, setting 2 (inner) and setting 3 (outer), we observe that these models perform best on the setting 2 case, *i.e.*, when only considering changes within the same class. For example, GraphCodeBERT's mAP score for setting 2 is 63.97, yet when considering only changes outside of the query method's class it achieves a score of 19.85. This result is intuitive, as there are many more detractors in the setting 3 than setting 2 when ranking methods for estimating the impact set.

One of the surprising observations is that TF-IDF performs quite strong in setting 1 and 2 compared to neural-based approaches to code representation. However, for setting 3, it suffers significantly achieving an mAP score of 14.97 compared to the next worst CodeBERT performance of 19.16. The reason behind this is that TF-IDF is particular good at keyword matching [183] rather than the understanding of underlying code semantics, and keyword overlapping is more common for the methods within the same file as the query as compared to those in different files, so TF-IDF ranked all these methods higher than others. Meanwhile, the methods within the same file are more likely to be actually affected based on the ratio of the size of ground-truth impact set to the corpus size from Table 5.1. Therefore, the performance of TF-IDF is comparable to neural-based models in setting 1. However, when based on keyword matching only, the relative positions of these methods hardly change, thus having little impact on the accuracy of settings 2 and 3. More details about this phenomenon are explained in $\mathbf{RQ}_3$. In contrast, neural-based models focus more on the underlying semantics understanding, which affects the relative positions for the methods in the same file as the query and the methods in other files, contributing to higher mRR and mAP in both setting 2 and 3.

5.3.2 $\mathbf{RQ}_2$: Athena Performance on IA

Table 5.2 also shows the results of each of the models with call graph information integrated. As observed, each neural-based Athena models improves significantly (Wilcoxon's paired test $p < 0.05$) compared to its corresponding baseline in setting 1 (whole). Specifically, Athena achieves an improvement of 3.61 mAP for CodeBERT, 3.85 mAP for GraphCode-BERT, and 3.63 mAP for UnixCoder, but only obtains an improvement of 1.07 mAP for TF-IDF because of the employed simpler similarity weighting strategy.

Similar to the previous analysis, when looking at the constituents of setting 1, we see that setting 2 (inner) is not greatly improved. The improvements largely come from setting 3 (outer), which is intuitive since the integration of the call graph information adds contextual information about the rest of the software system through connections that extend outside of the query method's class. For example, for the best performing overall neural-based model GraphCodeBERT, when using Athena to integrate the call graph information improves by 2.27 mAP for setting 3 (outer), yet sees a slight decrease for setting 2 (inner).

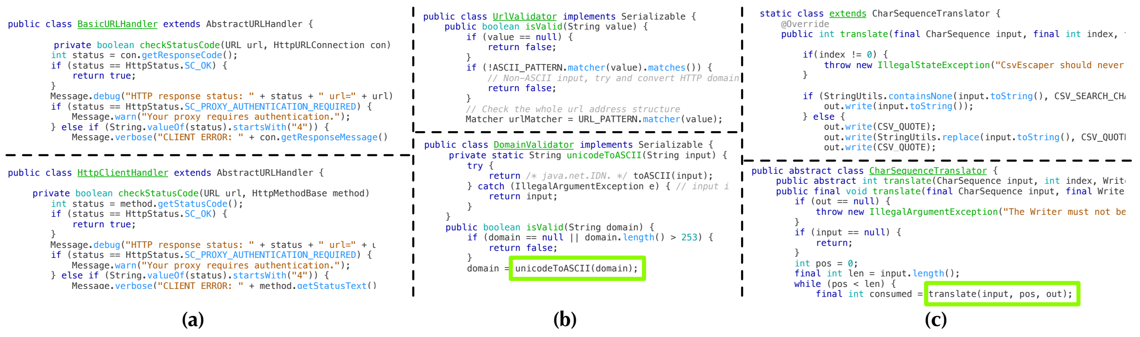
Table 5.3: Effectiveness of ATHENA for different configurations

Configuration	**mAP**	**mRR**	**HIT@10**
CodeBERT	23.77	46.25	67.20
ATHENA (CodeBERT)	**27.38**	**50.96**	**73.24**
+1 Neighbor	26.01	49.06	72.05
+3 Neighbor	27.23	50.67	73.49
GraphCodeBERT	24.42	46.70	68.79
ATHENA (GraphCodeBERT)	**28.27**	**51.28**	74.46
+1 Neighbor	26.75	49.33	73.39
+3 Neighbor	28.22	51.18	**74.53**
UnixCoder	23.65	45.96	66.95
ATHENA (UnixCoder)	27.28	49.91	72.83
+1 Neighbor	25.68	48.10	70.87
+3 Neighbor	**27.63**	**50.27**	**73.14**

The results from Table 5.2 of the ATHENA versions of the models utilize the best performing integration of neighborhood information, namely using second order neighbors. However, we are also interested in how the number of neighbor orders, *i.e.*, neighbors of neighbors *etc.*, impact performance. Table 5.3 shows the neural-based models under different number of neighbor orders, with the one marked ATHENA being two order neighbors used in Table 5.2. As shown, even with one order neighbors, the neural-based models are significantly improved over their baselines with an improvement of 2.24 mAP for CodeBERT, 2.33 mAP for GraphCodeBERT, and 2.03 mAP for UnixCoder on setting 1 (whole). Further increasing the number of orders to two also see another significant improvement across models of 1.22 mAP for CodeBERT, 1.52 mAP for GraphCodeBERT, and 1.6 mAP for UnixCoder on setting 1. However, increasing the order neighbors indefinitely does not continually lead to large gains as shown with three orders *vs* two: there is very little change in terms of mAP and for CodeBERT and GraphCodeBERT, we see a decrease.

We conducted additional experiments by concatenating the tokens of the method with its corresponding comment for CodeBERT and ATHENA (CodeBERT), but we did not observe an obvious benefit (the results are available in our online replication package [33]). Therefore, all the presented results are based on code only, without comment information.

Figure 5.2: Three qualitative examples for illustrating the effectiveness of ATHENA.



```
public class BasicURLHandler extends AbstractURLHandler {

        private boolean checkStatusCode(URL url, HttpURLConnection con)
        int status = con.getResponseCode();
        if (status == HttpStatus.SC_OK) {
            return true;
        }
        Message.debug("HTTP response status: " + status + " url=" + url)
        if (status == HttpStatus.SC_PROXY_AUTHENTICATION_REQUIRED) {
            Message.warn("Your proxy requires authentication.");
        } else if (String.valueOf(status).startsWith("4")) {
            Message.verbose("CLIENT ERROR: " + con.getResponseMessage()
```

```
public class HttpClientHandler extends AbstractURLHandler {

    private boolean checkStatusCode(URL url, HttpMethodBase method)
        int status = method.getStatusCode();
        if (status == HttpStatus.SC_OK) {
            return true;
        }
        Message.debug("HTTP response status: " + status + " url=" + u
        if (status == HttpStatus.SC_PROXY_AUTHENTICATION_REQUIRED) {
            Message.warn("Your proxy requires authentication.");
        } else if (String.valueOf(status).startsWith("4")) {
            Message.verbose("CLIENT ERROR: " + method.getStatusText()
```

(a)

```
public class UrlValidator implements Serializable {
    public boolean isValid(String value) {
        if (value == null) {
            return false;
        }
        if (!ASCII_PATTERN.matcher(value).matches()) {
            // Non-ASCII input, try and convert HTTP domain
            return false;
        }
        // Check the whole url address structure
        Matcher urlMatcher = URL_PATTERN.matcher(value);
```

```
public class DomainValidator implements Serializable {
     private static String unicodeToASCII(String input) {
        try {
            return /* java.net.IDN. */ toASCII(input);
        } catch (IllegalArgumentException e) { // input i
            return input;
        }
    }
    public boolean isValid(String domain) {
        if (domain == null || domain.length() > 253) {
            return false;
        }
        domain = unicodeToASCII(domain);
```

(b)

```
static class extends CharSequenceTranslator {
    @Override
    public int translate(final CharSequence input, final int index,

        if(index != 0) {
            throw new IllegalStateException("CsvEscaper should never
        }

        if (StringUtils.containsNone(input.toString(), CSV_SEARCH_CH
            out.write(input.toString());
        } else {
            out.write(CSV_QUOTE);
            out.write(StringUtils.replace(input.toString(), CSV_QUOT
            out.write(CSV_QUOTE);
```

```
public abstract class CharSequenceTranslator {
    public abstract int translate(CharSequence input, int index, Writ
    public final void translate(final CharSequence input, final Write
        if (out == null) {
            throw new IllegalArgumentException("The Writer must not be
        }
        if (input == null) {
            return;
        }
        int pos = 0;
        final int len = input.length();
        while (pos < len) {
            final int consumed = translate(input, pos, out);
```

(c)

Table 5.4: Effectiveness of three neural-based models by using class graphs

Type	Models	Setting	mAP	mRR	HIT@10
Class Graph	CodeBERT	Whole	32.94	56.15	80.32
		Inner	62.50	73.00	94.58
		Outer	19.16	37.81	53.55
	GraphCodeBERT	Whole	33.59	56.46	80.89
		Inner	63.97	73.04	94.32
		Outer	19.85	38.13	54.09
	UnixCoder	Whole	32.74	55.19	79.36
		Inner	61.95	72.07	93.58
		Outer	19.19	37.33	52.38

5.3.3 RQ$_3$: In-Depth Analysis of the Improvement

In Table 5.2, the improvement produced by ATHENA compared to the baseline in setting 1 (whole) is distributed between the improvements in setting 2 (inner) and in setting 3 (outer), with setting 3 accounting for the major part of the improvement.

Specifically, in setting 3, ATHENA (CodeBERT) outperforms CodeBERT by 3.62% on mRR, 2.44% on mAP and 5.94% on HIT@10, ATHENA (GraphCodeBERT) is better than GraphCodeBERT by 3.27% on mRR, 2.27% on mAP and 5.20% on HiT@10, and ATHENA (UniXCoder) outperforms UniXCoder by 2.87% on mRR, 2.16% on mAP and 5.17% on HIT@10, which accounts for 76.65% & 67.59%, 71.40% & 58.96%, and 72.66% & 59.50% of the improvement on mRR & mAP in setting 1 respectively. Since setting 3 is a more challenging setting than setting 2 which finds the actual affected method in other files, ATHENA exhibits a better ability to reason about change impact sets across file boundaries – which may be a more cognitively challenging task. Moreover, the ATHENA version of each neural model still performs *significantly* better than Athena (TF-IDF) in each of the three settings based on the Wilcoxon's paired test, and the improvement also comes mainly from the setting 3 (+7.92% on mRR, +6.33% on mAP, and 8.51% on HIT@10 when compared to Athena (GraphCodeBERT)).

To further validate the effectiveness of call graphs combined with code semantics when applied to IA, the performance of class graph-based CodeBERT, GraphCodeBERT, and UniXcoder is shown in Table 5.4. The class graph-based models improve their corresponding baselines by larger margins than our call graph-based models in setting 1, but as expected, they perform the same as their baselines in setting 2 and 3. Since all methods in the same file as the query are drawn closer to the query based on the similarity weighting strategy, they are all ranked higher than other methods. Given that the methods within the same file are more likely to be affected, we obtain a larger improvement in setting 1 as mentioned in $\mathbf{RQ}_1$. However, their relative positions in the ranked list do not change, so there is no impact on the performance in setting 2 and 3. This also explains why TF-IDF is comparable to neural-based models in setting 1, but worse than them in setting 2 and 3. In contrast, the improvement produced by our call graph-based models in setting 1 effectively contributes to the improvement in setting 2 and 3.

5.3.4 $\mathbf{RQ}_4$: Qualitative Analyses on IA Tasks

We begin our analysis of impact tasks by looking at the performance of our studied tehcniques across our different studied software projects. Table 5.5 provides a finer grained picture of the improvements per repository the best performing Athena model achieves over its corresponding baseline.

As shown, Athena improves performance on 22 of 25 repositories in terms of mAP and 20 of 25 in terms of mRR. We investigated the reasons for the failure of Athena in those repositories, especially for the project *commons-digester* and *gora* due to the relatively larger performance difference between Athena (GraphCodeBERT) and GraphCodeBERT. For *commons-digester*, this is mainly due to the number of method calls in the method,

Table 5.5: Effectiveness of ATHENA for each software system

Repo Name	Queries	*Baseline*$_{GCB}$		*Athena*$_{GCB}$	
		mAP	**mRR**	**mAP**	**mRR**
ant-ivy	785	19.14	29.59	**22.78**	**34.83**
archiva	43	14.21	41.07	**21.67**	**48.51**
commons-bcel	138	25.93	45.45	**30.31**	**48.28**
commons-beanutils	42	36.96	51.65	**37.22**	**53.49**
commons-codec	41	36.79	47.93	**41.55**	**48.87**
commons-collections	73	24.70	30.25	**24.88**	**30.52**
commons-compress	260	19.86	30.06	**27.8**	**40.91**
commons-configuration	253	21.95	30.43	**26.56**	**35.5**
commons-dbcp	91	48.45	60.00	**50.32**	**62.31**
commons-digester	22	**14.40**	**17.59**	11.64	15.83
commons-io	58	49.40	57.11	**54.48**	**63.33**
commons-jcs	221	**24.07**	**40.11**	23.16	39.12
commons-lang	115	42.45	51.28	**54.54**	**60.64**
commons-math	589	31.57	42.04	**38.03**	**48.64**
commons-net	171	28.56	40.00	**30.27**	**40.84**
commons-scxml	114	17.31	27.01	**25.48**	**38.25**
commons-validator	35	41.87	45.71	**43.02**	**46.29**
commons-vfs	166	22.29	30.27	**27.85**	**37.75**
deltaspike	5	34.08	41.86	**50.28**	**50.28**
giraph	527	27.43	44.48	**34.29**	**51.16**
gora	174	**23.76**	**36.95**	17.21	27.76
jspwiki	12	13.24	**52.12**	**15.54**	51.20
opennlp	141	23.66	34.40	**27.26**	**37.44**
parquet	324	19.97	37.24	**21.53**	**39.52**
systemml	5	46.19	**50.35**	**47.32**	50.10

e.g., there are more than 10 called methods in the query, while there is only one or two called methods in the method from the impact set. As for the project *gora*, we found that there are many semantically similar pairs in the changed method set, such as (serialize, deserialize), (encodeInt, decodeInt) *etc.*, but since their goals are opposite, the methods they called are quite different. Therefore, ATHENA does not improve in identifying the actual affected method after incorporating the call graph information into the original semantics.

Now that we have examined the performance of ATHENA across IA tasks at a repository level, we will now discuss some exemplars from our benchmark that showcase how incorporating both structural information and semantic information can benefit the task of automated impact analysis.

Example 1: The Importance of Semantics. Figure 5.2 (a) shows two methods from different classes. The top method `checkStatusCode (URL, HttpURLConnection)` from class `BasicURLHandler` is the query method in this exemplar and the bottom method `checkStatusCode (URL, HttpMethodBase)` is one of the impact set methods. They share no call graph information that a structural IA technique could use to determine these methods are coupled. Yet, a change in one requires a change in the other. This is representative of conceptual coupling [280], where the concepts of the two methods, *i.e.*, both performing a check on a status code, couples them together making it more likely that change in one would result in a change in the other. Utilizing the semantic information between the methods, *i.e.*, their keywords, either through a traditional TF-IDF or a neural based approach is necessary to determine that these two methods are highly coupled.

Example 2: The Importance of Combining Call Graph and Semantics. Figure 5.2 (b) shows a different example where The `isValid` method from the class `UrlValidator` is the query method, and the `unicodeToASCII` method from the class `DomainValidator` is part of the impact set. Note that the other `isValid` method from the class `DomainValidator` (which shares a name with our query method) is not part of the impact set. When examining the traditional TF-IDF and best performing neural-based GraphCodeBERT approaches, they both fail to rank the impacted method `unicodeToASCII` high, ranking it at 176 and 120, respectively. However, our ATHENA version of GraphCodeBERT achieved a ranking of 31, so we aimed to understand why this occured. We found that the method `isValid` in the `DomainValidator` calls the ground truth `unicodeToASCII` method. Our ATHENA (GraphCodeBERT), obtains a rank of 31 for the ground truth associated with the query even though there is no direct method invocation between the query and `unicodeToASCII` impacted method. Therefore, utilizing the embedding propagation strategy of our approach, `unicodeToASCII` was updated with the information from the method `isValid` (in the `DomainValidator` class) that is more semantically similar to the query, thus helping improve the ground truth rank.

Example 3: Call Graphs Improve IA with more code semantic overlap. Fig. 5.2 (c) presents an example with query being the method `int_translate` from the class `CsvEscaper` and the ground truth being the method `void_translate` from the class

`CharSequenceTranslator`. Owing to their similar code semantics, the original GraphCodeBERT ranks the ground truth 8th. In Athena (GraphCodeBERT), the information of the abstract method `int_translate` from the same class `CharSequenceTranslator` is incorporated into the method `void_translate` because of a call dependence between them, which thus improves the similarity between the query and the ground truth since the abstract method `int_translate` is more semantically similar to the query than `void_translate` in terms of the method signature. Therefore, Athena (GraphCodeBERT) achieves a higher rank of 6 for the target `void_translate` method.

As can be observed from these examples, there are clear benefits when the call graph information is combined with the local code semantics, and we saw this pattern hold after investigating additional cases where the Athena outperforms its corresponding baseline. The contextual information obtained from the global call dependencies among methods enriches the original semantics of the methods, which indeed helps to identify the impact set associated with the given query.

5.4 Threats to Validity

Threats to Internal Validity: To reduce potential issues from internal threats to validity, we studied three different DL models and a non-DL model when validating our proposed method of incorporating call graph information to improve IA. Additionally, we constructed our benchmark from commits that have been manually annotated and had the changes made to fix bugs untangled from other changes such as ones to documentation to ensure our ground truth labels are high quality. Additionally, we ensured there were no overlapping repositories between our training on the CodeSearchNet corpus and testing on our impact analysis dataset.

Threats to External Validity: To lessen the potential for threats to external validity, we used a significantly larger set of projects, 25 compared to previous work that used around five, and tested our method across different DL and non-DL models to show generalizability. One potential issue with generality is that we only evaluated our approach on Java and Apache projects, therefore, our approach may not generalize to other programming languages such as Python or to different types of projects. However, the DL models we used have shown success across multiple languages and so most likely the same would apply to our approach.

Limitations: We leave many of our approaches' limitations to future work. Particularly, utilizing more specialized DL architectures that might improve results. One type that we explored initially was Graph Neural Networks (GNNs), namely GraphSage [156] and Graph Attention Network (GAT) [345]. However, their performance was subpar so we refocused on the Transformer architecture, but additional investigation of such architectures is warranted. Another future direction is related to incorporating other software engineering artifacts into our approach such as change requests, test cases, architecture diagrams, *etc.*. Similarly, integrating other types of software specific information besides call graph is another potentially fruitful area. Another area that would cause for future work is how well this improves developer productivity when performing code changes. Lastly, due to the flexibility of our approach, future work could look into applying ATHENA to other software engineering tasks such as code search, code comprehension, or other maintenance tasks.

5.5 Bibliographical Notes

The paper supporting the content described in this Chapter was written in collaboration with other members of the SEMERU group at William & Mary and researchers from Universitá della Svizzera italiana. It is currently under review for publication.

Yan, Y., **Cooper, N.**, Moran, K., Poshyvanyk, D., & Bavota, G. (2023, March). Combining Call Graphs and Neural Code Semantics to Improve Automated Impact Analysis. Under Review.

Chapter 6

On the Generalizability of Transformer Models for Code Completion

Large Language Models (LLMs) for code have achieved state-of-the-art results across a variety of software engineering (SE) tasks such as code completion [364, 95], code reviews [342], clone detection [362], program repair [341, 91], testing [360, 340], and others [358, 253, 337]. The Transformer [343] has been at the center of these improvements due to its attention mechanism and ability to be highly parallelized, allowing for more efficient training compared to previous models such as Recurrent Neural Networks (RNN) [303].

When applied to code completion, Transformers take as input an incomplete code component and try to predict the missing code tokens (*e.g.*, [95, 85, 37, 131]). As already happened in the field of Natural Language Processing (NLP) [284], there has been a recent push in increasing the maximum length of the sequences (incomplete code components) on which Transformers are trained and tested. This is due to the fact that longer sequences (i) allow to provide the model with additional contextual information which can help with improving the prediction performance; and (ii) can help in simulating more variegate code completion scenarios. This, however, has a substantial cost to pay in terms of training time [283].

It comes without surprise that efforts have been made in the NLP literature to address this issue (*e.g.*, [105, 285, 283, 321]): The most recent work targets the generalization of Transformers to longer sequences than those they have been originally trained for [283, 321]. This allows to efficiently train a model on short sequences and, then, perform the inference on longer sequences without a significant performance degradation. This is, for instance, the goal of the ALiBi (Attention with Linear Biases) attention mechanism for Transformers [283], which has been successfully used in NLP.

If solutions such as ALiBi properly work on source code as well, they could substantially help reduce the training cost of code completion models such as the popular GitHub Copilot [142]. This is the focus of our work. We aim to investigate the extent to which solutions proposed in the NLP literature can support the generalization of Transformers on source code. We focus on three state-of-the-art solutions and one baseline. The first (baseline), Sinusoidal [343], uses Absolution Positional Encodings (APEs) by defining sine and cosine functions to generate positional embeddings that the authors original hypothesized would help the model to generalize. The second, xPOS [317, 321], is a hybrid between APEs and Relative Positional Encodings (RPEs) and applies rotations to the sine and cosine positional embedding to incorporate relative position information along with a attention resolution metric to improve generalization. The third, ALiBi [283], offers a simple solution of modifying the attention mechanism to weight positions far away as less important than ones closer. The last, T5 [285], similarly modifies the attention mechanism by adding a learned bias term that influences the attention given to a token.

We want to assess whether models trained on sequences of a specific length are able to generalize, *i.e.*, not incur a significant performance degradation, on sequences being longer (or shorter) than the training ones. To accomplish this, we built four datasets (*short*, *medium*, *long*, and *mix*) featuring incomplete Python and Java functions having different lengths. Then, we train 32 Transformers, namely four models (Sinusoidal, xPOS, ALiBi, and T5) for each of the four datasets and two programming languages. The performance of the 32 models has been evaluated on a series of test sets of previously unseen Python and Java functions of various different lengths, studying the generalization of their predictions. For example, we verified if the models trained on *short* datasets are able to work on instances in the test set having a length inline with the examples in the *long* dataset.

Overall, we make the following contributions:

1. A large systematic benchmark for evaluating the generalization of LLMs for code completion of different lengths across two programming languages;

2. An empirical study on whether current generalization approaches extend to encoder-decoder architectures for the task of code completion;

3. A set of results and implications that can be leveraged by researchers and developers of these models for navigating trade-offs.

6.1 Background

We introduce some mathematical background to understand the specifics of the position encoding schemes, namely Sinusoidal, xPOS, ALiBi, and Relative, and how they apply to our study.

Our paper focuses on the Sequence to Sequence Transformer [343] (see Fig. 6.1). It takes as input a sequence of tokens $C' = x_1, x_2, ..., x_n$ (in our case, the code to be completed), and outputs a target sequence, M, that is similarly decomposed into tokens $M = y_1, y_2, ..., y_m$. M represents in our context the missing piece of code to be predicted.

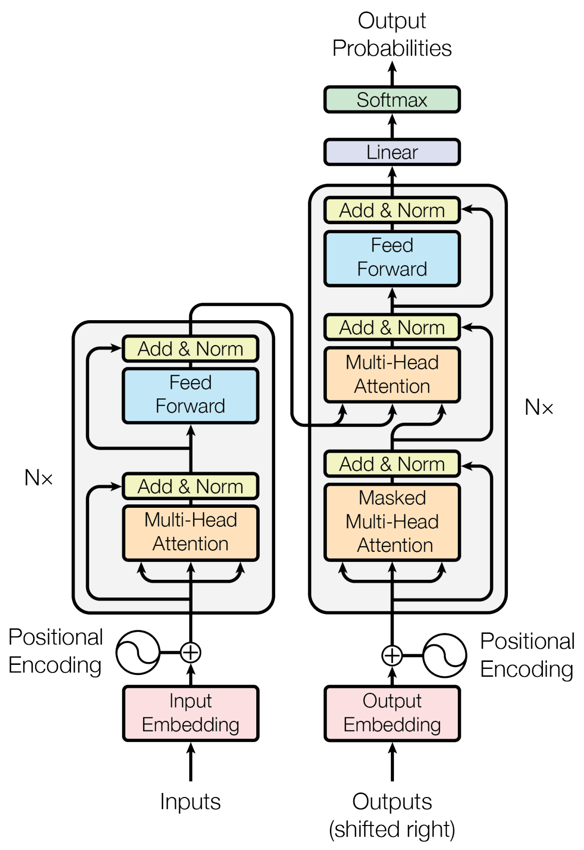


Figure 6.1: Sequence to Sequence Transformer Overview from the original paper [343]. The left part is the encoder and the right part is the decoder.

Thus, the combination $C = C' + M$ equals the complete code snippet. Note that the $+$ operator does not imply an append, but rather a combination regardless of where the missing part M is placed in C'. The decomposition of both C' and M is done through tokenization using a trained Byte Pair Encoding tokenizer [311] such that C' or $M \rightarrow s_0, s_1, ...s_T$, where $s_i \in \mathbb{V}$ and $\mathbb{V}$ is the vocabulary of the tokenizer. We use the same vocabulary for both C' and M. These tokens are passed through an Embedding layer, which is shared across the encoder and decoder, to get the token's vector representation, which will be updated progressive through various Attention and Feed Forward layers, we will explain further.

The output M is done in an autoregressive manner, where the output of one time step from the model is fed back into the decoder portion of the Transformer for generating the

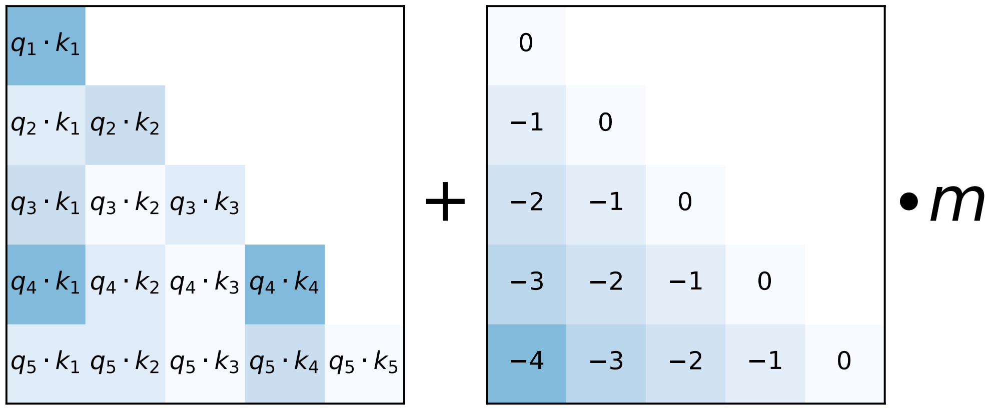

$q_1 \cdot k_1$
$q_2 \cdot k_1$ $q_2 \cdot k_2$
$q_3 \cdot k_1$ $q_3 \cdot k_2$ $q_3 \cdot k_3$
$q_4 \cdot k_1$ $q_4 \cdot k_2$ $q_4 \cdot k_3$ $q_4 \cdot k_4$
$q_5 \cdot k_1$ $q_5 \cdot k_2$ $q_5 \cdot k_3$ $q_5 \cdot k_4$ $q_5 \cdot k_5$
+
0
−1 0
−2 −1 0
−3 −2 −1 0
−4 −3 −2 −1 0
$\bullet m$

Figure 6.2: ALiBi Overview from the original paper [283]

probability distribution of the next token. Formally, the probability distribution of token y_i is conditioned on the output of the encoder Z and the $y_{<i}$ previously generated tokens: $p(M) = \prod_i^m p(y_i | y_{<i}, Z)$.

The Transformer architecture itself has no way of modeling sequential information. Therefore, sequential information is injected either at the bottom of the network (Fig. 6.1) or at each Transformer block of the network depending on the positional scheme. For our four positional schemes, only one injects the positional information at the bottom of the network, namely sinusoidal, the rest of the schemes inject the information at each Transformer block.

The Transformer block's composition depends on if it is part of the encoder or decoder. In the encoder portion, the attention is computed by transforming a given sequence, $s = (s_1, s_2, ..., s_n)$ where $s_i \in \mathbb{R}^{d_s}$ into a new sequence of the same size, $z = (z_1, z_2, ..., z_n)$ where $z_i \in \mathbb{R}^{d_z}$ via a weighted sum where the weight can be intuitively thought as the amount of attention to pay to the value of s_j in the sequence. $\mathbb{R}^{d_s}$ represents the embedding space of the Embedding layer that transforms the discrete token into a continuous vector of size d_s and similarly $\mathbb{R}^{d_z}$ represents the vector space that is composed of d_z dimensions where d_z can be the same or a different size than d_s. The weighted sum can be represented as follows:

$$z_i = \sum_{j=1}^{n} a_{ij}(s_j W^V) \tag{6.1}$$

With W^V being a learned value weight matrix and a_{ij} being calculated with the following softmax formula:

$$a_{ij} = \frac{\exp\ e_{ij}}{\sum_{k=1}^{n} \exp\ e_{ik}} \tag{6.2}$$

And e_{ij} being calculated by taking the corresponding s_i and s_j tokens and multiplying them by a learned query and key weight matrix, W^Q and W^K, to get the corresponding query and key vectors. These vectors are then put through a compatibility function, namely the scaled dot product:

$$e_{ij} = \frac{(s_i W^Q)(s_j W^K)^T}{\sqrt{d_k}} \tag{6.3}$$

As it can be seen, no positional information is present in these operations. So, the original Transformer injected positional information into the beginning of the network by adding a position vector to the token vector to encode this information for the rest of the network. Let us now discuss each positional encoding scheme.

Sinusoidal: Sinusoidal is the original scheme proposed in the Transformer paper [343]. The information is added directly to the token embeddings at the beginning of the network. Concretely, the same dimension of the token embeddings is used for the position embedding and the sinusoidals switch between sine and cosine:

$$PE_{pos,2i} = sin(pos/10,000^{2i/d_{model}}) \tag{6.4}$$

$$PE_{pos,2i+1} = cos(pos/10,000^{2i/d_{model}}) \tag{6.5}$$

Where *pos* refers to the position in the sequence and i refers to the specific dimension of the position embedding. The authors chose this scheme as they believed it would allow for the model to learn to use relative positions. However, this approach is generally considered an Absolute Positional Encoding (APE) scheme.

xPOS: Sun *et al.* [321] extended work from Su *et al.* [317] which made the observation that the dot product between queries and keys are where information is shared between different tokens. Therefore, position information can also be added of the relative position between the different tokens. Specifically, Su *et al.* [317] wanted to find an operation that satisfies the following:

$$(f_q(x_m, m), f_k(x_n, n)) = g(x_m, x_n, m - n) \tag{6.6}$$

Where functions f_q and f_k add this relative information to the token embeddings x_m and x_n. respectively. To accomplish this, the authors introduced Rotary, which uses rotations of the token embeddings based on their position so that the relative position of the tokens are preserved through this dot product. Namely, they define the function g that satisfies this to be

$$g(x_m, x_n, m - n) = Re((W_q \cdot x_m)(W_k \cdot x_n) * e^{i(m-n)\theta}) \tag{6.7}$$

where the function Re takes only the real part of the complex number. This is specifically for the 2D case, however, this is generalizable to any dimension that the token embeddings belong to.

For a complete discussion of the equations and their generalization, we refer readers to the original paper [317].

Sun *et al.* [321] extended this approach to have similar extrapolation abilities of ALiBi, discussed below, while still having better performance. To accomplish this, the authors introduced the idea of *attention resolution* where a model's attention should monotonically decrease as the pair wise distance between tokens increases, similar to ALiBi. To integrate this into the rotation matrix, they apply an exponential decay that adds this property. They show that this gives a good trade-off between the Rotary performance and ALiBi's ability to extrapolate to longer than seen during training sequences. For our experiments with xPOS, we use the original implementation from Su *et al.* [317] for the encoder and the extension, xPOS, by Sun *et al.* [321] for the decoder since the extension is unable to be applied directly to an encoder.

ALiBi: In ALiBi, the positional information is injected by modifying the equation above by adding a static bias:

$$e_{ij} = \frac{(s_i W^Q)(s_j W^K)^T}{\sqrt{d_k}} + m^h(j - i) \tag{6.8}$$

Where m is a head-specific scalar that is selected before training. We use the same geometric series for initializing these m values per head as in the original ALiBi paper, namely starting from $2^{\frac{-8}{n_{heads}}}$ and using the same value as the ratio. Intuitively, this static bias penalizes query and key vectors that are far away from each other. Figure 6.2 visualizes this process. Specifically, it shows how queries and keys corresponding to the same token do not receive any reduction whereas mismatched queries and keys receive a reduction proportional to their relative distance.

This process was originally designed for decoder-only Transformer models. However, since we use an encoder-decoder Transformer model, we additionally use the bidirectional version for the encoder portion as outlined in a post by the original author[1].

T5: Similar to ALiBi, T5 introduces a bias inside the softmax equation that is based on distance. Specifically, this bias is a learned scalar that is added to the query and key dot product $(s_i W^Q)(s_j W^K)^T + b^h_{ij}$ where each attention head, h, has a different learned bias. T5 introduces this idea of buckets, which are the different learned biases, where the different ij pairs logarithmically map up to a relative position of 128 beyond which the same ij pairs are mapped to the same bucket.

After each attention operation in the encoder Transformer block follows a normalized residual connection, a simple Feed Forward Multi-Layer Perceptron, and another normal-

[1]`https://github.com/ofirpress/attention_with_linear_biases/issues/5`

ized residual connection. A similar process happens in the decoder block. However, there is an additional attention mechanism that happens after the normal self-attention is applied to the transformed output token embeddings, which considers the encoder's representation of the input, Z. This is known as cross-attention and follows the same process as self-attention except the keys and values are constructed from the representation Z while the query is built from the output representation. Intuitively, this can be thought of as the output tokens requesting specific information from the input tokens.

6.2 Study Design

The *goal* of our study is to determine whether popular positional encoding schemes for length generalization work for the task of code completion. Namely, we seek to answer the following question:

RQ$_1$: *To what extent can different positional encoding schemes generalize to different code lengths for the task of code completion?*

In the context of code completion this means studying whether models trained on completing code sequences having a specific length generalize when used to complete shorter/-longer sequences. This is analogous to whether models are able to utilize different amounts of information, in terms of the input tokens, as compared to what they have seen during training. Naturally, shorter instances require shorter training time. Yet, it is unclear if a model trained on short code completions can generalize to also work on longer ones and *vice versa.*

While answering RQ$_1$ we also check whether our findings generalize to multiple programming languages. In particular, we contextualize our study to Python and Java, as languages often adopted in code completion studies (see *e.g.*, [95, 325]). Also, we consider two different code completion tasks recently used in the code completion study by Ciniselli *et al.* [95]: *statement-level* and *block-level* completion. The former is the classic code completion task in which the last n tokens of a single code statement are masked, with the model in charge of predicting them. The latter, instead, possibly extends the completion to multiple statements, masking the last n tokens in a block of code (*e.g.*, the body of an `if` statement) and asking the model to guess them.

In the following we detail the procedure used to (i) collect the training/testing datasets employed in our study (Section 6.2.1), and (ii) perform the required data collection and analysis (Section 6.2.2).

6.2.1 Dataset Construction

To build the datasets needed for our study, we mined data from GitHub open source projects. In particular we: (i) used the GitHub search tool by Dabić *et al.* [104] to identify all Python and Java GitHub projects having at least 100 commits, 10 contributors, 10 stars and 10 issues (to exclude toy projects); (ii) sorted the projects by number of stars; (iii) cloned the top 3k and extracted from each of them the functions in the master branch (to only rely on functions likely to be syntactically correct); (iv) removed all functions containing non-ascii characters (to avoid problems when reading data); (vi) removed all duplicates (to avoid leaking of information between the training, validation, and test sets we created out of this dataset); (vii) removed from the dataset all instances consisting of more than 1,024 tokens. The latter is a procedure usually adopted in applications of DL4SE (*e.g.*, [219, 336, 91]). Indeed, too long instances make the training of DL models too expensive, also motivating our investigation into the length generalizability.

Such a process resulted in the collection of ∼4M Python functions and ∼4.5M Java functions which we further processed to create datasets aimed at answering our research question.

6.2.1.1 Java dataset: statement-level code completion task

The Java dataset will cover the statement-level code completion task. The goal is to build three datasets featuring instances (*i.e.*, functions) having different lengths, namely *short*, *medium*, and *long* instances. Basically, with "length" we refer to the number of input tokens provided to the model. In all three datasets we keep constant the complexity of the prediction to generate (*i.e.*, the number of masked tokens). Indeed, only in this way we can "isolate" the impact on performance of changing the input length.

Since the code completion task, which we are focusing on, requires the masking of code statements, we start by removing for the set of collected Java functions all of those that do not contain any statements. We also removed all Javadoc comments, since we are focusing on completion tasks within the body of a function. We then compute the number of Java tokens within each of the remaining 3,833,445 functions: this results in a distribution of functions' *length*.

We then compute a second distribution representing the number of Java tokens within the code statements in the subject functions, observing a median of 11 tokens per statement. The idea is to mask in each instance of the three datasets we will create (*i.e.*, *short*, *medium*, *long*) the exact same number of tokens (11). As previously mentioned, this is done to keep constant the "complexity" of the completion task and better isolate the impact of the sequence length on the observed performance.

Given such a constraint, we remove all the functions not containing any valid statements to mask, *i.e.*, a statements consisting of at least 11 Java tokens. We sort the remaining 1,855,578 functions by their length and split them into three sets of the same size, obtaining: (i) a first set of functions with lengths ranging from 6 to 96 (*short* dataset); (ii) a second set of functions with lengths ranging from 97 to 180 (*medium*); and (iii) a third set of functions with lengths ranging from 181 to 1024 (*long*).

For each function F in each of the three datasets, we created n training instances, where n is the number of valid statements to mask F contains (*i.e.*, the statements having at least 11 tokens). This may end up in generating duplicates due to different functions from which we masked the only part being different, thus resulting in duplicates. For this reason, we perform a second deduplication round on all the datasets.

The final set of (masked) functions is then flattened to obtain the model input by replacing all new line characters (*i.e.*, '`\n`') with a special tag, ⟨NEW_LINE⟩, and remove all tabs (*i.e.*, '`\t`') and white spaces used to indent the code. We randomly split each set (*short*, *medium*, *long*) into training (80%), validation (10%) and test (10%) by making sure that all the instances obtained from the same function fall into the same set.

Finally, for each of the three datasets (*short*, *medium*, *long*), we limit the number of training instances to 280k and, proportionally, those of the test and evaluation set to 35k. This is done to reduce the training cost of our study, as explained later, required to train and test 32 DL models. These numbers (*i.e.*, 280k and 35k) are inherited from the (smaller) Python dataset that we will describe in the next section. In other words, we aligned the size of the Java and of the Python datasets towards the smallest one (Python), due to our computational resources.

To summarize, at the end of this process we have three datasets (*short*, *medium*, and *long*) each split into training, validation, and test, all containing the same number of instances having, however, different lengths.

We also built a fourth dataset, named *mixed*, consisting of a mix of the three lengths: 1/3 of instances comes from the *short* dataset, 1/3 from the *medium*, and 1/3 from the *long*. In this case we only built the training and the validation sets, since we will test the model trained on the mixed dataset on the *short*, *medium* and *long* test sets.

6.2.1.2 Python dataset: block-level code completion task

Similarly to what we discussed for Java, we build three Python datasets (*short*, *medium*, and *long*) featuring instances (functions) having different lengths but characterized by the same task complexity (*i.e.*, same number of masked tokens to predict). The main difference between the Java and the Python dataset is that the latter simulates block-level completion, thus possibly featuring completions spanning across multiple statements.

The process used to build the Python datasets resembles the one we presented for Java. Thus, we only briefly summarize it here. We removed all functions not containing any code block (2,833,017). For consistency with the Java datasets, we decided to keep the same task complexity, meaning that we target the masking of the last 11 Python tokens within a given block. Thus, we remove from the dataset all functions not containing any valid block to mask (*i.e.*, a block consisting of at least 11 tokens).

We then sort the remaining functions by their length and split them into three sets of the same size: *short* (featuring functions with length from 30 to 150), *medium* (from 151 to 309), and *long* (from 310 to 1024). For each function F we created n instances, each one having a different block featuring its last 11 tokens masked. For the same reasons previously explained, we remove any duplicates created at this stage.

We replace all characters used to indent the code (*i.e.*, '`\n`', '`\t`' and extra white spaces) with a special tag: `<TAB>`. This allows to flatten each function without losing information about the indentation, which is fundamental for the Python syntax.

Finally, we split each dataset (*short*, *medium*, *long*) into training, validation, and test using the same procedure described for Java. For each dataset, the training contains 280k instances, while the evaluation and test contain 35k instances. These numbers have been dictated by the smallest dataset involved in our study, being the *short* Python dataset. Aligning the size of all datasets removes another possible confounding factor.

6.2.2 Data Collection & Analysis

Table 6.1: Hyperparameters used and searched.

Hyperparameter	**Values**
Learning Rate	1e-4
Batch Size	256
Inner Dimension	512
Encoder Max Length	1,024
Encoder Layers	6
Encoder Heads	8
Decoder Max Length	128
Decoder Layers	8
Decoder Heads	6

To answer our research question, we train eight models (four per each of the subject languages, namely Python and Java) for each of the four experimented position encoding schemes (*i.e.*, Sinusoidal, xPOS, ALiBi, and T5). This leads to a total of 32 trained models. The four models for each language have been trained on datasets featuring code completions having inputs (*i.e.*, the Java or Python function to complete) characterized by different lengths (*i.e.*, *short*, *medium*, *long*, and *mix*). Then, each of these models have been used to generate predictions on three test sets featuring code completions of different lengths (*i.e.*, *short*, *medium*, and *long*). This allows us to verify if, for example, a model trained on *short* code completions can generalize to a test set containing *long* instances. Also, we can verify whether a model trained on code completions having a mixture of lengths (*i.e.*, featuring short, medium, and long sequences) can achieve on each of the three test sets (*i.e.*, *short*, *medium*, and *long*) results competitive with those of models specialized (*i.e.*, trained only) on instances having a specific length. For example, we can check whether the model trained on the mixture of lengths achieves on the *short* test set performance comparable to those of the model trained on the *short* training set. Remember that the amount of instances in each training set is fixed. Thus, observed differences should be due to the length of the employed training instances.

To reduce confounding factors, we used the same hyperparameters amongst all 32 models. The adopted hyperparameters are those suggested in the paper originally proposing the Transformer architecture [343] and are reported in Table 6.1. This design decision also avoided the need for an expensive hyperparameters tuning involving 32 different models.

All models have been trained with the Adam optimizer [199] with a cosine learning rate scheduler using a warmup of 2,000 steps. We used a vocabulary size of 50k for the tokenizer, which was shared across all the models.

For implementing and training the Transformers, we used x-transformers [351] and Pytorch Lightning [126]. Additionally, when generating samples, we used Nucleus Sampling [175] with a $top_p = 0.95$ and stopped generations once the ⟨`EOS`⟩ token was produced or the maximum number of tokens, 128, were produced. We trained all models for a maximum of five epochs and used the best performing checkpoint based on validation loss, which happened to always be the models trained on all five epochs.

To assess the performance of the models on the test sets, we collect the predictions they generate and measure the percentage of Exact Match (EM) with respect to the expected target. An EM indicates that the code generated by the model for the completion instance is identical to the target (*i.e.*, the one we masked). We also compute metrics usually adopted in the assessment of generative models, namely the BLEU [274], ChrF [279], Levenshtein Distance [224], ROUGE [235], and METEOR [42] score with respect to the target.

BLEU is a popular automatic metric for machine translation tasks due to the high correlation to human judgement. It has become a standard metric in code completion tasks [247] since it measures the overlap of a predicted sequence and a set of reference sequences in terms of n-grams. ChrF is a character level metric which averages the F-score of 1 to 6-grams of characters. Levenshtein Distance is a measure of the minimal edit operations (*i.e.*, insert, modify, and remove), that would be needed to convert the predicted sequence into the target one, and it has been used in assessing the models' performance in previous code completion studies [95]. ROUGE, and specifically RougeL, is a metric that measures the longest common subsequence between the predicted and ground truth sequences. Lastly, Meteor is also an F-score, where the recall is weighted nine times more than the precision. Additionally, predictions are penalized for not having adjacent unigrams that exist in the ground truth. We also measure the Cross-Entropy of the generated predictions (*i.e.*, a measure of the surprise of the model when predicting the ground truth sequence).

While we computed all the above-described metrics, we only discuss the results achieved in terms of EM, ChrF, and RougeL. The former (EM) is an easy-to-interpret proxy of the model's performance. ChrF and RougeL, instead, have been found to be best at measuring performance compared to human evaluation and allow to claim significance (95% confidence) if the difference between two models on code generation tasks is greater than two points [124]. Our full analysis can be found in our replication package [99]. For implementing these metrics, we used the Huggingface's datasets library [226], which contains a large selection of automated metrics for the evaluation of generative models.

6.3 Results and Discussion

Table 6.2: Exact Match Score (↑) achieved by the different position encoding schemes.

Test Set	Encoding	Java				Python			
		Short	Medium	Long	Avg. Δ	Short	Medium	Long	Avg. Δ
Short	**Sinusoidal**	**10.81%**	2.91%	0.50%	84.23%	**1.57%**	0.26%	0.03%	90.76%
	xPOS	**12.49%**	4.87%	0.85%	77.10%	**2.33%**	0.64%	0.07%	84.76%
	ALiBi	**10.97%**	3.09%	0.36%	84.28%	**1.48%**	0.26%	0.03%	90.20%
	T5	**18.39%**	6.66%	1.38%	78.14%	**5.77%**	3.19%	1.51%	59.27%
Medium	**Sinusoidal**	1.61%	**6.67%**	3.32%	63.04%	0.57%	**1.33%**	0.72%	51.50%
	xPOS	2.85%	**9.36%**	8.02%	41.93%	1.53%	**2.84%**	1.73%	42.61%
	ALiBi	1.56%	**6.61%**	3.68%	60.36%	0.49%	**1.36%**	0.64%	58.46%
	T5	5.29%	**14.06%**	11.33%	40.90%	3.16%	**5.71%**	5.99%	19.88%
Long	**Sinusoidal**	0.60%	2.85%	**8.27%**	79.14%	0.05%	0.61%	**8.81%**	96.25%
	xPOS	1.57%	5.47%	**11.33%**	68.93%	0.48%	1.80%	**11.34%**	89.95%
	ALiBi	0.57%	2.79%	**8.29%**	79.73%	0.08%	0.50%	**8.84%**	96.72%
	T5	3.62%	9.59%	**17.03%**	61.22%	3.10%	7.19%	**20.73%**	75.18%

Tables 6.2, 6.3, and 6.4 show the results in terms of EM, ChrF, and RougeL, respectively, achieved by the four positional encoding schemes when trained on datasets featuring code completions of different lengths (columns) and tested on the *short*, *medium*, and *long* test sets (rows).

The results are reported for both Java and Python. To provide a concrete example, let us consider the EM results reported in the Table 6.2. Here, the Sinusoidal schema

Table 6.3: ChrF Score (↑) achieved by the different position encoding schemes.

Test Set	Encoding	Java Short	Java Medium	Java Long	Java Avg. Δ	Python Short	Python Medium	Python Long	Python Avg. Δ
Short	Sinusoidal	**30.72%**	17.70%	13.19%	49.72%	**37.48%**	32.56%	30.00%	16.54%
	xPOS	**34.57%**	23.36%	16.89%	41.78%	**41.83%**	37.51%	33.06%	15.65%
	ALiBi	**30.95%**	17.46%	13.02%	50.76%	**37.50%**	33.10%	30.31%	15.45%
	T5	**42.97%**	27.43%	17.37%	47.87%	**50.86%**	48.41%	43.30%	9.84%
Medium	Sinusoidal	16.57%	**25.01%**	19.94%	27.01%	35.99%	**37.17%**	35.21%	4.22%
	xPOS	21.41%	**30.96%**	27.68%	20.72%	40.84%	**43.95%**	39.30%	8.83%
	ALiBi	16.39%	**24.93%**	20.29%	26.43%	35.65%	**36.88%**	34.67%	4.66%
	T5	29.18%	**39.67%**	35.95%	17.91%	47.45%	**52.61%**	51.36%	6.09%
Long	Sinusoidal	13.93%	17.19%	**25.03%**	37.83%	32.79%	34.72%	**41.71%**	19.07%
	xPOS	18.30%	25.18%	**32.38%**	32.86%	38.36%	42.00%	**50.58%**	20.56%
	ALiBi	13.47%	17.24%	**24.91%**	38.36%	32.36%	34.50%	**41.66%**	19.76%
	T5	27.14%	36.42%	**44.93%**	29.27%	48.14%	54.86%	**64.96%**	20.72%

Table 6.4: ROUGE-L Score (↑) achieved by the different position encoding schemes.

Test Set	Encoding	Java Short	Java Medium	Java Long	Java Avg. Δ	Python Short	Python Medium	Python Long	Python Avg. Δ
Short	Sinusoidal	**38.80%**	24.56%	17.68%	45.57%	**39.39%**	33.61%	30.16%	19.05%
	xPOS	**45.61%**	35.23%	25.66%	33.25%	**45.01%**	40.10%	32.96%	18.84%
	ALiBi	**39.24%**	24.71%	17.85%	45.77%	**39.41%**	33.81%	30.62%	18.26%
	T5	**52.47%**	38.99%	25.92%	38.15%	**52.84%**	50.46%	44.92%	9.75%
Medium	Sinusoidal	22.08%	**29.63%**	23.63%	22.87%	36.26%	**37.15%**	33.82%	5.68%
	xPOS	30.98%	**41.28%**	36.71%	18.01%	43.14%	**46.43%**	41.28%	9.09%
	ALiBi	21.39%	**29.57%**	24.18%	22.95%	35.98%	**36.74%**	33.81%	5.02%
	T5	39.12%	**49.36%**	45.66%	14.12%	49.04%	**54.49%**	53.14%	6.24%
Long	Sinusoidal	19.05%	22.27%	**26.60%**	22.33%	32.80%	34.07%	**39.83%**	16.06%
	xPOS	26.79%	36.05%	**41.01%**	23.38%	41.17%	44.39%	**51.65%**	17.17%
	ALiBi	18.69%	22.33%	**26.66%**	23.07%	32.70%	33.90%	**39.80%**	16.33%
	T5	36.20%	46.35%	**52.84%**	21.89%	49.71%	56.35%	**65.52%**	19.06%

trained on *short* Java completions generated 10.81% EM predictions when tested on *short* instances (*i.e.*, those resembling the training set instances).

Instead, when the training is performed on instances having a *medium* length, the percentage of EM predictions drops, on the same *short* test set, to 2.91%, finally falling at 0.50% when the training was performed on *long* instances.

Similar results are observed for Python in which, however, the percentage of EM predictions is substantially lower, moving from 1.57% achieved on the *short* test set when training on *short* instances down to the 0.03% when tested on the *long* ones.

Tables 6.2, 6.3, and 6.4 also contain two "Avg. Δ" columns (one per language). Given a row in one of the tables (*e.g.*, the first row in Table 6.2 reporting the performance of the Sinusoidal schema when run on the *short* test set), the Avg. Δ indicates the relative change in performance observed, on average, for the models trained on different lengths (in our example, those trained on *medium* and *long* completions) when compared the one specialized on lengths related to that row (*i.e.*, *short*). Indeed, the 84.22% shown as average Δ in the subject row is the result of:

$$\frac{\frac{10.81\%-2.91\%}{10.81\%}+\frac{10.81\%-0.50\%}{10.81}}{2}=84.22\%$$

where 10.81% is the percentage of EM predictions for Java generated by the Sinusoidal schema when trained and tested on code completions of *short* length, while 2.91% and 0.50% are the EM scores achieved by the Sinusoidal schema when trained on *medium* and *long* instances, respectively, while still being tested on *short* instances.

Finally, Tables 6.2, 6.3, and 6.4 adopt three styles to highlight findings in the context of a specific test set length. Let us focus on the Java results achieved on the *short* test set in terms of EM (Table 6.2). The black box shows the best-performing combination of $\langle$*encoding schema*, *training length*$\rangle$ for such a test set (*i.e.*, T5 trained on *short* completions). The bold values highlight, for each encoding schema, the best-performing training length for such a test set (*i.e.*, in all cases, training on *short* instances works better when testing on *short* instances). The red value in each "Avg. Δ" column highlights, instead, the encoding schema manifesting the lowest relative drop in performances when moving from a training length matching the instances in the test set (*e.g.*, training on *short*, testing on *short*) to the other training lengths. In this case, the lowest relative drop in terms of EM predictions is exhibited by xPOS. Note that a lowest relative drop indicates a better ability of the encoding schema to generalize to unseen lengths.

The first observation that can be made from the three tables is that T5's positional encoding schema performs better than all other approaches. Such a finding is consistently captured by all metrics, including ChfR and RougeL for which the difference is always substantially higher than two points, indicating a statistically significant difference at 95% confidence [124]. Being the best performing one, however, does not save T5 from a strong general observation that can be made across the board for all training schemes: They all suffer from a major degradation in performance when applied on code completions having a length different from the one they have been trained on. Interestingly, the degradation is not only observed when the models are tested on instances being longer (likely more complex to handle) than those they have been trained on, but also in the opposite direction. This can be easily seen in Tables 6.2, 6.3, and 6.4 by the fact that (i) the average Δ values are always positive, and (ii) the bold values in a given test set length are always associated with the same length in the training set.

The second observation concerns the encoding schema reporting the lowest average drop in performance (red values in the "Avg. Δ" column in the three tables). Overall, also from this perspective, T5 seems to be the best choice. There are a few exceptions to this trend, depending on the test set under analysis and on the metric used as proxy for performance.

For example, on the *short* Java test set, T5 is the second best in class in terms of EM and ChfR score, while confirming its leadership when looking at the RougeL score. By considering all 18 combinations of test set length (3), language (2), and evaluation metrics (3), T5 is the one exhibiting the lowest relative drop in 11 (61%) of cases, and the second-best in additional 3 cases (17%). Still, as observed, T5 also exhibits major drops in performance when working on sequence lengths unseen during training. For example, there is an absolute drop of 13.41% in terms of EM predictions when testing T5 on the *long* dataset when trained on *short* sequences as compared to the one trained on *long* sequences (17.03% *vs* 3.62%). The trend is confirmed when looking at the ChfR and the RougeL scores.

xPOS is the second best performing schema, both in terms of absolute performance and generalization to different lengths. ALiBi and Sonusoidal follow, exhibiting similar performances from both perspectives.

Take Away #1: T5's positional encoding scheme achieves the best overall performance across metrics, lengths, and languages. Also, it is also better at generalizing to unseen lengths. In general, however, all encoding schemes suffer generalization issues for unseen lengths.

Differences across languages. Overall, our main findings hold on both languages. These include: (i) the lack of generalizability to unseen lengths of any of the experimented encoding schemes; and (ii) the superiority of T5 both in terms of absolute performance and relative drop when dealing with unseen lengths.

We do not compare the absolute performance achieved on the two languages since (i) the test sets are different, (ii) the code completion tasks are different (statement-level *vs.* block-level), and (iii) the syntaxt of the two languages make the prediction tasks quite different, since Python requires the generation of the `<TAB>` indentation tokens while Java does not.

Take Away #2: On both languages, all encoding schemes struggle to generalize to unseen lengths. T5 confirms its superiority on both Java and Python code.

Table 6.5: Exact Match Mix (↑) achieved by the different position encoding schemes.

	Java - Test Set			**Python - Test Set**		
	Short	**Medium**	**Long**	**Short**	**Medium**	**Long**
Sinusoidal	9.85%	7.05%	8.81%	1.03%	1.04%	7.50%
Δ	-8.88%	+5.70%	+6.53%	-34.39%	-21.80%	-14.87%
xPOS	11.99%	9.73%	11.42%	2.49%	2.76%	10.67%
Δ	-4.00%	+3.95%	+0.79%	+6.87%	-2.82%	-5.91%
ALiBi	10.12%	6.85%	8.53%	1.00%	0.98%	7.34%
Δ	-7.75%	+3.63%	+2.90%	-32.43%	-27.94%	-16.97%
T5	19.11%	17.57%	18.63%	3.19%	3.69%	12.22%
Δ	+3.92%	+24.96%	+9.40%	-44.71%	-35.38%	-41.05%

Table 6.6: ChrF Mix (↑) achieved by the different position encoding schemes.

	Java - Test Set			**Python - Test Set**		
	Short	**Medium**	**Long**	**Short**	**Medium**	**Long**
Sinusoidal	30.49%	27.04%	26.60%	36.48%	36.24%	40.91%
Δ	-0.75%	+8.12%	+6.27%	-2.67%	-2.50%	-1.92%
xPOS	34.97%	33.20%	33.34%	43.83%	45.07%	51.33%
Δ	+1.16%	+7.24%	+2.96%	+4.78%	+2.55%	+1.48%
ALiBi	30.42%	26.72%	26.28%	36.44%	36.14%	40.93%
Δ	-1.71%	+7.18%	+5.50%	-2.83%	-2.01%	-1.75%
T5	45.47%	46.11%	47.73%	45.94%	48.33%	55.00%
Δ	+5.82%	+16.23%	+6.23%	-9.67%	-8.14%	-15.33%

Impact of training diversity. Tables 6.5 (EM), 6.6 (ChrF), and 6.7 (RougeL) report the results achieved by the four encoding schemas (rows) when trained on the *mix* dataset

Table 6.7: ROUGE-L Mix (↑) achieved by the different position encoding schemes.

	Java - Test Set			**Python - Test Set**		
	Short	**Medium**	**Long**	**Short**	**Medium**	**Long**
Sinusoidal	36.85%	31.54%	29.37%	37.85%	36.18%	39.35%
Δ	-5.03%	+6.45%	+10.41%	-3.91%	-2.61%	-1.21%
xPOS	45.26%	43.36%	42.60%	46.27%	47.33%	52.25%
Δ	-0.77%	+5.04%	+3.88%	+1.94%	+1.94%	+1.16%
ALiBi	36.83%	31.03%	28.82%	37.80%	36.10%	39.28%
Δ	-6.14%	+4.94%	+8.10%	-1.74%	-1.74%	-1.31%
T5	52.60%	53.96%	55.18%	48.20%	50.47%	55.94%
Δ	+0.25%	+9.32%	+4.43%	-8.78%	-7.38%	-14.62%

(*i.e.*, the one featuring a mixture of instances taken from the *short*, *medium*, and *long* datasets) and tested on the three datasets featuring sequences of different length (columns).

Note that the *mix* training dataset has exactly the same number of instances of the other length-specific datasets, thus do not introducing a confounding variable related to the training size.

The Δ associated to each combination of encoding schema and test set is the relative change in performance with respect to the same schema exclusively trained for sequences of the corresponding length. For example, in terms of ChrF score (Table 6.6), T5 trained on *short* Java sequences achieves a 42.97% ChrF score when tested on the *short* dataset. Such a score grows to 45.47% when T5 is trained on the *mix* dataset, with a relative improvement equal to (45.47% - 42.97%)/42.97% = +5.82%. The achieved findings confirm the superiority of T5 in this scenario as well.

Most importantly, we found that relying on a mixture of lengths during training is generally sufficient to achieve results approaching, and in some cases improving, than those achieved by specifically training the model for the target sequence length.

Indeed, by comparing the relative Δ reported, for example, in Table 6.6 (ChfR scores when training on the *mix* dataset) to those reported in Table 6.3 (ChfR scores when training on datasets featuring functions having different lengths), it is possible to observe a major difference in terms of magnitude of the deltas, with those in 6.6 being substantially smaller. This indicates that, while in some cases training on a specific length range l could help in achieving better performances on test instances fitting l, training on a mixture of lengths is a safe choice, since it would not result in dramatic lost of performances as those observed in Table 6.3.

Take Away #3: Training on a mixture of lengths being representative of those that will be seen during testing but also including other types of lengths might be the safest choice in most of cases. Only in scenarios in which even minor increases in performance are considered valuable, experimenting with a combination of models specialized on different lengths might be worthwhile, to then decide the best strategy to adopt.

6.4 Threats to Validity and Limitations

Threats to Internal Validity. In order to control for various levels of bias that can creep into our evaluation, we ensured to hold as many variables as possible in our datasets and models constant. This involved ensuring that there were no duplicates across the different training splits, both in the input and target [25]. Also, we held constant the hyperparameters across our different models and only changed the type of length they were trained on. However, despite these thorough mitigation strategies, bias can still be present in our empirical study.

Threats to Construct Validity. To mitigate threats to construct validity, we calculated a range of different metrics that have been commonly used in code completion literature. Additionally, we focus our discussion either on metrics that have been shown to correlate with human preference and that are more statistically stable [124] (*i.e.*, ChfR and RougeL) or that allow for a simple interpretation such as EM.

While there have been new recent metrics that are specific to code data for code completion, namely CodeBLEU [293] and functional-correctness [168, 85], we did not compute these for the following reasons. CodeBLEU has been shown to be not as stable as ChrF and RougeL [124]. Unfortunately, functional-correctness was not even an option for our evaluation due to the lack of unit tests for our test examples.

Besides the metric used, the type of code completion performed can result in bias as there has been some studies showing that synthetic benchmarks of code completion where the completions are randomized do not necessarily reflect the performance of real-world code completions [167].

Threats to External Validity. We investigated two popular Transformer architectures to mitigate the threats to external validity of our results. Additionally, we measured multiple types of metrics and constructed our datasets in such a way as to hopefully mimic realistic code completion scenarios. Additionally, we used two popular programming languages, namely Java and Python, to better ensure our results generalize across languages.

6.5 Bibliographical Notes

The paper supporting the content described in this Chapter was written in collaboration with other members of the SEMERU group at William & Mary and researchers from Universitá della Svizzera italiana. It is currently under review for publication.

Cooper, N., Tufano, R., Bavota, G., & Poshyvanyk, D. (2023, March). On the Generalizability of Transformer Models for Code Completion. Under Review.

Chapter 7

Conclusions & Future Research

In this dissertation, I have discussed my current progress and planned future work in my dissertation. Specifically, I discussed our work in first conducting a literature review and then using the findings to guide the development of intelligent software development and maintenance tools to assist software engineers in a variety of tasks. To give a succinct overview, I discuss each projects' conclusions and future work below.

7.1 DL4SE Literature Review

In Chapter 3, we presented a systematic literature review 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.

Future work. 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 described in this review should aid researchers and developers in understanding where DL can be applied and necessary considerations for applying these complex models to automate SE tasks.

7.2 Video-Based Bug Reporting

Chapter 4 presented TANGO, an approach that combines visual and textual information to help developers find duplicate video-based bug reports. Our empirical evaluation, conducted on 4,680 duplicate detection tasks created from 180 video-based bug reports from six mobile apps, illustrates that TANGO is able to effectively identify duplicate reports and save developer effort. Specifically, TANGO correctly suggests duplicate video-based bug reports within the top-2 candidate videos for 83% of the tasks, and saves 65.1% of the time that humans spend finding duplicate videos.

Future work. We will focus on addressing TANGO's limitations and extending TANGO's evaluation. Specifically, we plan to (1) explore additional ways to address the vocabulary overlap problem, (2) investigate the resilience of TANGO to different app characteristics such as the use of different themes, languages, and screen sizes, (3) extend TANGO for detecting duplicate bug reports that contain multimedia information (text, images, and videos), (4) evaluate TANGO using data from additional apps, and (5) assess the usefulness of TANGO in industrial settings.

7.3 Impact Analysis with Deep Learning and Call Graphs

In Chapter 5, we proposed Athena, an information retrieval technique for the task of impact analysis that combines neural code semantic embedding information with structural software call graph information. Additionally, we constructed a large benchmark on impact analysis that has been manually verified of bug fixing commits. On our new benchmark, Athena outperforms techniques without contextual call graph information by a large margin (+4.58% mRR, +3.85% mAP, and +5.67% HIT@10) and is robust across software systems with 22 out of 25 systems seeing improvement. Additionally, through our analysis, we found Athena's performance increase is on its ability to better find impacted methods to a change when the methods are outside of the query method's class.

Future work. We will explore using Athena for other tasks in software engineering such as code search, clone detection, traceability, *etc.*. Additionally, we will look at different ways of integrating the two types of information such as through the usage of Graph Neural Networks (GNNs). Lastly, we will explore Athena's ability to work with other languages to see if the approach generalizes.

7.4 Generalization of Code Completion Models

Lastly, in Chapter 6, we explored the generalization ability of popular decoder-only Transformer position encoding schemes that have shown success in Natural Language Processing that can be extended to the encoder-decoder Transformer and code completion task. Specifically, we investigated four different positional encoding schemes, namely Sinusoidal [343], xPOS [317, 321], ALiBi [283], and T5 [285], which have been proposed as a way to boost this generalization ability and represent the most popular position encoding types, *i.e.*, Absolute Positional Encoding and Relative Positional Encoding.

Overall, our results demonstrate that none of the studied positional encoding schemes has the ability to generalize to unseen lengths. While these findings suggest that there are currently no "shortcuts" for researchers or developers of tools utilizing these models to efficiently train on short lengths (which are less expensive to process) and generalize to longer lengths, it is interesting to note that training on a mixture of lengths should represent a safe compromise in most of cases. Still, the possible drop in performance this may result in should be considered and assessed case by case, depending on the context in which the models must be used, the targeted programming language, and the actual focus on performance.

Moreover, it is worth considering that our conclusions are only based on performance proxies we adopted (*i.e.*, EM, ChrF, and RougeL). Different trade-offs come into play if best performance is not the only constraint. For example, while T5 is the best performing positional encoding, it is also the slowest and most memory intensive both for training and inference. Therefore, in performance-critical settings it might be more beneficial to use xPOS, which achieves less performance, but is more efficient. Similar observations can be made for the Sinusoidal and ALiBi schemes for which, however, the cost to pay in terms of performance as compared to T5 is noticeably higher.

Future work. Given our findings and the potential impact that the generalizability of code completions models can have on the software engineering community in terms of training efficiency, we believe future research should explicitly target this problem with research focused outside of different positional encoding schemes and possibly involving additional architectural changes to the Transformer or even proposing completely novel architectures.

Bibliography

[1] Tesseract ocr library `https://github.com/tesseract-ocr/tesseract/wiki`.

[2] Acm artifact review policies `https://www.acm.org/publications/policies/artifact-review-badging`, 2019.

[3] Antennapod `https://github.com/AntennaPod/AntennaPod`, 2019.

[4] Droidweight `https://github.com/sspieser/droidweight`, 2019.

[5] Gnucash `https://github.com/codinguser/gnucash-android`, 2019.

[6] Growtracker `https://tinyurl.com/yy9oezom`, 2019.

[7] Time tracker `https://github.com/netmackan/ATimeTracker`, 2019.

[8] Token `https://github.com/markmcavoy/androidtoken`, 2019.

[9] Bird eats bugs `https://birdeatsbug.com/`, 2020.

[10] Bug squasher `https://thebugsquasher.com/`, 2020.

[11] Bugclipper `http://bugclipper.com`, 2020.

[12] Bugreplay `https://www.bugreplay.com/`, 2020.

[13] Bugsee `https://www.bugsee.com/`, 2020.

[14] Instabug `https://instabug.com/screen-recording`, 2020.

[15] Lucene's tfidfsimilarity javadoc - `https://tinyurl.com/ybhqqrqm`, 2020.

[16] Outklip `https://outklip.com/`, 2020.

[17] Python tesseract `https://github.com/madmaze/pytesseract`, 2020.

[18] Snaffu `https://snaffu.squarespace.com/`, 2020.

[19] Testfairy `https://testfairy.com`, 2020.

[20] Ubertesters `https://ubertesters.com/bug-reporting-tools/`, 2020.

[21] Welcome to core `https://www.core.edu.au/home`. 2023.

[22] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from data: a short course.* AMLbook.com, 2012.

[23] Mohammad Alahmadi, Abdulkarim Khormi, Biswas Parajuli, Jonathan Hassel, Sonia Haiduc, and Piyush Kumar. Code localization in programming screencasts. *EMSE'20*, 25(2):1536–1572, 2020.

[24] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.

[25] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.

[26] 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.

[27] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[28] Miltiadis Allamanis, Hao Peng, and Charles 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*, Maria-Florina Balcan and Kilian Q. Weinberger, editors, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2091–2100. JMLR.org, 2016.

[29] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[30] Amit Seal Ami, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 614–631. IEEE, 2022.

[31] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE'05*, pages 402–411, 2005.

[32] Plamen Angelov and Eduardo Soares. Towards explainable deep neural networks (xdnn). *Neural Networks*, 130:185–194, 2020.

[33] Anonymous. online appendix `https://anonymous.4open.science/r/Athena-6557/`, 2023.

[34] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. Identifying the starting impact set of a maintenance request: A case study. In *Proceedings of the fourth European conference on software maintenance and reengineering*, pages 227–230. IEEE, 2000.

[35] Forough Arabshahi, Sameer Singh, and Animashree Anandkumar. Combining symbolic expressions and black-box function evaluations in neural programs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[36] Erik Arisholm, Lionel C Briand, and Audun Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on software engineering*, 30(8):491–506, 2004.

[37] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[38] Gareth Ari Aye, Seohyun Kim, and Hongyu Li. Learning autocompletion from real-world datasets. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 131–139. IEEE, 2021.

[39] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: a control call graph based technique. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 9–pp. IEEE, 2005.

[40] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[41] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[42] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[43] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, and Bo Zhou. scvripper: video scraping tool for modeling developers' behavior using interaction data. In *ICSE'15*, pages 673–676. IEEE Press, 2015.

[44] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. Autopandas: Neural-backed generators for program synthesis. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[45] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. An empirical study on the developers' perception of software coupling. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 692–701. IEEE, 2013.

[46] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. volume 3951, pages 404–417, 07 2006.

[47] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 1–6, 2018.

[48] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *EICS'18*, page 3. ACM, 2018.

[49] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, pages 3589–3601, 2018.

[50] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. Translating video recordings of mobile app usages into replayable scenarios. In *ICSE'20*, 2020.

[51] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *FSE'08*, pages 308–318, New York, NY, USA, 2008. ACM.

[52] Nicolas Bettenburg, R. Premraj, T. Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful... really? In *ICSM'08*, pages 337–345, Sept 2008.

[53] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. In *MSR'08*, MSR '08, pages 27–30, New York, NY, USA, 2008. ACM.

[54] 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.

[55] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*, chapter An Introduction to Software Change Impact Analysis, pages 1–26. 1996.

[56] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. Discovering flaws in Security-Focused static analysis tools for android using systematic mutation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1263–1280, Baltimore, MD, August 2018. USENIX Association.

[57] Markus Borg, Per Runeson, Jens Johansson, and Mika V. Mäntylä. A Replicated Study on Duplicate Detection: Using Apache Lucene to Search Among Android Defects. In *ESEM'14*, pages 8:1–8:4, 2014.

[58] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, 2009.

[59] Lionel C Briand, Jurgen Wust, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 475–482. IEEE, 1999.

[60] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.

[61] Max Brunsfeld, Patrick Thomson, Andrew Hlynskyi, Josh Vera, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Hendrik van Antwerpen, Michael Davis, Ika, Tuan-Anh Nguyen, Stafford Brunk, Niranjan Hasabnis, bfredl, Mingkai Dong, Vladimir Panteleev, ikrima, Steven Kalt, Kolja Lampe, Alex Pinkus, Mark Schmitz, Matthew Krupcale, narpfel, Santos Gallegos, Vicent Martí, Edgar, and George Fraser. tree-sitter/tree-sitter: v0.20.7, September 2022.

[62] Lutz Büch and Artur Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104. IEEE, 2019.

[63] Nghi D. Q. Bui, Lingxiao Jiang, and Yijun Yu. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, volume WS-18 of *AAAI Workshops*, pages 758–761. AAAI Press, 2018.

[64] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. Bilateral dependency neural networks for cross-language algorithm classification. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 422–433, 2019.

[65] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[66] Lutz Büch and Artur Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104, 2019.

[67] Anabela Caetano, Neri Goulart, Manuel Fonseca, and Joaquim Jorge. Javasketchit: Issues in sketching the look of user interfaces. In *SSS'02*, pages 9–14, 2002.

[68] Haipeng Cai and Raul Santelices. A comprehensive study of the predictive accuracy of dynamic change-impact analysis. *Journal of Systems and Software*, 103:248–265, 2015.

[69] Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[70] Yang Cai, Linjun Yang, Wei Ping, Fei Wang, Tao Mei, Xian-Sheng Hua, and Shipeng Li. Million-scale near-duplicate video retrieval system. In *MM'11*, page 837–838, New York, NY, USA, 2011.

[71] Gerardo Canfora and Luigi Cerulo. Impact analysis by mining software and change request repositories. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 9–pp. IEEE, 2005.

[72] Gerardo Canfora and Luigi Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 105–111, 2006.

[73] Joao Carreira and Andrew Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. In *CVPR'17*, 2017.

[74] Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. Associating the visual representation of user interfaces with their internal structures and metadata. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 245–256, New York, NY, USA, 2011. ACM.

[75] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent

Ng. Assessing the quality of the steps to reproduce in bug reports. In *ESEC/FSE'19*, Bergamo, Italy, Ausgust 2019 2019.

[76] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. On the vocabulary agreement in software issue descriptions. In *ICSME'16*, pages 448–452, 2016.

[77] Oscar Chaparro, Juan Manuel Florez, Unnati Singh, and Andrian Marcus. Reformulating queries for duplicate bug report detection. In *SANER'19*, pages 218–229, 2019.

[78] K. Charmaz. *Constructing Grounded Theory.* SAGE Publications Inc., 2006.

[79] Gal Chechik, Varun Sharma, Uri Shalit, and Samy Bengio. Large scale online learning of image similarity through ranking. *JMLR'10*, 11:1109–1135, March 2010.

[80] 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.

[81] Chao Chen, Wenrui Diao, Yingpei Zeng, Shanqing Guo, and Chengyu Hu. Drlgencert: Deep learning-based automated testing of certificate verification in SSL/TLS implementations. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 48–58. IEEE Computer Society, 2018.

[82] 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.

[83] 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.

[84] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, and Liming Zhu. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In *ESEC/FSE'20*, page to appear, 2020.

[85] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[86] 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.

[87] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *ICML'20*, pages 1597–1607, 2020.

[88] X. Chen and C. Zhang. An interactive semantic video mining and retrieval platform–application in transportation surveillance video for incident detection. In *ICDM'06*, pages 129–138, 2006.

[89] Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from input-output examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of transformer models for code completion. *IEEE Transactions on Software Engineering*, 2021.

[96] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[97] Michael L Collard, Michael John Decker, and Jonathan I Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source

code: A tool demonstration. In *2013 IEEE International conference on software maintenance*, pages 516–519. IEEE, 2013.

[98] Aidan Connor, Aaron Harris, Nathan Cooper, and Denys Poshyvanyk. Can we automatically fix bugs by learning edit operations? In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 782–792. IEEE, 2022.

[99] N. Cooper, R. Tufano, G. Bavota, and D. & Poshyvanyk. Completeformer replication package `https://github.com/WM-SEMERU/completeformer`. *GitHub*, 2023.

[100] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. Tango's online appendix `https://github.com/ncoop57/tango`, 2020.

[101] 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)*, ICSME'15, pages 556–560, September 2015. ISSN:.

[102] 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.

[103] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. Open vocabulary learning on source code with a graph-structured cache. In *Proceedings of the 36th International Conference on Machine Learning*, Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, volume 97 of *Proceedings of Machine Learning Research*, pages 1475–1485. PMLR, 09–15 Jun 2019.

[104] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

[105] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[106] 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.

[107] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 46–57, 2019.

[108] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[109] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *UIST'17*, 2017.

[110] 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*, ICML'17, page 980–989. JMLR.org, 2017.

[111] 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.

[112] Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew J. Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, pages 2080–2088, 2017.

[113] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[114] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[115] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, Doina Precup and Yee Whye Teh, editors, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR, 2017.

[116] Morgan Dixon, Daniel Leventhal, and James Fogarty. Content and hierarchy in pixel-based methods for reverse engineering interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 969–978, New York, NY, USA, 2011. ACM.

[117] Pedro Domingos. Occam's two razors: The sharp and the blunt. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD'98, page 37–43. AAAI Press, 1998.

[118] M. Douze, H. Jegou, and C. Schmid. An image-based approach to video copy detection with spatio-temporal post-filtering. *TMM'10*, 12(4):257–266, 2010.

[119] Yann Dubois, Gautier Dagan, Dieuwke Hupkes, and Elia Bruni. Location Attention for Extrapolation to Longer Sequences. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 403–413, Online, July 2020. Association for Computational Linguistics.

[120] 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.

[121] 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.

[122] 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.

[123] C. Escobar-Velásquez, M. Osorio-Riaño, and M. Linares-Vásquez. Mutapk: Source-codeless mutant generation for android apps. In *ASE'19*, pages 1090–1093, 2019.

[124] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *arXiv preprint arXiv:2208.03133*, 2022.

[125] Sarah Fakhoury, Venera Arnaoudova, Cedric Noiseux, Foutse Khomh, and Giuliano Antoniol. Keep it simple: Is deep learning good for linguistic smell detection? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 602–611, 2018.

[126] William Falcon and The PyTorch Lightning team. PyTorch Lightning, 3 2019.

[127] Ming Fan, Xiapu Luo, Jun Liu, Meng Wang, Chunyin Nong, Qinghua Zheng, and Ting Liu. Graph embedding based familial analysis of android malware using unsupervised learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 771–782, 2019.

[128] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[129] Stephen Fink and Julian Dolby. Wala–the tj watson libraries for analysis, 2012.

[130] David Freedman, Robert Pisani, and Roger Purves. *Statistics (4th edn.)*. W. W. Norton & Company, 2007.

[131] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[132] Christian Frisson, Sylvain Malacria, Gilles Bailly, and Thierry Dutoit. Inspectorwidget: A system to analyze users behaviors in their applications. In *CHI'16*, pages 1548–1554. ACM, 2016.

[133] 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'17, pages 49–60, New York, NY, USA, 2017. ACM.

[134] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 13–23. IEEE, 2003.

[135] Keith Brian Gallagher. *Using program slicing in software maintenance.* University of Maryland, Baltimore County, 1990.

[136] 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.

[137] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. A neural model for method name generation from functional description. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 414–421, 2019.

[138] 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.

[139] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 430–440, 2012.

[140] Malcom Gethers and Denys Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *2010 IEEE International Conference on Software Maintenance*. IEEE, September 2010.

[141] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning, 2019.

[142] GitHub. Github copilot your ai pair programmer. 2023.

[143] 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.

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

[145] 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.

[146] 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.

[147] C. Guo, W. Wang, Y. Wu, N. Dong, Q. Ye, J. Xu, and S. Zhang. Systematic comprehension for developer reply in mobile system forum. In *2019 IEEE 26th Inter-*

national Conference on Software Analysis, Evolution and Reengineering (SANER), pages 242–252, Los Alamitos, CA, USA, feb 2019. IEEE Computer Society.

[148] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[149] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[150] 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.

[151] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep reinforcement learning for syntactic error repair in student programs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:930–937, 07 2019.

[152] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep reinforcement learning for syntactic error repair in student programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 930–937, 2019.

[153] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31, 2017.

[154] Dan Gusfield. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News*, 28(4):41–60, 1997.

[155] Huong Ha and Hongyu Zhang. Deepperf: Performance prediction for configurable software with deep sparse neural network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1095–1106, 2019.

[156] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[157] 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.

[158] Y. Hao, T. Mu, R. Hong, M. Wang, N. An, and J. Y. Goulermas. Stochastic multiview hashing for large-scale near-duplicate video retrieval. *TMM'17*, 19(1):1–14, 2017.

[159] 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.

[160] Youssef Hassoun, Roger Johnson, and Steve Counsell. A dynamic runtime coupling metric for meta-level architectures. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 339–346. IEEE, 2004.

[161] Erik Hatcher and Otis Gospodnetic. *Lucene in Action*. Manning Publications, 2004.

[162] Jianjun He, Ling Xu, Meng Yan, Xin Xia, and Yan Lei. Duplicate bug report detection using dual-channel convolutional neural networks. In *ICPC'20*, page to appear, 2020.

[163] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR'16*, pages 770–778, 2016.

[164] 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.

[165] 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.

[166] 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.

[167] Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, and Alberto Bacchelli. When code completion fails: A case study on real-world completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 960–970. IEEE, 2019.

[168] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He,

Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[169] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher A Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, et al. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering*, 27(6):125, 2022.

[170] Abram Hindle, Anahita Alipour, and Eleni Stroulia. A contextual approach towards more accurate duplicate bug report detection and ranking. *EMSE'16*, 21(2):368–410, 2016.

[171] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[172] Abram Hindle and Curtis Onuczko. Preventing duplicate bug reports by continuously querying bug reports. *EMSE'18*, pages 1–35, 2018.

[173] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: An end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45, 2019.

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

[175] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

[176] Daqing Hou and David M Pletcher. Towards a better code completion system by api grouping, filtering, and popularity-based ranking. In *Proceedings of the*

2nd International Workshop on Recommendation Systems for Software Engineering, pages 26–30, 2010.

[177] Gang Hu, Linjie Zhu, and Junfeng Yang. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *ESEC/FSE'18*, pages 269–282. ACM Press.

[178] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 200–210, New York, NY, USA, 2018. Association for Computing Machinery.

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

[180] Yuan Huang, Xiangping Chen, Qiwen Zou, and Xiaonan Luo. A probabilistic neural network-based approach for related software changes detection. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 279–286. IEEE, 2014.

[181] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. Deep transfer bug localization. *IEEE Transactions on Software Engineering*, 47(7):1368–1380, 2021.

[182] Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. Compositionality decomposed: How do neural networks generalise? *Journal of Artificial Intelligence Research*, 67:757–795, 2020.

[183] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

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

[185] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: Multi-token code completion by jointly learning from structure and naming sequences. *arXiv preprint arXiv:2202.06689*, 2022.

[186] He Jiang, Najam Nazar, Jingxuan Zhang, Tao Zhang, and Zhilei Ren. PRST: A PageRank-Based Summarization Technique for Summarizing Bug Reports with Duplicates. *International Journal of Software Engineering and Knowledge Engineering*, 27(06):869–896, 2017.

[187] Y. Jiang and J. Wang. Partial copy detection in videos: A benchmark and an evaluation of popular methods. *TBD'16*, 2(1):32–42, 2016.

[188] Yu-Gang Jiang, Chong-Wah Ngo, and Jun Yang. Towards optimal bag-of-features for object categorization and semantic video retrieval. In *IVR'07*, pages 494–501, 07 2007.

[189] Xianhao Jin and Francisco Servant. The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 70–73, 2018.

[190] Weizhen Jing, Xiushan Nie, Chaoran Cui, Xiaoming Xi, Gongping Yang, and Yilong Yin. Global-view hashing: harnessing global relations in near-duplicate video retrieval. *WWW'19*, 22(2):771–789, 2019.

[191] Jing Huang, S. R. Kumar, M. Mitra, Wei-Jing Zhu, and R. Zabih. Image indexing using color correlograms. In *CVPR'97*, pages 762–768, 1997.

[192] N. Jones. Seven best practices for optimizing mobile testing efforts. Technical Report G00248240, Gartner.

[193] H. Jégou, M. Douze, C. Schmid, and P. Pérez. Aggregating local descriptors into a compact image representation. In *CVPR'10*, pages 3304–3311, 2010.

[194] Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering*, 18:933–969, 2013.

[195] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[196] Li Kang. Automated Duplicate Bug Reports Detection - An Experiment at Axis Communication AB. Master's thesis, 2017.

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

[198] Deborah S. Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356, 2018.

[199] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

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

[201] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

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

[203] Shun Kiyono, Sosuke Kobayashi, Jun Suzuki, and Kentaro Inui. Shape: Shifted absolute position embedding for transformers. *arXiv preprint arXiv:2109.05644*, 2021.

[204] Nathan Klein, Christopher S. Corley, and Nicholas A. Kraft. New Features for Duplicate Bug Detection. In *MSR'14*, pages 324–327. ACM, 2014.

[205] G. Kordopatis-Zilos, S. Papadopoulos, I. Patras, and I. Kompatsiaris. Fivr: Fine-grained incident video retrieval. *TMM'19*, 21(10):2638–2652, 2019.

[206] G. Kordopatis-Zilos, S. Papadopoulos, I. Patras, and Y. Kompatsiaris. Near-duplicate video retrieval with deep metric learning. In *ICCVW'17*, pages 347–356, 2017.

[207] Giorgos Kordopatis-Zilos, Symeon Papadopoulos, Ioannis Patras, and Ioannis Kompatsiaris. Near-duplicate video retrieval by aggregating intermediate cnn layers. In *MMM'17*, volume 10132, pages 251–263, 01 2017.

[208] Jaakko Korpi and Jussi Koskinen. Supporting impact analysis by program dependence graph based forward slicing. In *Advances and innovations in systems, computing sciences and software engineering*, pages 197–202. Springer, 2007.

[209] W. Kraaij and G. Awad. Trecvid 2011 content-based copy detection: Task overview. in Online Proc. TRECVid, 2010,2011.

[210] 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.

[211] Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pages 2873–2882. PMLR, 2018.

[212] 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.

[213] Walter S. Lasecki, Juho Kim, Nick Rafter, Onkur Sen, Jeffrey P. Bigham, and Michael S. Bernstein. Apparition: Crowdsourced user interfaces that come to life as you sketch them. In *CHI'15*, page 1925–1934, New York, NY, USA, 2015. Association for Computing Machinery.

[214] Alina Lazar, Sarah Ritchey, and Bonita Sharif. Improving the Accuracy of Duplicate Bug Report Detection Using Textual Similarity Measures. In *MSR'14*, pages 308–311, 2014.

[215] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[216] 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.

[217] 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, page 106–117, New York, NY, USA, 2018. Association for Computing Machinery.

[218] 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, page 795–806. IEEE Press, 2019.

[219] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3931–3937, June 2019.

[220] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[221] 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.

[222] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 41–50, 2011.

[223] J. Lerch and M. Mezini. Finding Duplicates of Your Yet Unwritten Bug Report. In *CSMR'13*, pages 69–78, 2013.

[224] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[225] 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.

[226] Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, Joe Davison, Mario Šaško, Gunjan Chhablani, Bhavitvya Malik, Simon Brandeis, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Clément Delangue, Théo Matussière, Lysandre Debut, Stas Bekman, Pierric Cistac, Thibault Goehringer, Victor Mustar, François Lagunas, Alexander Rush, and Thomas Wolf. Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[227] 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.

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

[229] 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.

[230] Mingyang Li, Lin Shi, and Qing Wang. Are all duplicates value-neutral? an empirical analysis of duplicate issue reports. In *QRS'19*, pages 272–279. IEEE, 2019.

[231] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

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

[233] Tatiana Likhomanenko, Qiantong Xu, Gabriel Synnaeve, Ronan Collobert, and Alex Rogozhnikov. Cape: Encoding relative positions with continuous augmented positional embeddings. *Advances in Neural Information Processing Systems*, 34:16079–16092, 2021.

[234] 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.

[235] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[236] 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.

[237] 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.

[238] 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.

[239] 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.

[240] K. Liu, H. Beng Kuan Tan, and H. Zhang. Has this bug been reported? In *WCRE'13*, pages 82–91, 2013.

[241] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1–12, 2019.

[242] 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.

[243] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid C programs for fuzz testing. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 1044–1051. AAAI Press, 2019.

[244] Xiaoyu Liu, LiGuo Huang, Alexander Egyed, and Jidong Ge. Do code data sharing dependencies support an early prediction of software actual change impact set? *Journal of Software: Evolution and Process*, 30(11):e1960, 2018.

[245] Yibin Liu, Yanhui Li, Jianbo Guo, Yuming Zhou, and Baowen Xu. Connecting software metrics across versions to predict defects. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 232–243, 2018.

[246] David Lowe. Distinctive image features from scale-invariant keypoints. *JCV'04*, 60:91–, 11 2004.

[247] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[248] 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.

[249] L. MacLeod, M. Storey, and A. Bergen. Code, camera, action: How software developers document and share program knowledge using youtube. In *ICPC'15*, pages 104–114, 2015.

[250] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: Inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 304–315, 2019.

[251] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. *ACM Sigplan Notices*, 40(6):48–61, 2005.

[252] Ke Mao, Mark Harman, and Yue Jia. Crowd intelligence enhances automated mobile testing. In *ASE'17*, pages 16–26, Piscataway, NJ, USA, 2017. IEEE Press.

[253] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering*, pages 1–20, 2022.

[254] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347. IEEE, 2021.

[255] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[256] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo Frias. Training binary classifiers as data structure invariants. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 759–770. IEEE Press, 2019.

[257] K. P. Moran, C. Bernal-Cárdenas, 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.

[258] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Auto-completing bug reports for android applications. Bergamo, Italy, August-September 2015.

[259] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *ICST'16*, pages 33–44. IEEE, 2016.

[260] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 1287–1293. AAAI Press, 2016.

[261] 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.

[262] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[263] Maleknaz Nayebi. Eye of the mind: Image processing for social coding. In *ICSE'20*, page 49–52, 2020.

[264] Masato Neishi and Naoki Yoshinaga. On the relation between position information and sentence length in neural machine translation. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pages 328–338, 2019.

[265] Benjamin Newman, John Hewitt, Percy Liang, and Christopher D Manning. The eos decision and length extrapolation. *arXiv preprint arXiv:2010.07174*, 2020.

[266] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N. Nguyen. A deep neural network language model with contexts for source code. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 323–334, 2018.

[267] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling. In *ASE'12*, pages 70–79, 2012.

[268] Son Nguyen, Tien Nguyen, Yi Li, and Shaohua Wang. Combining program analysis and statistical language model for code statement completion. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 710–721. IEEE, 2019.

[269] Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with remaui. In *ASE'15*, pages 248–259, Washington, DC, USA, 2015.

[270] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[271] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

[272] 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.

[273] Annibale Panichella. A systematic comparison of search algorithms for topic modelling—a study on duplicate bug report identification. In *SSBSE'19*, pages 11–26. Springer, 2019.

[274] 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, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[275] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[276] Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 518–528, 2019.

[277] 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.

[278] L. Ponzanelli, G. Bavota, A. Mocci, R. Oliveto, M. D. Penta, S. Haiduc, B. Russo, and M. Lanza. Automatic identification and classification of software development video tutorial fragments. *TSE'19*, 45(5):464–488, 2019.

[279] Maja Popović. chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, 2015.

[280] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical software engineering*, 14:5–32, 2009.

[281] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[282] Devanbu Prem, Matthew Dwyer, Sebastian Elbaum, Michael Lowry, Kevin Moran, Denys Poshyvanyk, Baishakhi Ray, Rishabh Singh, and Xiangyu Zhang. Deep learning & software engineering: State of research and future directions. In *Proceedings of the 2019 NSF Workshop on Deep Learning and Software Engineering*, 2019.

[283] Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*, 2022.

[284] Ofir Press, Noah A Smith, and Mike Lewis. Shortformer: Better language modeling using shorter inputs. *arXiv preprint arXiv:2012.15832*, 2020.

[285] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020.

[286] M. S. Rakha, C. Bezemer, and A. E. Hassan. Revisiting the performance evaluation of automated approaches for the retrieval of duplicate issue reports. *TSE'18*, 44(12):1245–1268, 2018.

[287] Mohamed Sami Rakha, Cor-Paul Bezemer, and Ahmed E. Hassan. Revisiting the performance of automated approaches for the retrieval of duplicate re-

ports in issue tracking systems that perform just-in-time duplicate retrieval. *EMSE*, 23(5):2597–2621, 2018.

[288] Tharindu Ranasinghe, Constantin Orăsan, and Ruslan Mitkov. Semantic textual similarity with siamese neural networks. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2019)*, pages 1004–1011, 2019.

[289] 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.

[290] 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*, pages 419–428, 2014.

[291] 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, page 419–428, New York, NY, USA, 2014. Association for Computing Machinery.

[292] Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun, editors, 2016.

[293] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[294] Xiaoxia Ren, Barbara G Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings of the 27th international conference on Software engineering*, pages 664–665, 2005.

[295] J. Revaud, M. Douze, C. Schmid, and H. Jégou. Event retrieval in large video collections with circulant temporal encoding. In *CVPR'13*, pages 2459–2466, 2013.

[296] Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical software engineering*, 16:773–811, 2011.

[297] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4902–4912, Online, July 2020. Association for Computational Linguistics.

[298] Romain Robbes and Michele Lanza. How program history can improve code completion. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326. IEEE, 2008.

[299] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010.

[300] Irving Muller Rodrigues, Daniel Aloise, Eraldo Rezende Fernandes, and Michel Dagenais. A soft alignment model for bug deduplication. In *MSR'20*, pages 43–53, 2020.

[301] 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.

[302] Jan Rosendahl, Viet Anh Khoa Tran, Weiyue Wang, and Hermann Ney. Analysis of positional encodings for neural machine translation. In *Proceedings of the 16th International Conference on Spoken Language Translation*, 2019.

[303] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[304] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of Duplicate Defect Reports Using Natural Language Processing. In *ICSE'07*, pages 499–510, 2007.

[305] Barbara G Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, 2001.

[306] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: detection of clones in the twilight zone. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, pages 354–365. ACM, 2018.

[307] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., USA, 1986.

[308] Raul Santelices and Mary Jean Harrold. Probabilistic slicing for predictive impact analysis. Technical report, Georgia Institute of Technology, 2010.

[309] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[310] 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. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 584–593. IEEE Computer Society, 2017.

[311] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[312] Shu-Ting Shi, Ming Li, David Lo, Ferdian Thung, and Xuan Huo. Automatic code review by learning the revision of source code. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:4910–4917, 07 2019.

[313] 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.

[314] 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.

[315] Sivic and Zisserman. Video google: a text retrieval approach to object matching in videos. In *CCV'03*, pages 1470–1477 vol.2, 2003.

[316] 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.

[317] Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

[318] C. Sun, D. Lo, S. C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE'11*, pages 253–262, 2011.

[319] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval. In *ICSE'10*, pages 45–54, 2010.

[320] 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, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[321] Yutao Sun, Li Dong, Barun Patra, Shuming Ma, Shaohan Huang, Alon Benhaim, Vishrav Chaudhary, Xia Song, and Furu Wei. A length-extrapolatable transformer. *arXiv preprint arXiv:2212.10554*, 2022.

[322] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. A grammar-based structural cnn decoder for code generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:7055–7062, 07 2019.

[323] A. Sureka and P. Jalote. Detecting Duplicate Bug Report Using Character N-Gram-Based Features. In *ASPEC'10*, pages 366–374, 2010.

[324] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, 2014.

[325] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 329–340. IEEE, 2021.

[326] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2727–2735, 2019.

[327] I. Tetko, D. Livingstone, and A. I. Luik. Neural network studies, 1. comparison of overfitting and overtraining. *J. Chem. Inf. Comput. Sci.*, 35:826–833, 1995.

[328] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. Beir: A heterogenous benchmark for zero-shot evaluation of information retrieval models. *arXiv preprint arXiv:2104.08663*, 2021.

[329] Hannes Thaller, Lukas Linsbauer, and Alexander Egyed. Feature maps: A comprehensible software representation for design pattern detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 207–217, 2019.

[330] Ferdian Thung, Pavneet Singh Kochhar, and David Lo. DupFinder: Integrated Tool Support for Duplicate Bug Report Detection. In *ASE'14*, pages 871–874, 2014.

[331] Y. Tian, C. Sun, and D. Lo. Improved Duplicate Bug Report Identification. In *CSMR'12*, pages 385–390, 2012.

[332] 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.

[333] Frank Tip. *A survey of program slicing techniques.* Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[334] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri. Learning spatiotemporal features with 3d convolutional networks. In *ICCV'15*, pages 4489–4497, 2015.

[335] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.

[336] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *CoRR*, abs/2009.05617, 2020.

[337] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 25–36. IEEE, 2019.

[338] 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.

[339] 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.

[340] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–312, 2019.

[341] 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. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[342] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2291–2302, 2022.

[343] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[344] Stéphane Vaucher, Houari Sahraoui, and Jean Vaucher. Discovering new change patterns in object-oriented systems. In *2008 15th Working Conference on Reverse Engineering*, pages 37–41. IEEE, 2008.

[345] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[346] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Macro impact analysis using macro slicing. 2007.

[347] 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.

[348] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *IST*, 110:139–155, 2019.

[349] Junjie Wang, Ye Yang, Tim Menzies, and Qing Wang. isense2. 0: Improving completion-aware crowdtesting management with duplicate tagger and sanity checker. *TOSEM*, 29(4):1–27, 2020.

[350] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embeddings for program repair. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[351] Phil Wang. X-transformers. *GitHub*, 2023.

[352] Shaohua Wang, NhatHai Phan, Yan Wang, and Yong Zhao. Extracting api tips from developer question and answer websites. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 321–332, 2019.

[353] Shaowei Wang and David Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.

[354] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 46(12):1267–1293, 2020.

[355] Wei Wang, Yun He, Tong Li, Jiajun Zhu, and Jinzhuo Liu. An integrated model for information retrieval based change impact analysis. *Scientific Programming*, 2018:1–13, 2018.

[356] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information. In *ICSE'08*, pages 461–470, 2008.

[357] Cody Watson, Nathan Cooper, David Nader, Kevin Moran, and Denys Poshyvanyk. Data Analysis for the Systematic Literature Review of DL4SE, May 2021.

[358] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58, 2022.

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

[360] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1398–1409, 2020.

[361] 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.

[362] 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)*, ASE'16, pages 87–98, September 2016. ISSN:.

[363] Martin White, Michele Tufano, Matías Martínez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490, 2019.

[364] 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.

[365] Xiao Wu, Alexander G. Hauptmann, and Chong-Wah Ngo. Practical elimination of near-duplicates from web video search. In *MM'07*, page 218–227, New York, NY, USA, 2007. Association for Computing Machinery.

[366] Yuhuai Wu, Markus N Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. *arXiv preprint arXiv:2203.08913*, 2022.

[367] Franck Xia. A change impact dependency measure for predicting the maintainability of source code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, volume 2, pages 22–23. IEEE, 2004.

[368] Ning Xie, Gabrielle Ras, Marcel van Gerven, and Derek Doran. Explainable deep learning: A field guide for the uninitiated, 2020.

[369] Rui Xie, Long Chen, Wei Ye, Zhiyu Li, Tianxiang Hu, Dongdong Du, and Shikun Zhang. Deeplink: A code knowledge graph based deep learning approach for

issue-commit link recovery. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 434–444, 2019.

[370] Tao Xie and David Notkin. An empirical study of java dynamic call graph extractors. *University of Washington CSE Technical Report*, pages 02–12, 2002.

[371] Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In *SEKE*, volume 4, pages 123–128. Citeseer, 2004.

[372] Zhenchang Xing and Eleni Stroulia. Understanding class evolution in object-oriented software. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 34–43. IEEE, 2004.

[373] 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)*, ASE'16, pages 51–62, September 2016. ISSN:.

[374] Shir Yadid and Eran Yahav. Extracting code from programming tutorial videos. In *Onward!'16*, page 98–111, New York, NY, USA, 2016. ACM.

[375] Yixiao Yang, Yu Jiang, Ming Gu, Jiaguang Sun, Jian Gao, and Han Liu. A language model for statements of software code. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 682–687. IEEE, 2017.

[376] Zebin Yang, Aijun Zhang, and Agus Sudjianto. Enhancing explainability of neural networks through architecture constraints, 2019.

[377] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software*

Repositories, MSR '18, page 476–486, New York, NY, USA, 2018. Association for Computing Machinery.

[378] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. Learning to represent edits. *arXiv preprint arXiv:1810.13337*, 2018.

[379] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Learning to represent edits. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[380] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering*, 30(9):574–586, 2004.

[381] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 70–80. IEEE Press, 2019.

[382] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 70–80. IEEE, 2019.

[383] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. Explainability in graph neural networks: A taxonomic survey, 2021.

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

[385] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax

tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[386] 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.

[387] 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.

[388] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. Cnn-fl: An effective approach for localizing faults using convolutional neural networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445–455, 2019.

[389] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. Actionnet: Vision-based workflow action recognition from programming screencasts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 350–361, 2019.

[390] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Seenomaly: Vision-based linting of gui animation effects against design-don't guidelines. In *ICSE'20*, 2020.

[391] Gang Zhao and Jeff Huang. Deepsim: Deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, page 141–151, New York, NY, USA, 2018. Association for Computing Machinery.

[392] Gang Zhao and Jeff Huang. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 141–151, 2018.

[393] Guoying Zhao and Matti Pietikäinen. Dynamic texture recognition using local binary patterns with an application to facial expressions. *PAMI'07*, 29:915–28, 07 2007.

[394] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. Seqfuzzer: An industrial protocol fuzzing framework from a deep learning perspective. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 59–67, 2019.

[395] Jianjun Zhao, Hongji Yang, Liming Xiang, and Baowen Xu. Change impact analysis to support architectural evolution. *Journal of software maintenance and evolution: research and practice*, 14(5):317–333, 2002.

[396] Jinman Zhao, Aws Albarghouthi, Vaibhav Rastogi, Somesh Jha, and Damien Octeau. Neural-augmented static analysis of android communication. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, pages 342–353. ACM, 2018.

[397] Jian Zhou and Hongyu Zhang. Learning to rank duplicate bug reports. In *CIKM '12*, pages 852–861, New York, NY, USA, 2012. ACM.

[398] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

[399] 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.