Deep Learning Applied to the Assessment of Online Student Programming Exercises

Benjamin Trevett

Submitted for the degree of Doctor of Philosophy

Heriot-Watt University
Institute of Sensors, Signals and Systems
January, 2022

Primary Supervisor: Dr. Donald Reay
Secondary Supervisor: Prof. Nick Taylor

Research Thesis Submission

Please note this form should be bound into the submitted thesis.

Name*:*	**BENJAMIN TREVETT**		
School:	**EPS ISSS (INSTITUTE OF SENSORS, SIGNALS AND SYSTEMS)**		
Version: *(i.e. First, Resubmission, Final)*	**FINAL**	Degree Sought:	**PHD**

Declaration

In accordance with the appropriate regulations I hereby submit my thesis and I declare that:

1. The thesis embodies the results of my own work and has been composed by myself
2. Where appropriate, I have made acknowledgement of the work of others
3. The thesis is the correct version for submission and is the same version as any electronic versions submitted*.
4. My thesis for the award referred to, deposited in the Heriot-Watt University Library, should be made available for loan or photocopying and be available via the Institutional Repository, subject to such conditions as the Librarian may require
5. I understand that as a student of the University I am required to abide by the Regulations of the University and to conform to its discipline.
6. I confirm that the thesis has been verified against plagiarism via an approved plagiarism detection application e.g. Turnitin.

ONLY for submissions including published works

Please note you are only required to complete the Inclusion of Published Works Form (page 2) if your thesis contains published works)

7. Where the thesis contains published outputs under Regulation 6 (9.1.2) or Regulation 43 (9) these are accompanied by a critical review which accurately describes my contribution to the research and, for multi-author outputs, a signed declaration indicating the contribution of each author (complete)
8. Inclusion of published outputs under Regulation 6 (9.1.2) or Regulation 43 (9) shall not constitute plagiarism.

* *Please note that it is the responsibility of the candidate to ensure that the correct version of the thesis is submitted.*

Signature of Candidate*:*	**BENJAMIN TREVETT**	Date:	**12/01/2022**

Submission

Submitted By *(name in capitals):*	BENJAMIN TREVETT
Signature of Individual Submitting:	**BENJAMIN TREVETT**
Date Submitted:	**12/01/2022**

For Completion in the Student Service Centre (SSC)

Limited Access	Requested	Yes		No		Approved	Yes		No	
E-thesis Submitted ***(mandatory for final theses)***										
Received in the SSC by *(name in capitals):*						Date:				

HERIOT WATT UNIVERSITY

Inclusion of Published Works

Please note you are only required to complete the Inclusion of Published Works Form if your thesis contains published works under Regulation 6 (9.1.2)

Declaration

This thesis contains one or more multi-author published works. In accordance with Regulation 6 (9.1.2) I hereby declare that the contributions of each author to these publications is as follows:

Citation details	Ben Trevett, Donald Reay, Nick Taylor (2017) Automatically Correcting Semantic Errors in Programming Assignments. 10th European Conference on Machine Learning & Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2017)
Author 1: Ben Trevett	Primary Author
Other Authors: Donald Reay and Nick Taylor	Supervision
Signature:	BENJAMIN TREVETT
Date:	12/01/2022

Citation details	Ben Trevett, Donald Reay, Nick Taylor (2020) The Effectiveness of Pre-Trained Code Embeddings. 16th International Conference on Data Science (ICDATA'20)
Author 1: Ben Trevett	Primary Author
Other Authors: Donald Reay and Nick Taylor	Supervision
Signature:	BENJAMIN TREVETT
Date:	12/01/2022

Please included additional citations as required.

Abstract

Massive online open courses (MOOCs) teaching coding are increasing in number and popularity. They commonly include homework assignments in which the students must write code that is evaluated by functional tests. Functional testing may to some extent be automated however provision of more qualitative evaluation and feedback may be prohibitively labor-intensive. Provision of qualitative evaluation at scale, automatically, is the subject of much research effort.

In this thesis, deep learning is applied to the task of performing automatic assessment of source code, with a focus on provision of qualitative feedback. Four tasks: language modeling, detecting idiomatic code, semantic code search, and predicting variable names are considered in detail.

First, deep learning models are applied to the task of language modeling source code. A comparison is made between the performance of different deep learning language models, and it is shown how language models can be used for source code auto-completion. It is also demonstrated how language models trained on source code can be used for transfer learning, providing improved performance on other tasks.

Next, an analysis is made on how the language models from the previous task can be used to detect idiomatic code. It is shown that these language models are able to locate where a student has deviated from correct code idioms. These locations can be highlighted to the student in order to provide qualitative feedback.

Then, results are shown on semantic code search, again comparing the performance across a variety of deep learning models. It is demonstrated how semantic code search can be used to reduce the time taken for qualitative evaluation, by automatically pairing a student submission with an instructor's hand-written feedback.

Finally, it is examined how deep learning can be used to predict variable names within source code. These models can be used in a qualitative evaluation setting where the deep learning models can be used to suggest more appropriate variable names. It is also shown that these models can even be used to predict the presence of functional errors.

Novel experimental results show that: fine-tuning a pre-trained language model is an effective way to improve performance across a variety of tasks on source code, improving performance by 5% on average; pre-trained language models can be used as zero-shot learners across a variety of tasks, with the zero-shot performance of some architectures outperforming the fine-tuned performance of others; and that language models can be used to detect both semantic and syntactic errors. Other novel findings include: removing the non-variable tokens within source code has negligible impact on the performance of models, and that these remaining tokens can be shuffled with only a minimal decrease in performance.

Acknowledgements

Without the support of a great, many number of people, I don't believe this thesis would have ever been completed. I am eternally grateful to everybody I have interacted with at Heriot-Watt University, Edinburgh, and Cambridge over the last few years. I don't think I could've asked for a better group of friends, mentors and peers.

First, I'd like to thank my supervisors Donald Reay and Nick Taylor. Without their weekly meetings, I believe I would've thrown in the towel a long time ago.

I'd also like to thank all of my friends in Edinburgh, especially: Rahil Joshi, Ruben Kruiper, Alexis Costouris, Lucas Kirschbaum, Alisdair Hill, and Filip Bartoszewski.

My thanks also goes out to Arm and the ESPRC, as without their support and funding this PhD wouldn't have even been possible. Thanks to Khaled Benkrid, Sean Hong, Shuojin Hang and Mark Allen in particular.

I'd also like to thank Mustafa Suphi Erden and Miltos Allamanis for acting as my viva examiners.

Finally, I would like to thank my family for always being supportive of me, no matter what.

Contents

1 Introduction

1.1 Motivation

Massive online open courses (MOOCs) have become a popular method for distance learning, the most popular of which are related to programming and computer science related topics [1]. Most MOOCs are offered by universities or high-profile companies working with collaborators, for example the ARM University Program [2] – which sponsors this work – collaborates with companies in its ecosystem to provide online courses in embedded programming. With the large number of students – potentially in the thousands – enrolled in a MOOC, providing detailed feedback at scale becomes an interesting challenge. Many MOOCs offer a coarse level of feedback for programming assignments in the form of test suites in which the student's code is executed with an input and the output is compared against the expected value. These test suites allow for quantitative feedback, in which the student is simply given a numerical score – however, they do not provide any form of qualitative feedback. The most common form of qualitative feedback is when an instructor hand-grades the student's submission by providing annotations and comments on the student's submission where appropriate. A student submission that receives a low score on a set of test suites (quantitative feedback), and no qualitative feedback, provides little information to the student – they know their submission is incorrect, but are either not sure why or were unable to reach a correct solution; in this case, qualitative feedback can be used to explain the mistakes made by the student, so they can learn from them. Similarly, a student submission that receives a high score on a set of test suites – again, with no qualitative feedback – also receives little information, they know their submission is correct but have received no feedback on where their submission could potentially be improved in terms of qualitative metrics, e.g. if they are using idiomatic code or using appropriate data structures.

The downside of qualitative feedback is that it is time-consuming and must be carried out manually, unlike quantitative feedback, which can be done automatically without the instructor's input. This issue is exacerbated in MOOCs due to the potentially large number of students enrolled. One so-

[1]http://www.onlinecoursereport.com/the-50-most-popular-moocs-of-all-time/
[2]https://www.arm.com/resources/education/education-kits

lution to this would be to create a fully automated system which would provide qualitative feedback on student submissions with no input from the instructor. Unfortunately, creating an automatic qualitative feedback system is significantly challenging compared to a quantitative system. An infinite number of test inputs and expected outputs can be created for the quantitative feedback system using a single correct submission – provided by the instructor – whereas no such analogy exists for qualitative feedback – it must be uniquely created for each submission depending on its contents. However, what if there existed a system that could learn from a corpus of student submissions that have been qualitatively graded? This system could take what it has learned from these submissions and then extrapolate what it has learned to new, unseen submissions and apply relevant qualitative feedback. If this system is able to learn when and where appropriate qualitative feedback should be applied, then it could be used as a fully automatic qualitative feedback system – solving the issue of applying qualitative feedback at scale.

Machine learning is a method of learning from a collection of examples and then inferring learned features to unseen data. Framing the automatic qualitative feedback problem in terms of machine learning involves: using a collected dataset of qualitatively graded submissions, training a machine learning algorithm on these submissions in order to replicate the instructor's method of providing qualitative feedback, and then applying the machine learning algorithm on unseen student submissions – which would then have qualitative feedback automatically inferred. Machine learning has gained a large amount of interest in recent years due to the rise of deep learning – a subset of machine learning that focuses on artificial neural networks – which has shown promising breakthroughs in research areas such as computer vision and natural language processing, and has successfully been applied to source code for applications including: detecting bugs, improving code auto-completion and translation between source code and natural language. These techniques can now be applied to the task of learning how to provide qualitative feedback.

Assuming it is possible to develop a machine learning system that can automatically apply qualitative feedback, what type of feedback should it provide? The aim of feedback is to provide information which can be used, by the student, to improve their performance in terms of grades received. This information, when provided by human graders, usually consists of multiple natural language comments, each of which have a given location – e.g. "incorrect type used here", "a 'for' loop would be more appropriate here" – and a general comment on the submission as a whole with a grade – e.g. "good overall but some small mistakes, 8/10". Feedback provided can be broken down into two main categories, feedback on errors and feedback on inefficiencies.

Errors come in two types – syntax errors and semantic errors (also known as functional errors). Syntax errors occur when the source code does not follow the correct language syntax, e.g. not ending a line with a semicolon. Modern integrated development environments (IDEs) are able to continuously check syntax for correctness whilst the user is writing code and any syntax errors detected are then usually highlighted to the user. If not detected by the IDE, then the syntax errors will be detected by the compiler. Semantic errors arise when the source code is syntactically correct but produces an incorrect output, e.g. an off-by-one error. These cannot be checked by IDEs, however in the common case of a student being able to compile and run their code locally as many times as they want, they are able to check for semantic errors themselves – provided that there are example inputs and outputs. Syntax and semantic errors are also tested by the quantitative analysis obtained from test suites. As both types of errors already have existing methods in place to detect them and provide quantitative feedback on them, this work focuses more on the provision of qualitative feedback in regard to inefficiencies within code.

In this work an inefficiency within code is defined as a section of code which: a) does not contain a syntax error, b) does not contain a semantic error, and c) would cause most human graders to apply qualitative feedback to this section of code – usually in the form of a natural language comment, e.g. "be sure to use consistent naming conventions". This definition implies that inefficiencies should be dealt with via qualitative feedback techniques as quantitative feedback techniques, i.e. test suites, cannot detect them.

A major challenge faced by a system that learns how to apply qualitative feedback from human examples is the level of subjectivity and variance in the feedback. Qualitative feedback is subjective in that multiple instructors may not agree on what counts as inefficient, and the feedback also varies widely due to the comments provided being in a natural language with no standardization, i.e. even if the instructors agree on which sections of a submission are deemed inefficient there is little chance their associated natural language comments will be identical. The desired use of the source code also determines where qualitative feedback should be given, as what counts as inefficient in one scenario may not be inefficient in another, e.g. code written to optimize execution time is inefficient if the goal is to optimize for memory consumption, and vice versa.

The techniques presented in this work attempt to solve these challenges not by directly learning from pairs of source code and relevant feedback, but instead learning from source code only. Specifically, this work focuses on three main areas: appropriate naming of functions, appropriate naming of variables, and detecting unnatural coding conventions. Features learned by these techniques can then be used to highlight areas of code which potentially require qualitative feedback. One potential use of this is that a human instructor can then review these highlighted areas only, providing natural language comments where appropriate – thus significantly reducing the time taken to provide qualitative feedback.

It is also shown how machine learning models can be used on examples of code and natural language pairs. Using these techniques, a human instructor can provide qualitative feedback to a subset of all student submissions, and this feedback can be propagated to all similar student submissions. This greatly reduces the amount of submissions that need to be manually graded, and also reduces the time taken to provide qualitative feedback.

Challenges. Locating inefficient source code within a program poses numerous challenges. To locate inefficiencies, both the syntax and the semantics of the programming language must be understood. Syntax is consistent across all programs within a language, no matter what their application is, however the range of syntactic constructs used can be diverse. Thus, the proposed system needs to understand correct syntax of a given program. Semantics require understanding the control flow of a program in order to determine both what the most suitable control flow should be and locating where the student has deviated from this. There is also the issue of multiple programs being semantically equivalent, yet syntactically diverse, through multiple different solutions to a single problem, giving a wide range of different control flow graphs to a particular problem.

Encouraging Progress. Despite the potential difficulty of automatically locating inefficiencies within code, there has been a large amount of work in this area recently due to the popularity of deep learning. Using architectures such as recurrent neural networks (RNNs) and inspired by natural language processing (NLP) techniques, such as neural machine translation (NMT), deep learning has been used to locate and correct a wide variety of syntax errors in a range of applications.

Remaining Challenges. Although rapid progress has been made on determining syntax errors, both semantic errors and inefficiencies still pose a challenge. It can be argued that using deep learning for syntax errors, although useful, is not entirely necessary, as IDEs and compilers have robust syntax error checking due to their rigorous parsing which has been built on for decades. However, compiler error messages are not always exact, as sometimes the error cascades down a program, causing multiple errors to appear from a single fault.

Long-term motivations. The long-term applications can be seen from both the machine intelligence and practical point-of-view. Having a model that can automatically determine and correct errors and inefficiencies in code allows a step to be taken toward fully automatic programming where the user can write a brief outline of their code with the desired functionality and the remaining code can automatically write itself. From the practical side, having the ability for all errors and inefficiencies to be instantly highlighted with the correction already prepared would vastly increase the speed in which programs are written, as well as reducing the number of bugs which make it into production code.

Short-term motivations. In terms of what can be reachable in the near future, determining the location of errors and inefficiencies, and automatically pairing code to feedback, is useful from a pedagogical view. Inefficiencies can be highlighted to an instructor, reducing the time spent to searching for them, allowing instructors to focus on providing detailed feedback. These areas can also be highlighted to the student, allowing them to see which areas of their code need improvement before submission. Automatically pairing code with relevant feedback also significantly reduces the time taken to hand-grade student submissions.

1.2 Research Aims and Objectives

The aim of the work in this thesis is to develop machine learning based methods that will assist in providing qualitative feedback for programming languages. Specifically, the aims are to provide qualitative feedback assistance methods which focus on three main areas: suggesting relevant existing qualitative feedback, automatic detection of deviation from standard naming conventions, and automatic detection of unnatural code which does not fit common coding conventions.

Thus, the objectives are to implement several machine learning models and apply them to three tasks: learning to pair relevant source code and natural language pairs, learning correct naming conventions by training models to predict function and variable names from surrounding context, and learning coding conventions by learning to predict code from surrounding context.

1.3 Outline of Contributions

This work aims to study techniques which can be used to improve the performance of models using natural language processing techniques applied to programming assignments in provide feedback on code. Concretely, the novel contributions of this thesis are:

- Showing how masked language modeling outperforms standard language modeling on code
- Confirming the results of Hindle et al. [109], that programming languages achieve a lower perplexity than natural languages, on the CodeSearchNet dataset [116]
- Showing that pre-training on either natural language data, programming language data which matches the downstream task, or programming language data consisting of multiple programming languages improves performance for semantic code search and variable name prediction
- Demonstrating that multi-task learning improves performance for semantic code search and variable name prediction, both when no downstream task data is used and when using a dataset consisting of multiple programming languages
- Analyzing the effect that two data augmentation techniques (stripping punctuation and shuffling tokens) has on the performance of semantic code search and variable name prediction

1.4 Thesis Outline

Chapter 2 provides an overview of the current research on applying machine learning models to programming languages for a variety of different tasks.

In Chapter 3, the four main models used throughout this thesis: neural bag-of-words, recurrent neural networks, convolutional neural networks and the Transformer, are all described in detail.

The following three chapters focus on the applications of models introduced in Chapter 3, to different tasks. Each task is designed to align with the overall goal of helping provide quantitative feedback at scale.

In Chapter 6, semantic code search is examined. This task involves learning to automatically pair programming language code with a relevant natural language description. It is shown how models trained on semantic code search can be used to automatically pair new submissions with existing qualitative feedback. Experiments also show how: transfer learning can be used to improve performance of models, exploring how pre-training on different data affects the performance; how multi-task learning can be successfully used even when no examples in the desired programming language are used to train the models; and, how the models rely on information contained within the source code, shown by applying two data augmentation techniques.

In Chapter 7, the models are applied to the task of predicting variable names from a given method body. It is shown how this can be applied to the task of providing qualitative feedback for variable naming conventions, suggesting a more suitable name if applicable, or even finding a potential semantic error when the predicted variable name does not match to the variable name actually used. Experiments also show the effectiveness of transfer learning for improving the performance of models; the use of multi-task learning to improve performance; and, how this task uses more information contained within the existence of non-variable tokens [3] and the order of tokens than in semantic code search.

[3] A token is an atomic part of a sequence and is determined by a tokenization function. For example the string "Hello, world!" can be represented as the four tokens `Hello`, `,`, `world` and `!`. See Chapter 4 for further discussion on tokenization functions used in this thesis.

Chapter 8 contains the conclusions of the work contained in this thesis, where the experimental findings are summarized and ideas for future directions are presented.

1.5 Publications

The work in this thesis builds on the following peer-reviewed publications:

1. **Ben Trevett**, Donald Reay, Nick Taylor (2017) Automatically Correcting Semantic Errors in Programming Assignments. *10th European Conference on Machine Learning & Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2017)*

2. **Ben Trevett**, Donald Reay, Nick Taylor (2020) The Effectiveness of Pre-Trained Code Embeddings. *16th International Conference on Data Science (ICDATA'20)*

2 Background

Machine learning applied to source code has seen a rapidly increasing popularity in recent years. `ml4code` [4], a website that attempts to list all research papers that apply machine learning to source code, shows that the number of papers published has increased from nine papers in 2013 to 87 in 2020.

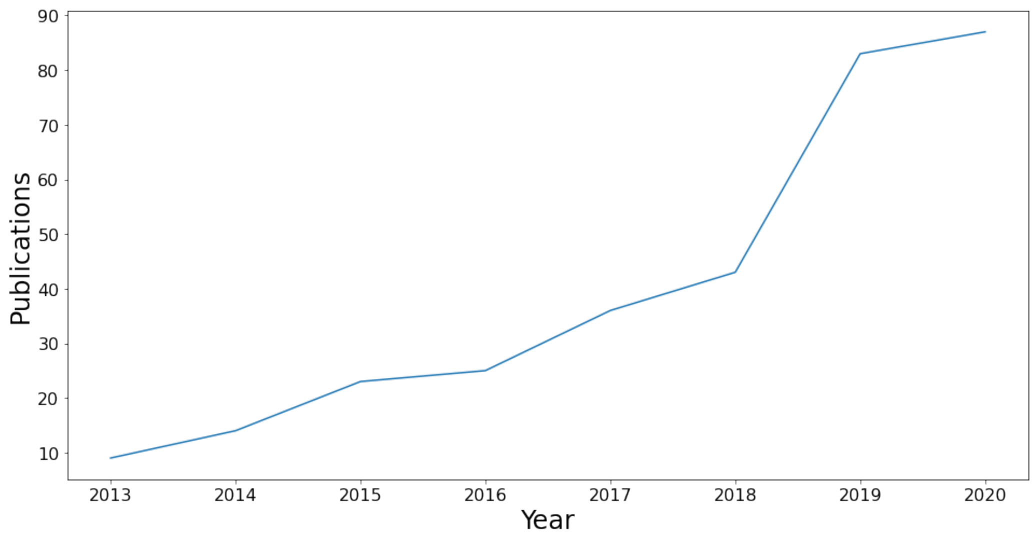

Figure 1: Number of machine learning for code publications since 2013. Data taken from `https://ml4code.github.io`.

This chapter separates research on applying machine learning to code into seven sections: feedback generation; bug finding, correcting and verification; language modeling; predicting method and variable names; sequence-to-sequence models; code mining; and code search. Also included is a section on "meta" research for machine learning applied to source code, i.e. research specifically examining artefacts in machine learning for code datasets, representations or models.

[4]https://ml4code.github.io

2.1 Automatic Feedback Generation for Code

Ideally, a system used for providing qualitative feedback on code would be trained in a supervised learning setting, with code as input and feedback serving as the labels. Unfortunately, there is a limited number of systems that directly learn to apply feedback to source code. This is mostly due to the lack of a suitable dataset – both in size and in quality. The systems below do provide feedback directly to code, however they suffer from limitations in terms of the size of the dataset used, the complexity of the data, and the quality of the data.

Automata [249] is believed to be the first system that uses machine learning for automatically grading code. They use hand-crafted features that are built on counting the occurrence of the number of certain keywords, the number of certain expressions, the number of data-dependency statements, the number of lines of source code, etc. They feed these features into shallow machine learning models which predict the numerical grade, between 1 and 5, given to a program by an expert human grader. Initially, a separate model needed to be trained for each task, however further work [245] builds a task independent model for grading source code. The authors show that if Automata is trained on a set of 'good' code spanning across multiple tasks, then task independent features are learned. Automata can then use these task independent features to accurately predict grades across tasks not seen in the training set. As these methods only use shallow machine learning models they require significantly fewer data – 3500 examples across 8 tasks – however are limited by their ability to only produce a numerical grade, whereas the work in this thesis is more interested in providing a form of qualitative feedback.

Instead of predicting a numerical grade, Piech et al. [207] use neural networks to propagate feedback given to a small sample of code submissions. This is similar to OverCode [79] which used static and dynamic analysis tools. Piech et al., however, use a two-step process where they first use a recursive neural network to transform their code into a sparse representation using input-output examples and then, using a sample of hand-graded code, learn how to use this sparse representation to predict which hand-written feedback should be applied to it. They show that they are able to propagate up to 214 examples for each one hand-graded whilst maintaining over 90% precision. The downside is that Piech et al.'s work is only applied to a toy language which has control flow – if/else statements and for loops – but no variable declaration, whilst OverCode uses real Python code.

DeepCodeReviewer [90] learns from a dataset made of code snippets paired with code reviews, which can be viewed as feedback. Their model, an LSTM, is trained to take the code snippet – which includes the line indicated by the review as well as lines above and below – and the review as input and produce a binary output indicating if the review is relevant or not. The model is trained on both positive and negative examples scraped from open-source repositories. During inference, a submitted code snippet is fed through the model with each review in the dataset of code reviews. The most relevant reviews, above a threshold value, are shown to the user. In theory, this method does not require any hand-written feedback, providing the dataset of code snippets and code reviews in the desired domain already exists. However, the number of examples required for training deep learning models is significant – DeepCodeReviewer itself is trained on over 30,000 examples. Using a model trained on examples scraped from open-source repositories would raise issues where the feedback from code reviews are not necessarily suitable for pedagogical feedback. There is also the issue of the model being unable to produce original feedback, it can only provide feedback already in the dataset. There is room for further research on this work, exploring how many examples are required for sufficient domain adaptation.

Ahmed et al. [6] propose a three stage model for automatically parsing, correcting and typing incomplete, and potentially erroneous code from StackOverflow. They train a Transformer model on correct code that has synthetic errors inserted to generate correct code. They find it beneficial to split their correction into two stages: one for correcting the nesting of code, and the other to fix syntax errors. This is following by a third stage in which they predict the correct types of variables.

2.2 Automatic Bug Finding, Correcting and Verification

A popular application for machine learning applied to source code is learning how to locate, correct and verify bugs/errors. These traditionally used supervised machine learning methods using a large annotated dataset in which the inputs are the source code and the targets are, e.g., the line number of the bug, the correct line of code without a bug, or a binary label denoting if the code compiles or not.

2.2.1 Automatic Bug Location and Correction

The ability to automatically locate bugs can either be used by other bug correction and verification techniques or the location of the discovered bugs presented to the user, so they can correct and verify the bugs themselves. Correcting bugs is more challenging, but is the most commonly researched area relating to machine learning on code for bugs.

In regard to providing feedback, consider a model which can accurately locate bugs in submitted code. Feedback can be provided by highlighting the location of the bugs found by the model and by allowing the user to fix the bug themselves. This could also potentially find bugs which would not be caught by unit tests, such as obscure edge cases. If the model is also able to generate corrections for the found bugs, a pedagogical aspect could be implemented in which the user has to choose between a valid correction and two invalid corrections.

Kremenek et al. [141] propose a system that builds a factor graph using static program analysis in order to build a specification – the rules that must be obeyed – from a program. They use these specifications to find bugs within a program.

SK_P [212] uses a neural sequence-to-sequence model to correct syntactic and semantic errors within student submissions. It is trained to output the correct line of code using the previous and following lines of code as context. They find that their approach corrects 29% of all incorrect submissions, outperforming current state-of-the-art approaches which require manual input by the user.

DeepRepair [283] fixes bugs in source codes by selecting and transforming statements using the redundancy assumption – that large programs already contain the exact lines of code available within them to be used for their repair. They use deep leaning to rank the potential repairs using a learned code similarity metric.

The *Share, Specialize and Compete* (SSC) network by Devlin et al. [67] is designed to correct semantic errors in code without having access to the information about the intended correct behavior of a program. SSC uses a RNN over the AST and then scores each candidate repair with a specialized module, before comparing scores from each of the specialized modules and performing the predicted correction.

DeepFix [93] uses an LSTM with attention mechanisms trained to fix syntax errors within source code. The model takes in source code annotated with line numbers and outputs both a line number and generated line of code. The original source code at the line number output by the model is replaced with the generated line of code. This process is repeated until all the syntax errors are removed – determined when the code is compiled without errors – or a maximum number of iterations are reached. DeepFix manages to completely fix 27% of programs within their dataset, and partially fixes another 19%.

SynFix [32] uses a recurrent neural network to correct syntax errors within code. The model is first trained on a corpus of code without any syntax errors. Inference is performed by starting at the line number parsed from the compiler error, where the model outputs tokens using the previous tokens in the code as context. They find that their model can fix 32% of all syntax errors within their dataset.

Santos et al. [234] use a language model to detect and correct syntax errors within code. They test their model to automatic fix syntax errors in a dataset consisting of student code before and after a syntax error has been fixed. They find that their models can correctly predict the token type and location around 30% of the time, but can only predict the correct identifier name around 3% of the time.

DeepBugs [210] uses a simple feed-forward neural network to learn embeddings on source code tokens in order to predict three different types of bugs: swapped arguments, incorrect operator and incorrect operand. They test their model on a synthetic dataset, which consists of real code with the bugs artificially seeded, as well as real-world code – where they find 102 previously undiscovered bugs. They show that pre-training the embedding layer on the source code tokens with the WORD2VEC algorithm improves performance over randomly initialized embedding weights.

Bhatia et al. [31] combine SynFix [32] and AutoProf [246]. The fixes proposed by SynFix are then used with the error model of AutoProf to find the minimal program repair for the submitted program to be functionally correct.

Harer et al. [96] apply generative adversarial networks (GANs) [82] to correct errors in source code. The GAN's generator is taught to generate correct code conditioned on incorrect code, whilst simultaneously trying to fool the discriminator, which is trained to predict if the generated code is output by the generator or is actual correct code.

Ratchet [98] uses a sequence-to-sequence model to learn how to fix errors via the changes between two versions of a given source code. They find that their model almost always generates valid statements and improves upon existing pattern-matching techniques, however Ratchet is limited in that it must be told the line containing an error and can only generate a single line of code.

Tufano et al. [265] develop a method that learns to mutate, i.e. adds bugs, to correct code, by learning from bug fix commits. These mutants can be used for assessing the effectiveness of test suites. They find their model is able to generate a diverse set of mutants.

RLAssist [92] uses reinforcement learning techniques to find and correct errors. The agent navigates through the code using a cursor, and actions consist of either moving the cursor or modifying the token at the current cursor position – using a restricted set of modifications. Their agent can be provided expert demonstrations to speed up the training time. They compare their model against DeepFix [93], which it outperforms.

Codeit [50] is a tree-to-tree model which learns from changes in source code in order to automatically generate patches. They show that their tree-based model beats traditional sequence-to-sequence models used in NLP applications. The authors show that, even though not explicitly trained to, Codeit can correct 43% of bugs in a dataset of defects.

Tufano et al. [263] develop a method that trains on a dataset of code changes scraped from open-source repositories. They turn the task of fixing bugs to a translation problem, translating from buggy code to fixed code. They show that, with enough time, their model can fix 82% of bugs in their dataset.

Chen et al. [53] apply sequence-to-sequence learning to the problem of correcting bugs in source code. They focus on fixing bugs that appear in larger classes and methods which require their model to learn long-range dependencies, however their model can only correct bugs which occur across a single line. For the buggy line, which they get from a compiler error message, they generate multiple candidate correct lines which they check against the compiler in turn. They find that the use of a direct copy mechanism significantly improves the performance of their model.

SampleFix [95] is a tool for correcting errors within programs. The tool outputs the line of the potential fix as well as the corrected line. It generates multiple candidate solutions with a generative model – a variational auto-encoder [137] – however, in contrast to other approaches which generate multiple candidates, the authors bias their model towards generating solutions which are diverse as possible which is supposed to reflect the fact that each bug can have multiple valid fixes. They show that their model beats the previous state-of-the-art, DeepFix [93], on the same dataset.

Tufano et al. [266] argue that most work in automated bug fixing measure quantitatively (e.g. how many bugs can be fixed?) rather than qualitatively (e.g. what kind of bugs can be fixed?). They train an RNN model on methods obtained before and after a pull request and find that their model is able to perfectly predict the fixed code up to 21% of the time with one prediction and 36% of the time with ten predictions. By examining the perfect predictions they find that the majority of them are refactoring or bug fixes – however their experiments are only on "small" and "medium" methods (under 50 and between 50-100 tokens, respectively).

Habib and Pradel [94] examine an LSTM's ability to predict if a method contains a bug or not. They find that some classes of bugs are easy for a neural network to predict, whereas others are not. They find that the types of bugs where the neural network performs poorly are usually rarer, although some bug patterns can be accurately detected with a relatively small amount of examples.

NeuralBugLocator [91] uses a CNN over the AST represented as paths – similar to [19] – in order to localize a bug given a method and the test case it failed. It is able to correctly localize the bug 80% of the time when using the top 10 predictions, though this rapidly decreases to 57% and 20% when using top 5 and top 1 predictions, respectively.

Vasic et al. [268] expand on the work by Allamanis on the VarMisuse task – for each variable in a method, predict if the correct variable is used. They find their model, an RNN with two pointer networks, beats the GNN used by Allamanis on the VarMisuse task.

Hellendoorn et al. [103] introduce GREAT combines GNNs and Transformers for bug location and correction. They find that neural network architectures which combine local and global information outperform previous graph neural networks, which are only able to leverage local information.

2.2.2 Automatic Program Verification

Program verification is the task of predicting if a code snippet contains a bug or not, a less popular task due to compilers and unit tests already being able to detect syntax and semantic errors, respectively. However, research in this area also focuses on highlighting patterns in code which cause the verification process to fail.

A system that is able to highlight why a code fails program verification could be used in a pedagogical setting. For example, using the output of the program verification model, students can find the causes of failure in their own code submissions, or in the submissions provided by other students.

Gated Graph Sequence Neural Networks [154] are designed to be used in program verification. They are applied to the heap of a program using the memory addresses accessed by the program to form a graph.

Wang et al. [278] train a deep belief network on the AST of a program to perform binary classification on code in order to determine if it contains a bug or not. They find that their model is able to learn features that are common across multiple projects.

Murali et al. [187] use an RNN to encode source code into a "reference distribution", they then compare the reference distribution of correct code and provided code. The distance between these distributions is used to produce an "anomaly score" which indicates the probability that a given program contains potential bugs.

Koc et al. [138] use machine learning along with a static code analysis (SCA) tool on source code in order to predict whether the output of the SCA tool is a false positive. They use their results to identify common coding patterns that lead to false positives.

Russel et al. [230] use a CNN on the obfuscated tokens of source code to detect whether the code contains a security vulnerability. They find that their model significantly outperforms static analysis tools.

2.3 Language Modeling for Code

Language modeling source code is the most popular area of research for machine learning on source code. The task is to learn a probability distribution over source code, usually achieved by training a *language model* to predict the next token from the previous tokens. Trained language models have numerous applications for source code, such as: detecting idioms, finding bugs, autocompletion, and code generation.

2.3.1 Language Models for Code

Initially, language modeling was performed using n-gram language models. These have now been replaced with neural language models which use recurrent neural networks. Recently, recurrent neural networks have found themselves being replaced by Transformer models trained as *masked language models*. See Sections 3.3 and 3.5 for a description of the recurrent neural network and Transformer architectures, and Chapter 5 for a description of standard and masked language modeling.

The loss per token over a given sequence can be used to calculate the perplexity per token. Higher perplexity, also known as "surprise", over a sequence of tokens implies that the sequence is uncommonly encountered within the training data. Thus, by training the language model only on correct, bug-free code the model should, in theory, produce higher perplexity for code containing bugs than code without bugs. Thus, perplexity can be used to find potential bugs in code and provide feedback to students by highlighting areas of their code with a high perplexity value. Experiments on this task are detailed in Section 5.4.

Hindle et al. [108] were the first to apply language models to source code. They propose the hypothesis that "programming languages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models" which can be "leveraged for software engineering tasks". They show that programming languages are more predictable than natural languages, such as English, and can be modeled with simple n-gram language models. This is because software is not unique, and that for a sequence of six code tokens there is over 50% chance of finding the exact same sequence in another project [75].

Allamanis and Sutton [12] expand on work by Hindle et al. by showing that the performance of a language model scales with the amount of data (in terms of lines of code) it is trained on.

SYZYGY [201] is a tool for modeling source code that models code as syntax trees, using information about syntax, types and program context. Using this information, it is constrained to output a conditional distribution that has non-zero probability only for valid outputs.

SLAMC [199] use an n-gram language model to not only model the lexemes (the individual tokens) but also the sememes (semantic information and common idioms that occur in code). They find that their SLAMC model significantly outperforms standard n-gram language models that only take lexemes into account.

Hsiao et al. [114] perform n-gram language modeling on the program dependence graphs of code. They use their language model to obtain a measurement of "importance" for a given code snippet. Commonly occurring sequence of code tokens have low importance, whilst rare sequences have high importance. An example use-case of using high importance areas to indicate likely faults is proposed.

Campbell et al. [47] show a similar finding, where code that contains errors – specifically syntax errors in this case – is more "unnatural" – has low probability of being predicted by an n-gram language model. They argue that an n-gram language model should be used to "enhance a compiler's ability to locate missing tokens, extra tokens and replaced tokens."

Maddison and Tarlow [175] create a model that learns probabilistic context-free grammar over the abstract syntax tree of source code. This can then be used for code generation. They show that, in terms of log probabilities, their model outperforms n-gram generative models.

Tu et al. [262] hypothesize that n-gram language models do not take the localness of software into account – that code is locally repetitive, i.e. a variable will most likely be used several times within multiple consecutive lines. They propose adding a cache to an n-gram language model, which increases the output distribution over recent tokens. The show that their cache language model significantly improves performance over standard n-gram language models. Later, CACHECA [74] use a cache n-gram language model for auto-completion of code. They find that by combining the suggestions with the IDE's native code suggestion tool, they double the accuracy of suggestions.

Naturalize [14] learns an n-gram language model for modeling coding conventions of a code-base. It can then be used to suggest variable names that fit the current conventions used in the project. Their novel contribution is the introduction of cross-project language models, which use two language models: one trained globally across many projects and one trained locally across the current project only.

Instead of training a language model in an autoregressive style Slang [221] is trained by learning to fill in gaps within code. It gains context from the code both preceding and following the gap. They find that a recurrent neural network language model achieves higher performance than an n-gram language model, though it takes considerably longer to train, hours rather than seconds.

Hellendoor et al. [100] evaluate n-gram language models on GitHub pull requests. They find that: rejected pull requests are less natural than accepted ones, debated pull requests are less natural than undebated ones, code added during revisions is less natural, and that experienced contributors' code is more natural.

Karpathy et al. [133] visualize the activations of a character RNN language model on a dataset consisting of the entire Linux kernel. They find several interpretable neurons which activate in certain conditions, such as: when inside an 'if' statement, when inside comments and quotations, with a sensitivity depends on current nesting depth, and at the end of lines. They show that the reason RNNs achieve superior performance to n-gram language models is their ability to correctly model long range structural dependencies.

White et al. [284] also show that RNN language models beat n-gram language models on the task of modeling source code. They find that the performance improvements offered by RNNs increases drastically for lower sequence lengths.

Saraiva et al. [235] analyze n-gram language models across different developers, time periods and applications. They conclude that: a) an application specific language model performs better on that application than one trained across the entire code base, b) a language model trained on a single developer's code still performs better when it is application specific, implying that developers do not re-use patterns and idioms across applications, and c) language models trained on the latest version of the code base performs as well as ones trained on the entire code base across all versions.

Ray et al. [219] use n-gram language models to measure the unnaturalness of "buggy" code by comparing the entropy between code before and after a bug fix. They find that code containing bugs has a higher entropy than correct code, and that a static bug finder which prioritizes searching lines by their entropy value improves its performance over the standard heuristics.

Allamanis et al. [13] train a bi-modal language model which jointly models code and natural language. They find that modeling from code to natural language is easier than natural language to code. They also show that models which take the structure of code into account consistently beat models which do not.

PHOG [34] builds a probabilistic model of code by modeling the probabilistic context-free grammar of code. It encodes source code tokens via modeling context-free grammar of underlying the programming language, conditioned on the context of the AST.

BugGram [279] also applies n-gram language models to the task of discovering bugs within code. They find that their approach is complementary to existing methods that use rule-based techniques.

Hellendoorn and Devanbu [101] argue that explicit language models, such as n-gram language models, are superior to implicit language models, such as RNNs and LSTMs, for modeling source code. This is because explicit language models are better at handling the locality of source code and can deal with the out-of-vocabulary problem easier. They also state that the limited vocabularies used by implicit models misleadingly boost performance in terms of evaluation metrics, but not actual performance on the task of auto-suggestion.

Lanchantin and Gao [147] perform a similar experiment as Ray et al. [219], but use an LSTM language model instead of an n-gram language model. They find that the findings of Ray et al. still hold, that lines containing bugs have a higher entropy than those which do not.

Craic [167] aims to remove uninformative code comments by proposing the comment entailment task – predicting if the code snippet logically implies the natural language comment. The task is carried out for each of the individual sentences within the comment. They train a language model on the natural language comments, conditioned on the associated source code. The perplexity over each sentence determines how predictable the sentence is, with easier to predict sentences deemed to be uninformative and thus removed.

Parvez et al. [202] propose a model to improve the performance of neural language models for code tokens. Their model is augmented with a second language model that predicts the token type. Their model beats the perplexity of SLP-Core [101] by 22%.

Alon et al. [21] use a language model over the AST of code by generating paths from the root to the leaves. They use the Transformer [270] model augmented with the ability to directly copy sub-tokens, which they find greatly benefits the model's performance.

Lin et al. [156] examine the "naturalness" of refactored code. Although "buggy" code is less natural than correct code, they investigate whether refactoring code improves naturalness. They find that refactoring does not necessarily improve naturalness, and that different refactoring methods change the naturalness in different ways.

Rahman et al. [218] revisit the work of Hindle et al. [108] by: increasing the size of the training corpora, expanding the experiments to six more languages, investigating the removal of syntax tokens (parentheses, semi-colons, etc.), examining the naturalness of API calls, and measuring the naturalness of code represented as ASTs instead of a sequence of code tokens. They find that the naturalness hypothesis holds – code is repetitive and predictable

– across all languages used, but the amount of repetition varies across languages. With the removal of syntax tokens – which account for up to 60% of tokens in a programming language, the naturalness of programming languages significantly decreases, closing the gap between programming languages and natural languages. However, Java API calls are predictable as they are repetitive across programs. Finally, they find that AST representations of code are more repetitive than sequential representations, and argue that researchers use graph model that correctly capture how control and data flow within code.

Brockschmidt et al. [39] perform language modeling on source code using the AST. They use learned representations of each note to output to iteratively generate subsequent nodes using expansion rules to ensure generated code is semantically correct.

CuBERT [129] is a Transformer model trained as a masked language model on source code. When fine-tuned, CuBERT outperforms an LSTM model trained for 100 epochs on the five classification tasks described in the paper, even after only being trained for two epochs.

CodeBERT [73] is a masked language model trained on both source code and natural language doc-strings simultaneously. They show that using this bi-modal data allows them to achieve state-of-the-art results on semantic code search and code to doc-string generation.

2.3.2 Code Auto-completion

Once a language model has been trained, it can be used to generate code by repeatedly sampling from the learned conditional probability distribution. This can be used for code auto-completion, i.e., suggesting the next code token based on the previous code tokens. Related to locating areas of high perplexity using a language model, potential bugs can be found in cases where the token suggested by the auto-completion model differs from the actual token used in a student submission.

Bruch et al. [41] compare three different methods of automatic code completion – one based on uni-gram frequency, one based on association rules and another based on the k-nearest-neighbors (kNN) algorithm. They find the kNN algorithm, which predicts the next token by ranking the nearest "contexts" and predicting the most commonly used token within each context.

Bhoopchand et al. [33] train a neural language model augmented with a sparse pointer network over a neural memory consisting of representations from the last few identifiers. They find the addition of the pointer network allows their model to more effectively capture long-term dependencies.

Li et al. [153] train an LSTM as a language model for auto-completion of AST leaf nodes. They find that an LSTM augmented with a pointer network [271] helps modeling out-of-vocabulary (OoV) words more accurately, leading to improved performance. OoV words are common in source code as they have significantly larger number of unique identifiers.

Murali et al. [186] use a Gaussian Stochastic Neural Network (GSNN) for auto-completion. Their model learns to output a distribution which is used to select code templates which they combine together to write code.

Pythia [255] is a code autocompletion system for Python code. The authors train an LSTM on the extracted AST from code with a focus on usability via low-latency inference. To accommodate this, they use techniques such as model quantization and weight sharing. Using their Markov Chain baseline, they find that the frequency of methods inside `if` statements is considerably different from those outside of `if` statements.

Hussain et al. [117] use transfer learning to perform code autocompletion. They first pre-train two language models, an RNN and a GRU, as language models, then fine-tune them by concatenating the representations from both language models and training a learned attention layer to predict the next code token in a given sequence of code tokens.

2.3.3 Code Generation

Code generation is related to code auto-completion – both are performed by sampling a language model – however, the task of auto-completion is to predict the next token that would have been input by the user, whereas the code generation task involves generating an entire code snippet, usually from scratch.

Abstract Syntax Networks (ASNs) [213] generate source code – conditioned on an input – by iteratively generating nodes within the AST of the desired output language.

In Learning to Execute [297] the authors propose using LSTMs that take in a sequence of code tokens and generate the output of that code on character at a time. The code examples are simple and generated via some heuristics, with operations being limited to: addition, subtraction, multiplication, conditional statements and loops that only have a single statement within their body. They find that the model's performance improves if it is trained using a curriculum of data that gets progressively more difficult.

CLGen [61] learns a code generation model from a suite of OpenCL benchmarks to test the optimization of GPU parallelization. They find that models trained on their generated benchmarks have a 4.3 times increase in performance (although subsequent work [80] has shown that this performance increase could be improved further by simply using the data mined to train CLGen). The authors follow up on CLGen with DeepTune [60], an LSTM based model which takes source code as input and predicts whether it should be run on the CPU or GPU. In 89% of cases, DeepTune surpasses state-of-the-art hand-crafted features, providing up to 14% improved performance.

DeepSmith [59] train a language model on code that performs unit tests in order to generate novel unit tests. They find that as well as finding more errors than hand-written test cases, their generated tests were shorter and thus evaluated faster.

2.4 Predicting Method and Variable Names

Predicting method and variable names is another common task for machine learning applied to source code. This task involves scraping data from open-source repositories and creating a dataset using the method/variable names as targets and the code – with the relevant method/variable names removed – as the input.

2.4.1 Method and Variable Name Prediction

The main use-case for method and variable name prediction models is to suggest appropriate method/variable names given the context of the surrounding code. This can be applied to provide feedback by indicating where a model's predicted method/variable name differs from the method/variable name used by the student. This could potentially highlight the use of a poorly chosen variable name, or perhaps even a bug where the incorrect variable has been used. See Sections 7.1 and 7.2 for a description of the approach taken to predicting method/variable names used in this thesis.

Allamanis et al. [16] introduce the 'method naming problem', with the goal of automatically predicting a descriptive name for a method or class. They use a log-bilinear language model which incorporates non-local information from the method's body. They also train a sub-token model which can generate neologisms, names that do not appear within the training set.

Raychev et al. [220] present an approach to predict method and variable names from obfuscated code. Their method first constructs an AST of the code in question, before building a dependency graph between the variables and then predicting the missing identifiers.

Allamanis et al. [10] use deep learning to predict method name sub-tokens from the associated method body. They call this task "extreme summarization" as it can be thought of as summarizing the method body in to a small amount of tokens – three on average – which make up the sub-tokens of the method name. They find that their model performance improves when the model is augmented with the ability to copy tokens directly from the method body.

Allamanis et al. [9] propose using graph neural networks for the variable naming task. They argue that graph neural networks take advantage of the structured information inside ASTs within source code, and show that their model beats those which treat source code as a flat sequence of tokens. They also show how their model can be used to detect errors in code by searching for where their model's predicted variable does not match the one used within the source code.

Liu et al. [161] train two models in order to predict method names in an unsupervised way. The first model directly converts the method name into a vector and then looks up similar method names based on cosine similarity. The second model, that has been trained to predict the method name from the method body, outputs a vector which is also used to look up method names based on cosine similarity. If these two sets have no intersection, then the method name is deemed inconsistent with the method body and a new one is chosen as the closest method name from the method body embedding.

CODE2VEC [20] learn to represent method names from their corresponding method bodies. They represent their methods as a bag-of-paths, as in [19], and train a relatively simple attention-based model which allows them to train on significantly more data than previous models which they find considerably improves performance over more complex models.

CODE2SEQ [18] improves on CODE2VEC [20] by now predicting method names as a sequence of sub-tokens by using the output of a CODE2VEC model as the initial state of an LSTM. They find that this significantly improves the performance of CODE2VEC whilst still being relatively lightweight and outperforming more complex models.

MNIRE [197] is a tool for predicting a method name from a method body and also checking the consistency of a method name with its method body. The authors achieve improved performance over CODE2VEC [20] by including information from the method's parameter and return types, and the method's class name.

2.4.2 Code De-obfuscation

Models that predict method and variable names can also be used for code de-obfuscation – to convert an obfuscated/minified source code back into a human-readable form. This task is more difficult than the previous method and variable name prediction task as models are unable to rely on existing method or variable names for context.

JSNAUGHTY [269] builds upon AUTONYM and JSNICE and uses SMT techniques to de-obfuscate code. CONTEXT2NAME [29] uses an LSTM to learn how to de-obfuscate code by learning from source code and its automatically obfuscated equivalent. It has comparable performance to JSNAUGTY whilst only taking a fraction of the time. JSNEAT [261] improves on the performance of JSNAUGHTY and CONTEXT2NAME and is faster than both of them.

2.5 Sequence-to-Sequence Models for Code

There are many ways that sequence-to-sequence based models can be applied to source code. They are commonly used to translate from source code to natural language, and vice versa. They're also used to translate from source code in one language to source code in another.

2.5.1 Sequence-to-Sequence Models for Code and Natural Language

The most common use of sequence-to-sequence models for source code and natural language is automatically generating comments/doc-strings for a method given the code tokens, or vice versa. This would allow programmers to write code and have the documentation automatically written for them, or alternatively, write the documentation which specifies what the relevant method should do and have the code automatically generated.

For providing feedback, one of the most straightforward applications is by generating natural language feedback given a method submitted by a student. Unfortunately, this requires a large dataset of method-feedback pairs, and one does not exist yet. However, advancements in this research path will be useful for generating feedback when such a dataset does become available.

Kushman and Barzilay [146] present a technique that parses natural language text searches into regular expressions.

Movshovitz and Cohen [185] use an n-gram model to predict comments from the relevant source code. They use this similarly to code auto-complete systems, with the goal of reducing the characters a programmer needs to write to produce relevant comments.

Oda et al. [200] present the first work on translating from code to pseudo-code. They use SMT tools to generate English descriptions from lines of Python code. They show that the addition of their automatically generated pseudo-code increases how well developers understand code compared to code snippets without any pseudo-code or comments.

Code-NN [122] use a neural attention LSTM model to translate from source code to natural language – for automatic captioning of code – as well as from natural language to source code – for code retrieval from a natural language query. They find their model significantly outperforms previous work [13].

Ling et al. [160] present their Latent Prediction Network, similar to the Pointer Network [271] used in [153], with the goal of translating from meta-data to source code. Their model allows for character generation, directly copying a field from the input or partially copying a field from the input.

DeepAPI [88] is used to translate natural language sequences to API calls.

The CODE2TEXT CHALLENGE dataset [227] is a dataset consisting of a parallel corpus of code and documentation, designed to be used for training models on automatically producing documentation from a provided code snippet and vice versa. The authors followed up this work with a larger dataset [226] which they then used to create FUNCTIONASSISTANT [225], a toolkit for building an API on a corpora of source code allowing users to build their own search API that takes in a natural language query and produces the relevant code snippet from the provided corpora.

TELLINA [159] is a system designed to generate code from natural language snippets. It first uses an entity recognition model to extract the names of files and directory or quantities such as size or time from the natural language snippet. These entities are then obfuscated within the snippet and an RNN is used to generate a program template. The arguments in the template are then filled in with nearest neighbor classification.

The neural attribute machine (NAM) [22] is a variant of an RNN designed for generating source code. It explicitly contains methods which forces the generation of grammatically correct sequences.

Jiang et al. [126] set up the task of generating commits from changes between two versions of a given source code as a NMT task. They find that most commit messages are of low quality and impact the model's performance. They solve this by only using a small subset of their dataset – the ones in which the commit messages begin with a verb-direct object. Loyola et al. [169] also aim to generate natural language commits from source code changes using the model proposed in [123]. They do not filter out any low-quality commit messages and note that their model has a "tendency to choose more general terms over specific ones, meanwhile also avoiding irrelevant words such as numbers or names".

Barone and Sennrich introduce a parallel corpus of Python functions and their doc-strings [28]. They show that translating from code to doc-string is easier than doc-string to code.

Yin and Neubig [294] directly generate source code AST nodes from a natural language input. They model the natural language with a bi-directional LSTM before feeding it into a grammar aware decoder RNN.

Richardson et al. [224] performed experiments on translating natural language to code across ten different programming languages. Their experiments show that polyglot models (trained across all languages) exceeded the performance of monolingual models.

NNGen [165] examine the results of Jiang et al. [126] and discover that there is a large overlap between training and testing data, and that a significant portion of the commit messages within the dataset are noisy or contain little information. Removing the noisy and low information examples causes a large drop in the model's performance. They propose a new model which takes advantage of the training/test data overlap by simply finding the nearest example from the training set which is similar to the proposed example and then outputting that example. This significantly outperforms the results from Jiang et al.

NL2Bash [158] is a dataset comprised of bash scripts with human-annotated labels. The two baselines put forth by the authors show that a sequence-to-sequence network augmented with a copying mechanism [86] beats existing approaches based on heuristics.

CONCODE [121] is a dataset of over 100,000 examples of code, natural language and context tuples – with the task of generating code from natural language and context. The authors propose a new model – that takes context into account and produces code as a set of production rules – which outperforms all the proposed baseline models.

Wan et al. [273] use reinforcement learning on the AST of source code to generate a natural language description. They argue that sequence-to-sequence models suffer from *exposure bias* – that they are not repeatedly given the ground-truth next token in the output sequence when testing, but are during training – which cause them to greedily output the next token in a sequence without maximizing the overall sequence accuracy. The use of reinforcement learning solves this issue by having the agent attempt to maximize the overall sequence accuracy, in terms of BLEU score. They find that their model significantly outperforms their baseline sequence-to-sequence models.

Loyola et al. [170] study the generation of commit messages from the changes between two versions of a given source code. Their hypothesis is that the docstring is as important as the source code itself as it describes the functionality of the code, whilst the changes are only used to describe what has changed in the file.

Hashimoto et al. [97] propose using a retrieval model to aid in translating from natural language to code. They first train the retrieval model as an auto-encoder, which encodes the natural language and attempts to reconstruct it. This can be used to retrieve the closest natural language and code pair within the dataset. Then, they train the code generation model that is conditioned on both the natural language and the retrieved code, but not the retrieved natural language snippet. They show that this model does not beat AST methods, but outperforms all non-AST based methods.

LeClair and McMillan [151] argue that a lack of standardized datasets consisting of parallel examples of code and natural language provide a barrier in terms of research, making it difficult to compare the performance of models. As well as introducing a standardized dataset, they also examine the effect of splitting code datasets by project vs. by method, and the effect of removing automatically generated code. They find that splitting by method leaks information into the test set, artificially inflating BLEU scores, and that automatically generated code should also be removed. They also find that tokens in code are used more often than in natural language, but this repetition is within the same method and not across methods, and that 70% of words in code summary comments do not appear in the code tokens, where the comments represent a high-level summary and the code tokens represent a low-level representation.

Iyer et al. [120] improve on CONCODE [121] by learning to mapping natural language queries directly to source code, using learned code idioms. They define an idiom as a commonly occurring depth-two sub-tree of the AST. Now, instead of decoding their encoded natural language directly as nodes in the AST, they instead decode either a node or an idiom. They find that augmenting their model to generate idioms significantly reduces the training time as well as improving performance.

Agashe et al. [4] introduce JuICe, a dataset of interleaved natural language and code that requires using a long context history. Using baseline neural network models, LSTM with attention and a Transformer, they find that the performance significantly improves with longer context history and the amount of training data used.

CORE [247] learns to pair appropriate code reviews with code changes. They use four bi-directional LSTMS, one for the tokens and one for the characters in both the code tokens and the natural language review. They find that using characters helps their model handle out-of-vocabulary tokens. The output of each pair of LSTMs is concatenated together and fed through an attention layer, and the output of both attention layers are used to calculate similarity. They show their approach significantly outperforms DeepMem [90].

CoDiSum [288] learns to generate commit messages from changes between versions of a given source code. They encode the source code tokens with a bi-directional GRU and then decode the commit message tokens using a GRU with an attention layer and copying mechanisms. They find it beneficial to replace variable names with placeholder tokens, learn separate representations of the variable names and sum them with the GRU output of the placeholder token.

Shido et al. [242] propose a multi-way Tree-LSTM for generating natural language from code ASTs. Their novel multi-way Tree-LSTM architecture does not impose limitations on the number of children within the nodes of the tree and natural language decoder uses attention over the sub-trees of the AST representation, both of which improve performance over standard Tree-LSTM based approaches.

Kacmajor and Kelleher [128] propose using unit tests as a method of automatically extracting code and natural language descriptions. They hypothesize that unit test method names are an apt description of the method being tested and should be used as the natural language description as they are usually self-documenting.

LeClair et al. [150] introduce a model for code to natural language that uses both the code tokens and the AST representation, encoding each separately and then allowing the decoder to apply attention to both individually. They find that their method marginally outperforms a model that only uses the code tokens, however by ensembling a model that uses the code tokens only and one that uses both code tokens and the AST they find they are able to improve performance.

Sun et al. [251] propose generating code from natural language by using a grammar-based CNN. Their model generates code by iteratively generating AST nodes by predicting grammar rules. They show that their approach significantly outperforms recurrent neural network based models.

TAG [45] learns to generate comments from code by using an augmented

AST which has type information embedded within the nodes. The decoder also has a copying mechanism that is type restricted, i.e. it can only copy valid tokens from the input representation.

2.5.2 Sequence-to-Sequence Models for Code to Code

Sequence-to-sequence models can also be used to "translate" between two programming languages. Although not directly related to providing feedback, major advances in encoding or decoding source code with sequence-to-sequence models would also be applicable to sequence-to-sequence models between source code and natural languages.

Nguyen et al. [194] build a SMT system that translates between C# and Java. They note that although they achieve a high translation accuracy of over 80% they find that over 50% of their translations contain basic syntax errors. They note that future models should be constrained when generating translations in order to reduce the amount of syntax errors. StatMiner [195] follows on from that work by translating between C# and Java using a learned mapping between API usage in both languages. They further build on this with mppSMT [193] that translates between programming languages in three steps: first it produces a high level DSL that represents the syntax required, then it maps desired variables with their data and types, finally it maps the desired variable names from the input API call.

An SMT system was also used by Karaivanov et al. [131] who explicitly used the grammar of the desired output programming language to ensure translated code parses correctly. They find that whilst the model without explicit grammar produced similar BLEU scores, it produced significantly fewer programs that parsed and compiled correctly. This leads them to the conclusion that BLEU scores are not necessarily a good indicator of quality in SMT systems for programming languages.

Aggarwal et al. [5] show a pilot study on translating between Python 2 and Python 3. They find their SMT method provides a small performance improvement over a rule-based system.

api2vec [198] uses the word2vec algorithm to learn representations for API elements within API calls within source code. They show that API elements that are nearby in the vector space of the learned representations reflects the similar usage contexts of the API calls used by those elements.

DeepAM [89] uses bi-directional LSTMs to map APIs from one programming language to another. They first train the LSTM, jointly on both languages, to produce natural language documentation from a relevant API call. To translate the APIs they use the trained model to extract the representations from the code tokens in the API call of the input language and then find the closest API call, by taking the cosine similarity of the representations, in the desired output language.

SAR [43] aims to train an API translator from a source language to a target language using less parallel data. They train a generative adversarial network (GAN) [81] to generate embeddings from the AST of the source language and train them to be close together – in vector space – to the embedded AST of the matching target language snippet.

TraFix [134] is a sequence-to-sequence model designed to decompile programs, i.e. go from a low-level representation, such as assembly, to a high-level representation, such as C. They use a novel technique of dynamically increasing the size of the training set by adding in incorrect high-level representation outputs and their matching low-level representations obtained from passing the outputs through a compiler. They state that this teaches the model the correct translations for those high-level representations, thus reducing the chances that the model will generate those translations again when given the original low-level representation.

TransCoder [229] performs unsupervised translation between programming languages. They first train a single masked language model using data from all three languages considered (Python, Java and C++), Then they use the masked language model weights as the initial weights to a de-noising auto-encoder trained to produce a given sequence of code which has been corrupted. Finally, they use back-translation – using a sequence-to-sequence model to translate from one language to another and then back to the original language – to generate supervised data to be used for translation. They find their model significantly outperforms existing rule-based transcompilers.

AthenaTest [267] learns to generate test cases for a given method. They find that pre-training their code-to-test model as a masked language model on a combination of English and Java outperforms pre-training on either language individually, or training their model from scratch.

Wei et al. [281] investigate both code summarization (code to natural language) and code generation (natural language to code). They propose a technique which performs code summarization and generation simultaneously, enforcing that the outputs of each model – an LSTM with attention – are similar by using regularization. They find this regularization allows their models to outperform previous approaches in both code summarization and code generation.

2.6 Mining Source Code

Mining source code allows patterns within source code to be extracted. There are a wide variety of applications that involve mining source code, such as: extracting code snippets, detecting common idioms, predicting types, and classifying programs.

2.6.1 Mining Idioms from Source Code

Mining code idioms is commonly used to perform exploratory analysis on source code by finding common patterns of code tokens which frequently occur. This could potentially be used in a pedagogical setting by applying automatic exploratory data analysis techniques on a corpus of student submissions and then using the results of this data analysis in order to provide feedback to a large group of students.

On a study of repetitiveness in code changes [196] the authors find that the vast majority, over 70% of changes made by programmers are small changes that are identical to another change made in the history of that project.

Irish [49] is a system designed to extract code snippets that are embedded within natural language documents, e.g. e-mails. Irish achieves comparable performance to other systems, without the need to manually write regular expressions or grammars.

Haggis [11] mines idioms from source code, an idiom being an abstract syntax tree fragment that is commonly used. From a large dataset of source code they learn a probability distribution over abstract syntax trees which they model as a probabilistic context-free grammar.

Allamanis et al. [15] mine loop idioms, discovering that 90% of loops have fewer than 15 lines of code, 90% of them have no nesting and that 50 idioms covered 50% of all loops within their dataset.

StaQC [292] is a dataset of programming related question and answer pairs. A bidirectional LSTM was used to mine the correct code snippet provided in the answer to the question. Their approach is over 15% more accurate than previous methods which rely on heuristics.

The CoNaLa Corpus [295] is an improved version of the StaQC dataset [292], where heuristics have been added to remove extraneous lines of code which appeared in the answer to the provided question.

Louis et al. [168] perform the task of predicting which code blocks should be commented or not.

Another aspect of feedback that can be applied to code is formatting. Generally, formatting should follow conventions that are consistent throughout the entirety of a student submission. By training models which can automatically detect inconsistent formatting and apply the correct formatting conventions, feedback in terms of code formatting can be provided.

Style-Analyzer [178] trains a random forest to correctly format code by creating examples from functions with their formatting removed. They train their model on a per-project basis, which allows it to adapt to the specific formatting conventions used within the project.

2.6.2 Predicting Types

Feedback can also be applied to the variable types used in static programming languages. By training a model to predict types and then applying it to student code and examining where the model and the code disagree it can potentially highlight areas where the students have used the incorrect variable types.

RefiNym [64] mines and assigns meta-types to variables within source code. Variables with the same meta-type are assumed to be used in a similar context within the source code and thus can be explicitly typed and type-checked.

DeepTyper [102] explores the task of applying optional types to dynamic programming languages, such as Python or JavaScript. They use a two-layer bidirectional LSTM with a novel consistency layer between the two layers of the LSTM. The consistency layer causes the hidden state for a token to be equal to the average hidden states of all occurrences of that token.

This allows the model to predict types based on both a sequentially local representation and the consensus judgment for all other locations where this identifier occurs.

TypeWriter [211] learns to predict argument and return types for dynamic languages, such as Python and JavaScript. The model uses both static analysis – to extract the AST of the code – followed by a neural network applied to the code and doc-strings to predict missing type annotations, which have incorrect predictions filtered out by a second stage of static analysis. They find their approach outperforms DeepTyper [102] and NL2Type [176].

Schrouff et al. [238] compare graph convolutional networks (GCNs) and gated graph neural networks (GGNNs) to predict types in JavaScript code from the relevant AST using the type, property and value of each node. They find that the GGNN has marginally improved performance over the GCN, although they propose this could be due to the simplicity of the task.

Typilus [17] uses a GGNN to predict type annotations in Python code. It is able to predict a type in 70% of cases, and out of those predictions is correct enough to pass a type-checker 95% of the time.

2.6.3 Classifying Programs

Programs can also be classified using supervised learning. The most common use-case of classifying programs is detecting which algorithm was implemented by a student. This can potentially be used to provide feedback by detecting if an inefficient algorithm was used, e.g. bubble sort, and automatically applying feedback to suggest a more efficient implementation.

Peng et al. [204] present one of the first uses of deep learning in programming languages. They teach a deep neural network to classify programs based on their abstract syntax tree. They qualitatively show that the deep neural network has learned to cluster similar abstract syntax tree nodes together within its intermediate – hidden – representations. Furthermore, they show their deep neural network performs better than linear regression or an SVM. The authors later apply convolutional neural networks [184] to the same task and find that it significantly outperforms their previous work.

Bui et al. [42] apply the model of Mou et al. [184] to the tasks of determining if two programs implement the same algorithm, where the programs are in different programming languages.

Bui et al. [192] propose the dependency tree-based convolutional neural network (DTBCNN) for the task of classifying programs. Their model, as well as using the AST, uses the dependency tree for calculating the representations for each node. They show that their model outperforms the graph neural network (GNN) [154] and the previous state-of-the-art, the TBCNN model [184].

TreeCaps [125] uses a capsule network [110, 231] to classify programs. The work is the first to apply capsule networks to trees and beats the performance of the GNN, TBCNN and DTBCNN.

Tan et al. [172] use graph neural networks to classify programs, improving on the gated graph neural network by adding attention mechanisms.

Sharma et al. [241] classify code, with the predicted class being the occurrence of a "code smell". A code smell is an indication of quality issues within the code. They find that the performance of deep learning models is smell-specific. RNNs are better at detecting some smells than CNNs, however take considerably longer to train. They also find that it is possible to train a model to detect smells in one programming language and perform transfer learning, applying it to a second programming language, achieving similar results to directly training on the second programming language.

DyPro [277] is a neural network model designed for classifying source code. DyPro differs from previous approaches by not classifying code based on their source code tokens or AST, but by using the sequence of variable states as the program is tested with random inputs.

2.7 Source Code Retrieval

Source code retrieval covers using source code, either as an input or a target, to look up a relevant entry from a database. This can be used to: look up a natural language snippet from a source code snippet (or vice versa), and look up a source code snippet from another source code snippet – such as in the task of clone detection for plagiarism. See Sections 6.1 and 6.2 for how source code retrieval is used in this thesis.

2.7.1 Semantic Code Search

Semantic code search involves training a model which can find the most relevant source code example when given a natural language query, or vice versa. This can be used to save developers time by having them search through a database of previously implemented methods instead of writing their own from scratch. For applying feedback, if a dataset of code-feedback pairs existed then a model could be trained to automatically find the most relevant human-written feedback when given a student submission. This is significantly easier than generating human-written feedback using a sequence-to-sequence model so would require significantly less training data, and hence is a more realistic short-term goal for providing automatic feedback.

CODEnn [87] is a tool used to search for relevant code snippet given a natural language query. Instead of using traditional information retrieval techniques, they use machine learning. The model transforms both the code and the query into a high dimensional representation. The cosine similar between these high dimensional representations is used to tell how similar the code and query are. Their model outperforms non-machine learning based approaches.

NCS (Neural Code Search) [232] work on the hypothesis that "tokens in source code contain enough natural language information to make retrieval possible". They use the FastText [37] embeddings to get an embedding vector for each token within a given function and then take a weighted average over these embedding vectors to get a final vector representation of that function. They find that this simple approach works well when retrieving a paired natural language query that has been processed in the same way, showing that unsupervised learning of code representations is able to achieve acceptable performance for the code retrieval task.

Cambronero et al. [46] perform an evaluation for code retrieval across NCS [232], CODEnn [87], SCS [115] and their own novel model, UNIF – which is the NCS model with added supervision. They find that their UNIF model outperforms the other three models and that performance of the supervised models significantly improves when the supervised dataset consists of actual queries instead of using the method doc-strings as queries – which is the default in code retrieval tasks due to the ability easily harvest vast amounts of examples.

Kulal et al. [144] focus on mapping pseudo-code to actual source code. They note that previous metrics used to measure the quality of code generation are insufficient as they provide a value that represents what percentage of the target code was successfully generated, whereas the actual metric of interest is functional correctness. For each pseudo-code input, they generate multiple candidate source code translations, each of which they run against a suite of test cases. They find that performance, in terms of the number of passed tests, can be improved by using a neural network across the generated code and the error messages provided by the test suite to localize which lines are causing an error and down-weight candidates containing that line.

MMAN [274] uses the sequential tokens, the AST and the CFG of source code to create a representation of a method to be used for source code retrieval. The sequence of tokens is fed through an LSTM, the AST through a Tree-LSTM and the CFG through a GGNN. These three representations have self-attention applied and are then concatenated together and fed through a single layer linear network to get a final representation. They show this multi-modal representation outperforms previous source code retrieval methods.

CodeSearchNet [116] is a dataset of code and doc-string pairs designed for training semantic code search systems. The dataset consists of over two million code-doc-string pairs in six different programming languages. The authors of CodeSearchNet experiment with different models, using an identical model to encode the code and doc-strings, and find that the performance on CodeSearchNet improves with model complexity, i.e. using a pair of Transformer models performs better than a pair of NBoW models. However, they find that most of this complexity is required by the doc-strings as using an NBoW model to encode the code tokens and a Transformer to encode the doc-strings provides comparable performance to using a pair of Transformer models.

CoaCor [291] combines code retrieval (obtaining a relevant code using a natural language) and code annotation (generating a natural language snippet from a code snippet). Their model first performs code annotation and then uses the resulting natural language to perform code retrieval, using both the original code and the generated natural language. By training their model using reinforcement learning to optimize the mean reciprocal rank (MRR) they find their generated natural language captures more of the semantic meaning of code and improve performance over previous approaches.

TransS3 [280], like CoaCor [291], performs code annotation for improving code retrieval. Unlike CoaCor, which uses an LSTM on code tokens, TransS3 uses a Transformer over the AST of the code. TransS3 significantly outperforms CoaCor in both the code annotation and code retrieval tasks.

2.7.2 Clone Detection

Clone detection is similar to semantic code search, however instead of the using code-query pairs the dataset is now code-code pairs and the largest the similarity between two code examples the higher the probability that they are clones, i.e. perform the same functionality. This could potentially be used for plagiarism detection, or assist in classifying programs by determining if they perform the same functionality as some provided reference method.

White et al. [282] are the first to apply neural language models for code clone detection. They first use a recurrent neural network to obtain embeddings of the individual tokens before feeding these embeddings, along with the AST, to a recursive neural network. They compare the final output of the recursive neural network between two potential clones to determine if they are actually clones, or not.

Oreo [233] uses Siamese neural network to detect if two functions are semantic clones or not. They do not input the tokens of the suspected clone programs, but instead the inputs are results obtained via static analysis of the two programs.

Tufano et al. [264] develop a machine learning clone detection algorithm that uses the source code at multiple levels of abstraction, namely: identifiers, ASTs, CFGs, and byte-code. It is the first work to apply machine learning simultaneously to multiple levels of source code abstraction. They show that each representation is complementary to each other, and the addition of each improves performance.

2.8 Meta

There is also research focused on adapting machine learning techniques to the domain of programming languages. These involve research on learning representations of source code, which improve the performance of applications, and examining the closed vocabulary problem which is an issue in source code.

2.8.1 Code Datasets

The majority of machine learning on source code research uses data scraped from open-source repositories. These open-source repositories allow a large unlabeled dataset to be collected easily, however it is non-trivial to obtain the exact dataset used by researchers in order to replicate their work. Fortunately, standardized datasets, such as CodeSearchNet [116] and CodeXGLUE [173], are beginning to appear. See Chapter 4 for information about the datasets used in this thesis.

Allamanis [8] conducted a study into code duplication in machine learning on code applications. They find that commonly used source code datasets contain examples that exist in both the training and test sets, which artificially inflates the evaluation performance of these models.

2.8.2 Code Embeddings

Embeddings are distributed representations of code that can potentially be used in downstream tasks performed on source code. These are commonly used as the input to a machine learning model applied to code. Learning efficient embeddings is common in natural language processing by using language models trained on a large corpus of text and fine-tuned on the desired downstream task. These embeddings usually provide improved performance over learning embeddings from scratch.

inst2vec [30] is a method of training embeddings for source code which can be used for machine learning models. The source code is first processed into a language independent intermediate representation as well as the contextual flow graph. They then apply the skip-gram model [182] to learn embeddings for each instruction in the intermediate representation.

import2vec [259] is a method for learning representations of libraries with the aim of having semantically similar libraries have similar representations. Their representations are learned using the word2vec model [182, 183] on the co-occurrence of `import` statements within code snippets scraped from open-source repositories.

IdBench [272] is a benchmark for evaluating learned representations of source code tokens using a dataset of identifier pairs hand labeled by expert humans in terms of similarity and relatedness. They find that learned code representation techniques accurately represent the relatedness of identifiers, but perform poorly when measuring similarity. They also show that learned code representations significantly outperform string distance functions and that an ensemble of learned representations outperform individual learned representations, implying the techniques compliment each other.

COMMIT2VEC [171] learns representations of code changes to be used in downstream tasks. These representations represent code, before and after a change, as a bag-of-paths, similar to [19] and feed both pre- and post-change code into an LSTM to obtain a distributed representation for each. The difference between these two representations are used as the commit representation after being fed through a linear layer. They find that representing code as a bag-of-paths outperforms representing code as a sequence of tokens. They find that pre-training their representations on a dataset more relevant to the downstream task provides improved performance than representations pre-trained on a larger dataset that is significantly different to the downstream task.

2.8.3 Vocabularies

In machine learning, a vocabulary is a unique mapping between a code token and an integer. One of the issues with machine learning applied to code is the large number of unique tokens within a dataset due to method and variable names usually consisting of a combination of multiple sub-tokens.

Cvitkovic et al. [62] study the vocabulary problem that arises when applying machine/deep learning to programming languages. As the majority of source code variables usually only appear very few times within a dataset, they will be "out-of-vocabulary" and indistinguishable from other out-of-vocabulary variables. They propose adding a vocabulary cache on to a graph neural network. The cache performs character level embedding on each variable name. They find their cache improves performance over baseline models – even without adjusting any hyperparameters – as well as over graph neural network models augmented with a pointer network.

Karampatsis and Sutton [132] also tackle the open vocabulary problem by breaking code tokens into small sub-word units with byte-pair encoding (BPE). This allows any code token to be represented by sub-words, the smallest of which are single characters, without ever having to replace it with an out-of-vocabulary token. They show that a language model using their "open" sub-word vocabulary outperforms "closed" vocabulary models, even when the size of the sub-word vocabulary is relatively small.

2.8.4 Novel Code Representations

When source code is used as input to machine learning models it is traditionally represented as a sequence of tokens, AST or CFG. However, researchers have also come up with novel representations specifically to be used for machine learning on code.

Alon et al. [19] introduce a novel method of representing programs – as a sequence of paths between each leaf node with every other leaf node. This representation is programming language agnostic, and the authors show it improves performance over previous models on the tasks of: variable name prediction, method name prediction and full type prediction. These representations allow information from the AST to be used by non-graph neural networks. They also introduce a tool, Pigeon, which implements their representations. A similar method was used in code2vec [20] and code2seq [18].

PathMiner [139] is a library which implements path-based representations similar to Pigeon and works for Python, Java, C/C++ and JavaScript.

Yin et al. [296] use a neural network to learn representations of source code edits. Using a GGNN as an auto-encoder, they learn *edit representations* which can be applied to encoded representations of source code and then decoded to receive the source code with the appropriate edit.

Zhang et al. [300] proposes splitting full ASTs into a sequence of smaller *statement trees*. They show that this representation has improved performance over models that consider the full AST.

Chen and Monperrus [52] provide a short literature review on distributed representations (embeddings) of source code.

2.8.5 Adversarial Attacks

Finally, research on how to train models is followed by research on how to break those models. Neural networks are susceptible to adversarial attacks [256, 145] – inputs which are designed to "trick" the neural network into producing an incorrect output.

DAMP [293] is a method of performing adversarial attacks on machine learning models designed to predict a method name from a method body. They show that by either changing a variable name or adding a declaration of an unused variable, both of which do not change the semantics of the method, they are able to change the model's predicted output to an adversarial target method name 89% of the time and any incorrect method 94% of the time. They also propose a method of defending against these adversarial attacks – by inserting a variable outlier detection model which detects adversarially swapped or inserted variables and replacing them with an out-of-vocabulary token.

2.9 Conclusion

Table 1 shows an overview of the work covered in this literature review.

Reference	Task	Model	Method
AUTOMATA [249, 245]	FG	MLP	SL
Piech et al. [207]	FG	RNN	SL
OVERCODE [79]	FG	Clustering	UL
DEEPCODEREVIEWER [90]	FG	LSTM	SL
Ahmed et al. [6]	FG	Transformer	UL
Kremenek et al. [141]	BL	Static Analysis	SL
SK_P [212]	BC	LSTM	SL
DEEPREPAIR [283]	BC	Autoencoder	UL
Devlin et al. [67]	BC	LSTM	SL
DEEPFIX [93]	BC	LSTM	SL
SYNFIX [32]	BC	LSTM	UL
Santos et al. [234]	BC	LSTM	UL
DEEPBUGS [210]	BL	MLP	SL
Bhatia et al. [31]	BC	LSTM	SL
Harer et al. [96]	BC	GAN	UL
RATCHET [98]	BC	LSTM	SL

Reference	Task	Model	Method
Tufano et al. [265]	BC	LSTM	SL
Chen et al. [53]	BC	LSTM	SL
SampleFix [95]	BC	VAE	UL
Tufano et al. [266]	BC	LSTM	SL
Habib and Pradel [94]	BC	LSTM	SL
NeuralBugLocator [91]	BL	CNN	SL
Vasic et al. [268]	BC	LSTM	SL
GREAT [103]	BC	Transformer	UL
Li et al. [154]	PV	GNN	SL
Wang et al. [278]	PV	DBN	SL
Murali et al. [187]	PV	RNN	UL
Koc et al. [138]	PV	LSTM	SL
Russel et al. [230]	PV	CNN	SL
Hindle et al. [108]	LM	n-gram LM	SSL
Allamanis and Sutton [12]	LM	n-gram LM	SSL
Syzygy [201]	LM	n-gram LM	SSL
SLAMC [199]	LM	n-gram LM	SSL
Hsiao et al. [114]	LM	n-gram LM	SSL
Campbell et al. [47]	LM	n-gram LM	SSL
Maddison and Tarlow [175]	LM	CFG	SSL
Tu et al. [262]	LM	n-gram LM	SSL
CACHECA [74]	LM	n-gram LM	SSL
Naturalize [14]	LM	n-gram LM	SSL
Slang [221]	LM	RNN	SSL
Hellendoor et al. [100]	LM	n-gram LM	SSL
Karpathy et al. [133]	LM	LSTM	SSL
White et al. [284]	LM	RNN	SSL
Saraiva et al. [235]	LM	n-gram LM	SSL
Ray et al. [219]	LM	n-gram LM	SSL
Allamanis et al. [13]	LM	NBoW	SSL
PHOG [34]	LM	CFG	SSL
BugGram [279]	LM	n-gram LM	SSL
Hellendoorn et al. [101]	LM	n-gram LM	SSL
Lanchantin and Gao [147]	LM	LSTM	SSL
Craic [167]	LM	LSTM	SSL
Parvez et al. [202]	LM	LSTM	SSL
Alon et al. [21]	LM	Transformer	SSL

Reference	Task	Model	Method
Rahman et al. [218]	LM	n-gram LM	SSL
Brockschmidt et al. [39]	LM	GNN	SSL
CuBERT [129]	LM	Transformer	SSL
CodeBERT [73]	LM	Transformer	SSL
Bruch et al. [41]	AC	kNN	UL
Bhoopchand et al. [33]	AC	LSTM	SSL
Li et al. [153]	AC	LSTM	SSL
Murali et al. [186]	AC	GSNN	SSL
Pythia [255]	AC	LSTM	SSL
Hussain et al. [117]	AC	GRU	SSL
Rabinovich et al. [213]	AC	LSTM	SSL
Zaremba et al. [297]	AC	LSTM	SSL
CLGen [61]	AC	LSTM	SSL
DeepSmith [59]	AC	LSTM	SSL
Allamanis et al. [16]	VP	MLP	SL
Raychev et al. [220]	VP	CFG	SL
Allamanis et al. [10]	VP	CNN	SL
Allamanis et al. [9]	VP	GNN	SL
Liu et al. [161]	VP	CNN	UL
code2vec [20]	VP	MLP	SL
code2seq [18]	VP	LSTM	SL
MNire [197]	VP	RNN	SL
JSNaughty [269]	CDO	SMT	SL
Context2Name [29]	CDO	LSTM	SL
JSNeat [261]	CDO	SMT	SL
Kushman and Barzilay [146]	SSNL	CFG	SL
Movshovitz and Cohen [185]	SSNL	n-gram LM	SL
Oda et al. [200]	SSNL	SMT	SL
Code-NN [122]	SSNL	LSTM	SL
Ling et al. [160]	SSNL	LSTM	SL
DeepAPI [88]	SSNL	RNN	SL
Tellina [159]	SSNL	RNN	SL
NAM [22]	SSNL	RNN	SL
Jiang et al. [126]	SSNL	LSTM	SL
Yin and Neubig [294]	SSNL	LSTM	SL
Richardson et al. [224]	SSNL	LSTM	SL
NNGen [165]	SSNL	LSTM	SL

Reference	Task	Model	Method
Wan et al. [273]	SSNL	LSTM	RL
Loyola et al. [170]	SSNL	LSTM	SL
Hashimoto et al. [97]	SSNL	LSTM	SL
Iyer et al. [120]	SSNL	LSTM	SL
CORE [247]	SSNL	LSTM	SL
CoDiSum [288]	SSNL	GRU	SL
Shido et al. [242]	SSNL	LSTM	SL
LeClair et al. [150]	SSNL	LSTM	SL
Sun et al. [251]	SSNL	CNN	SL
TAG [45]	SSNL	LSTM	SL
Nguyen et al. [194]	SSC	SMT	SL
StatMiner [195]	SSC	SMT	SL
mppSMT [193]	SSC	SMT	SL
Karaivanov et al. [131]	SSC	SMT	SL
Aggarwal et al. [5]	SSC	SMT	SL
api2vec [198]	SSC	MLP	SL
DeepAM [89]	SSC	LSTM	SL
SAR [43]	SSC	GAN	UL
TraFix [134]	SSC	LSTM	SL
TransCoder [229]	SSC	Transformer	UL
AthenaTest [267]	SSC	Transformer	SL
Wei et al. [281]	SSC	LSTM	SL
Irish [49]	MI	HMM	UL
Haggis [11]	MI	CFG	SL
Louis et al. [168]	MI	LSTM	SL
Style-Analyzer [178]	MI	RF	SL
RefiNym [64]	PT	CFG	UL
DeepTyper [102]	PT	LSTM	SL
TypeWriter [211]	PT	LSTM	SL
Schrouff et al. [238]	PT	GNN	SL
Typilus [17]	PT	GNN	SL
Peng et al. [204]	PC	MLP	SL
Bui et al. [192]	PC	CNN	SL
TreeCaps [125]	PC	CN	SL
Tan et al. [172]	PC	GNN	SL
Sharma et al. [241]	PC	RNN	SL
DyPro [277]	PC	LSTM	SL

Reference	Task	Model	Method
CODEnn [87]	SCS	MLP	SL
NCS [232]	SCS	MLP	SL
Cambronero et al. [46]	SCS	LSTM	SL
Kulal et al. [144]	SCS	LSTM	SL
MMAN [274]	SCS	GNN	SL
CoaCor [291]	SCS	LSTM	SL
TranS3 [280]	SCS	Transformer	SL
White et al. [282]	CD	RNN	SL
Oreo [233]	CD	LSTM	SL
Tufano et al. [264]	CD	LSTM	SL

Table 1: A summary of the work covered in the literature review. Tasks: FG (feedback generation), BL (bug location), BC (bug correction), PV (program verification), LM (language modeling), AC (autocompletion), VP (variable prediction), CDO (code de-obfuscation), SSNL (sequence-to-sequence models between code and natural language), SSC (sequence-to-sequence models between code), MI (mining idioms), PT (predicting types), PC (program classification), SCS (semantic code search), CD (clone detection). Models: MLP (multi-layer perceptron), RNN (recurrent neural network), LSTM (long short-term memory), GAN (generative adversarial network), VAE (variational autoencoder), CNN (convolutional neural network), GNN (graph neural network), DBN (deep belief network), CFG (context free grammar), NBoW (neural bag-of-words), kNN (k nearest neighbours), GRU (gated recurrent unit), SMT (statistical machine translation), HMM (hidden Markov model), RF (random forest). Methods: SL (supervised learning), UL (unsupervised learning), SSL (self-supervised learning), RL (reinforcement learning).

As shown, there is little research on applying machine learning directly to the application of grading or providing feedback on source code, and the research that does exist is limited in its use. Automata [249] uses hand-crafted features and only produces a numerical grade. OverCode [79] also relies on existing, non-machine learning based static and dynamic code analysis. Work by Piech et al. [207] uses machine learning without hand-crafted features, but is limited to a toy language. DeepCodeReviewer [90] also is a pure machine learning system, but cannot produce original feedback and simply returns what it believes are the closest one from a database of feedback.

There are significantly more bug location, correction, and verification tools that use machine learning, most of which are sequence-to-sequence models over sequences of tokens or the abstract syntax trees of programs. These models learn from examples containing bugs which are either taken from source code changes in open-source repositories or introduced in the code by some heuristic. These systems show promise, however their performance is relatively low – usually only able to fix under 50% of examples – and also produce a fix, but do not provide any feedback to the user.

Applying language models to source code is the most popular application of machine learning to source code. The majority of work in this space uses recurrent neural networks – such as an LSTM, although Transformer models are becoming more popular in language modeling for natural language processing – to learn from a sequence of source code tokens. Another popular topic is using recursive neural networks to model the nodes in an abstract syntax tree, although these are being replaced by graph neural networks, which are also now being replaced by Transformers. This thesis focuses on a pedagogical setting where users may want to query a model on code that is potentially incomplete and thus will not compile and produce an abstract syntax tree, which limits the relevance of models applied to trees obtained from code. Most of the work in this area also focuses on language modeling with very little thought put into the use case of these systems outside a casual mention of being a glorified code auto-complete tool, and no mention of use in a pedagogical setting is mentioned.

Due to the prevalence of scaling laws in neural networks (see Chapter 8.3), the future of language modeling in natural language processing is heading towards using large pre-trained language models that are only fine-tuned on a desired task. There is no doubt that this will also apply to language models for code. One of the major issues with training large language models is ensuring a high-quality dataset to train on – "garbage in, garbage out" – this may be something language models for source code can easily avoid as open-source repositories can have their quality judged by the number of "stars" or "followers", although scarce research has been done on the quality of language models with respect to their datasets.

Language models such as GPT-3 [40] are so prohibitively large that they cannot even be fine-tuned and are evaluated in a "one-shot" setting where their inputs are written in such a way that by predicting the next token in the given sequence will produce a task specific answer, e.g. to predict the sentiment of the sentence "I didn't like this movie at all" the example is rewritten as "I didn't like this movie at all. I am feeling `[BLANK]`" and the predicted sentiment is either "positive" or "negative" depending on which of those two words the language model believes is more likely to appear in the blank. There has been no formal research on how to most effectively craft examples in such a way for optimal one-shot performance, or how this can be applied to tasks involving source code.

Predicting method and variable names is one of the tasks which could be most easily applied to a pedagogical setting – its application is relatively unobtrusive as it makes no functional changes to the users code and providing sensible variable names is an important task which new programmers commonly struggle with.

Sequence models for source code, such as models which translate from natural language to source code or vice versa, have the potential to be significant educational tools – if a model was trained on a large parallel corpus of code and hand-written feedback with a sufficient performance it would effectively replace the majority of work done by human graders. However, no such datasets exists and the majority of the work in this area is focusing on automatic documentation generation via learning to generate doc-strings for functions.

Mining source code has a wide variety of applications, some of which have potential pedagogical applications, such as: learning to automatically format code to follow conventions, predicting the correct types for variables, and classifying code. Again, however, most of the work is focused on a specific task with no thought to the pedagogical aspects.

Semantic code search, which is one of the most common tasks for machine learning on code, is another task which has significant research interest and a large amount of potential to be applied in a pedagogical setting. However, most applications only pair up methods with doc-strings – as this data is relatively easy to obtain – with no focus on how these models could be used to apply feedback.

As shown, a common theme is the increasing use of machine learning systems on source code, but most of which provide little to no insight into how these applied to a pedagogical setting or perform any qualitative analysis in regard to how these models can be used in an educational setting. This thesis takes the three most common applications of machine learning on source code – language modeling, semantic code search, and predicting variable names – which show the most promise to be adapted for a pedagogical setting, and focus on how to improve models on these tasks whilst suggesting ways they can be deployed for providing feedback.

3 Models

3.1 Introduction

The experiments in this thesis use four main models: neural bag-of-words (NBoW), a recurrent neural network (RNN), a convolutional neural network (CNN), and a Transformer model. Each model is a type of neural network[5] and were chosen as they have distinct architectures from each other. By showing how some findings in this thesis hold true across all four architectures, it strengthens the potential of these findings being universal across all neural network architectures. Conversely, by showing findings which differ between architectures, it allows for a discussion on why these findings have different outcomes on the different architectures.

Each of the four architectures can be represented as:

$$\boldsymbol{Z} = f_{\theta_m}(\boldsymbol{X})$$

where $\boldsymbol{X}$ is a length N sequence of tokens. The tokens within this sequence can be words, characters or code tokens, e.g. $\boldsymbol{X} = [$`print`, `"`, `Hello`, `"`, `;`$]$. Before being passed to the model, the sequence is *numericalized*, converted from a string to an integer, using a *vocabulary*, a unique mapping of strings to integers, e.g. `print` $\rightarrow 0$, `"` $\rightarrow 1$, `Hello` $\rightarrow 2$, `;` $\rightarrow 3$, and then usually converted to a *one-hot vector*, a V-dimensional vector with all elements being zero except the one at the position indicated by the integer, which is a one. V is the size of the vocabulary, thus $\boldsymbol{X} \in \mathbb{R}^{N \times V}$. $\boldsymbol{Z} \in \mathbb{R}^{N \times F}$ is the sequence of F-dimensional *features* output by the model, f, which is parameterized by the model's parameters (also commonly called the *weights*), θ_m. Each of the N features is a distributed representation of the corresponding input element. Depending on the architecture, each feature may be calculated independent of the other elements in the sequence, or calculated using information from other elements in the sequence.

For each task, the features are passed through a task-specific *head*:

$$\hat{\boldsymbol{Y}} = g_{\theta_h}(\boldsymbol{Z})$$

[5]Specifically, each of the four models are a *deep* neural network, using the definition that a deep neural network has more than one hidden layer of neurons.

g is the task specific head, a neural network parameterized by θ_h, the architecture of which depends on the task at hand. $\hat{\boldsymbol{Y}}$ is the output of the neural network, it can either be a sequence of outputs, one corresponding to each element in the input sequence, $\mathbb{R}^{N \times C}$, or a single output $\mathbb{R}^C$, depending on the task. C is the number of classes. The final layer in g is usually the *softmax* activation function:

$$\text{softmax}(\boldsymbol{z}_i) = \frac{e^{\boldsymbol{z}_i}}{\sum_{j=1}^{C} e^{\boldsymbol{z}_j}}$$

where $\boldsymbol{z}$ represents the input vector the softmax function. Softmax is applied across the C-dimension and *normalizes* the vector $\hat{\boldsymbol{Y}}$ so that each component will be between zero and one, and the entire vector will sum to one. This effectively converts the raw output (commonly called *logits*) into a probability distribution.

To update the parameters, θ_m and θ_h, a *loss function* is used to calculate the *loss*, a value to be minimized that measures "how close" our model is to a correct prediction, between the neural network's predicted output, $\hat{\boldsymbol{Y}}$, and the desired output of the model, $\boldsymbol{Y}$. When the final layer of a model is the softmax activation function the loss function used is almost always *cross-entropy loss*, and when the output is a sequence the loss is calculated across the length dimension and averaged:

$$\mathcal{L}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = -\frac{1}{N} \sum_{i=1}^{N} \boldsymbol{Y}_i \cdot \log(\hat{\boldsymbol{Y}}_i)$$

The *backpropagation* algorithm is then used to find the gradient of the loss with respect to each of the parameters, i.e. how much "influence" each parameter had on the loss value. Hence, the loss function and the entire neural network must be differentiable. Gradient descent is then used to update the parameters of the neural network by taking a "step" in the direction of the negative gradient for each parameter:

$$\theta_i \leftarrow \theta_i - \eta \nabla_\theta J(\theta_i)$$

η is known as the *learning rate*, and controls the size of the step taken. $\nabla_\theta J(\theta_i)$ is the gradient of the parameter θ_i with respect to the loss.

In practice, to fully utilize the parallelism of graphics processing units (GPUs), which are used to train modern algorithms, a *batch* of examples are fed through the model in parallel and the loss is the averaged across the batch.

3.2 Neural Bag-of-Words

The neural bag-of-words (NBoW) (sometimes called *continuous bag-of-words* (CBoW)) model is a relatively simple model that serves as popular baseline in natural language processing [124].

In the NBoW model, the one-hot encoded input, $\boldsymbol{X} \in \mathbb{R}^{N \times V}$ is used to obtain an *embedding vector* for each element in the sequence, $\boldsymbol{E} \in \mathbb{R}^{N \times D}$, where $\boldsymbol{E} = \boldsymbol{XW}$. $\boldsymbol{W} \in \mathbb{R}^{V \times D}$ is known as the embedding matrix, and $D <<< V$. As each element of $\boldsymbol{X}$ is a one-hot vector, the operation $\boldsymbol{XW}$ can be thought of as treating $\boldsymbol{W}$ as a look-up table and retrieving the i-th row of $\boldsymbol{W}$ where i is the non-zero element in each one-hot encoded vector. The embeddings vectors have shown to be able to capture semantic and syntactic features of natural languages [183, 182, 181].

Traditionally, for classification tasks the embedding vectors are averaged across the sequence length, multiplied by a weight matrix and then passed through a softmax layer. As each embedding vector is calculated independently, the output of the NBoW model would be identical for every permutation of the input sequence, i.e. the NBoW model output is calculated without regard to the order of the items in the input sequence, hence the *bag* in bag-of-words. However, the NBoW model implemented for the experiments in this thesis simply outputs the embeddings vectors, hence:

$$\boldsymbol{Z} = f_{\theta_m}(\boldsymbol{X}) = \boldsymbol{XW} = \boldsymbol{E}$$

Hence, the NBoW model can be thought of as a single linear layer, with no bias term. It is commonly used as the first layer in neural network architectures applied to natural language processing where it is usually known as an *embedding layer*, and each element of $\boldsymbol{E}$, denoted by $\boldsymbol{e} \in \mathbb{R}^D$, is known as a *word embedding*, or simply an *embedding*.

3.3 Recurrent Neural Networks

The most common neural network architectures used when the input is a sequence are recurrent neural networks (RNNs). RNNs have been used for neural machine translation [26, 55, 252, 285], speech recognition [85], generating image captions [287], natural language inference [38, 164], named entity recognition [205, 7], part-of-speech tagging [208, 36], text classification [113, 205], summarization [189, 57] and language modeling [84, 179, 205].

Unlike standard neural networks, RNNs have *temporal* connections. This allows them to process sequences one element at a time and propagate information through time. The RNN steps along the sequence of inputs, usually after they have passed through an embedding layer, and at each time-step the RNN calculates a *hidden state*, $\boldsymbol{h}^{(t)} \in \mathbb{R}^H$, recurrently as:

$$\begin{aligned}\boldsymbol{h}^{(t)} &= \sigma(\boldsymbol{e}^{(t)}\boldsymbol{W}_h + \boldsymbol{h}^{(t-1)}\boldsymbol{U}_h + \boldsymbol{b}_h)\\ &= \mathrm{RNN}(\boldsymbol{e}^{(t)}, \boldsymbol{h}^{(t-1)})\end{aligned}$$

The hidden state is calculated using the input embedding at the current time-step, $\boldsymbol{e}^{(t)} \in \mathbb{R}^D$, an input weight matrix, $\boldsymbol{W}_h \in \mathbb{R}^{D\times H}$, the hidden state from the previous time-step, $\boldsymbol{h}^{(t-1)} \in \mathbb{R}^H$, and recurrent weight matrix, $\boldsymbol{U}_h \in \mathbb{R}^{H\times H}$, and a bias term, $\boldsymbol{b}_h \in \mathbb{R}^H$. Notice how the input and recurrent weight matrices are shared between each time-step. σ is the *sigmoid function*, a non-linear activation function given by:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

The sigmoid function has the property of constraining the output to be between zero and one. The initial hidden state, $\boldsymbol{h}^{(0)}$, is usually initialized to a zero vector. The hidden states at each time-step $t \geq 1$ are concatenated together to give a sequence of hidden states, $\boldsymbol{H} \in \mathbb{R}^{N\times H}$.

These hidden states can be used as input to another RNN, creating a *multi-layer RNN*, i.e. for RNNs after the first:

$$\boldsymbol{h}^{(t,l)} = \mathrm{RNN}(\boldsymbol{h}^{(t,l-1)}, \boldsymbol{h}^{(t-1,l)})$$

l denotes the layer index, $\boldsymbol{h}^{(t,l-1)}$ is the input to the RNN from the previous layer, and $\boldsymbol{h}^{(t-1,l)}$ is the hidden state from the previous time-step. The input to the first RNN layer is always the embedded input tokens, $\boldsymbol{e}^{(t)}$.

Each hidden state can be thought of as a summary of the sequence up to time-step t. It is often beneficial for each hidden state to contain information about time-steps $> t$. This can be implemented using a second RNN which steps over the sequence in reverse and the hidden state at each time-step is now the concatenation, $[\cdot, \cdot]$, of the initial *forward RNN* and additional *backward RNN* for the corresponding time-step:

$$\boldsymbol{h}^{(t)} = [\boldsymbol{h}^{(t)}_{\rightarrow}; \boldsymbol{h}^{(t)}_{\leftarrow}]$$

$\boldsymbol{h}^{(t)}_{\rightarrow}$ is the hidden state obtained from the forward RNN after it has seen inputs $< t$, whereas $\boldsymbol{h}^{(t)}_{\leftarrow}$ is the hidden state obtained from the backward RNN after has seen the inputs $> t$. A model containing both a forward and backward RNN is known as a *bi-directional RNN*, and each hidden state is now $\boldsymbol{h}^{(t)} \in \mathbb{R}^{2H}$.

The "RNN" term actually covers a family of models. The one detailed above is the most basic RNN, and is also called an *Elman network* [70]. However, Elman networks are not commonly used in practice as they suffer from the *exploding/vanishing gradient problem* when handling long sequences. When performing backpropagation through time, the gradient for early time-steps is calculated by repeatedly multiplying partial derivatives from all subsequent time-steps. If these partial derivatives have values greater than one then the gradient will exponentially increase over time, or *explode*, causing the model parameters to increase exponentially and making the output extremely unstable. Conversely, if the partial derivatives are less than one then the gradient will exponentially decrease, or *vanish*, over time, preventing the parameters from updating and effectively halting the learning process.

Ideally, an RNN should "remember" information from previous time-steps by being able to multiply the hidden state by an identity mapping, i.e. $\boldsymbol{U}_h$ should be an identity matrix. The partial derivative of an identity function is one, hence this would prevent exploding/vanishing gradients. However, the non-linear sigmoid activation function in the Elman network prevents an identity mapping being learned. A variant of the RNN, the long-term short-term memory (LSTM) [111, 77] architecture is designed with the ability to remember information across time-steps by explicitly allowing an identity function to be learned.

The key concepts introduced in the LSTM are the use of *gates* to control information flow in and out of the LSTM, and a *memory cell*, a vector which stores information over time. The LSTM computes hidden states as:

$$\begin{aligned}
\boldsymbol{f}^{(t)} &= \sigma(\boldsymbol{e}^{(t)}\boldsymbol{W}_f + \boldsymbol{h}^{(t-1)}\boldsymbol{U}_f + \boldsymbol{b}_f) \\
\boldsymbol{i}^{(t)} &= \sigma(\boldsymbol{e}^{(t)}\boldsymbol{W}_i + \boldsymbol{h}^{(t-1)}\boldsymbol{U}_i + \boldsymbol{b}_i) \\
\boldsymbol{o}^{(t)} &= \sigma(\boldsymbol{e}^{(t)}\boldsymbol{W}_o + \boldsymbol{h}^{(t-1)}\boldsymbol{U}_o + \boldsymbol{b}_o) \\
\hat{\boldsymbol{c}}^{(t)} &= \tanh(\boldsymbol{e}^{(t)}\boldsymbol{W}_c + \boldsymbol{h}^{(t-1)}\boldsymbol{U}_c + \boldsymbol{b}_c) \\
\boldsymbol{c}^{(t)} &= \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}_t \odot \hat{\boldsymbol{c}}^{(t)} \\
\boldsymbol{h}^{(t)} &= \boldsymbol{o}^{(t)} \odot \tanh(\boldsymbol{c}^{(t)}) \\
&= \mathrm{LSTM}(\boldsymbol{e}^{(t)}, \boldsymbol{h}^{(t-1)}, \boldsymbol{c}^{(t-1)})
\end{aligned}$$

The *forget gate*, $\boldsymbol{f} \in \mathbb{R}^H$, controls how much information in the cell, $\boldsymbol{c} \in \mathbb{R}^H$, is kept from the previous time-step. If the forget gate is a zero vector, then the current contents of the cell are erased, and if it's one vector then the current contents of the cell are remembered. The *input gate*, $\boldsymbol{i} \in \mathbb{R}^H$, controls how much information from the cell's input, $\hat{\boldsymbol{c}} \in \mathbb{R}^H$, should be written to the cell. If the input gate is a zero vector then no new information is added to the cell. The *output gate*, $\boldsymbol{o}$, controls how much information should be output by the LSTM from the cell. $\boldsymbol{W} \in \mathbb{R}^{D\times H}$ and $\boldsymbol{U} \in \mathbb{R}^{H\times H}$ are the input and recurrent weight matrices, the bias terms are denoted by $\boldsymbol{b} \in \mathbb{R}^H$, $\odot$ is element-wise multiplication and tanh is the hyperbolic tangent function, a non-linear activation function which bounds outputs between zero and one.

When the forget gate is a one vector and the input gate is a zero vector, then the current cell is equal to the previous cell $\boldsymbol{c}^{(t)} = \boldsymbol{c}^{(t-1)}$. Hence, the partial derivative is one and the gradient can flow freely without exploding or vanishing.

The LSTM is ubiquitous in natural language processing, so much so that when researchers and practitioners mention recurrent neural networks, they are usually referring to LSTMs. In this thesis a *multi-layer bi-directional LSTM* is used as the RNN of choice, unless explicitly noted otherwise, hence:

$$
\begin{aligned}
\boldsymbol{Z} &= f_{\theta_m}(\boldsymbol{X}) = \boldsymbol{H}^{(L)} \\
\boldsymbol{H}^{(l)} &= [\boldsymbol{h}_{\rightarrow}^{(t,l)}; \boldsymbol{h}_{\leftarrow}^{(t,l)}]_t^N \\
\boldsymbol{h}_{\rightarrow}^{(t,l)} &= \mathrm{LSTM}_{\rightarrow}^{l}(\boldsymbol{z}_{\rightarrow}^{(t,l)}, \boldsymbol{h}_{\rightarrow}^{(t-1,l)}, \boldsymbol{c}_{\rightarrow}^{(t-1,l)}) \\
\boldsymbol{h}_{\leftarrow}^{(t,l)} &= \mathrm{LSTM}_{\leftarrow}^{l}(\boldsymbol{z}_{\leftarrow}^{(t,l)}, \boldsymbol{h}_{\leftarrow}^{(t-1,l)}, \boldsymbol{c}_{\leftarrow}^{(t-1,l)}) \\
\boldsymbol{E} &= [\boldsymbol{e}^{(t)}]_t^N \\
\boldsymbol{E} &= \boldsymbol{X}\boldsymbol{W}
\end{aligned}
$$

In other words, the features, $\boldsymbol{Z}$, are the concatenation of the forward and backward hidden states from the final layer LSTMs across all time-steps, $\boldsymbol{H}^{(L)}$. L denotes the number of layers, $\boldsymbol{z}^{(t,l)} = \boldsymbol{e}^{(t)}$ when $l = 1$, i.e. the first layer LSTM inputs are the embedding vectors, and $\boldsymbol{z}^{(t,l)} = \boldsymbol{h}^{(t,l-1)}$, when $l > 1$, i.e. the subsequent layer LSTM inputs are the hidden states output by the LSTM from the previous layer.

3.4 Convolutional Neural Networks

Convolutional neural networks (CNNs) are extensively used in modern approaches to computer vision. They have been used in image classification [143, 244, 99], image segmentation [228] and object detection [78, 222], but have also been applied to natural language processing in applications such as text classification [135] and neural machine translation [76].

A CNN consists of multiple *convolutional layers*. Each convolutional layer contains J convolutional kernels (also called *convolutional filters* or *receptive fields*) with a defined height, width and depth. However, when applied to sequences they only have a width and depth, with the depth usually equal to D, the size of the embedding vectors. This is known as a 1-dimensional CNN. These kernels "slide" over the embedded input sequence and at each time-step multiply the kernel weights by the embedding values covered by the kernel, e.g. a $3 \times D$ kernel covers three embedding vectors at once and calculates a hidden state for all kernels, $\boldsymbol{h}_3^{(t)} \in \mathbb{R}^J$ as:

$$\boldsymbol{h}_3^{(t)} = \boldsymbol{e}^{(t-1)}\boldsymbol{W}_1 + \boldsymbol{e}^{(t)}\boldsymbol{W}_2 + \boldsymbol{e}^{(t+1)}\boldsymbol{W}_3 + \boldsymbol{b}$$

$\boldsymbol{e}^{(t)} \in \mathbb{R}^D$ is the embedding of input element t, $\boldsymbol{W}_i \in \mathbb{R}^{D\times J}$ are the kernel weights, and $\boldsymbol{b} \in \mathbb{R}^J$ is the bias term. The values of the kernel are shared across every time-step, making the convolutional layer translation invariant, i.e. a sequence of three input elements will give the same $\boldsymbol{h}$ value no matter where they appear in the input sequence. However, the convolutional layer does not contain information about where those three input elements appear within a sequence. Each of the J filters in the example above can be thought of as learning to extract features from a bag of *tri-grams* from an embedded sequence.

More generally, the equation to calculate the hidden state at time-step t using a convolutional layer with kernel width k is:

$$\boldsymbol{h}_k^{(t)} = \boldsymbol{b} + \sum_{i=1}^{k} \boldsymbol{e}^{t+(i-k+1)} \boldsymbol{W}_i$$

The hidden states at each time-step can be concatenated together to get a sequence of hidden states, $\boldsymbol{H} \in \mathbb{R}^{N\times J}$. To ensure input and output are the same lengths, each sequence is padded with $k-1$ zeros, thus all e^t for $t \leq 0$ and $t > N$ are zero vectors, and only odd kernel widths are used.

For the CNN architecture used in this thesis, K convolutional layers, all with J kernels but each with different widths, are applied directly to the input sequence. All CNN layers use a stride of one, and no dilation is used. The outputs of each of these convolutional layers are concatenated together and then passed through a rectified linear unit (ReLU), a non-linear activation function given by:

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if x} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Hence, the CNN architecture calculates features, $\boldsymbol{Z}$, as:

$$
\begin{aligned}
\boldsymbol{Z} &= f_{\theta_m}(\boldsymbol{X}) = \boldsymbol{H} \\
\boldsymbol{H} &= [\mathrm{ReLU}(\boldsymbol{h}^{(t)}))]_t^N \\
\boldsymbol{h}^{(t)} &= [\boldsymbol{h}_k^{(t)}]_k^K \\
\boldsymbol{h}_k^{(t)} &= \boldsymbol{b} + \sum_{i=1}^{k} \boldsymbol{e}^{t+(i-k+1)} \boldsymbol{W}_i \\
\boldsymbol{E} &= [\boldsymbol{e}^{(t)}]_t^N \\
\boldsymbol{E} &= \boldsymbol{X}\boldsymbol{W}
\end{aligned}
$$

The features are the sequence of hidden states, $\boldsymbol{H}$, where each hidden state, $\boldsymbol{h}^{(t)}$, is the concatenation of K convolutional layers, each with J kernels and width k, passed through a rectified linear unit. Each convolutional layer is applied directly to the input embeddings, $\boldsymbol{e}^{(t)}$.

3.5 Transformer

The Transformer is a relatively recent neural network architecture, initially designed for neural machine translation [270] although Transformer variants have been used to achieve hold state-of-the-art performance across almost all natural language processing tasks, such as: summarization [217, 301], question answering [66, 217], language modeling [214, 215, 63, 140, 40], named entity recognition [66, 289], natural language inference [290, 188] and general language understanding [66, 290, 217].

Transformers do not contain any recurrence or convolution operations, and can be parallelized at scale, allowing them to be scaled up considerably. This, along with the growing amount of data and compute, has led to Transformer-based models frequently being used to achieve state-of-the-art results in many branches of machine learning research.

The Transformer heavily relies on the use of *attention*, which was popularized in neural networks by Bahdanau et al. [26]. Attention is a normalized vector, obtained by passing an *alignment vector* through a softmax function. The attention vector is then applied to other elements to get a weighted sum of those elements. Bahdanau et al. use attention in neural machine translation, where they first generate a sequence of hidden states for the input sequence, then at each time-step, when generating the translated output sequence one word at a time, the attention is heavily weighted on the hidden states corresponding to the aligned words within the input sequence. For example, when translating the phrase "Ich habe Hunde" from German to English, after outputting the words "I have", the attention vector should be concentrated on the hidden state for the word "Hunde" for translating it to the English word "dog". The key concept is that the alignment vector is learned by the model itself and is dependent on the current context.

In a Transformer, the embeddings, $\boldsymbol{E} \in \mathbb{R}^{N \times D}$, are passed through L *blocks* (sometimes called layers). Within the first block the embeddings are used as input to three linear layers, q, k, and v to get $\boldsymbol{Q}, \boldsymbol{K}$ and $\boldsymbol{V}$, all $\mathbb{R}^{N \times D}$, i.e.:

$$\begin{aligned}\boldsymbol{Q} &= q(\boldsymbol{E}) = \boldsymbol{E}\boldsymbol{W}_q + \boldsymbol{b}_q \\ \boldsymbol{K} &= k(\boldsymbol{E}) = \boldsymbol{E}\boldsymbol{W}_k + \boldsymbol{b}_k \\ \boldsymbol{V} &= v(\boldsymbol{E}) = \boldsymbol{E}\boldsymbol{W}_v + \boldsymbol{b}_v\end{aligned}$$

$\boldsymbol{W}_q, \boldsymbol{W}_k$ and $\boldsymbol{W}_v$ are $\mathbb{R}^{D \times D}$ and $\boldsymbol{b}_q, \boldsymbol{b}_k$ and $\boldsymbol{b}_v$ are $\mathbb{R}^{D}$. $\boldsymbol{Q}, \boldsymbol{K}$ and $\boldsymbol{V}$ are known as the *query*, *key* and *value*, respectively.

The alignment vector is then calculated as the *scaled dot product* between the query and the key, as:

$$\text{alignment}(Q, K) = \frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{D}}$$

This alignment (sometimes called *score* or *energy*) is then passed through a softmax activation function to get the attention:

$$\boldsymbol{A} = \text{softmax}\Big(\text{alignment}(Q, K)\Big)$$

$\boldsymbol{A} \in \mathbb{R}^{N \times N}$, and each row, i, contains the attention between input element i and all N other elements. As the input sequence is calculating attention across itself, this is sometimes known as *self-attention*. This interconnection between all items in a sequence, has also led researchers to find parallels between Transformers and *graph neural networks* [69, 127].

The attention is then applied to the value:

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{V}$$

$\boldsymbol{C} \in \mathbb{R}^{N \times D}$ is known as the *context* and each element of $\boldsymbol{C}$ is a weighted sum of $\boldsymbol{V}$.

The Transformer uses a novel variant of attention called *multi-head attention*. Before the alignment vector is calculated, $\boldsymbol{Q}, \boldsymbol{K}$ and $\boldsymbol{V}$ are reshaped to $\mathbb{R}^{N \times h \times D/h}$ where h is the number of *heads* and must be a multiple of D. The context is then calculated across each of the multiple heads individually, concatenated together and then reshaped to $\mathbb{R}^{N \times D}$. This can be seen as performing h attention calculations in parallel.

The context is then summed with the input embedding to the block via a residual connection and then passed through a *layer normalization* layer, which normalizes the residual sum across the D-dimension by subtracting the mean and dividing by the variance.

$$\boldsymbol{R} = \text{LayerNorm}(\boldsymbol{E} + \boldsymbol{C})$$

$\boldsymbol{R} \in \mathbb{R}^{N \times D}$ is then passed through two linear layers with a non-linear activation function between them:

$$\boldsymbol{S} = \alpha(\boldsymbol{R}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2$$

$\boldsymbol{S} \in \mathbb{R}^{N \times D}, \boldsymbol{W}_1 \in \mathbb{R}^{D \times S}, \boldsymbol{W}_2 \in \mathbb{R}^{S \times D}$, with $S >>> D$. α is the non-linear activation function, which is ReLU in the original Transformer model, although GELU [104] is now more commonly used. This can be seen as "expanding" the D-dimensional states into S dimensions before contracting them again.

The final operation within the block is another residual and layer normalization:

$$\boldsymbol{H} = \text{LayerNorm}(\boldsymbol{R} + \boldsymbol{S})$$

$\boldsymbol{H} \in \mathbb{R}^{N \times D}$ is then used as input to the next Transformer block which contains the exact same set of operations, hence:

$$\boldsymbol{H}^{(l)} = \text{TransformerBlock}(\boldsymbol{H}^{(l-1)})$$

Where $\boldsymbol{H}^{(l)} = \boldsymbol{E}$ when $l = 1$.

One thing to note is that the Transformer has no method of knowing the position of each element within a sequence. The solution is to calculate a *position embedding* for each position and add them to the word embeddings.

$$\boldsymbol{E} = \boldsymbol{X}\boldsymbol{W}_e + \boldsymbol{P}\boldsymbol{W}_p$$

$\boldsymbol{P} \in \mathbb{R}^{N \times P}$ are the position indices, i.e. the first element of $\boldsymbol{P}$ is a one-hot vector with the one in the first element. $\boldsymbol{W}_p \in \mathbb{R}^{P \times D}$ is the position weight matrix, and P is the maximum length sequence the Transformer can handle, usually 256 or 512.

In the original Transformer implementation [270], the position embeddings are calculated as different frequency sine and cosine waves. It is now more common to learn $\boldsymbol{W}_p$ [66] from scratch, however some recent research [119, 236] has shown position embeddings actually decrease the performance of Transformer models.

Hence, the Transformer calculates features, $\boldsymbol{Z}$ as:

$$\begin{aligned}
\boldsymbol{Z} &= f_{\theta_m}(\boldsymbol{X}) = \boldsymbol{H}^{(L)} \\
\boldsymbol{H}^(l) &= \text{TransformerBlock}(\boldsymbol{H}^{(l-1)}) \\
\boldsymbol{E} &= [\boldsymbol{e}^{(t)}]_t^N \\
\boldsymbol{E} &= \boldsymbol{X}\boldsymbol{W}
\end{aligned}$$

The features $\boldsymbol{Z}$ are the output of the final block, $\boldsymbol{H}^{(L)}$. The input to a block is the output of the previous block, $\boldsymbol{H}^{(l-1)}$, and the first block's input is the sequence of embedding vectors, $\boldsymbol{H}^{(1)} = \boldsymbol{E}$. Within each block: the query, key and value are calculated and then split into h heads; the scaled dot product is used to obtain the alignment between the query and key; the attention is calculated from the alignment and applied to the value to get the context; the context has a residual connection applied and then passed through a layer normalization layer; the result of this is then expanded and contracted through two linear layers with an activation function between them; and finally, another residual connection and layer normalization is calculated.

3.6 Hyperparameters

Each of the four models are defined by their own set of *hyperparameters*. A *parameter* refers to the weights and biases that are learned via the architecture. A hyperparameter is used to define the specifics of the architecture or training set-up, e.g. the number of neurons in a hidden layer, the learning rate, etc. The model hyperparameters are shown in Table 2.

The models are all trained using the *Adam* optimizer [136] using the default learning rate of 0.001. Adam adapts the learning rate of each parameter individually over time, reducing the learning rate for parameters updated more frequently and increasing it for parameters updated less frequently. It also applies *momentum* to the parameter updates, making them a weighted sum of the current and previous parameter update. Although there is no single optimization algorithm considered the "best", empirical results [237, 56, 248] have shown Adam to consistently perform well across a range of tasks.

Training is performed using a batch size of 128. This is chosen as it is the largest batch size that fits into GPU memory whilst using the largest (in terms of number of parameters) model, the Transformer. The gradients are clipped to a value of 1.0 before used for parameter updates. This is used to prevent exploding gradients in some architectures. Models are trained for 5 epochs and each experiment is run 3 times using different random seeds, with the results averaged across the 3 runs.

For all models, the embedding vectors are 128-dimensional, and the dropout rate is set to 0.1. Dropout [250] is a *regularization* technique used to prevent *over-fitting* in neural network models. It works by setting a proportion (defined by the dropout rate) of the neurons to zero during each forward pass when training. Dropout reduces the ability for neurons to depend on other neurons from previous layers, making the parameters more robust, and can also be thought of as training multiple smaller neural networks and ensem-

	Model			
Hyperparameter	NBoW	LSTM	CNN	Transformer
Embedding Dim.	128	128	128	128
Hidden Dim.	-	256	-	256
Layers	-	2	-	3
Heads	-	-	-	8
Filters	-	-	100	-
Filter Sizes	-	-	3, 5, 7	-
Dropout Rate	0.1	0.1	0.1	0.1
Parameters (M)	6.6	6.8	9	7.1

Table 2: Hyperparameters used for the NBoW, LSTM, CNN and Transformer models, and the number of parameters in each model.

bling them together when dropout is "turned off" during evaluation.

For the NBoW model, the only parameters are contained within the embedding layer. Dropout is applied to the output of the embedding layer.

The LSTM model used is a two-layer bi-directional LSTM. The hidden states are 256-dimensional. Dropout is also applied to the output of the embedding layer and on the hidden states passed the two LSTM layers.

The CNN passes the input through the embedding layer and then applies 100 filters, of widths 3, 5 and 7, and depths equal to the embedding dimension, over the sequence. The output of each filter is passed through a ReLU activation function, and then the result of the 300 filters at each time-step is concatenated together. Dropout is applied to the output of the embedding layer and to the concatenation of the filters.

In the Transformer, positional embeddings are learned and the GELU activation function is used over ReLU to calculate $\boldsymbol{S}$. The hidden dimension, S is set to 256. The Transformer contains three layers ("blocks") and the multi-head attention layer uses 8 heads.

Generally, these hyperparameters were chosen using values which are commonly used across machine learning research, as well as values which were suitable for the compute resources used – most experiments were performed on a single NVIDIA GTX 1080 Ti GPU. Extensive hyperparameter tuning was not performed, due to limitations in time (due to the number of experiments performed) and available compute resources. Fine-tuning was performed using the same hyperparameters used to train the pre-trained models.

All experiments are implemented in PyTorch [203] and use the PyTorch's default weight initialization method, which is usually $\mathcal{U}(-\frac{1}{\sqrt{\text{fan in}}}, \frac{1}{\sqrt{\text{fan in}}})$, where *fan in* is the number of input features into the layer. The exception to this is the Transformer model which is initialized from a *truncated normal distribution* with a standard deviation of 0.02, following BERT [66]. The reason for the truncated normal distribution with a low standard deviation is that Transformer models have issues with stability when training [299, 298, 302] and require initializing values from very small distributions.

4 Data

4.1 CodeSearchNet

In computer vision and natural language processing, several popular benchmarks are used to compare performance between models. For example computer vision has MNIST [152], CIFAR10/100 [142], ImageNet [65] for image classification, and natural language processing has SuperGLUE [275] for natural language understanding. One issue in machine learning for code is the lack of standardized benchmark datasets. The majority of papers scrape their own dataset from open-source repositories, usually using some quality metric such as number of "stars", and then apply their own methods to this dataset. LeClair and McMillian [151] argue that this reduces the progress of research as each researcher must run all compared models on their scraped dataset. They also briefly discuss how the performance of models across different datasets varies wildly. There also exists an issue with code duplication in open-source repositories, as Lopes et al. [166] show that 70% of code on GitHub – the most commonly used website for scraping code for datasets – is a duplicate of existing code. Allamanis examined the effect of this duplication on machine learning for code models [8] and found that removing duplicates significantly reduces the reported performance of the examined models.

Due to the above issues, this thesis primarily uses the CodeSearchNet dataset[6] [116] throughout. Using a standardized dataset, instead of scraping a novel dataset, allows for easier replication of results obtained in this thesis. This dataset has all duplicates removed using the techniques described by Allamanis in [8], is available in both raw and pre-tokenized formats, and has already been split into training, validation and test sets. It is also available through the HuggingFace Datasets library[7], a popular library for accessing standardized datasets, further increasing standardization. As such, this work uses CodeSearchNet via the HuggingFace library.

The CodeSearchNet dataset consists of over two million functions across six programming languages: PHP, Java, Python, Go, JavaScript and Ruby[8].

[6]https://github.com/github/CodeSearchNet

[7]https://huggingface.co/datasets/code_search_net

[8]The CodeSearchNet dataset also contains an extra 99 evaluation examples manually collected by human experts. These are not used for the experiments in this thesis

Language	**Functions**	**Average Tokens**	**Total Tokens (M)**	**Unique Tokens (M)**
PHP	578,118	127	73.5	2
Java	496,688	112	55.8	1.9
Python	457,461	117	53.7	2.4
Go	346,365	107	37.1	0.9
JavaScript	138,625	170	23.7	0.8
Ruby	53,279	69	3.7	0.2
All	2,070,536	119	247.3	7.7

Table 3: CodeSearchNet dataset statistics using the TreeSitter tokenisation method.

Available for each function is: the raw code string, code tokens obtained with TreeSitter[9], the raw documentation (the code comments associated with the function), and the documentation tokens obtained via a regular expression[10] that splits a string on whitespace and punctuation. The dataset statistics are shown in Table 3

One issue faced when using the TreeSitter tokens is the large number of unique tokens within the dataset. Natural language processing models have a bound vocabulary size, usually within the tens of thousands, and any time a natural language processing model encounters a token not within its vocabulary that token is replaced by an *out-of-vocabulary* (OoV) token (also commonly called an *unknown token* and denoted by `<unk>`). In natural languages, OoV tokens are relatively rare when using a sensible vocabulary size, however in programming languages the number of unique tokens is considerably higher as function and variable names are usually made of a concatenation of several words describing the intended purpose of the function or variable, e.g. `getAspectRatio`, `getScreenWidth`, `getScreenHeight`.

There have been some approaches to the OoV problem which dynamically update the vocabulary [62, 101], however the most common approach to reducing the vocabulary size is by splitting tokens into sub-tokens, e.g. splitting on camel case `getAspectRatio` becomes the three sub-tokens `get`, `Aspect` and `Ratio`, and splitting on snake case `set_frequency` becomes the two sub-tokens `set` and `frequency`. Instead of naively splitting the TreeSit-

[9]https://github.com/tree-sitter/tree-sitter

[10]https://github.com/github/CodeSearchNet/blob/master/src/dataextraction/utils.py#L7

Language	**Average Tokens**	**Total Tokens (M)**
PHP	187	108.3
Java	174	86.5
Python	284	130
Go	159	55.2
JavaScript	274	38
Ruby	134	7.2
All	205	425

Table 4: CodeSearchNet dataset statistics using BPE tokenization method on the raw code string.

ter tokens further into sub-tokens, the work in this thesis uses *byte pair encoding* (BPE) to tokenize the raw code string, ignoring the TreeSitter tokens completely. BPE tokenization involves learning a tokenizer from data by splitting tokens based on commonly found sub-strings. For example, `getScreenWidth` and `getScreenHeight` would become four unique tokens under the naive sub-token approach – `get`, `Screen`, `Height` and `Width` – but under BPE tokenization would become only three unique sub-tokens – `getScreen`, `Width` and `Height`. Using BPE has been shown by Karampatsis and Sutton [132] to significantly improve the performance of neural language models on programming languages, and is now becoming commonly used on natural languages [240, 214, 215, 40].

The BPE tokenizer used is from the HuggingFace Transformers library[11]. It has been trained on the training data from all six languages in the CodeSearchNet dataset with a fixed vocabulary size of 52,000 sub-tokens. There are now no OoV tokens within the data used, as the BPE tokenizer will repeatedly split the raw code string until it matches an item in the vocabulary, with the shortest sub-tokens being individual characters. One disadvantage of this is the increased length in the number of tokens per function. Table 4 shows the average tokens per function and total tokens per language when the BPE tokenizer is used. It can be seen that, across all six languages, the average tokens per function and the total number of tokens increases by 1.7×.

[11]https://huggingface.co/huggingface/CodeBERTa-small-v1

To narrow the scope of experiments in this thesis, most of the work is focused on the Java data within CodeSearchNet. Java was chosen as it has the second most number of functions within the CodeSearchNet dataset, and, compared to PHP, it is commonly used in machine learning applied to programming languages [12, 21, 20, 18, 247, 288, 151] and is one of the most popular programming languages used by software developers[12,13,14]. However, advances in natural language processing on English data also apply to other natural languages with the same roots, e.g. French and German, so it is believed that the findings obtained in this thesis are not be unique to Java and should also be applicable to other imperative programming languages.

4.2 WikiText-103

In some experiments, natural language data is used to provide a comparison to the Java code. The natural language data used is the WikiText-103 dataset [180], a collection of 28,595 articles from English Wikipedia. WikiText-103 was chosen over other natural language datasets as it commonly used for language modeling[15] and is comparable in size to the Java dataset from CodeSearchNet – it is actually 1.7 times larger, in terms of number of tokens, however other language modeling datasets are considerably smaller (the Penn Treebank dataset [177] is 100 times smaller) or larger (the Billion Word Benchmark [51] is 100 times larger). BPE tokenization is also applied to this dataset, using the same BPE tokenizer as the one applied to the CodeSearchNet data. As this tokenizer is trained on programming languages, it leads to longer sequences of tokens than if a BPE tokenizer trained on English language data were used (WikiText-103 is described in [180] as having a total of 104M tokens, however our BPE tokenization of WikiText-103 has closer to 149M tokens). However, this was chosen so the data from all six programming languages and the natural language share a single vocabulary and tokenization method.

[12]https://insights.stackoverflow.com/survey/2020/#most-popular-technologies
[13]https://insights.stackoverflow.com/survey/2019/#most-popular-technologies
[14]https://insights.stackoverflow.com/survey/2018#most-popular-technologies
[15]http://nlpprogress.com/english/language_modeling.html#wikitext-103

5 Language Modeling for Source Code

5.1 Introduction

The most common application of machine learning for source code is *language modeling* (see Chapter 2, Section 2.3). A *language model* is a statistical model which assigns a probability to a sequence consisting of one or more tokens, x. The goal of a language model is to estimate the probability of a sequence of tokens:

$$P(x_1, x_2, \cdots, x_N) = \prod_{n=1}^{N} P(x_n | x_1, \cdots, x_{n-1})$$

Language models can also be thought of as generative probabilistic models, as given a sequence of tokens they produce a probability distribution over the next token in the sequence. Using a well-trained language model, tokens can be repeatedly sampled to generate likely sequences. For example, given the sequence of tokens `printf`, `"`, `Hello`, `World`, `!`, a well-trained language model should predict the token `"` with a high probability.

Traditionally, P was an *n-gram language model*, in which the probability of a token given a sequence was the proportion of times that same token followed that same sequence within some training data. n-gram language models usually have *smoothing* applied to ensure all sequences have a non-zero probability, even if they did not occur with in the training data.

Hindle et al. [109] were among the first to use language models for source code. In their paper they refer to the language model as measuring *naturalness* of code. They note that although natural language is complex, humans tend to stick to a set of common utterances and idioms. This repetition and predictability allows natural language processing techniques to work effectively across all natural languages on all tasks. Programming languages are also complex, however programs are written by humans in a way that is meant to be understood by other humans. Because of this, they mostly use simple, repetitive, predictable idioms and thus the same techniques applied to natural languages should also be applicable to programming languages. For example, the code `for(int i=0;i<10` is most likely followed by `;i++){`, and a well-trained language model should be able to predict the sequence of tokens that form `;i++){` with a reasonably high probability.

The performance of a language model is typically measured using *cross-entropy*:

$$H(x_1, x_2, \cdots, x_N) = -\frac{1}{N}\sum_{n=1}^{N} \log P(x_n|x_1, \cdots, x_{n-1})$$

Or, *perplexity*, $PP = e^H$. In both of which, lower is better.

One problem is that n-gram language models are unable to obtain any context further than the n token window. Increasing n leads to sparsity in the occurrence counts as longer sequences are less likely to occur in the training data, causing n-gram language models to generalize poorly across relatively small domain shifts, i.e. where common sequences in the test data did not appear in the training data.

This problem with n-gram language models, along with the advances in neural networks, mean that recurrent neural networks have become the most common approach to language modeling, i.e. P is now a recurrent neural network model which takes in a sequence of tokens and outputs a probability distribution over the next token in the sequence. In theory, a recurrent neural network's hidden state allows it to keep information over arbitrarily long sequence lengths.

Language modeling is a common task in natural language processing as it has now become common to take a trained language model and *fine-tune* it on the desired downstream task, e.g. text classification, question answering, etc. This is known as *transfer learning* [303]. Transfer learning is frequently able to achieve improved results compared to training a model on the downstream task from scratch, and usually allows for sufficient performance even with a low number of training examples.

Due to language models being trained using a form of *self-supervised learning* [163] – where the labels come from the data itself – they can leverage large amounts of unlabeled data. Language models trained on natural language usually get this data by scraping it from the internet. Language models for source code are able to leverage websites that host open-source repositories, such as GitHub[16] or GitLab[17]. As this large amount of data is available, language models for source code are able to be trained and then fine-tuned to improve the performance on any downstream task, making them a common research topic for machine learning on source code as language model perplexity correlates with transfer learning performance [190, 3].

A relatively new approach to language modeling is *masked language modeling* [66]. Instead of predicting the next token in a sequence, one (or more) tokens within the sequence are replaced by a `<mask>` token and the masked language model is trained to predict what the original token was. For example, given the sequence `for(<mask> i=0;i<10;i++){` the masked language model should predict `int` with a high probability. The benefits of masked language modeling is that it allows the prediction to use information from both sides of the masked token, whereas standard language modeling can only use information preceding the token to be predicted.

This chapter investigates language modeling applied to the CodeSearchNet dataset (detailed in Chapter 4, Section 4.1) using the models discussed in Chapter 3. Experiments aim to answer three main questions: do masked language models outperform standard language models? Does the Transformer, the state-of-the-art language model for natural languages, also provide the best performance on programming languages? Can the findings of Hindle et al. [109], that the perplexity of source code is less than that of natural language, be independently replicated on this dataset with these models?

5.2 Methods

In order to prepare the data to be used for language modeling, all functions are tokenized, using the tokenizer detailed in Chapter 4, Section 4.1, and then appended together into a single sequence with an *end of sequence*token, `<eos>` appended between each individual sequence. The sequence is input into the models as chunks of N tokens, where N is chosen uniformly between 45 and 55, similar to [179]. For the standard language modeling task, the target for each element in the sequence is the next token in the sequence, i.e. the target is the input shifted by one time-step. For the masked lan-

[16]https://github.com/
[17]https://gitlab.com/

guage modeling task, 15% of tokens are replaced by `<mask>` tokens and the model has to predict each of the original masked tokens. This is done both for the Java data in the CodeSearchNet dataset and the entirety of the WikiText-103 dataset.

As standard language modeling requires architectures that are unable to "see" the inputs from future time-steps (or else they can simply see the next token in the sequence and don't need to learn it) only the NBoW and uni-directional LSTM models are trained as standard language models. The NBoW model does not pass information between time-steps, so is predicting a token based only on the previous token. The uni-directional LSTM model only passes information forward in time, so can use information from the "past", but not the "future", to make predictions. The CNN and Transformer are able to see information from future time-steps, so are not trained as standard language models.

Masked language models are able to use information from the whole sequence, so a bi-directional LSTM, CNN and a Transformer are trained as masked language models. Training an NBoW model as a masked language model would involve training a model to predict the true token behind a `<mask>` with no other information and would involve simply predicting the most common token in the dataset. Due to this, an NBoW model is not trained as a masked language model.

The task specific head in each of the language models is a linear layer which takes in the sequence of features from the model, $\boldsymbol{Z} \in \mathbb{R}^{N \times F}$, and for each feature applies a weight and bias followed by the softmax function: $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{z}\boldsymbol{W} + \boldsymbol{b})$. With $\boldsymbol{z} \in \mathbb{R}^{F}$, $\boldsymbol{W} \in \mathbb{R}^{F \times V}$, and $\boldsymbol{b} \in \mathbb{R}^{V}$. F is the features output by the model, V is the size of the vocabulary which is defined by the tokenizer.

The model and training hyperparameters are detailed in Chapter 3, Section 3.6.

Model	Type	Language	Train Perplexity	Validation Perplexity	Test Perplexity
NBoW	LM	Java	69.79	89.10	97.55
NBoW	LM	WikiText-103	106.70	95.55	100.13
LSTM	LM	Java	24.24	28.83	30.14
LSTM	LM	WikiText-103	62.71	46.71	48.57
LSTM	MLM	Java	15.09	17.56	18.71
LSTM	MLM	WikiText-103	22.63	16.55	**17.96**
CNN	MLM	Java	30.65	36.36	39.56
CNN	MLM	WikiText-103	35.68	24.55	26.16
Transformer	MLM	Java	13.77	14.15	**14.67**
Transformer	MLM	WikiText-103	34.91	18.88	23.01

Table 5: Results for the language modeling task. Lower perplexity is better. It can be seen that the Transformer trained as a masked language model (MLM) outperforms all other models and that the LSTM trained as a masked language model outperforms the LSTM trained as a standard language model (LM). Generally, the perplexity across the natural language data is lower than across programming language data – with the exception of the CNN MLM. The best result for the natural language data (obtained by the LSTM MLM) and the Java data (obtained by the Transformer MLM) is shown in bold.

The expected results are, that for both datasets, masked language models outperform standard language models, and that the masked language model Transformer outperforms all models. This is a common finding in the natural language processing literature, but has not been verified on programming languages. It is also expected that the test perplexity across the Java data is lower than WikiText-103, as Hindle et al. [109] state that "software is far more regular than English".

5.3 Results and Discussion

Table 5 shows the results for standard and masked language modeling tasks in terms of train, validation and test set perplexities. Figure 2 shows the perplexities obtained on the validation set during training for the standard language models. Figure 3 shows the perplexities obtained on the validation set during training for the masked language models.

First, looking at the results across the Java data, it can be seen that the masked language model Transformer provides the lowest test perplexity.

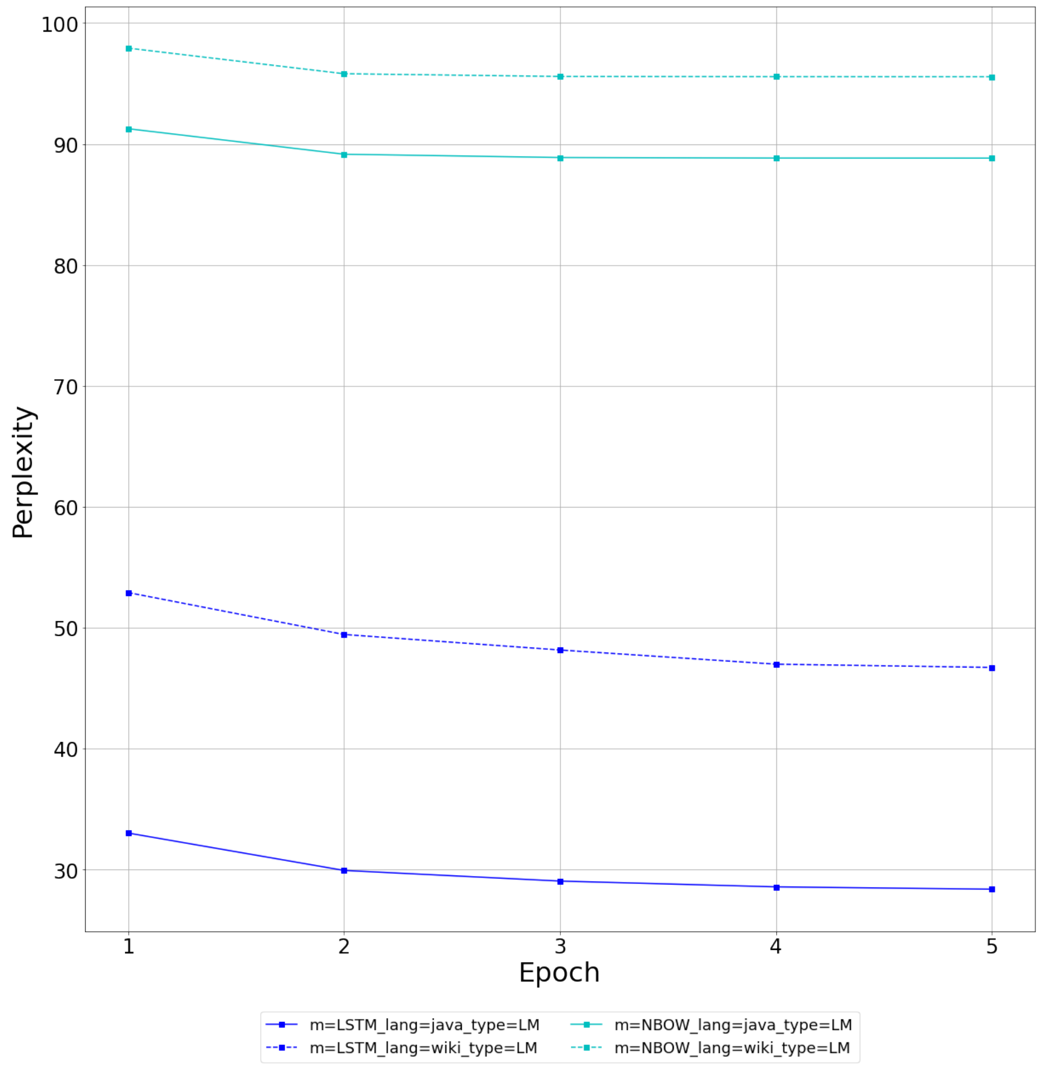

Figure 2: Language model results obtained on the validation set for the LSTM (blue) and NBOW (cyan) architectures on Java (solid lines) and WikiText103 (dashed lines) data. It can be seen that programming languages obtain a lower validation perplexity than natural languages.

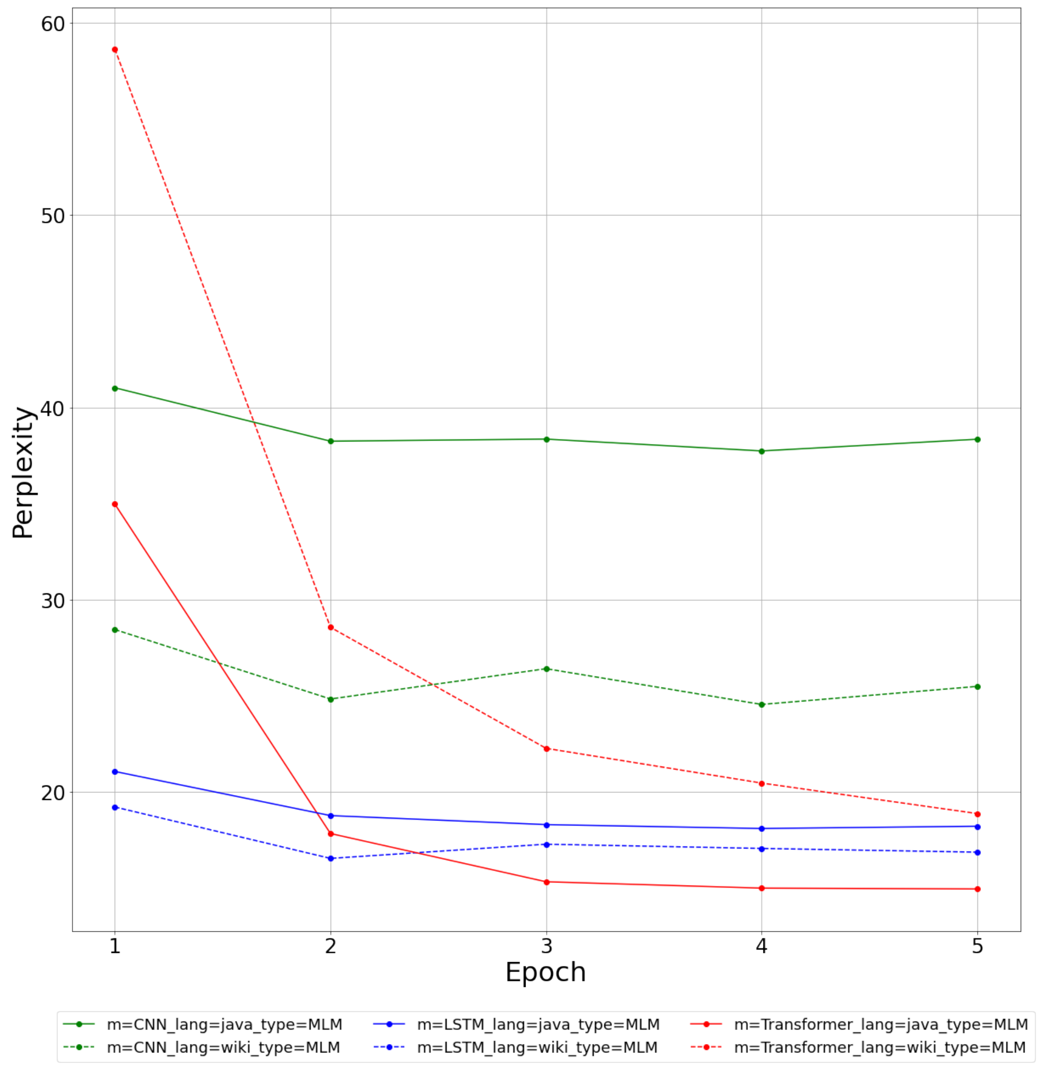

Figure 3: Masked language model results obtained on the validation set for the CNN (green), LSTM (blue), NBOW (cyan) and Transformer (red) architectures on Java (solid lines) and WikiText103 (dashed lines) data. Unlike standard language modeling, the perplexity obtained for programming languages is not always lower than that of natural languages.

However, it is the LSTM trained as a masked language model which provides the lowest test perplexity for the WikiText-103 data. It has been shown that a bi-directional LSTM can outperform a Transformer when both are only trained with a small amount of data [72, 68]. However, these findings – that LSTMs outperform Transformers on small datasets – are only on natural language. On the programming language data, which is even smaller than the natural language data, the Transformer outperforms the bi-directional LSTM.

One potential explanation for this novel finding – that Transformers outperform LSTMs on small, programming language datasets – is that the Transformer models are inherently superior for performing language modeling on source code. Within the Transformer architecture, all tokens are interconnected in the multi-headed attention layer, leading some researchers to find parallels between Transformers and graph neural networks [69, 127]. Graph neural networks are commonly applied to programming languages to make use of the internal representations of source code, e.g. abstract syntax trees, control-flow graphs, etc. Perhaps this internal graph representation leveraged by the Transformer allows it to outperform LSTMs, which view source code as a sequence of tokens, even with a small amount of data. The Transformer is able to do this for programming languages more so than natural languages, as in programming languages the tree structures are more explicit, rather than implicit.

Comparing the results of standard language models against masked language models by looking at the LSTM model specifically, it can be seen that the masked language model has significantly lower perplexity. This echoes the findings in the natural language processing literature, where masked language modeling is now commonly used over standard language modeling, and Transformers are used as they are easier to scale up with increased computation compared to LSTMs.

Looking at the test perplexities obtained for each of the language model trained on Java and WikiText-103 it can be seen that, generally, the language models perform equally or better on the programming language than they do on the natural language, even though there are 1.7 times more tokens in the natural language dataset than the programming language one. The NBoW and masked language model LSTM have comparable performance between Java and WikiText-103, whilst the Transformer and standard language model LSTM give significantly lower test perplexities for Java. The one exception to this is the CNN masked language model, where the test perplexity for the Java data is significantly higher than the WikiText-103 data.

CNN language models have had little attention, and the published work on them notes that they still underperform LSTMs [206]. One reason why they perform better on natural language than programming languages is due to their limited filter widths. In the hyperparameters for the experiments used in this thesis, the largest filter width in the CNN is 7. Hence, each `<mask>` prediction can only use information from three tokens either side. This limited context may be less of an issue on natural language than in programming languages. Hindle et al. [109] show that the cross entropy of n-gram language models plateaus at a lower *order* for natural language than programming languages, implying that programming languages require larger contexts but are easier to predict when given a suitably large context.

Shown below is an example output by the best performing model, the Transformer trained on Java code. The initial input is the code shown in blue with a `<mask>` token appended to the end. The model predicts masked token, which is then appended to the input and fed back into the Transformer. This is repeated until the model predicts an end-of-sequence, `<eos>`, token. The model predictions are shown in red, and code formatting was added manually.

```
public static String reverseString(String str) {
  StringBuilder sb = new StringBuilder();
  for (int i = str.length() - 1; i >= 0; i--) {
    sb.append(str.charAt(i));
  }
  return sb;
}
```

The generated code shows the model is able to understand the desired functionality from the method name alone, correctly construct an instance of the `StringBuilder` class and loop through the given string in reverse, appending each character to `sb`. However, there is a subtle bug in the code as `sb` should be converted to a string, using `sb.toString();` before being returned and hence the function returns a `StringBuilder` object and not a `String` object, as denoted by the method declaration. It also appears the model is not aware that the `StringBuilder` class has a built-in `.reverse` method as the above code can also be written in a more concise and less error prone way as:

```
public static String reverseString(String str) {
  String reversedStr = new StringBuilder(str).reverse().toString();
  return reversedStr;
}
```

Using another example, where the model must generate code to convert a given string to uppercase:

```
public static String upperCaseString(String str) {
  return str.toUpperCase();
}
```

Here, the model is able to output the optimal solution by using the `toUpperCase` method from the `String` class.

The above examples highlight how language models for code are able to mostly generate correct code, especially when the desired functionality is relatively simple, but sometimes introduce subtle bugs or miss methods which would provide a simpler solution.

5.4 Applications

As language models can be used to generate sequences the most obvious choice of application when applied to source code is code autocompletion [41, 33, 153, 186, 255, 117]. With a masked language model, the input is the original sequence with a `<mask>` token appended to the end. Most language models are relatively large in terms of memory and computational requirements, hence take a non-negligible time to make a next token prediction [254].

One method of applying language models in order to assist in providing feedback is by feeding an input sequence to a language model, having it predict each token in the sequence, and measure the average loss across the whole sequence. The hypothesis is that a well-trained language model is able to predict the tokens within correct code more accurately than in code containing bugs, thus the correct code has a lower loss value. This has been shown to be true for n-gram language models [219, 279].

To test this hypothesis on the language models trained for this thesis, the best performing language model, the masked language model Transformer, is used to measure the average loss across two statements, one containing a bug and one that does not. The "accuracy" is the number of times the statement containing the bug received the higher loss. The dataset used for this experiment is the test set from Hata et al. [98][18], which contains 568 pairs of buggy and corrected Java statements from actual open-source repositories. Note that the weights of the language model are not updated at all, this can be thought of as a *zero-shot* classification problem. Zero-shot implies that a model is performing a task which it has not been explicitly trained to do.

Figure 4 shows a histogram of the average per token losses obtained for the correct and buggy statements. It can be seen that there is significant overlap between the two and this is reflected in the classification accuracy, a mere 57%. However, by looking at the data, it is not surprising to see why. An example buggy and correct pair is `expect(arg).andReturn(arg);` and `expect(arg).andReturn(arg).atLeastOnce(arg);`. Without any wider context, it is difficult to tell that it is because of the lack of `.atLeastOnce(arg)` that the first statement is not correct.

To test if this approach is a viable proof-of-concept at all, the experiment is repeated by taking the correct code and synthetically adding bugs by randomly changing one of the tokens to another within the vocabulary[19]. Figure 5 shows the histogram for this experiment. In this scenario, the accuracy jumps to 93%, implying that the task might now be too easy, however it does show the ability for language models to detect "unnatural" code.

[18] Available at https://github.com/hideakihata/NMTbasedCorrectivePatchGenerationDataset

[19] This does not take the "type" of the token into account, i.e. it may switch a variable with a semicolon causing the example to have a syntax error, reducing the difficulty of the task by making the error significantly easier to locate as all the examples should be syntactically correct.

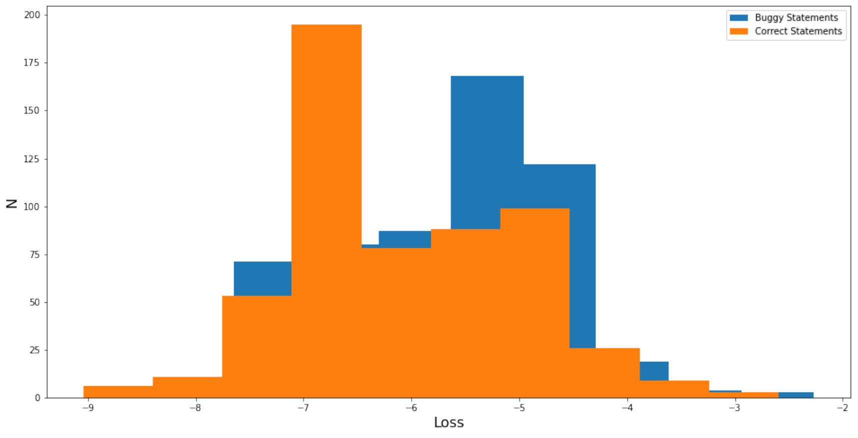

Figure 4: Histogram of average per token losses for the masked language model Transformer across correct (orange) and statements containing actual bugs (blue).

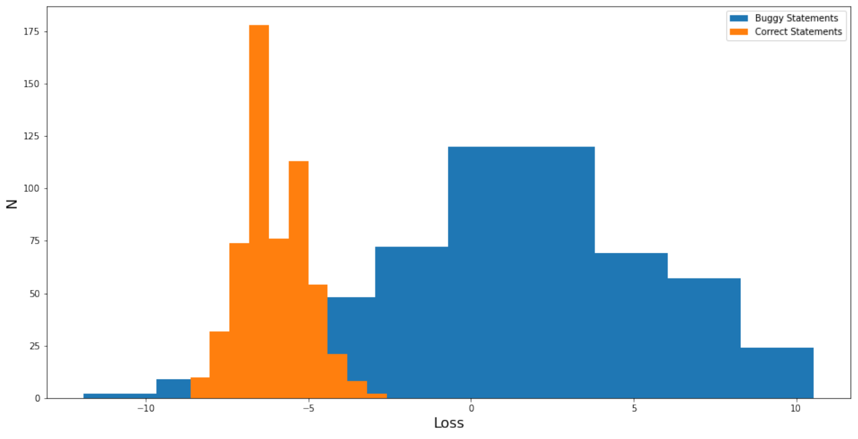

Figure 5: Histogram of average per token losses for the masked language model Transformer across correct (orange) and statements containing synthetic bugs (blue).

A hypothetical use-case for a model that can detect how idiomatic a function is would be to prioritize provision of feedback to students that produce the least idiomatic code. This code would be least "natural" and would usually imply that it is written in a non-idiomatic style that the student should aim to correct in the future.

5.5 Conclusion

This section has detailed how language models, both standard auto-regressive language models and the recently discovered masked language models, can be applied to code. It has shown how masked language models outperform standard language models for both natural and programming languages, and that Transformers outperform NBoW, CNN and LSTM architectures when used as language models on source code, which contradicts some research stating that Transformers perform poorly on small datasets. It has also shown the results of Hindle et al. [109], that programming languages are more predictable than natural languages, to hold true. Though, the gap between the two is smaller than it was for the n-gram language models used by Hindle et al., and the lesser studied CNN language model actually finds natural languages easier to predict.

Whilst discussing how these language models can be used to provide feedback, it has been shown that they are not effective in detecting subtle bugs within single statements, however are able to easily detect synthetic bugs which make statements very "unnatural". A hypothetical use-case has been suggested, where student submissions containing the most unnatural code should be prioritized, as these are more likely to be written in a non-idiomatic style.

6 Semantic Code Search

6.1 Introduction

One method that can be used to boost the productivity of software developers is the use of *semantic code search*, also known as *semantic code retrieval* or *neural code search* when using neural networks (see Chapter 2, Section 2.7). Instead of a developer writing methods from scratch, they should be able to search for existing methods which provide the same functionality. This is most commonly seen when developers search a natural language query, such as "how to reverse a string in Java", using a search engine and receiving search results which contain code snippets on how to perform the desired task – usually from the popular question-and-answer site, StackOverflow[20].

The key challenge is understanding the *semantics* of both the source code and the natural language query. To perform semantic code search effectively, the methods must be able to understand that "reverse" in natural language is related to the `.reverse()` method in Java. Simple queries like this are aided by the fact that, as Hindle et al. [109] pointed out, programs and programming languages are designed to be written and read by other humans, so in theory the method to reverse a string could be called anything. However, more complex semantic understanding of code is often required for some simple natural language queries. For example, determining if a number is odd or even requires performing modulo two and then checking if the result is equal to zero.

Fortunately, advances in machine learning, specifically using deep neural networks, have led to progress in *natural language understanding* [260] through the task of *natural language inference* [38] – determining if two natural language sentences have the same semantic meaning. A deep neural network is used to build a representation of each sentence, and then a *similarity metric* is used to measure how "similar" the sentences are. As this has been shown to successfully measure the similarity between natural language sentences where the two sentences are in different languages [58], e.g. English and French, it has also been applied between natural languages and programming languages [46, 274, 116, 291, 280] in the context of semantic code search.

[20]For example: https://stackoverflow.com/questions/7569335/reverse-a-string-in-java

The most common method of applying natural language inference to semantic code search is to train two neural networks. The first encodes the natural language sequence, typically called a *query*, into a single *query embedding* tensor. The second encodes the programming language snippet into a single *code embedding* tensor. A similarity metric is then used to measure the *similarity* between the two embeddings, where relevant query and code embeddings should have a high similarity and unmatched query and code embeddings should have a low similarity. In other words, the code and query encoders should map the two matching sequences nearby in high-dimensional space.

$$\begin{aligned}\boldsymbol{Z}_c &= \text{CodeEncoder}(\boldsymbol{X}_c)\\ \boldsymbol{Z}_q &= \text{QueryEncoder}(\boldsymbol{X}_q)\\ s &= \text{Similarity}(\boldsymbol{Z}_c, \boldsymbol{Z}_q)\end{aligned}$$

$\boldsymbol{Z}_c \in \mathbb{R}^{N\times F}$ is the code embedding obtained from the code tokens $\boldsymbol{X}_c$, $\boldsymbol{Z}_q \in \mathbb{R}^{M\times F}$ is the query embedding obtained from the $\boldsymbol{X}_q$ query tokens, and $s \in \mathbb{R}$ is the similarity.

One of the main issues with machine learning models is that they require large datasets in order to successfully train them. For natural language inference tasks on natural languages this data has to be collected manually, limiting the size of the datasets used. However, collecting natural language query and programming language snippet pairs can be done automatically, such as mining questions and answers from StackOverflow [292, 295]. The most common method of automatically mining natural language and code pairs is by leveraging the vast amount of code available from open-source repositories, which have their methods and corresponding *doc-strings* – the natural language comments written within the method, usually used by automatic documentation tools – scraped to be used for constructing large datasets.

Transfer learning has become incredibly popular in natural language processing using deep neural networks. A model is first trained on a task, T_A, and then the weights of this model are fine-tuned on the desired downstream task, T_B. Compared to using randomly initialized weights for T_B, fine-tuning the weights from T_A often leads to faster convergence and improved performance. In natural language processing the most common pre-training task,

T_A, is language modeling due to its ability to learn from unlabeled data.

Transfer learning for machine learning on source code has received little attention, which is ironic as the large amount of code available in open-source repositories makes gathering data for language modeling relatively easy compared to gathering data for natural language. To test the effectiveness of transfer learning on source code, several experiments are performed by fine-tuning a pre-trained language model. Each of the four models are trained on semantic code search, in Java, using: randomly initialized weights, transfer learning from a language model trained on Java, transfer learning from a language model trained on six different programming languages, and transfer learning from a language model trained on natural language data.

Another technique for natural language processing that has yet to receive attention in machine learning for source code is data augmentation [223]. Data augmentation artificially increases the size of the dataset used to train a model. It is commonly used in computer vision, where images are flipped, cropped, skewed and color shifted. In natural language processing, words are swapped for a synonym or their position is swapped around in a sentence.

For machine learning on source code, either swapping a single variable with another variable or randomly swapping two tokens around will more than likely cause the code to become incorrect, by either causing it to be uncompilable or evaluate to a different result. However, how much does "correctness" matter for the semantic code retrieval task?

Consider the query "how to reverse a string" and the relevant snippet `reversedString = myString.reverse()`. What if all "punctuation", tokens not containing alphabetic characters, were removed? The code would then be `reversedString myString reverse`. Obviously this code would no longer compile, however it would still contain all the relevant tokens to be able to correctly "pair" it with the natural language query. What if instead of removing the punctuation, the tokens were randomly shuffled? An example result of the shuffling would be `= reversedString () reverse . myString .` Again, this would not compile, but all the information is still within the code snippet – just not in the right order. In this thesis, semantic code search is also performed using these two augmentations, dubbed *strip punctuation* and *shuffle tokens*. The performance of the models using these augmentations indicate how much information the models are using about the punctuation or the order of the code tokens.

The experiments carried out in this chapter explore semantic code search, using the CodeSearchNet dataset and the four models discussed in Chapter 3. The experiments aim to answer four research questions: which model performs the best on semantic code search? Does transfer learning provide a noticeable performance improvement? How should the models be pre-trained to for optimal transfer learning performance? What effect do the proposed data augmentation techniques, *strip punctuation* and *shuffle tokens*, have on the performance of the models?

6.2 Methods

The experiments focus on the Java data within the CodeSearchNet dataset. Each Java method has a corresponding doc-string. The doc-strings are used as the query, and the method is used as the code. Both the code and query are tokenized with the same tokenizer, detailed in Chapter 4, Section 4.1. Each of the four models are used as both the code and query encoder, i.e. performance reported for the Transformer model is where a Transformer is used as both the code and query encoder. The two encoders have their own distinct set of parameters, but share the same model architecture and hyperparameters.

The code and query embeddings are obtained directly from the output of each model, i.e. $\boldsymbol{Z}_c \in \mathbb{R}^{N \times F}$ and $\boldsymbol{Z}_q \in \mathbb{R}^{M \times F}$ are the sequence of features and no task-specific head is used. Similarity is measured by averaging the features across the sequence length for both the code and query embeddings, and then calculating the inner product between the two averaged feature vectors.

$$\boldsymbol{c}_c = \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{z}_c^{(i)}$$
$$\boldsymbol{c}_q = \frac{1}{M} \sum_{i=1}^{M} \boldsymbol{z}_q^{(i)}$$
$$s = \boldsymbol{c}_c \boldsymbol{c}_q^T$$

$\boldsymbol{z} \in \mathbb{R}^F$ is the embeddings of each individual token, $\boldsymbol{c} \in \mathbb{R}^F$ is the averaged embeddings across the sequence length, and $s \in \mathbb{R}$ is the similarity.

When training the models, a batch of code and query pairs are input into the model at the same time. The loss used to update the model parameters is calculated by maximizing the similarity of the actual code-query pairs whilst minimizing the similarity between code i and query j, where $i \neq j$.

$$\mathcal{L}_i(\boldsymbol{s}) = -\log \operatorname{softmax}(\boldsymbol{s}_i)$$

$\boldsymbol{s} \in \mathbb{R}^B$ is the similarity between an embedded code and all embedded queries within a batch of size B, and $\mathcal{L}_i$ is the loss for element i in the batch, calculated using the negative log-likelihood of the similarity between code i and query i. The loss is averaged across all elements in the batch.

Performance on semantic code search is measured using *mean reciprocal rank* (MRR). First, $\boldsymbol{s}_i$ is calculated between a code and all queries within the batch. Then the similarity values are then sorted in descending order, and the position of the actual matching query is calculated. This position is called the *rank* and is used to calculate the *reciprocal rank*, $\frac{1}{\text{rank}}$. The average reciprocal rank across the entire batch is the mean reciprocal rank:

$$\text{MRR} = \frac{1}{B} \sum_{b=1}^{B} \frac{1}{\text{rank}_b}$$

MRR takes a value between 1 and 0, where higher values are better. A MRR value of 1.0 implies the correct query had the highest similarity value for each code example within the batch. A MRR value of 0.5 implies that, on average, the correct query had the second-highest similarity value for each code example within the batch.

To perform experiments with transfer learning, the code and query encoder are both initialized with the parameters obtained via the language modeling task detailed in Chapter 5. Each model is trained with four different initialization settings: the parameters initialized randomly, the parameters initialized using a language model trained only on the Java data from the CodeSearchNet dataset, the parameters initialized using a language model trained on the WikiText-103 dataset, and a language model trained on all six languages in the CodeSearchNet dataset. For the LSTM model, only the masked language model is fine-tuned, not the standard language model. This is because semantic code search can take advantage of the bi-directionality of the LSTM (the standard language model was uni-directional), and – as seen in Chapter 5, Section 5.3 – that the masked language model outperformed the standard language model (research has shown that lower language modeling perplexity positively correlated with downstream task performance [190, 3]).

It is slightly unconventional to fine-tune a language model on the exact same data as the downstream task, as done when fine-tuning the language model trained only on the CodeSearchNet dataset. However, the belief is that this experiment shows less about transferring generalized knowledge between two distinct datasets, but instead how to obtain a good initial set of parameters for a downstream task.

Researchers have had success in transferring knowledge between distinct natural languages [304, 157] but transferring between natural languages and programming languages is a novel concept. It should be possible to successfully transfer knowledge between a natural language and a programming language as code is written and designed to be read by humans, and thus function and variable names should be self-descriptive, e.g. a method called `reverseString` (which will be tokenized to `reverse` and `string`) will most probably reverse a string. If the query also contains the tokens `reverse` and `string` then it is expected that this query will have a high similarity to the `reverseString` method code. Thus, performance on semantic code search depends on the similarity of the features extracted between the code and queries, and well-performing features can also be learned from natural language data where the tokens carry the same semantic meaning as the programming language data. This is not always the case, for example the word "string" in natural language usually has a different meaning to that in a programming language.

The ability to transfer knowledge from non-Java programming languages to Java shows how general programming languages are when analyzed with machine learning. Consider two functionally identical methods in two different programming languages, both with identical or similar method and variable names. A query that has a high similarity to one of the methods should also have a high similarity to the other method. Thus, learning similarity between methods in one programming language should assist in determining the similarity between methods in another programming language.

Language	**Average Tokens**	**Reduction**
PHP	74	2.5
Java	85	2.0
Python	144	1.9
Go	74	2.1
JavaScript	120	2.3
Ruby	62	2.2
All	95	2.2

Table 6: CodeSearchNet dataset statistics with BPE tokenization on the raw code string after applying the *strip punctuation* data augmentation, as well as the reduction in the number of average tokens compared to non-augmented methods. It can be seen that stripping punctuation approximately reduces the number of tokens by half.

If the above claim holds true, that it is only the existence of method and variable names which provide information for semantic code search, then the strip punctuation data augmentation should have negligible impact on the performance of models. Stripping punctuation is performed by first performing tokenization and then removing all tokens which do not contain any alphabetic characters, e.g. semicolons, parentheses and braces. This is similar to removing *stop words* – the most common words which usually contain low semantic information – in natural languages, used to reduce sequence lengths and thus speed up computation. Removing stop words is less common in modern machine learning NLP pipelines as the models themselves are capable of learning which tokens can be ignored and when, instead of these tokens being explicitly removed. This data augmentation technique reduces the average number of tokens from 174 to 84 for the Java data. Table 6 shows the reduction in tokens for all six languages, and it can be seen that this augmentation cuts the amount of code tokens per example in half. The strip punctuation augmentation is applied to the code tokens for the training, validation and test data, and never to the query tokens.

If only the methods and variables provide information for semantic code search, then does the order in which they appear matter? This is examined using the shuffle tokens augmentation by randomly shuffling the code tokens whilst training. This causes our models to treat the code tokens as a bag-of-words, i.e. order is irrelevant, which should cause a negligible impact on the performance of models if they are simply looking for matching tokens appearing in the code and query, but a larger decrease in performance if the models are learning semantic representations by taking the order of tokens into account. Data augmentation that involves shuffling rarely appears in natural language processing – outside of learning to summarize via training a model to output a correctly ordered sentence after receiving a shuffled sentence [162, 217]. Shuffling tokens is only applied to the code tokens only, not the query tokens, and the validation and test data are not shuffled.

The expected results are that when the models are randomly initialized the LSTM model should outperform the Transformer model as it is usually the case that Transformers require larger datasets or need to be initialized from pre-trained models; performing transfer learning should improve the overall performance of every model, just as it does on natural languages, even when transferring from natural language to programming language data, and that a fine-tuned Transformer should outperform a fine-tuned LSTM; pre-training on just Java should outperform pre-training on WikiText-103 due to the domain-shift required when transferring from a natural language to a programming language; pre-training on all six languages in the CodeSearchNet dataset should provide a larger performance improvement than just pre-training on the Java data as the domain-shift from one programming language to another is smaller than natural language to a programming language, and this domain-shift is offset by the 4 times increase in the number of examples; the strip punctuation data augmentation technique should provide a small decrease in performance as the punctuation tokens contain little information; and the shuffle tokens data augmentation technique should provide a larger decrease in performance as treating the code tokens as a bag-of-words is detrimental to learning good semantic representations.

6.3 Results and Discussion

Table 7 shows the results obtained on the semantic code search experiments. Figure 6 shows the MRR values obtained on the validation set during training when using different initialization techniques, i.e. transfer learning. Figures 7, 8, 9 and 10 compare the results of the data augmentation techniques when using different initialization methods.

Model	**Initialization**	**MRR**	**ΔSP**	**ΔST**
NBoW	Random	0.2179	0.0217	0.0852
NBoW	Java	0.2808	0.0615	0.0626
NBoW	Wiki	0.3423	-0.0311	-0.0032
NBoW	All	0.3093	-0.0059	0.0519
LSTM	Random	**0.5754**	-0.0224	-0.0709
LSTM	Java	0.6057	-0.0068	-0.0862
LSTM	Wiki	**0.6010**	-0.0242	-0.1044
LSTM	All	0.6396	-0.0045	-0.0905
CNN	Random	0.4433	-0.0291	-0.0551
CNN	Java	0.4514	-0.0231	-0.0830
CNN	Wiki	0.4705	-0.0445	-0.0833
CNN	All	0.4528	-0.0878	-0.0658
Transformer	Random	0.5327	0.0089	-0.1990
Transformer	Java	**0.6505**	-0.0143	-0.0961
Transformer	Wiki	0.5869	-0.0010	-0.0764
Transformer	All	**0.6875**	-0.0047	-0.0739

Table 7: MRR values obtained on the test set for semantic code search using different initialization techniques and the increase (green) or decrease (red) in the obtained MRR when using strip punctuation (**ΔSP**) or shuffle tokens (**ΔST**) data augmentation techniques. *Random* implies the weights are randomly initialized, *Java* initialization uses a language model pre-trained on Java code, *Wiki* initializes the weights from a language model pre-trained on the WikiText-103 dataset, and *All* uses a language model pre-trained on all six languages in the CodeSearchNet dataset. It can be seen that the *All* initialization generally performs best, the strip punctuation data augmentation reduces performance by a negligible amount, and the shuffle tokens data augmentation reduces performance more than the strip punctuation data augmentation. The best result for each initialization technique is shown in bold.

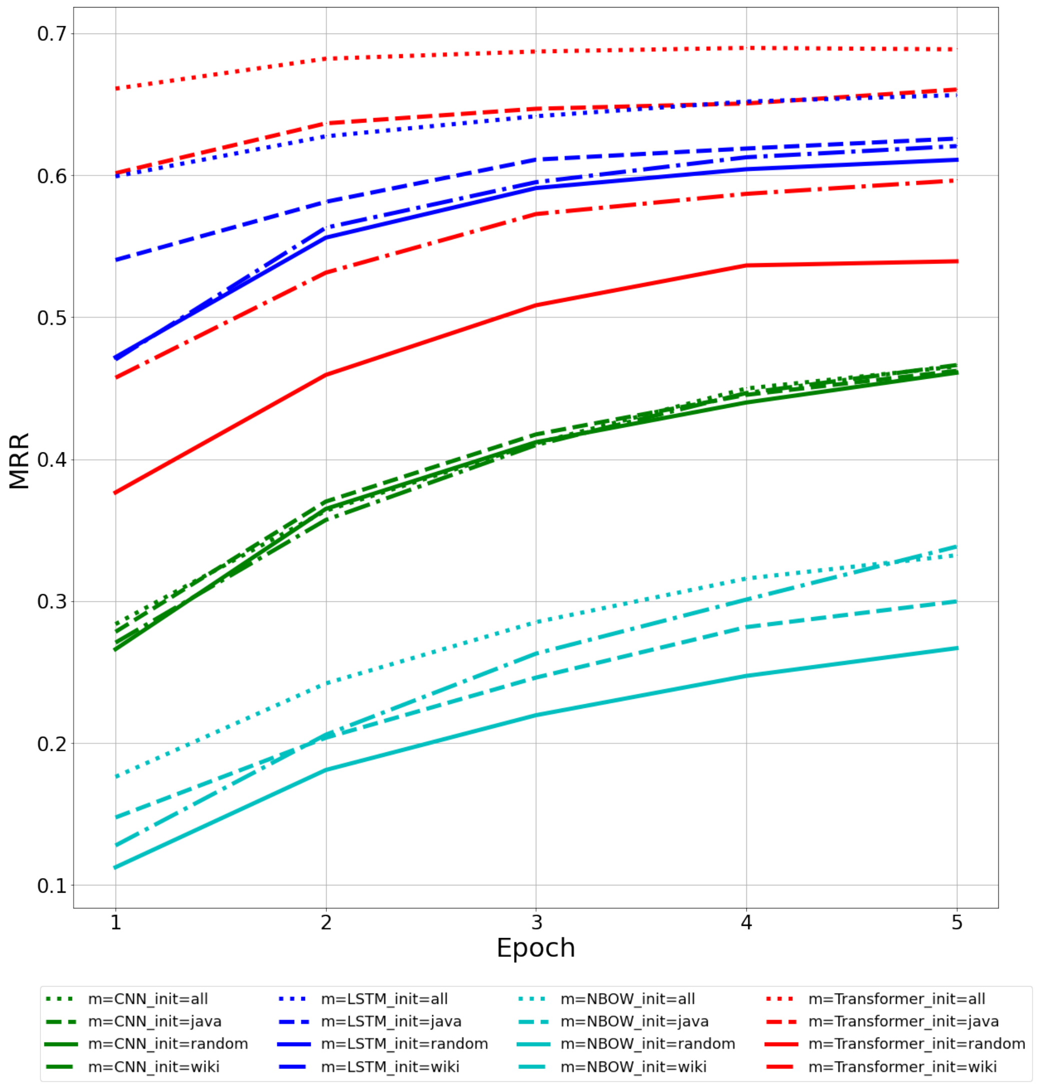

Figure 6: MRR values obtained on the validation set for semantic code search across all four models – CNN (green), LSTM (blue), NBoW (cyan), and Transformer (red) – when initialized from: random weights (solid line), a language model pre-trained on Java (dashed line), a language model pre-trained on all six languages in the CodeSearchNet dataset (dotted line), and a language model pre-trained on WikiText-103 (dot dashed line).

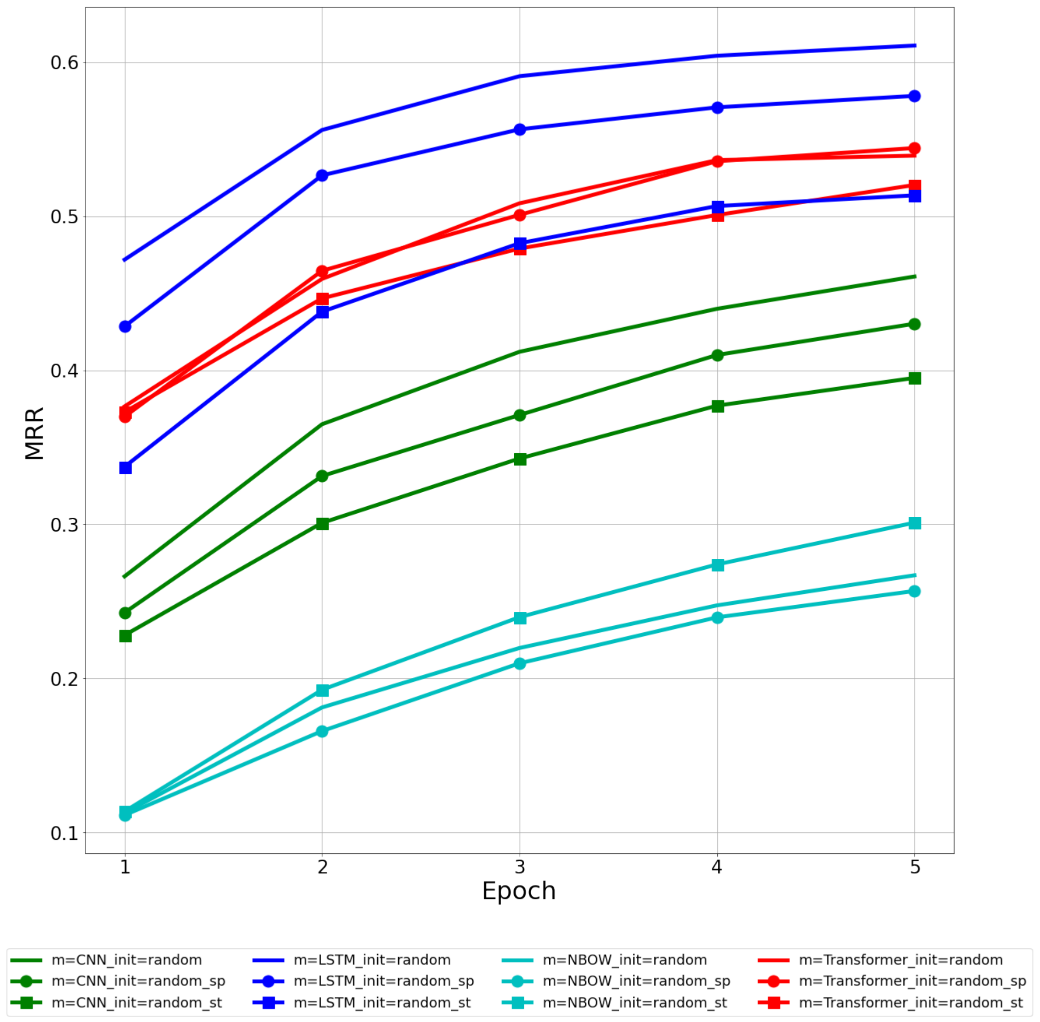

Figure 7: MRR values obtained on the validation set for semantic code search across all four models – CNN (green), LSTM (blue), NBoW (cyan), and Transformer (red) – initialized from random weights when using no augmentation (straight line), the strip punctuation (circle markers) and shuffle tokens (square markers) data augmentation.

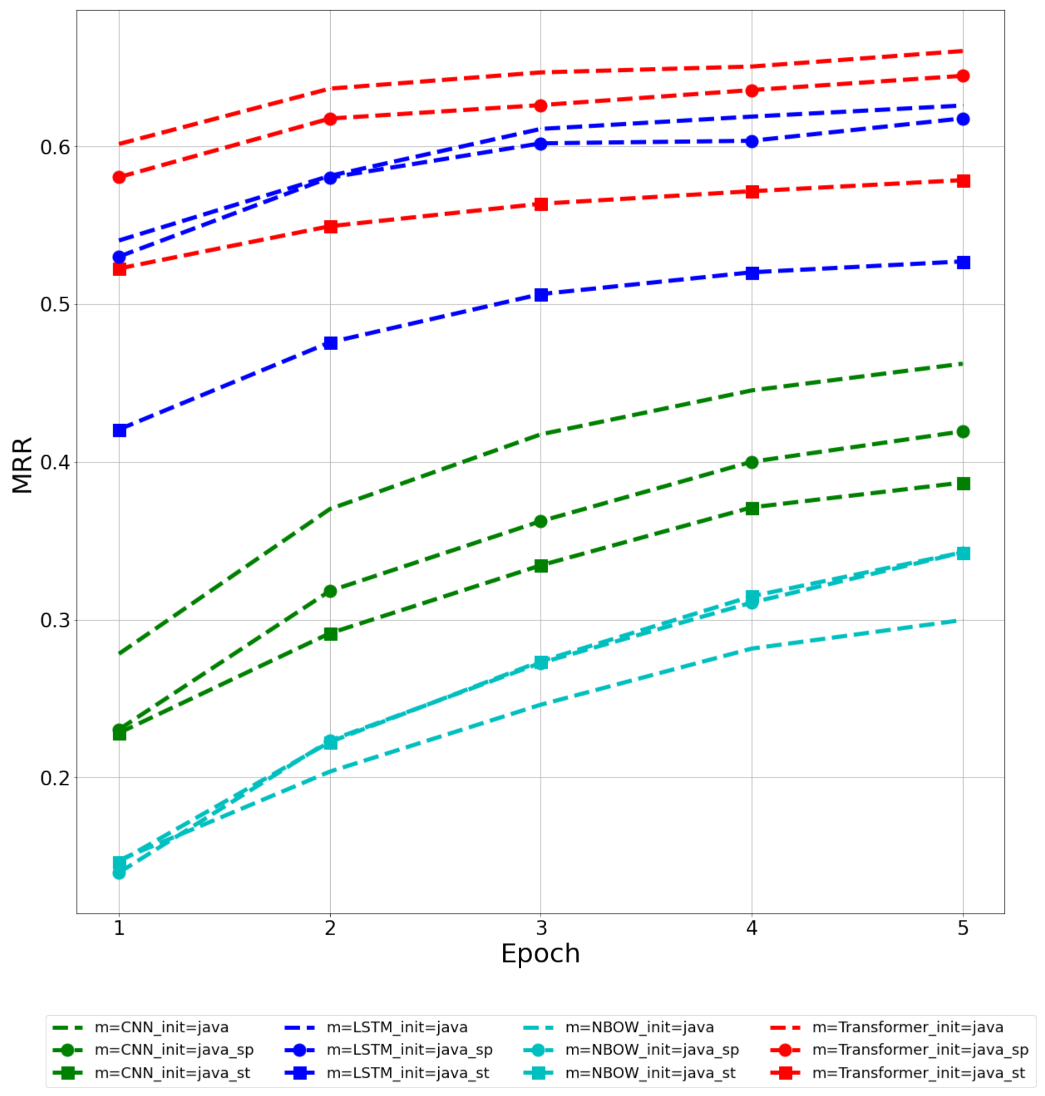

Figure 8: MRR values obtained on the validation set for semantic code search across all four models – CNN (green), LSTM (blue), NBoW (cyan), and Transformer (red) – initialized from a language model pre-trained on Java when using no augmentation (straight line), the strip punctuation (circle markers) and shuffle tokens data augmentation (square markers).

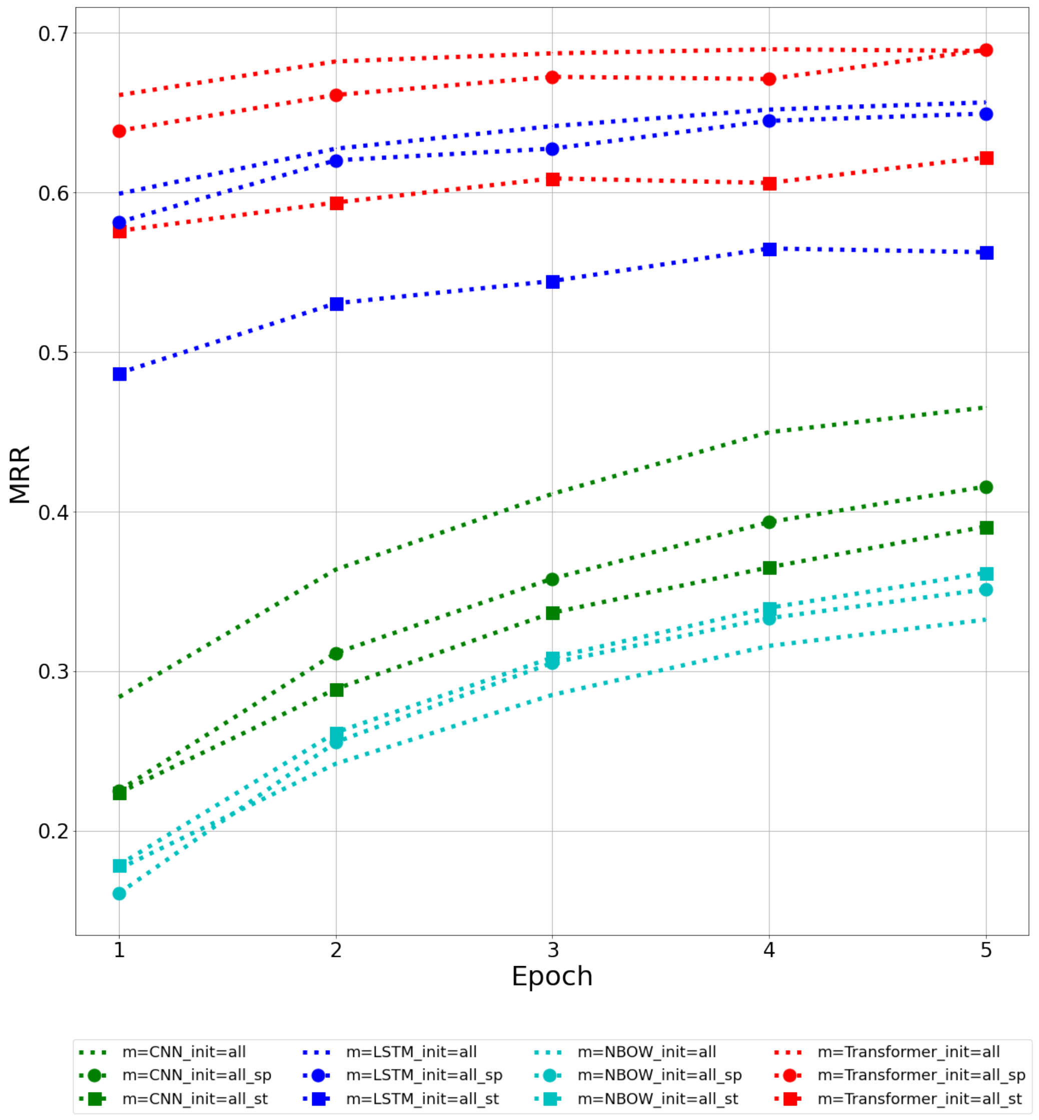

Figure 9: MRR values obtained on the validation set for semantic code search across all four models – CNN (green), LSTM (blue), NBoW (cyan), and Transformer (red) – initialized from a language model pre-trained on WikiText-103 when using no augmentation (straight line), the strip punctuation (circle markers) and shuffle tokens (square markers) data augmentation.

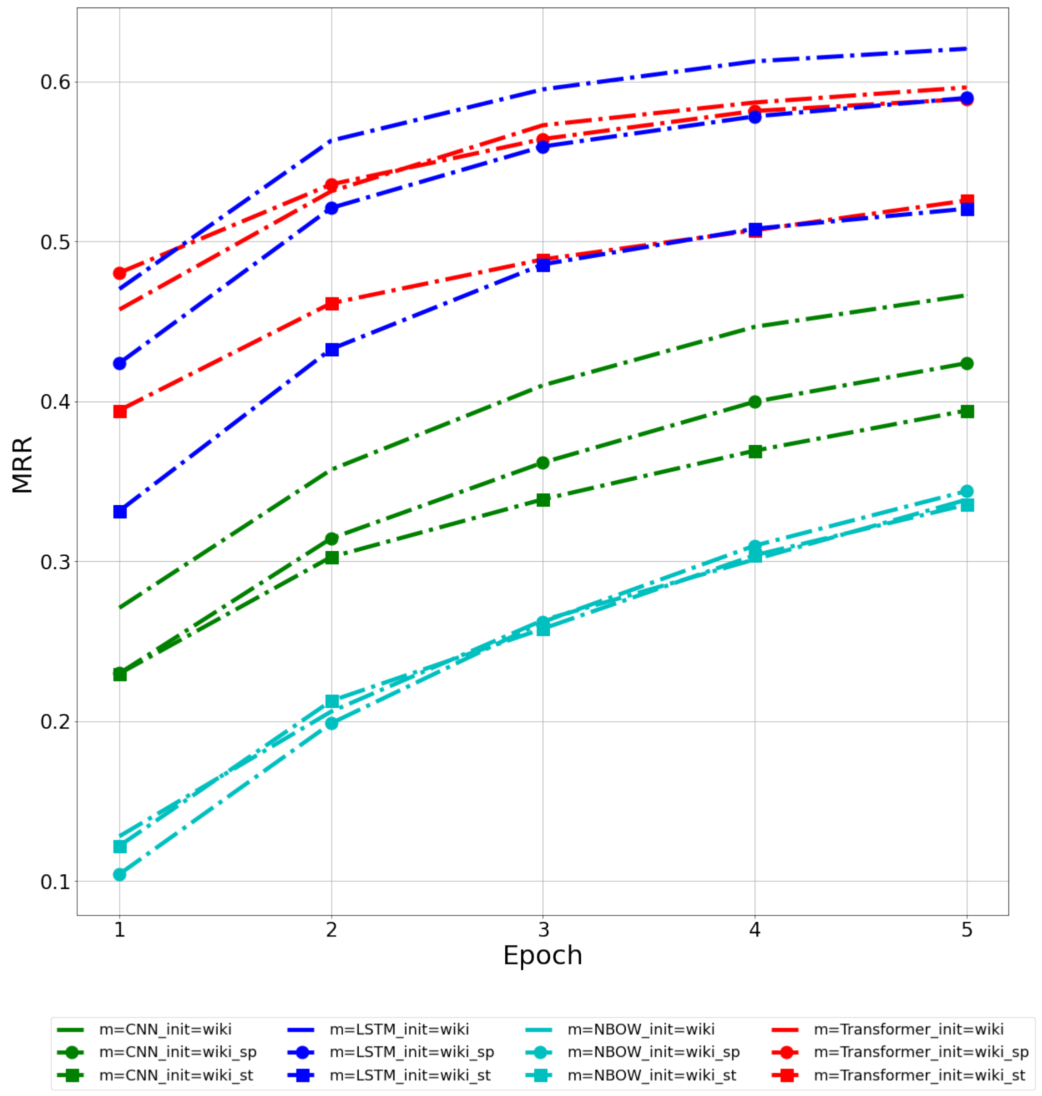

Figure 10: MRR values obtained on the validation set for semantic code search across all four models – CNN (green), LSTM (blue), NBoW (cyan), and Transformer (red) – initialized from a language model pre-trained on all six languages in the CodeSearchNet dataset when using no augmentation (straight line), the strip punctuation (circle markers) and shuffle tokens (square markers) data augmentation.

From Table 7 and Figure 6 it can be seen that when no transfer learning is performed – the models are randomly initialized – then the LSTM model outperforms the Transformer model. This echoes the findings of Ezen-Can [71], where LSTMs outperform Transformers when trained from scratch on relatively small datasets. As with the language modeling task, both the LSTM and Transformer outperform the CNN model, which outperforms the NBoW model.

Table 7 and Figure 6 also show that all forms of transfer learning used in these experiments, even transferring from a natural language to a programming language, showed performance improvements. The Transformer model received the largest performance improvements when transfer learning was used, whereas the CNN model improved the least. This reflects the empirical research results, that pre-trained Transformers such as BERT [66] are able to achieve state-of-the-art results in many natural language processing tasks when fine-tuned. Transfer learning for CNNs applied to natural languages is not a commonly researched area, though Semwal et al. [239] have shown that it requires techniques such as carefully tuning the dropout ratio and only transferring parts of the weights – which is not done in these experiments. The results also show that when performing transfer learning, the Transformer outperforms the LSTM – except when initialized with weights learned from the WikiText-103 dataset.

Comparing the performance improvement gained from pre-training on WikiText-103 to pre-training on the Java data in the CodeSearchNet dataset it can be seen that for the NBoW and CNN models, pre-training on WikiText-103 offers greater performance, significantly so for the NBoW model; for the LSTM model, both the WikiText-103 and the CodeSearchNet Java data provide similar performance; and for the Transformer the Java data offers significantly greater performance. In fact, for the NBoW and CNN models, transferring from WikiText-103 provides the best overall performance. The explanation that transferring from Java is superior to natural language due to the latter requiring a larger domain-shift holds for the LSTM and Transformer models, but not for the NBoW and CNN models. One potential reason for this is that the NBoW and CNN models are unable to capture the domain-specific knowledge required for programming languages and instead benefit from the larger WikiText-103 dataset. However, even the best performing NBoW and CNN models underperform the randomly initialized LSTM and Transformer models.

Comparing the performance improvement gained from pre-training on the Java data in CodeSearchNet compared to the data from all six programming languages in CodeSearchNet it can be seen that transferring from a language model trained on all six languages always offers superior performance. This shows that there are features which can be learned from a programming language that generalize across different programming languages. Hence, researchers should be training large language models across a variety of programming languages to be used for transfer learning, and not restricting these language models to being trained on a single programming language.

If it is possible to generalize across programming languages, then why not use the non-Java data for fine-tuning the models instead of only using it to pre-train a language model? Taking that idea one step further, do our models need to be trained on any Java data at all to achieve acceptable performance? Table 8 shows results of models fine-tuned on either all six languages in the CodeSearchNet, or all five languages that are not Java. The MRR shown is across the Java test set. Experiments are run both when training from scratch – random initialization – or when using transfer learning, where the pre-training dataset is the same as the fine-tuning dataset. Hence, the results with a **Dataset** of "No Java" have not been trained on a single example in Java and are similar to a zero-shot learning set-up [286].

For the NBoW and CNN models, fine-tuning a language model trained on all six languages on a dataset also consisting of all six languages provides better performance than simply fine-tuning the model on the Java data. For the LSTM and Transformer models, these results are comparable to fine-tuning on just the Java data. However, as these models are trained on significantly more data, they take considerably longer to train. Hence, the multi-task learning set-up should not be used for this task. As expected, when no Java data is used, the performance decreases. However, when the NBoW and Transformer models are pre-trained and fine-tuned on the No Java data, they outperform training those models from scratch on just the Java data.

Generally, the strip punctuation data augmentation reduces the performance of a model by a negligible amount, and in some cases the performance slightly increases. The average performance change is a decrease of only 0.013 MRR, despite the fact that, on average, half of the tokens are removed from the code. This has two potential explanations: the code tokens which only contain

Model	**Initialization**	**Dataset**	**MRR**
NBoW	Random	All	0.2788
NBoW	All	All	0.3854
NBoW	Random	No Java	0.1876
NBoW	No Java	No Java	0.2391
LSTM	Random	All	**0.5853**
LSTM	All	All	0.6384
LSTM	Random	No Java	0.2393
LSTM	No Java	No Java	0.4153
CNN	Random	All	0.5082
CNN	All	All	0.5119
CNN	Random	No Java	0.3176
CNN	No Java	No Java	0.3439
Transformer	Random	All	0.5745
Transformer	All	All	**0.6612**
Transformer	Random	No Java	**0.4442**
Transformer	No Java	No Java	**0.5504**

Table 8: MRR values obtained on the test set for semantic code search using different initialization techniques and different datasets for fine-tuning. The test MRR shown is across the Java data only. *Random* initialization implies the weights are initialized randomly, *All* initialization uses a language model pre-trained on data from all six languages in the CodeSearchNet dataset, *No Java* initialization uses a language model pre-trained on the five languages in the CodeSearchNet dataset that are not Java. A **Dataset** of *All* refers to training on all six languages in the CodeSearchNet dataset, and *No Java* refers to using the five languages in the CodeSearchNet dataset that are not Java. The best result for each initialization-dataset combination is shown in bold.

"punctuation" carry very little information for performing semantic code search, and/or the models examined in this thesis have no mechanisms to take advantage of the existence of "punctuation" tokens. When viewing code as an abstract syntax tree the punctuation tokens are removed as they are implicit within the structure of the tree, hence it is the relationship between the named variables (represented as edges within the tree) that is used to build features. However, models that view code as a sequence of tokens are unable to implicitly learn the tree structure and simply ignore punctuation tokens within the code. This is also a potential reason why the Transformer model performs best on this task, as research has shown [69, 127] that Transformers are implicitly building graphs between tokens within their internal representations.

The shuffle tokens data augmentation also reduces the model's performance when applied. The average performance change is a decrease in 0.056 MRR, a 4.3 times larger decrease than the strip punctuation data augmentation. This means that the order of the code tokens within the dataset matters more than the presence of "punctuation" tokens. Thus, the models are not treating the code tokens as a bag-of-words, but are building features dependent on the semantic meaning of the code tokens captured using information on the order of which the tokens appear.

One way the model can be tricked is by using deceptive variable names. Calculating the similarity between the doc-string (shown in green below) and the two functions (shown in red and blue), the model (the best performing Transformer model from Table 7) provides a higher similarity to the code shown in red – which converts a string to uppercase but uses method and variable names incorrectly implying it has something to do with reversing a string – than to the code shown in blue – which reverses a string but uses method and variable names incorrectly implying it has something to do with reversing an array.

```
Reverse a string

public static String reverseString(String str) {
       String reversedString = str.toUpperCase();
       return reversedString
}

public static String reverseArray(String array) {
       String reversedArray = new StringBuilder(array).reverse().toString();
       return reversedArray;
}
```

However, when passed an example which does correctly reverse the string use appropriate variable names, such as:

```
public static String reverseString(String str) {
       String reversedStr = new StringBuilder(str).reverse().toString();
       return reversedStr;
}
```

The model then does allocate it a higher similarity than to the incorrect example which uses relevant variable names. This highlights how the model depends on appropriate variable names being used to match the given query.

6.4 Applications

Consider a well-trained semantic code search model that can: accurately provide a relevant method given a natural language snippet, and also provide a relevant natural language snippet when given a method. How can this be used to provide feedback for students in an educational setting?

One use-case is propagating feedback at scale. Consider a scenario where the number of students, and thus the number of student submissions to a programming assignment, is significantly higher than the number of graders. By using a semantic code search model, the average of the token features, $\boldsymbol{c}_c$, can be used to map each student submission into F-dimensional space where similar submissions are close together in this space. A clustering algorithm, such as *k-means clustering* can then be used to automatically locate k clusters of student submissions. The graders then only have to provide feedback to

k submissions, one from each cluster, and this feedback is propagated to all submissions within that cluster. To increase the granularity of feedback, i.e. to make it more specific instead of general, k can be increased. Thus, there is a trade-off between the granularity of feedback and the amount of grading to be done.

Another approach would be to show a student similar (close together in the F-dimensional space) submissions to their own. This would allow a student to see submissions that have received the same feedback and would either show an alternative approach to solve the same problem, or examples written by other students which contains the same mistakes they themselves made. By showing an alternative approach, or another example of their own mistakes, it is likely that the student can learn something from seeing these other submissions.

Finally, two submissions from different clusters and one piece of written feedback belonging to one of the clusters could be display to the student. They would then have to correctly determine which of the submissions the feedback belongs to. Alternatively, two pieces of feedback and one submission could be shown. This allows students to learn for themselves what "good" and "bad" code look like, so they can potentially avoid mistakes and patterns commonly seen in "bad" code.

The above approaches however would require training a semantic code search model on a dataset of code-feedback pairs. A dataset of this kind does not currently exist, however all advances made in the task of semantic code search would also apply to a dataset of code-feedback pairs. Thus, advances such as improved performance (obtained by using state-of-the-art models, such as Transformers) and increased sample efficiency (obtained by advances in transfer learning, pre-training, fine-tuning, etc.), like the ones detailed in this thesis, would also improve performance on semantic code search on a dataset of code-feedback pairs.

6.5 Conclusion

This section has detailed experiments on semantic code search. It has shown how, when randomly initialized – i.e. no transfer learning is performed, an LSTM model outperforms the Transformer, CNN and NBoW models. However, when transfer learning is used, the Transformer model outperforms the

LSTM, CNN and NBoW models. It has been shown that to achieve performance improvements when using transfer learning, the pre-trained model can be trained on either the same data used for the downstream task, data that is in a different domain (a natural language instead of a programming language), or a combination of multiple different languages within the same domain (i.e. a mixture of different programming languages). However, the best results are achieved using the mixture of different programming languages, which implies that there are features that the models can learn which generalize across programming languages. Experiments also show that models pre-trained and fine-tuned on no Java data at all manage to outperform models randomly initialized and trained directly on Java.

It has also been shown that removing all "punctuation" tokens (tokens which do not contain alphabetic characters) – corresponding to removing over half of the code tokens – has a negligible impact on performance. Finally, performing data augmentation where the code tokens are randomly shuffled has a detrimental impact on performance, implying that although the "punctuation" tokens do not provide much semantic information, the order in which the named tokens appear does provide useful and significant semantic information for the models used in these experiments.

It has also been discussed how semantic code search models can be applied to provide feedback – by propagating feedback to clustered similar student submissions – and also be applied in a pedagogical setting – by allowing the student to view similar submissions and having the students learn to construct the correct code-feedback pairs themselves.

7 Predicting Method and Variable Names

7.1 Introduction

Semantically meaningful method and variable names assist programmers in comprehending the behavior and intent of a given piece of code [35, 257, 155, 23, 48, 149, 148] (see Chapter 2, Section 2.4). In fact, poorly named methods and variables make code harder to understand and maintain [112, 24, 25], and lead to an increased likelihood in the number of bugs [1, 2, 44]. One method of reducing the burden of programmers to repeatedly come up with suitable names is to have a tool which will automatically generate method and variable names based on the surrounding context, i.e. code tokens and any natural language comments.

As with semantic code search (detailed in Chapter 6) the key challenge is understanding the *semantics* of the context. For example, given a method which contains the code snippet which accepts a string argument called `myString`, contains the line `myString = myString.reverse()` and then returns `myString`, what should the method be called? Ideally, something similar to `reverseString`. This requires understanding the relationship between the variable names and the methods called on them, i.e. a variable called `myString` is most probably a string, and it has the `reverse` method called on it. Consider a method which takes in an integer, `i`, and contains the line `return i % 2 == 0`, i.e. the method returns true if `i` is even, and false if `i` is odd, so an appropriate name is `isEven`. However, being able to correctly predict the method name `isEven` requires understanding the interaction between the modulo 2 operation and the comparison to 0.

In the literature it is common to predict method names only, and not the names of the variables [16, 20, 18, 10]. Allamanis et al. [10] refer to this as *extreme summarization*, as a method name can be thought of as a summarization of the method body. This has led to the use of summarization techniques used in natural languages to be applied to method name prediction in programming languages. There are two types of text summarization techniques: extractive summarization and abstractive summarization. Extractive summarization can only copy tokens directly from the input, it cannot create original tokens. Abstractive summarization produces the summary using tokens from a defined vocabulary. Among those, when used for predicting method names, abstractive summarization is used, as method names frequently use tokens which do not exist inside the method body. However, it is common to use a mechanism, such as a *pointer network* [271] which allows for copying tokens directly from the method body if required.

Abstractive summarization in natural language processing is commonly performed with an *encoder-decoder architecture* [252, 55], which consists of an *encoder* – which creates a sequence of features from the input sequence (the method body) – and a *decoder*, which takes the encoded sequence and then produces an output sequence (the method name), one token at a time.

$$\begin{aligned}\boldsymbol{Z} &= \text{Encoder}(\boldsymbol{X}) \\ \hat{\boldsymbol{Y}} &= \text{Decoder}(\boldsymbol{Z})\end{aligned}$$

$\boldsymbol{X} \in \mathbb{R}^N$ is the code tokens, $\boldsymbol{Z} \in \mathbb{R}^{N \times F}$ is the sequence out features output by the encoder, and $\hat{\boldsymbol{Y}} \in \mathbb{R}^{M \times V}$ is the sequence of predictions for the M method name tokens as a probability distribution over the vocabulary, V. This probability distribution can be sampled to obtain actual method tokens from the vocabulary.

The decoder can either output the entire method name in a single prediction, e.g. the model outputs $\boldsymbol{Y} = \texttt{reverseString}$, or it can output the method name by outputting a sequence of tokens that can be combined to obtained the predicted method name, e.g. the model outputs the sequence $\boldsymbol{Y} = [\texttt{reverse}, \texttt{String}]$. Research has shown that predicting the method as a sequence of tokens outperforms predicting the entire method name in a single prediction [18]. As the decoder's output is constrained by a fixed vocabulary size, V, a model that can only predicting entire method names in a single prediction can only predict those within its vocabulary, whereas a model predicting individual tokens can predict any method name that consists of tokens within the decoder vocabulary.

The encoder-decoder architecture can be applied to any problem where the input and output are both sequences, such as translation from one language to another. Hence, this can be applied to variable names as well as method names. This work focuses on predicting variable names over method names. Allamnanis et al. refer predicting variable names as the VarNaming task [9]. The only condition is that the variable name that the model is attempting to predict has to be replaced by a `<mask>` token, so the model knows the location of the variable name it is attempting to predict, but not what the actual variable is named.

As with all modern machine learning tasks, abstractive summarization requires large datasets. Unlike natural language summarization where the summary has to be hand-written, method and variable names can be automatically mined from open-source repositories. Every single named method and variable can be used as an example when performing summarization to programming languages. Hence, the task is more similar to the masked language modeling task, but with the `<mask>` tokens only replacing method and variable names and potentially being used to mask out several tokens if the tokenized method or variable consists of multiple tokens.

Transfer learning has also been successfully used in natural language summarization [301], but has received little attention when applied to machine learning for method and variable name prediction. This is despite the fact that obtaining a large amount of code from open-source repositories is significantly easier than obtaining a large amount of natural language text, and thus making it easier to train language models for programming languages. Masked language modeling would seem like an appropriate pre-training task, as predicting the single token replaced by a `<mask>` token is similar to predicting multiple tokens which were replaced by `<mask>` token.

As with semantic code search, effectiveness of transfer learning is tested for each model by performing variable name prediction using: randomly initialized weights, transfer learning from a language model trained on Java, transfer learning from a language model trained on six different programming languages, and transfer learning from a language model trained on natural language data.

Finally, the two data augmentation techniques, strip punctuation and shuffle tokens, explored in the experiments on semantic code search, are also applied to variable name prediction. This is done to see how much information contained in the punctuation or the order of the code tokens is used by the models for variable name prediction.

The experiments carried out in this chapter explore variable name prediction, using the four models discussed in Chapter 3 on the CodeSearchNet dataset. The experiments aim to answer the following questions: which model performs the best on variable name prediction? Does transfer learning provide a noticeable performance improvement? Does transfer learning provide more or less performance improvement than semantic code search? How should models be pre-trained to use for optimal transfer learning performance? What effect do the proposed data augmentation techniques have on the performance of the models? How do these data augmentation techniques effect the performance compared to semantic code search?

7.2 Methods

The experiments focus on the Java data within the CodeSearchNet dataset. Each Java method – including the method name, types, arguments and entire body – is used as an example. During each iteration the method is tokenized and the list of all named variables and the method name are obtained. These variables may consist of multiple tokens produced by the tokenizer – hence the task is more similar to span prediction [290, 217, 258] rather than masked language modeling [66]. A random named variable is then chosen to be predicted. All occurrences of this variable within the method are then replaced by a `<mask>` token. The masked variable is chosen at random for that example at every epoch. If the variable consists of multiple sub-tokens, as tokenized by the BPE tokenizer, then all all sub-tokens are replaced by a single `<mask>` token.

Using the encoder-decoder architecture, each of the four models are used as the encoder. They accept the masked code tokens as input and produce a sequence of features, $\boldsymbol{Z} \in \mathbb{R}^{N \times F}$. The decoder used by all four of the models is a gated recurrent unit (GRU) [55], as used by Allamanis et al. [10] (who use a single CNN-based encoder for predicting method names only, use a different dataset to this work, and do not experiment with pre-training, multi-task learning or data augmentation), and produces the target sequence, the variable tokens, one token per time-step.

GRUs are variants of recurrent neural networks, similar to LSTMs. They have fewer parameters than LSTMs as they only have two gates – an update and a reset gate – instead of three, and do not have a recurrent cell state.

$$
\begin{aligned}
\boldsymbol{u}^{(t)} &= \sigma(\boldsymbol{e}^{(t)}\boldsymbol{W}_u + \boldsymbol{h}^{(t-1)}\boldsymbol{U}_u + \boldsymbol{b}_u) \\
\boldsymbol{r}^{(t)} &= \sigma(\boldsymbol{e}^{(t)}\boldsymbol{W}_r + \boldsymbol{h}^{(t-1)}\boldsymbol{U}_r + \boldsymbol{b}_r) \\
\hat{\boldsymbol{h}}^{(t)} &= \tanh(\boldsymbol{e}^{(t)}\boldsymbol{W}_h + (\boldsymbol{r}^{(t)} \odot \boldsymbol{h}^{(t-1)})\boldsymbol{U}_h + \boldsymbol{b}_h) \\
\boldsymbol{h}^{(t)} &= (1 - \boldsymbol{u}^{(t)}) \odot \boldsymbol{h}^{(t-1)} + \boldsymbol{u}^{(t)} \odot \hat{\boldsymbol{h}}^{(t)} \\
&= \mathrm{GRU}(\boldsymbol{e}^{(t)}, \boldsymbol{h}^{(t-1)})
\end{aligned}
$$

The update gate, $\boldsymbol{u} \in \mathbb{R}^H$, controls how much information from the previous hidden state, $\boldsymbol{h}^{(t-1)} \in \mathbb{R}^H$, will be used in the new hidden state. If the update gate is a zero vector, then all information from the previous hidden state is retained. The reset gate, $\boldsymbol{r} \in \mathbb{R}^H$, controls how much information from the previous hidden state is discarded. If the reset gate is zero, then all information from the previous hidden state is discarded. $\boldsymbol{e} \in \mathbb{R}^D$ are the embedded tokens. $\boldsymbol{W} \in \mathbb{R}^{D \times H}$ and $\boldsymbol{U} \in \mathbb{R}^{H \times H}$ are the input and recurrent weight matrices, the bias terms are denoted by $\boldsymbol{b} \in \mathbb{R}^H$, $\odot$ is element-wise multiplication, σ is the sigmoid activation function, and tanh is the hyperbolic tangent function.

The initial hidden state of the decoder is usually referred to as the *context vector* that is assumed to contain the information within the input sequence, obtained from the encoder. When the encoder takes the order of the input sequence into account, the context vector is usually the final encoder hidden state, else the context vector is the mean of the hidden states returned by the encoder. To make a fair comparison between the models used in this thesis, the experiments used here all provide a context vector, which is the average of all the hidden states.

An issue faced by traditional encoder-decoder models is their attempt to compress all the information contained in the input within a single context vector, which usually leads to a loss in information. This issue is amplified with longer input sequences [54, 209].

The most common solution to the information compression problem introduced by the context vector is the use of *attention* [26]. When using attention, the decoder is able to access the entire input sequence (via the sequence of hidden states produced by the encoder) during each decoding time-step. This reduces the amount of information that needs to be compressed into the context vector and improves performance of encoder-decoder models on longer sequences [26].

Attention mechanisms consist of calculating an *alignment* between the current decoder hidden state and each encoder hidden state [174]. The alignment is a normalized vector (values between zero and one and sums to one) which is calculated via the softmax function over a sequence of *scores*. A score is calculated as the *similarity* between each encoder hidden state and each decoder hidden state.

$$\begin{aligned}\boldsymbol{a}^{(t)} &= \text{align}(\boldsymbol{h}_d^{(t-1)}, \boldsymbol{H}_e) \\ &= \text{softmax}(\text{score}(\boldsymbol{h}_d^{(t-1)}, \boldsymbol{H}_e))\end{aligned}$$

$\boldsymbol{a} \in \mathbb{R}^N$ can be thought of as the attention weights over the input sequence, i.e. how much the model should focus on each of the individual input tokens. The score function can be calculated in multiple ways, the simplest method being a dot product between the individual hidden states. The experiments in this thesis use the *concat* method introduced by Bahdanau et al. [26]:

$$\text{score}_i = \boldsymbol{v} \tanh(\boldsymbol{W}_a[\boldsymbol{h}_d^{(t-1)}; \boldsymbol{h}_e^{(i)}])$$

Each of the i scores is calculated as the concatenation $[\cdot\,;\,\cdot]$ between the current decoder hidden state, $\boldsymbol{h}_d^{(t)} \in \mathbb{R}^H$, and each encoder hidden state, $\boldsymbol{h}_e^{(i)} \in \mathbb{R}^H$, multiplied by a weight matrix, $\boldsymbol{W}_a \in \mathbb{R}^{H \times 2H}$, passed through the tanh activation function and finally multiplied by a learned weight vector, $\boldsymbol{v} \in \mathbb{R}^H$.

Once the attention vector, $\boldsymbol{a}$, is obtained it can then be used to calculate a *weighted context* of the input sequence by multiplying it with the encoder's hidden states.

$$\boldsymbol{c}^{(t)} = \boldsymbol{a}^{(t)} \boldsymbol{H}_e$$

The weighted context, $\boldsymbol{c} \in \mathbb{R}^H$, is a hidden state constructed via a weighted sum of encoder hidden states, where the weights are learned, and are context dependent.

The hidden state obtained from the GRU at each time-step during decoding is given by passing the concatenation of the decoder's current input token, $\boldsymbol{e}^{(t)}$, with the weighted context, $\boldsymbol{c}^{(t)}$, along with the hidden state output from the GRU from the previous time-step:

$$\boldsymbol{h}^{(t)} = \text{GRU}([\boldsymbol{e}^{(t)}; \boldsymbol{c}^{(t)}], \boldsymbol{h}^{(t-1)})$$

The initial input token to the decoder is always a *start-of-sequence token*, `<sos>`. Each prediction, $\hat{\boldsymbol{y}}$, is made by concatenating the decoder input token embedding, the weighted context and the current hidden state, through a linear layer and a softmax activation function:

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(\boldsymbol{W}_p[\boldsymbol{e}^{(t)}; \boldsymbol{c}^{(t)}; \boldsymbol{h}^{(t)}] + \boldsymbol{b}_p)$$

$\boldsymbol{W}_p \in \mathbb{R}^{V \times D+2H}$ and $\boldsymbol{b}_p \in \mathbb{R}^{V}$, where V is the size of the vocabulary, D is the size of the embedding dimension and H is the hidden state size. The final target token is always an *end-of-sequence token*, `<eos>`, and the loss is calculated using the cross-entropy loss between the predicted sequence, $\hat{\boldsymbol{Y}}$, and the actual target sequence, $\boldsymbol{Y}$. When calculating the loss during validation and testing, the decoder input token embedding is calculated from the predicted token from the previous time-step, but during training the actual ground-truth token from the previous time-step is used with a probability of 0.1. Using the ground-truth token is known as *teacher forcing*, and using a constant teacher forcing ratio of 1.0 during training can lead to poor inference results [83].

The metric used to calculate the performance of the variable prediction models is the *F1 score*. The F1 score is calculated using the harmonic mean of the *precision* and *recall* of the predicted sequence and the actual sequence. The precision and recall are calculated using the *true positive* (TP), *false positive* (FP) and *false negative* (FN) rates of the prediction. The true positives rate is the number of sub-tokens predicted that are actually in the ground-truth sequence. The false positive rate is the number of tokens predicted that are not in the ground-truth sequence. The number of tokens which are in the ground-truth sequence but are not in the predicted sequence is the false negative rate. The `<sos>` and `<eos>` tokens are not counted towards the true positive, false positive and false negative rates.

$$\text{F1 score} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$
$$\text{precision} = \frac{TP}{TP + FP}$$
$$\text{recall} = \frac{TP}{TP + FN}$$

F1 score is commonly used for predicting variable names via predicting their individual tokens [10, 20, 18]. One of the disadvantages of the F1 score is that it does not take the order of the sequence into account, e.g. if the target sequence is `reverse, String` then a predicted sequence of `String, reverse` achieves a perfect F1 score, so it should be thought of as the number of relevant tokens retrieved rather than the position aware accuracy of the prediction.

Transfer learning is performed similarly to semantic code search (Chapter 6). Only the encoders use pre-trained models when performing transfer learning, and the decoder is trained from scratch in every experiment. The hypothesis is that the encoder hidden states from models which use transfer learning provide representations which are better for determining the missing variable names. As the masked variable is replaced by the same `<mask>` token as in the pre-training task – masked language modeling – the representation of the masked token should be a good representation for the first token of the variable name to be predicted. However, the masked language modeling task only predicts a single masked token, whereas variable name prediction involves predicting multiple tokens that have been obfuscated by the masked token. Hence, there is a degree of domain mismatch between the masked language modeling and the variable prediction task, as the model cannot rely on surrounding tokens which may be part of the variable name. This may cause the representations for the masked token to be less than optimal for variable name prediction. Even so, performance should be higher than models initialized from scratch.

Again, the data augmentation is performed similar to semantic code search. Both augmentations, strip punctuation and shuffle tokens, are applied after the named variable has been selected and masked. The hypothesis is that both of these data augmentation techniques reduce performance more than in semantic code search. For predicting the name of a masked variable the assumption is that the position of the `<mask>` token with respect to other tokens, both punctuation and other non-punctuation tokens, is more important, and thus carries more information, than in semantic code search.

The expected results are that: when the models are randomly initialized then, the LSTM model outperforms the Transformer model, as seen in semantic code search and previous research which shows the Transformers require larger datasets; models which use transfer learning will outperform randomly initialized models, no matter the pre-training language; pre-training on Java will outperform pre-training on WikiText-103 as the information obtained from training on tokenized code will be more useful than natural languages; pre-training on all six languages in the CodeSearchNet dataset will outperform pre-training on just the Java data as the representations learned from variables within the six languages will generalize to Java, and the domain-shift will be offset by the larger number of examples in the pre-training dataset; both data augmentation techniques will reduce performance more than in semantic code search due to variable name prediction requiring more context in which the `<mask>` tokens appear; the shuffle tokens data augmentation technique will provide a larger decrease in performance compared to the strip punctuation data augmentation as more contextual information is lost when using this data augmentation technique.

7.3 Results and Discussion

Table 9 shows the results for the variable prediction task. Figure 11 shows the F1 score obtained on the validation set during training when using different initialization techniques. Figures 12, 13, 14 and 15 compare the results of the data augmentation techniques when using different initialization methods.

From Table 9 and Figure 11 it can be seen that when no transfer learning is performed, the LSTM model is the best performing model. Interestingly, the Transformer model is unable to learn anything useful for the variable prediction task and even performs worse than the CNN and NBoW models.

Table 9 and Figure 11 also show that all forms of transfer learning used in these experiments provide performance improvements over randomly initialized models. The Transformer's performance increases the most, out of the four models experimented on, however even a pre-trained Transformer is

Model	**Initialization**	**F1 Score**	**ΔSP**	**ΔST**
NBoW	Random	0.0858	-0.0180	-0.0043
NBoW	Java	0.1018	-0.0123	0.0016
NBoW	Wiki	0.1006	-0.0011	0.0056
NBoW	All	0.1047	-0.0089	0.0053
LSTM	Random	**0.2159**	-0.0683	-0.2030
LSTM	Java	**0.2857**	-0.0598	-0.1563
LSTM	Wiki	**0.2682**	-0.0585	-0.1601
LSTM	All	**0.2896**	-0.0576	-0.1609
CNN	Random	0.1879	-0.0160	-0.0994
CNN	Java	0.2003	-0.0121	-0.0854
CNN	Wiki	0.2007	-0.0331	-0.1080
CNN	All	0.2093	-0.0255	-0.0974
Transformer	Random	0.0152	0.0009	0.0067
Transformer	Java	0.1622	-0.0515	-0.0787
Transformer	Wiki	0.1603	-0.0916	-0.0838
Transformer	All	0.2069	-0.0360	-0.1191

Table 9: F1 scores obtained on the test set for variable prediction using different initialization techniques and the increase (green) or decrease (red) in the obtained F1 score when using strip punctuation (**ΔSP**) or shuffle tokens (**ΔST**) data augmentation. *Random* implies the weights are randomly initialized, *Java* initialization uses a language model pre-trained on Java code, *Wiki* initializes the weights from a language model pre-trained on the WikiText-103 dataset, and *All* uses a language model pre-trained on all six languages in the CodeSearchNet dataset. It can be seen that the *All* initialization generally performs best, the strip punctuation data augmentation reduces performance, and the shuffle tokens data augmentation reduces performance more than the strip punctuation data augmentation. These performance decreases are larger than in semantic code search (see Chapter 6, Section 6.3, Table 7. The best result for each initialization technique is shown in bold.

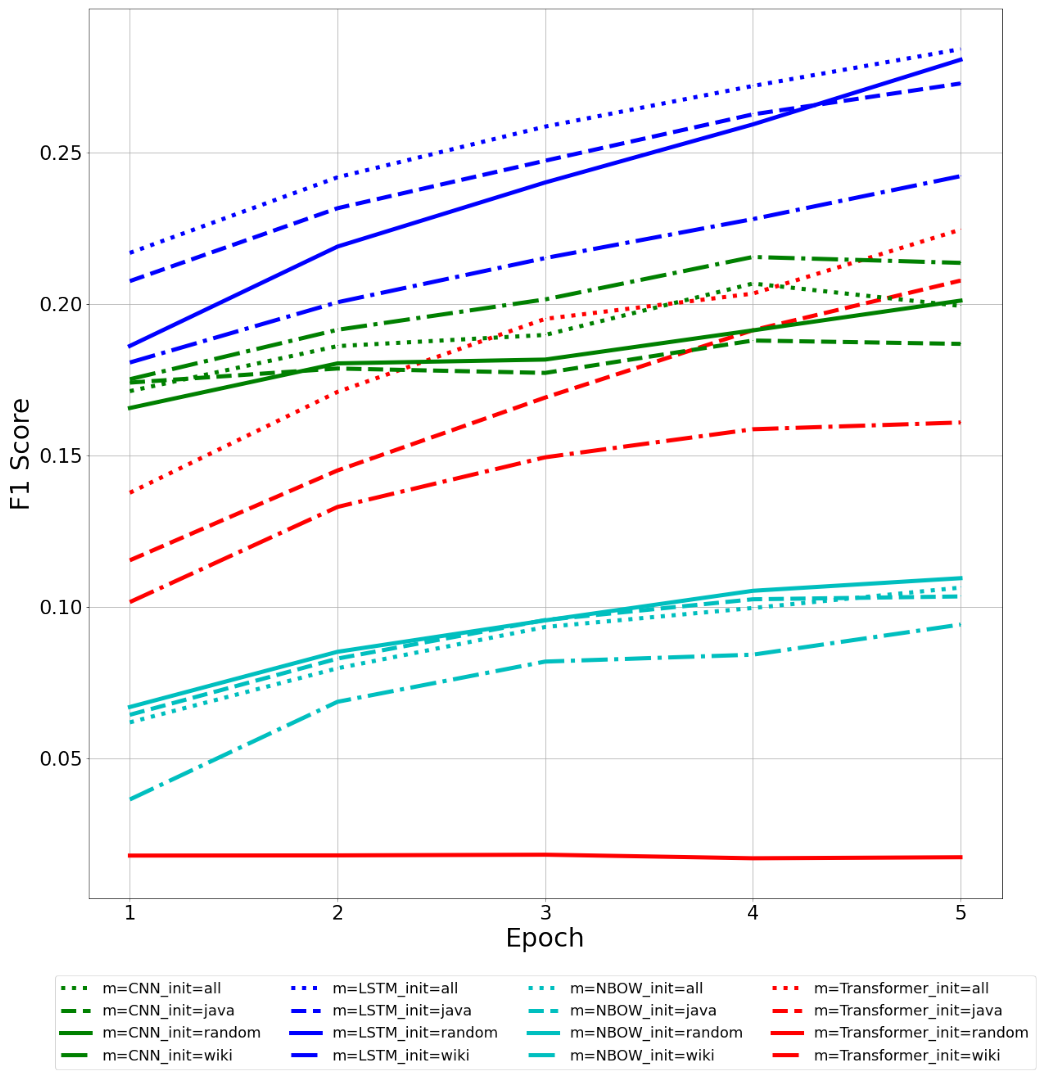

Figure 11: F1 score obtained on the validation set for variable prediction across all four models – CNN (green), LSTM (blue), NBoW (cyan), and Transformer (red) – when initialized from: random weights (solid line), a language model pre-trained on Java (dashed line), a language model pre-trained on all six languages in the CodeSearchNet dataset (dotted line), and a language model pre-trained on WikiText-103 (dot dashed line).

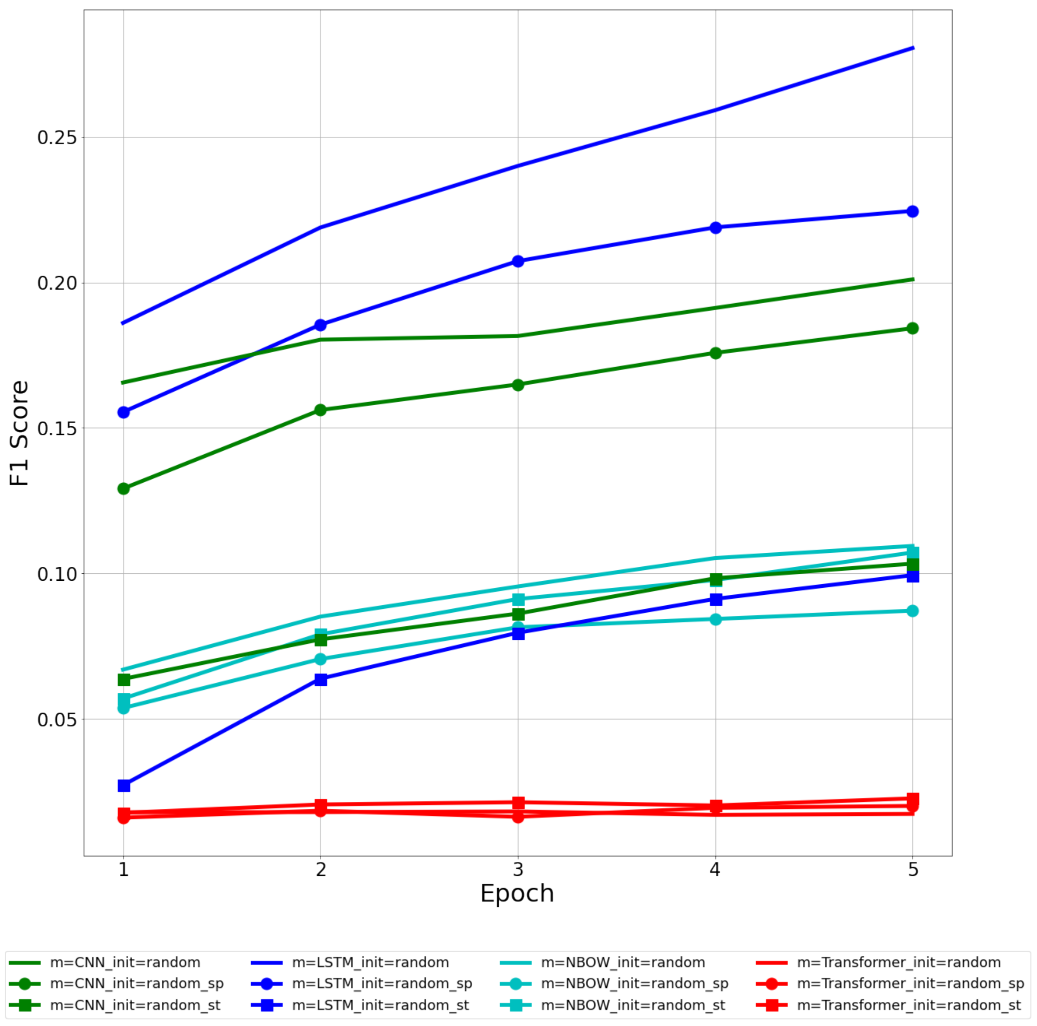

Figure 12: F1 score obtained on the validation set for variable prediction across all four models – CNN (green), LSTM (blue), NBoW (cyan), and Transformer (red) – initialized from a language model pre-trained on WikiText-103 when using no data augmentation (straight line), the strip punctuation (circle markers) and shuffle tokens (square markers) data augmentation.

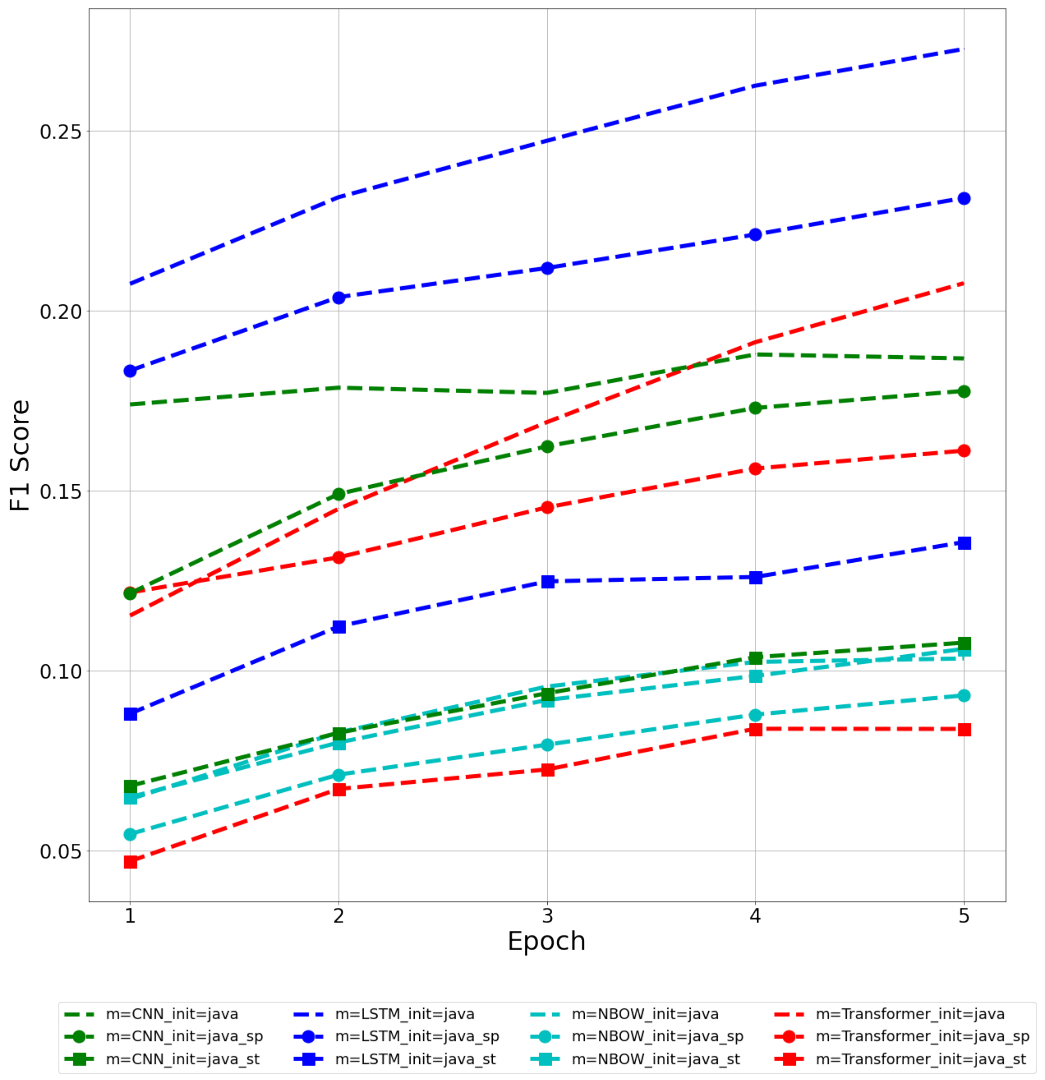

Figure 13: F1 score obtained on the validation set for variable prediction across all four models – CNN (green), LSTM (blue), NBoW (cyan), and Transformer (red) – initialized from a language model pre-trained on WikiText-103 when using no data augmentation (straight line), the strip punctuation (circle markers) and shuffle tokens (square markers) data augmentation.

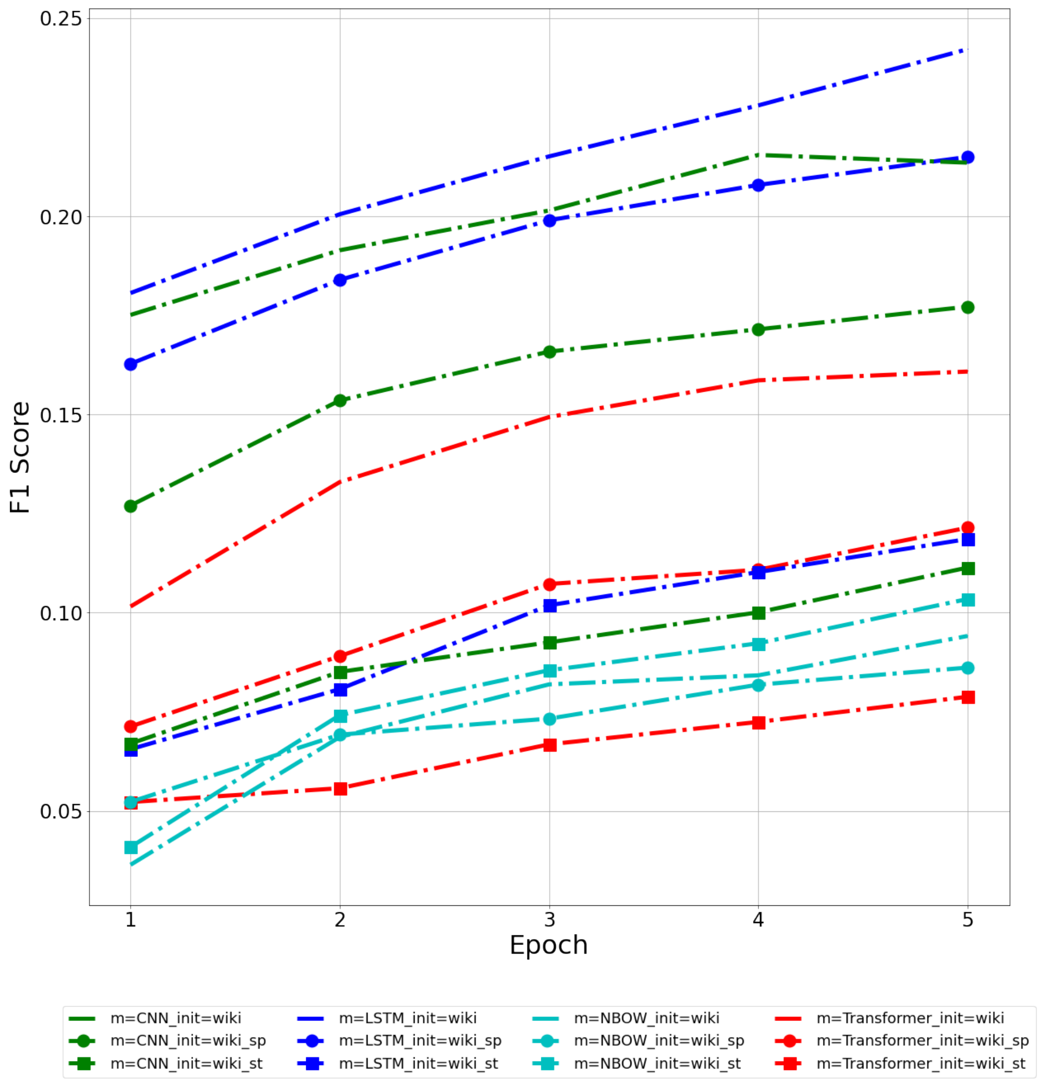

Figure 14: F1 score obtained on the validation set for variable prediction across all four models – CNN (green), LSTM (blue), NBoW (cyan), and Transformer (red) – initialized from a language model pre-trained on WikiText-103 when using no data augmentation (straight line), the strip punctuation (circle markers) and shuffle tokens (square markers) data augmentation.

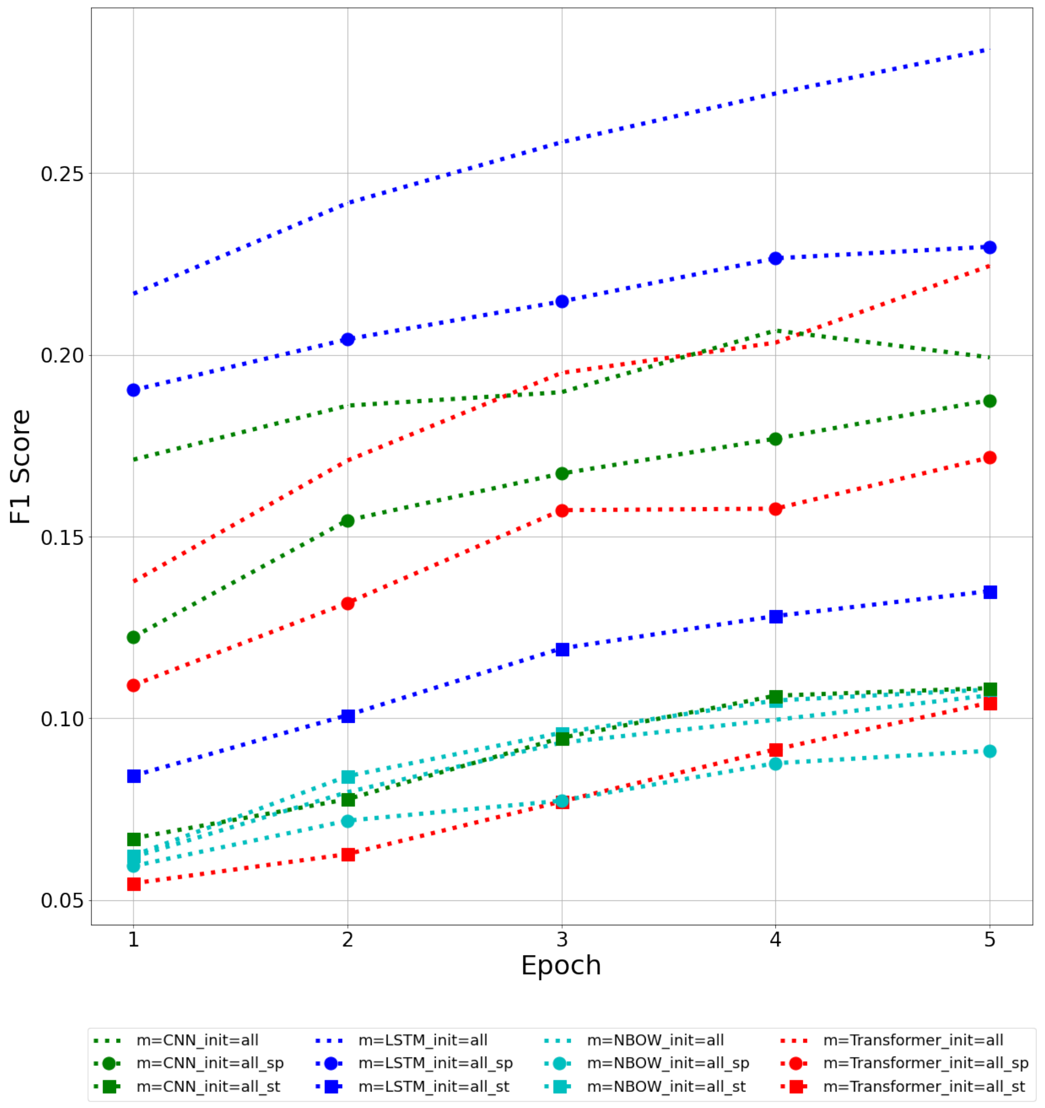

Figure 15: F1 score obtained on the validation set for variable prediction across all four models – CNN (green), LSTM (blue), NBoW (cyan), and Transformer (red) – initialized from a language model pre-trained on WikiText-103 when using no data augmentation (straight line), the strip punctuation (circle markers) and shuffle tokens (square markers) data augmentation.

unable to outperform the LSTM, or even the CNN model, and only outperforms the NBoW model. The results show that a pre-trained LSTM encoder performs the best on the variable prediction task, followed by the pre-trained CNN, pre-trained Transformer, and finally, the pre-trained NBoW model.

Why does the Transformer, even when pre-trained, perform so poorly at the variable prediction task? One potential reason is that Transformer only performs well in a sequence-to-sequence task when both the encoder and decoder are Transformer models, and not when the decoder is a different architecture, such as the GRU used here. Wang et al. [276] have successfully used a Transformer encoder and GRU encoder for neural machine translation and shown that it outperforms their RNN-based encoder-decoder architecture, but only by a small amount. However, their encoder is a single-layer unidirectional GRU model, whereas the encoder used here is a multi-layer bi-directional LSTM. It is possible that by upgrading their RNN-based encoder to have multiple layers and be bi-directional their RNN-based model would outperform the Transformer-RNN model.

Why does the Transformer fail to learn anything when initialized randomly? The original Transformer implementation [270] and popular Transformer variants – such as BERT [66], RoBERTa [188], T5 [217], GPT-2 [216] and GPT-3 [40] – require Transformer specific initialization strategies, such as layer-wise initialization or *warm-up stages*, where the initial learning rate is gradually increased from zero during the early stages of training. To avoid using architecture specific initialization strategies, a default initialization scheme and an optimizer with a fixed learning rate (see Chapter 3, Section 3.6 for more details) is used for all four models in these experiments. These have the potential to lead to unstable training in Transformer models [299, 298, 302], which may explain why the Transformer used here is unable to learn to perform the variable naming task unless initialized from a pre-trained language model.

Comparing the performance between models pre-trained on WikiText-103 and the Java data from CodeSearchNet it can be seen that: comparable performance is achieved for the NBoW and CNN models; the Transformer performs slightly better when transferring from the Java data; and that the LSTM pre-trained on the Java data significantly outperforms an LSTM pre-trained on the WikiText-103 data. The reason for the comparable performance in the NBoW and CNN models may be due to these models not having enough capacity to capture the domain-specific knowledge required for predicting variable names and instead benefiting from the larger dataset provided by WikiText-103. However, all models benefit from any form of pre-training from either a natural language or a programming language, and the use of transfer learning helps the Transformer now outperform the NBoW model by overcoming the instability that arises when randomly initializing a Transformer.

Comparing performance between models pre-trained on only the Java data in CodeSearchNet and those pre-trained on the data from all six languages in CodeSearchNet, it can be seen that using all six languages for pre-training gives the best performance for all models. As with semantic code search, this shows that for variable name prediction, there are features which generalize across different programming languages that are useful for variable name prediction in Java. When initialized from a model pre-trained on all six languages in the CodeSearchNet dataset, the Transformer is now able to perform on par with the CNN model. This now reveals that even with a good initialization, the Transformer still underperforms the LSTM model, implying that the poor performance of the Transformer in this task is most likely due to the interaction between the Transformer encoder and GRU decoder.

As with semantic code search, experiments are performed by training models in the multi-task learning scenario – where models are trained on variable name prediction using all six languages from the CodeSearchNet dataset, or the five languages that are not Java, and then evaluated only on the Java data, i.e. zero-shot learning [286] on Java code.

Table 10 shows the results for the multi-task learning experimental set-up. It can be seen that all models achieve their best performance when performing multi-task learning on all six languages after being pre-trained on all six languages. As expected, models which are not been trained on any Java data perform poorly, with the NBoW model failing to generalize at all. Both Transformer models that use randomly initialized weights also perform poorly, again highlighting the need for Transformers to be pre-trained on the variable prediction task as the Transformer pre-trained and fine-tuned on a dataset with no Java examples at all outperforms a Transformer initialized randomly and trained only on Java. Unlike semantic code search, performance for the LSTM and Transformer models trained and fine-tuned

Model	**Initialization**	**Dataset**	**F1 score**
NBoW	Random	All	0.1078
NBoW	All	All	0.1106
NBoW	Random	No Java	0.0139
NBoW	No Java	No Java	0.0130
LSTM	Random	All	**0.2782**
LSTM	All	All	**0.3150**
LSTM	Random	No Java	**0.1244**
LSTM	No Java	No Java	**0.1261**
CNN	Random	All	0.2003
CNN	All	All	0.2117
CNN	Random	No Java	0.0922
CNN	No Java	No Java	0.0939
Transformer	Random	All	0.0391
Transformer	All	All	0.2689
Transformer	Random	No Java	0.0189
Transformer	No Java	No Java	0.1148

Table 10: F1 scores obtained on the test set for the variable name prediction using different initialization techniques and different datasets for fine-tuning. The test F1 score shown is across the Java data only. *Random* initialization implies the weights are initialized randomly, *All* initialization uses a language model pre-trained on data from all six languages in the CodeSearchNet dataset, *No Java* initialization uses a language model pre-trained on the five languages in the CodeSearchNet dataset that are not Java. A **Dataset** of *All* refers to training on all six languages in the CodeSearchNet dataset, and *No Java* refers to using the five languages in the CodeSearchNet dataset that are not Java. The best result for each initialization-dataset combination is shown in bold.

on all six languages is significantly higher than training on all six languages and only fine-tuning on Java, hence multi-task learning should be used for variable name prediction where available.

When applying the strip punctuation data augmentation, the F1 score decreases by 0.0343. This is 3 times more than the decrease of the MRR on semantic code search. It can be seen that the best performing model, the LSTM, suffers the worst performance degradation when applying the strip punctuation data augmentation – however it still manages to outperform all the other models. This would imply that the LSTM model requires information from these punctuation tokens in order to obtain its performance, more than the other models experimented on, and thus the punctuation tokens carry significantly more information for variable name prediction than they do in semantic code search. The NBoW and CNN models have the least performance decrease, implying that they use the least amount of information contained in the punctuation tokens.

When performing the shuffle tokens data augmentation, F1 score decreases by 0.0835. This is over 2 times more degradation than the strip punctuation data augmentation and 1.5 times more than the shuffle tokens data augmentation when applied to semantic code search. Again, the best performing model, the LSTM, suffers the worst performance decrease, implying that it uses the most information contained in the order of the tokens. As the NBoW model does not use any information from the order of tokens the performance remains the same, but the other three models – which do take the order of tokens into account – decrease in performance, implying that the order of tokens does matter in variable name prediction, and more information is contained in the order of tokens than in semantic code search.

Shown below is an example input given to the best performing model (the LSTM which achieves the highest F1 score in Table 10):

```
public static String reverseString(String str) {
  String <mask> = new StringBuilder(str).reverse().toString();
  return <mask>;
}
```

The top predicted variable names are:

```
1. result
2. str
3. reverse
4. reversed
5. s
```

Any variable name could be used to produce syntactically correct code, however the model does predict variable names that fit the appropriate context. To see how much information contained in the method name is used to predict the variable names, the below example is passed to the model:

```
public static String x(String str) {
        String <mask> = new StringBuilder(str).reverse().toString();
        return <mask>;
}
```

Now, the top predicted variable names are:

```
1. x
2. str
3. s
4. result
5. string
```

Notice how none of the predicted variables mention reverse, or reversing, implying that the model makes significant use of the method name when predicting the variable names.

If, instead, the method name is replaced by a masked token, such as:

```
public static String <mask>(String str) {
        String reversedString = new StringBuilder(str).reverse().toString();
        return reversedString;
}
```

Then the top predicted method names are:

1. `reverse`
2. `reversed`
3. `convert`
4. `Reverse`
5. `string`

The first two predictions are appropriate, however `convert` is a sub-optimal method name as nothing in the code is converted, `Reverse` does not follow Java's camel case naming conventions, and `string` is a vague name which does not describe what the method.

Repeating the above example, but with less appropriate variable names:

```
public static String <mask>(String str) {
        String x = new StringBuilder(str).reverse().toString();
        return x;
}
```

Now, the top predicted method names are:

1. `reverse`
2. `reversed`
3. `str`
4. `string`
5. `parse`

As before, the top two predictions are appropriate, but `str` and `string` are vague, and `parse` would not be an accurate description of the method's functionality. However, as the examples show, the model is usually able to output an appropriate variable or method name within the top predictions.

7.4 Applications

How can a well-trained variable name prediction model – one that can accurately predict a suitable variable name for a given method – be used in an educational setting? The two main use-cases are covered by the VarNaming and VarMisuse task of Allamanis et al. [9].

The VarNaming task is to predict the name of a variable given the surrounding context, i.e. how the variable is used within code. Consider a novice programmer who has difficulty deciding what to name their variables. By leveraging a variable name prediction model, a suitable variable name can be suggested to the programmer, which can then be accepted and automatically applied. This reduces the burden on the programmer to continuously come up with informative variable names, allowing them to focus on other aspects of their code. As there is usually more than one valid name for a variable, the model can suggest multiple variable names and the programmer can select which they believe is the most appropriate. This model will suggest names that follow coding conventions for the programming language used, educating the programmer on the general naming conventions used in that language.

The VarMisuse task is to predict whether the correct variable has been used in a position, for example if performing a `for` loop that increments the value of variable `i` on each iteration then an array inside that loop should most probably be indexed with `i` and not another variable that does not increment. How can the models tell when an incorrect variable has been used at a position? By iterating over all variables in scope and measuring the loss obtained by the model for each variable. If the variable that appears in the position has the lowest loss, then it can be assumed it is the correct variable. If not, then the incorrect variable might have been used. Whenever the model believes an incorrect variable has been used, this can be indicated to the user. This, in theory, allows for real-time bug checking of code, and is able to provide both the location of errors and how they should be fixed (which variable they should be replaced by).

7.5 Conclusion

This section has detailed experiments on variable name prediction. It has shown that when randomly initialized, an LSTM model outperforms the Transformer, CNN and NBoW models. In fact, when randomly initialized the instability in the Transformer models causes them to fail to learn to perform this task at all, causing them to perform worse than even the NBoW model. When performing transfer learning, the LSTM model still outperforms the other three models, and then models can be pre-trained on either natural language data, programming language data where the languages match that of the downstream task, and a combination of different programming languages. Using a combination of multiple, different programming languages achieves the best performance, implying that models are able to learn features that generalize across different programming languages. When performing transfer learning, the Transformer is able to learn to perform this task, however still underperforms the LSTM and CNN models, implying that the interaction between a Transformer encoder and a GRU decoder is sub-optimal for this task. By performing multi-task learning, both pre-training and fine-tuning on all six languages achieves the best overall performance across the Java data. It has been shown that a Transformer pre-trained and fine-tuned on no Java data at all outperforms a randomly initialized Transformer that has been fine-tuned on the Java data only, highlighting the instability in the Transformers when initialized randomly. It has also been shown that removing punctuation tokens decreases performance, more so than semantic code search, showing that variable name prediction requires information contained in the punctuation tokens. Finally, by showing how randomly shuffling the code tokens significantly decreases the performance on this task, it has been shown that the order of the code tokens contains more information for this task than semantic code search.

It has also been described how variable name prediction models can be used to provide feedback using the VarNaming and VarMisuse tasks. The VarNaming task can be used to suggest appropriate variable names to a student, allowing them to focus on other parts of their code. The VarMisuse task can be used to highlight potential errors in code, showing where a variable name prediction model disagrees with the choice of variable used by the programmer and the expected variable name can be suggested.

8 Conclusion

8.1 Synopsis

Machine learning, especially deep learning, when applied to programming languages is still a relatively novel area of research. In this thesis it has been shown how deep learning can be applied to three of the most common tasks in this field: language modeling, semantic code search, and variable name prediction. The experiments have been performed with four distinct machine learning models: NBoW, CNN, LSTM and the Transformer. It has been shown that the use of transfer learning from a pre-trained language model is beneficial to machine learning on programming languages – much like it is with natural languages – and how pre-training effects the outcome of different training, fine-tuning and multi-task learning regimes. It has also been shown how data augmentation techniques, strip punctuation and shuffle tokens, effect the performance on semantic code search and variable name prediction. These data augmentation techniques show how much information is contained within the punctuation tokens and in the order of the tokens.

8.2 Summary of Findings

Chapter 1 introduces the motivations, aims and objects, and contributions of this thesis. Chapter 2 provides a literature review of machine learning on source code. Chapter 3 describes the models and hyperparameters used in this work, and Chapter 4 describes the datasets used for the experiments.

Chapter 5 describes how language modeling is applied to source code. It shows a novel result, that masked language modeling outperforms standard language modeling, in terms of achieving a lower perplexity metric. Experiments confirm the findings of Hindle et al. [109], that programming languages are more predictable than natural languages by showing how the perplexity obtained on programming language data is lower than natural language data, even with a smaller programming language dataset. A novel contribution is showing that the findings of Hindle et al. hold across a variety of models, and on the CodeSearchNet dataset. As expected, a masked language model Transformer outperforms an LSTM on the programming language data. However, the LSTM outperforms the Transformer when applied to natural language data. This novel comparison of LSTM and Transformer models across natural and programming language datasets potentially highlights some inherent ability contained within the Transformer architecture which makes it more suited for performing masked language modeling on programming languages. Finally, the chapter shows how masked language models can be used to detect potential errors in code by detecting areas of high perplexity in statements with synthetic errors added.

Chapter 6 experiments show how machine learning can be applied to semantic code search. When randomly initialized, an LSTM model outperforms all other models experimented on. However, as expected, a pre-trained Transformer model outperforms a pre-trained LSTM. A novel finding discovered is that when performing transfer learning from a pre-trained language model, performance improvements can be obtained by pre-training on either natural language data, programming language data from the same language as the downstream task, or on a dataset consisting of multiple, different programming languages. Experiments show that pre-training on a combination of multiple languages provides the optimal performance, a novel finding highlighting the ability for these models to generalize across programming languages, which indicates that researchers should pre-train their language models on a large dataset consisting of multiple programming languages, instead of one only containing examples in the downstream language. Experiments performed in the multi-task learning scenario show a novel result, that even models pre-trained and fine-tuned on no Java language data outperform randomly initialized models fine-tuned on Java examples only. This implies that using a dataset consisting of multiple programming languages can also be used for low-resource, uncommon programming languages with only a small amount of data available. Finally, by examining two different data augmentation techniques – which have not been applied to source code previously – it has been shown that removing punctuation tokens slightly decreases performance – despite more than halving the number of code tokens – and that shuffling the code tokens also decreases performance, but more so than removing the punctuation tokens. This novel finding shows that more information is contained within the order of the tokens than within the punctuation tokens for this task. The chapter also describes how a semantic code search model can be applied to provide feedback: by propagating instructor feedback between similar submissions, and automatically generating examples in which students must match relevant code-feedback pairs.

In Chapter 7 the experiments show how to apply machine learning models to predict variable names. When randomly initialized, as with semantic code search, the LSTM model outperforms the Transformer, CNN and NBoW models. The experiments shown how the Transformer fails to learn to effectively perform this task when randomly initialized, highlighting the instability that arises when randomly initializing a Transformer model. The LSTM model also outperforms the other three models when fine-tuned from a pre-trained model, and the Transformer manages to effectively learn when fine-tuned, however still underperforms the LSTM and CNN models. The chapter also shows a novel result in that, as with semantic code search, all forms of pre-training performed in the experiments – from natural language, the same programming language as the downstream task and a mixture of six different programming languages – improve the performance across all four models. Again, a novel finding shows that fine-tuning from the mixture of programming languages provides the best performance, indicating that models trained for semantic code search are also able to generalize across programming languages. The performance of the models in a multi-task learning scenario is also measured. Unlike semantic code search, the best performance is achieved when performing both pre-training and fine-tuning on a mixture of distinct programming languages. This novel finding indicates that researchers should not only pre-train, but also fine-tune their models on datasets consisting of multiple programming languages, and not just a dataset containing examples in the downstream task language. Finally, the data augmentation performed by stripping punctuation and shuffling tokens is shown to significantly reduce performance in this task compared to semantic code search. This novel result highlights the importance of the punctuation tokens and the order of the code tokens for variable name prediction, and show that they contain more information for effectively performing this task than they do in semantic code search. The chapter also describes how to apply variable name prediction models for feedback, using the VarNaming and VarMisuse tasks, by either automatically naming variables for the programmer, or by highlighting where the model disagrees with the variable used by the programmer and providing suggestions on what it should be replaced with.

8.3 Limitations

One potential limitation on the work presented in this thesis is the models selected. The models were selected to represent a broad spectrum of general architectures, and none are designed specifically for source code. A model designed specifically for source code will no doubt have improved performance over a model without the same built in biases, if the number of parameters in the two models are similar. However, in *The Bitter Lesson* [253], Sutton argues "general methods that leverage computation are ultimately the most effective, and by a large margin". Hence, any performance gained by using a source code specific model will eventually be overcome by models without the source code specific biases within them simply by using more computation.

The two main methods to allow a neural network model to use more computation are by increasing the number of parameters within the model, or the amount of data used to train the model. Research on increased computation is known as *scaling* [107, 130, 105, 106, 118, 27], and most research into scaling for deep learning focus on finding a *scaling law*: the relationship between compute, dataset size, number of model parameters, and the model performance.

The other limitations of the work in this thesis are: the relatively small size of the models and dataset used. The model sizes were chosen to be able to train on a single GPU. This in contrast to modern deep learning models which now have billions [40, 243], or even trillions [191] of parameters. The arguments for using the CodeSearchNet dataset are detailed in Section 4.1, and the main focus is on the Java dataset, which consists of just under half a million examples. This is a relatively small number of examples for modern machine learning applications as most work in this domain uses tens of millions of examples, e.g. the dataset used by code2vec [20] uses 14 million Java methods. However, the research on scaling has shown that as long as the ratio between the model and dataset size is kept consistent, both can be scaled up with increased compute. Thus, the findings in this thesis should also hold for larger models and datasets.

Finally, the model hyperparameters are chosen using values commonly used across machine learning research, and that are suitable for the compute resources used. This potentially means suboptimal hyperparameters were used for some models and tasks, and that the performance of certain models could be improved by selecting more optimized hyperparameters. Ideally, extensive hyperparameter tuning would be performed in order to obtain the best set of hyperparameters for each model on each task. However, by keeping the hyperparameters constant, it is shown how well the models can generalize across tasks, datasets and training regimes.

8.4 Future Directions

The results contained in this thesis show the importance of fine-tuning pre-trained language models when performing machine learning on programming languages. This has received relatively little attention compared to transfer learning in natural language processing tasks, despite the fact that a vast amount of programming language data is readily available from open-source code repositories. Future work would more thoroughly investigate novel methods of pre-training in order to determine if masked language modeling is the optimal method of pre-training these models, or if there exists a novel pre-training method which outperforms masked language modeling.

It has also been shown that the optimal method of pre-training is by using a model trained on a mixture of different programming languages. This was done by simply using all languages contained within the CodeSearchNet dataset. Future work would examine if this effect is universal – all programming languages benefit a downstream task in any language – or that some downstream tasks would benefit from a specific mixture of programming languages within the pre-training dataset. Perhaps performing downstream tasks on a functional programming language requires more functional programming languages to be within the pre-training dataset, or that if the pre-training dataset is large enough then only data in the downstream task language is required.

The experiments have also focused on a single dataset and most results are focused on the downstream performance in Java. Future work would investigate if these results hold true for each language in the CodeSearchNet dataset.

The two data augmentation techniques used, strip punctuation and shuffle tokens, have been shown to reduce performance on both the semantic code search and variable name prediction. Although this has shown the importance of the punctuation tokens and the order of code tokens within the examples, ideally future work would be to determine if there exists a data augmentation technique that actually improves performance for these tasks much like data augmentation improves performance for computer vision and natural language processing.

Finally, future work could also focus on applying the assessment techniques discussed, and measuring how effective each one is. The rise in the popularity of remote learning means that online classes are no longer restricted in size. However, the growing size of these classes places the burden of applying detailed feedback to each individual learner on the instructors. Automated assessment, using methods such as the ones detailed in this thesis, can be used to alleviate some of the effort required by instructors to provide feedback, and in some cases can be used to offer novel methods of providing feedback. The hope is that future, automated assessment approaches can, potentially, go beyond giving a simple pass/fail or numerical grade.

9 Bibliography

[1] S. Abebe et al. “Can Lexicon Bad Smells Improve Fault Prediction?” In: *2012 19th Working Conference on Reverse Engineering* (2012), pp. 235–244.

[2] S. Abebe et al. “The Effect of Lexicon Bad Smells on Concept Location in Source Code”. In: *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation* (2011), pp. 125–134.

[3] D. Adiwardana et al. “Towards a Human-like Open-Domain Chatbot”. In: *ArXiv* abs/2001.09977 (2020).

[4] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. “JuICe: A Large Scale Distantly Supervised Dataset for Open Domain Context-based Code Generation”. In: *CoRR* abs/1910.02216 (2019). arXiv: `1910.02216`. URL: `http://arxiv.org/abs/1910.02216`.

[5] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. “Using machine translation for converting *Python 2* to *Python 3* code”. In: *PeerJ PrePrints* 3 (2015), e1459. DOI: `10.7287/peerj.preprints.1459v1`. URL: `https://doi.org/10.7287/peerj.preprints.1459v1`.

[6] Toufique Ahmed, Vincent Hellendoorn, and Premkumar T. Devanbu. “Learning Lenient Parsing & Typing via Indirect Supervision”. In: *CoRR* abs/1910.05879 (2019). arXiv: `1910.05879`. URL: `http://arxiv.org/abs/1910.05879`.

[7] Alan Akbik, Duncan Blythe, and Roland Vollgraf. “Contextual String Embeddings for Sequence Labeling”. In: *Proceedings of the 27th International Conference on Computational Linguistics*. Santa Fe, New Mexico, USA: Association for Computational Linguistics, Aug. 2018, pp. 1638–1649. URL: `https://www.aclweb.org/anthology/C18-1139`.

[8] Miltiadis Allamanis. “The Adverse Effects of Code Duplication in Machine Learning Models of Code”. In: *CoRR* abs/1812.06469 (2018). arXiv: `1812.06469`. URL: `http://arxiv.org/abs/1812.06469`.

[9] 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. URL: `https://openreview.net/forum?id=BJOFETxR-`.

[10] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 2091–2100. URL: `http://proceedings.mlr.press/v48/allamanis16.html`.

[11] Miltiadis Allamanis and Charles A. Sutton. "Mining idioms from source code". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 472–483. DOI: `10.1145/2635868.2635901`. URL: `https://doi.org/10.1145/2635868.2635901`.

[12] Miltiadis Allamanis and Charles A. Sutton. "Mining source code repositories at massive scale using language modeling". In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. Ed. by Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim. IEEE Computer Society, 2013, pp. 207–216. DOI: `10.1109/MSR.2013.6624029`. URL: `https://doi.org/10.1109/MSR.2013.6624029`.

[13] Miltiadis Allamanis et al. "Bimodal Modelling of Source Code and Natural Language". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 2123–2132. URL: `http://proceedings.mlr.press/v37/allamanis15.html`.

[14] Miltiadis Allamanis et al. "Learning natural coding conventions". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on*

Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 281–293. DOI: `10.1145/2635868.2635883`. URL: `https://doi.org/10.1145/2635868.2635883`.

[15] Miltiadis Allamanis et al. "Mining Semantic Loop Idioms". In: *IEEE Trans. Software Eng.* 44.7 (2018), pp. 651–668. DOI: `10.1109/TSE.2018.2832048`. URL: `https://doi.org/10.1109/TSE.2018.2832048`.

[16] Miltiadis Allamanis et al. "Suggesting accurate method and class names". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans. ACM, 2015, pp. 38–49. DOI: `10.1145/2786805.2786849`. URL: `https://doi.org/10.1145/2786805.2786849`.

[17] Miltiadis Allamanis et al. "Typilus: Neural Type Hints". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 91–105. ISBN: 9781450376136. DOI: `10.1145/3385412.3385997`. URL: `https://doi.org/10.1145/3385412.3385997`.

[18] Uri Alon, Omer Levy, and Eran Yahav. "code2seq: Generating Sequences from Structured Representations of Code". In: *CoRR* abs/1808.01400 (2018). arXiv: `1808.01400`. URL: `http://arxiv.org/abs/1808.01400`.

[19] Uri Alon et al. "A general path-based representation for predicting program properties". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 404–419. DOI: `10.1145/3192366.3192412`. URL: `https://doi.org/10.1145/3192366.3192412`.

[20] Uri Alon et al. "code2vec: Learning Distributed Representations of Code". In: *CoRR* abs/1803.09473 (2018). arXiv: `1803.09473`. URL: `http://arxiv.org/abs/1803.09473`.

[21] Uri Alon et al. “Structural Language Models for Any-Code Generation”. In: *CoRR* abs/1910.00577 (2019). arXiv: `1910.00577`. URL: `http://arxiv.org/abs/1910.00577`.

[22] Matthew Amodio, Swarat Chaudhuri, and Thomas W. Reps. “Neural Attribute Machines for Program Generation”. In: *CoRR* abs/1705.09231 (2017). arXiv: `1705.09231`. URL: `http://arxiv.org/abs/1705.09231`.

[23] N. Anand. “Clarify Function!” In: *SIGPLAN Not.* 23.6 (June 1988), pp. 69–79. ISSN: 0362-1340. DOI: `10.1145/44546.44552`. URL: `https://doi.org/10.1145/44546.44552`.

[24] Venera Arnaoudova, M. D. Penta, and G. Antoniol. “Linguistic antipatterns: what they are and how developers perceive them”. In: *Empirical Software Engineering* 21 (2014), pp. 104–158.

[25] Venera Arnaoudova et al. “REPENT: Analyzing the Nature of Identifier Renamings”. In: *IEEE Transactions on Software Engineering* 40 (2014), pp. 502–532.

[26] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: `http://arxiv.org/abs/1409.0473`.

[27] Y. Bahri et al. “Explaining Neural Scaling Laws”. In: *ArXiv* abs/2102.06701 (2021).

[28] Antonio Valerio Miceli Barone and Rico Sennrich. “A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation”. In: *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan, November 27 - December 1, 2017, Volume 2: Short Papers*. Ed. by Greg Kondrak and Taro Watanabe. Asian Federation of Natural Language Processing, 2017, pp. 314–319. URL: `https://www.aclweb.org/anthology/I17-2053/`.

[29] Rohan Bavishi, Michael Pradel, and Koushik Sen. "Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts". In: *CoRR* abs/1809.05193 (2018). arXiv: `1809.05193`. URL: `http://arxiv.org/abs/1809.05193`.

[30] 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, 3-8 December 2018, Montréal, Canada*. Ed. by Samy Bengio et al. 2018, pp. 3589–3601. URL: `http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics`.

[31] 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 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Michel Chaudron et al. ACM, 2018, pp. 60–70. DOI: `10.1145/3180155.3180219`. URL: `https://doi.org/10.1145/3180155.3180219`.

[32] Sahil Bhatia and Rishabh Singh. "Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks". In: *CoRR* abs/1603.06129 (2016). arXiv: `1603.06129`. URL: `http://arxiv.org/abs/1603.06129`.

[33] Avishkar Bhoopchand et al. "Learning Python Code Suggestion with a Sparse Pointer Network". In: *CoRR* abs/1611.08307 (2016). arXiv: `1611.08307`. URL: `http://arxiv.org/abs/1611.08307`.

[34] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. "PHOG: Probabilistic Model for Code". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 2933–2942. URL: `http://proceedings.mlr.press/v48/bielik16.html`.

[35] D. Binkley et al. "The impact of identifier style on effort and comprehension". In: *Empirical Software Engineering* 18 (2012), pp. 219–276.

[36] Bernd Bohnet et al. “Morphosyntactic Tagging with a Meta-BiLSTM Model over Context Sensitive Token Encodings”. In: *CoRR* abs/1805.08237 (2018). arXiv: `1805.08237`. URL: `http://arxiv.org/abs/1805.08237`.

[37] Piotr Bojanowski et al. “Enriching Word Vectors with Subword Information”. In: *CoRR* abs/1607.04606 (2016). arXiv: `1607.04606`. URL: `http://arxiv.org/abs/1607.04606`.

[38] Samuel R. Bowman et al. “A large annotated corpus for learning natural language inference”. In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 632–642. DOI: `10.18653/v1/D15-1075`. URL: `https://www.aclweb.org/anthology/D15-1075`.

[39] Marc Brockschmidt et al. “Generative Code Modeling with Graphs”. In: *CoRR* abs/1805.08490 (2018). arXiv: `1805.08490`. URL: `http://arxiv.org/abs/1805.08490`.

[40] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: `2005.14165 [cs.CL]`.

[41] 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 International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. Ed. by Hans van Vliet and Valérie Issarny. ACM, 2009, pp. 213–222. DOI: `10.1145/1595696.1595728`. URL: `https://doi.org/10.1145/1595696.1595728`.

[42] 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*. Vol. WS-18. AAAI Workshops. AAAI Press, 2018, pp. 758–761. URL: `https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/17338`.

[43] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. "SAR: learning cross-language API mappings with little knowledge". In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Marlon Dumas et al. ACM, 2019, pp. 796–806. DOI: `10.1145/3338906.3338924`. URL: `https://doi.org/10.1145/3338906.3338924`.

[44] Simon Butler et al. "Relating Identifier Naming Flaws and Code Quality: An Empirical Study". In: *2009 16th Working Conference on Reverse Engineering* (2009), pp. 31–35.

[45] Ruichu Cai et al. "TAG : Type Auxiliary Guiding for Code Comment Generation". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020, pp. 291–301. DOI: `10.18653/v1/2020.acl-main.27`. URL: `https://www.aclweb.org/anthology/2020.acl-main.27`.

[46] José Cambronero et al. "When Deep Learning Met Code Search". In: *CoRR* abs/1905.03813 (2019). arXiv: `1905.03813`. URL: `http://arxiv.org/abs/1905.03813`.

[47] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. "Syntax errors just aren't natural: improving error reporting with language models". In: *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. Ed. by Premkumar T. Devanbu, Sung Kim, and Martin Pinzger. ACM, 2014, pp. 252–261. DOI: `10.1145/2597073.2597102`. URL: `https://doi.org/10.1145/2597073.2597102`.

[48] B. Caprile and P. Tonella. "Nomen est omen: analyzing the language of function identifiers". In: *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)* (1999), pp. 112–122.

[49] Luigi Cerulo et al. "Irish: A Hidden Markov Model to detect coded information islands in free text". In: *Sci. Comput. Program.* 105 (2015), pp. 26–43. DOI: `10.1016/j.scico.2014.11.017`. URL: `https://doi.org/10.1016/j.scico.2014.11.017`.

[50] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. “Tree2Tree Neural Translation Model for Learning Source Code Changes”. In: *CoRR* abs/1810.00314 (2018). arXiv: `1810.00314`. URL: `http://arxiv.org/abs/1810.00314`.

[51] Ciprian Chelba et al. “One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling”. In: *CoRR* abs/1312.3005 (2013). arXiv: `1312.3005`. URL: `http://arxiv.org/abs/1312.3005`.

[52] Zimin Chen and Martin Monperrus. “A Literature Study of Embeddings on Source Code”. In: *CoRR* abs/1904.03061 (2019). arXiv: `1904.03061`. URL: `http://arxiv.org/abs/1904.03061`.

[53] Zimin Chen et al. “SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair”. In: *CoRR* abs/1901.01808 (2019). arXiv: `1901.01808`. URL: `http://arxiv.org/abs/1901.01808`.

[54] KyungHyun Cho et al. “On the Properties of Neural Machine Translation: Encoder-Decoder Approaches”. In: *CoRR* abs/1409.1259 (2014). arXiv: `1409.1259`. URL: `http://arxiv.org/abs/1409.1259`.

[55] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. ACL, 2014, pp. 1724–1734. DOI: `10.3115/v1/d14-1179`. URL: `https://doi.org/10.3115/v1/d14-1179`.

[56] Dami Choi et al. “On Empirical Comparisons of Optimizers for Deep Learning”. In: *CoRR* abs/1910.05446 (2019). arXiv: `1910.05446`. URL: `http://arxiv.org/abs/1910.05446`.

[57] Sumit Chopra, Michael Auli, and Alexander M. Rush. “Abstractive Sentence Summarization with Attentive Recurrent Neural Networks”. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, June 2016, pp. 93–98. DOI: `10.18653/v1/N16-1012`. URL: `https://www.aclweb.org/anthology/N16-1012`.

[58] Alexis Conneau et al. "XNLI: Evaluating Cross-lingual Sentence Representations". In: *CoRR* abs/1809.05053 (2018). arXiv: `1809.05053`. URL: `http://arxiv.org/abs/1809.05053`.

[59] Chris Cummins et al. "Compiler fuzzing through deep learning". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by Frank Tip and Eric Bodden. ACM, 2018, pp. 95–105. DOI: `10.1145/3213846.3213848`. URL: `https://doi.org/10.1145/3213846.3213848`.

[60] Chris Cummins et al. "End-to-End Deep Learning of Optimization Heuristics". In: *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*. IEEE Computer Society, 2017, pp. 219–232. DOI: `10.1109/PACT.2017.24`. URL: `https://doi.org/10.1109/PACT.2017.24`.

[61] Chris Cummins et al. "Synthesizing benchmarks for predictive modeling". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. Ed. by Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang. ACM, 2017, pp. 86–99. URL: `http://dl.acm.org/citation.cfm?id=3049843`.

[62] 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, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 1475–1485. URL: `http://proceedings.mlr.press/v97/cvitkovic19b.html`.

[63] Zihang Dai et al. "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context". In: *CoRR* abs/1901.02860 (2019). arXiv: `1901.02860`. URL: `http://arxiv.org/abs/1901.02860`.

[64] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. "RefiNym: using names to refine types". 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*. Ed.

by Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu. ACM, 2018, pp. 107–117. DOI: `10.1145/3236024.3236042`. URL: `https://doi.org/10.1145/3236024.3236042`.

[65] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[66] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: `1810.04805`. URL: `http://arxiv.org/abs/1810.04805`.

[67] Jacob Devlin et al. "Semantic Code Repair using Neuro-Symbolic Transformation Networks". In: *CoRR* abs/1710.11054 (2017). arXiv: `1710.11054`. URL: `http://arxiv.org/abs/1710.11054`.

[68] I. Drozdov et al. "Supervised and unsupervised language modelling in Chest X-Ray radiological reports". In: *PLoS ONE* 15 (2020).

[69] Vijay Prakash Dwivedi and Xavier Bresson. "A Generalization of Transformer Networks to Graphs". In: *ArXiv* abs/2012.09699 (2020).

[70] J. Elman. "Finding Structure in Time". In: *Cogn. Sci.* 14 (1990), pp. 179–211.

[71] Aysu Ezen-Can. *A Comparison of LSTM and BERT for Small Corpus*. 2020. arXiv: `2009.05451 [cs.CL]`.

[72] Aysu Ezen-Can. "A Comparison of LSTM and BERT for Small Corpus". In: *ArXiv* abs/2009.05451 (2020).

[73] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *ArXiv* abs/2002.08155 (2020).

[74] Christine Franks et al. "CACHECA: A Cache Language Model Based Code Suggestion Tool". In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*. Ed. by Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum. IEEE Computer Society, 2015, pp. 705–708. DOI: `10.1109/ICSE.2015.228`. URL: `https://doi.org/10.1109/ICSE.2015.228`.

[75] Mark Gabel and Zhendong Su. "A Study of the Uniqueness of Source Code". In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 147–156. ISBN: 9781605587912. DOI: 10.1145/1882291.1882315. URL: https://doi.org/10.1145/1882291.1882315.

[76] Jonas Gehring et al. "Convolutional Sequence to Sequence Learning". In: *CoRR* abs/1705.03122 (2017). arXiv: 1705.03122. URL: http://arxiv.org/abs/1705.03122.

[77] Felix A. Gers, Jürgen A. Schmidhuber, and Fred A. Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Neural Comput.* 12.10 (Oct. 2000), pp. 2451–2471. ISSN: 0899-7667. DOI: 10.1162/089976600300015015. URL: http://dx.doi.org/10.1162/089976600300015015.

[78] Ross B. Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524.

[79] Elena L. Glassman et al. "OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale". In: *ACM Trans. Comput. Hum. Interact.* 22.2 (2015), 7:1–7:35. DOI: 10.1145/2699751. URL: https://doi.org/10.1145/2699751.

[80] Andres Goens et al. "A Case Study on Machine Learning for Synthesizing Benchmarks". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 38–46. ISBN: 9781450367196. DOI: 10.1145/3315508.3329976. URL: https://doi.org/10.1145/3315508.3329976.

[81] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014, pp. 2672–2680. URL: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf.

[82] Ian J. Goodfellow et al. "Generative Adversarial Networks". In: *CoRR* abs/1406.2661 (2014). arXiv: 1406.2661. URL: http://arxiv.org/abs/1406.2661.

[83] Anirudh Goyal et al. "Professor Forcing: A New Algorithm for Training Recurrent Networks". In: *NIPS*. 2016.

[84] Alex Graves. "Generating Sequences With Recurrent Neural Networks". In: *CoRR* abs/1308.0850 (2013). arXiv: `1308.0850`. URL: `http://arxiv.org/abs/1308.0850`.

[85] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. "Speech Recognition with Deep Recurrent Neural Networks". In: *CoRR* abs/1303.5778 (2013). arXiv: `1303.5778`. URL: `http://arxiv.org/abs/1303.5778`.

[86] Jiatao Gu et al. "Incorporating Copying Mechanism in Sequence-to-Sequence Learning". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. DOI: `10.18653/v1/p16-1154`. URL: `https://doi.org/10.18653/v1/p16-1154`.

[87] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. "Deep code search". In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Michel Chaudron et al. ACM, 2018, pp. 933–944. DOI: `10.1145/3180155.3180167`. URL: `https://doi.org/10.1145/3180155.3180167`.

[88] Xiaodong Gu et al. "Deep API learning". In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. Ed. by Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su. ACM, 2016, pp. 631–642. DOI: `10.1145/2950290.2950334`. URL: `https://doi.org/10.1145/2950290.2950334`.

[89] Xiaodong Gu et al. "DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. Ed. by Carles Sierra. ijcai.org, 2017, pp. 3675–3681. DOI: `10.24963/ijcai.2017/514`. URL: `https://doi.org/10.24963/ijcai.2017/514`.

[90] Anshul Gupta and Neel Sundaresan. "Intelligent code reviews using deep learning". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*. 2018.

[91] Rahul Gupta, Aditya Kanade, and Shirish Shevade. “Neural Attribution for Semantic Bug-Localization in Student Programs”. In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/f29a179746902e331572c483c45e5086-Paper.pdf.

[92] Rahul Gupta, Aditya Kanade, and Shirish K. Shevade. “Deep Reinforcement Learning for Programming Language Correction”. In: *CoRR* abs/1801.10467 (2018). arXiv: 1801.10467. URL: http://arxiv.org/abs/1801.10467.

[93] Rahul Gupta et al. “DeepFix: Fixing Common C Language Errors by Deep Learning”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. Ed. by Satinder P. Singh and Shaul Markovitch. AAAI Press, 2017, pp. 1345–1351. URL: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603.

[94] Andrew Habib and Michael Pradel. “Neural Bug Finding: A Study of Opportunities and Challenges”. In: *CoRR* abs/1906.00307 (2019). arXiv: 1906.00307. URL: http://arxiv.org/abs/1906.00307.

[95] Hossein Hajipour, Apratim Bhattacharyya, and Mario Fritz. “SampleFix: Learning to Correct Programs by Sampling Diverse Fixes”. In: *CoRR* abs/1906.10502 (2019). arXiv: 1906.10502. URL: http://arxiv.org/abs/1906.10502.

[96] Jacob Harer et al. “Learning to Repair Software Vulnerabilities with Generative Adversarial Networks”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. Ed. by Samy Bengio et al. 2018, pp. 7944–7954. URL: http://papers.nips.cc/paper/8018-learning-to-repair-software-vulnerabilities-with-generative-adversarial-networks.

[97] Tatsunori B. Hashimoto et al. “A Retrieve-and-Edit Framework for Predicting Structured Outputs”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. Ed. by Samy Bengio et al. 2018, pp. 10073–10083.

URL: `http://papers.nips.cc/paper/8209-a-retrieve-and-edit-framework-for-predicting-structured-outputs`.

[98] Hideaki Hata, Emad Shihab, and Graham Neubig. "Learning to Generate Corrective Patches using Neural Machine Translation". In: *CoRR* abs/1812.07170 (2018). arXiv: `1812.07170`. URL: `http://arxiv.org/abs/1812.07170`.

[99] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 770–778.

[100] Vincent Hellendoorn, Premkumar T. Devanbu, and Alberto Bacchelli. "Will They Like This? Evaluating Code Contributions with Language Models". In: *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. Ed. by Massimiliano Di Penta, Martin Pinzger, and Romain Robbes. IEEE Computer Society, 2015, pp. 157–167. DOI: `10.1109/MSR.2015.22`. URL: `https://doi.org/10.1109/MSR.2015.22`.

[101] Vincent J. Hellendoorn and Premkumar T. Devanbu. "Are deep neural networks the best choice for modeling source code?" In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. Ed. by Eric Bodden et al. ACM, 2017, pp. 763–773. DOI: `10.1145/3106237.3106290`. URL: `https://doi.org/10.1145/3106237.3106290`.

[102] Vincent J. Hellendoorn et al. "Deep learning type inference". 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*. Ed. by Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu. ACM, 2018, pp. 152–162. DOI: `10.1145/3236024.3236051`. URL: `https://doi.org/10.1145/3236024.3236051`.

[103] Vincent J. Hellendoorn et al. "Global Relational Models of Source Code". In: *ICLR*. 2020.

[104] Dan Hendrycks and Kevin Gimpel. "Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units". In: *CoRR* abs/1606.08415 (2016). arXiv: `1606.08415`. URL: `http://arxiv.org/abs/1606.08415`.

[105] T. Henighan et al. “Scaling Laws for Autoregressive Generative Modeling”. In: *ArXiv* abs/2010.14701 (2020).

[106] Danny Hernandez et al. “Scaling Laws for Transfer”. In: *ArXiv* abs/2102.01293 (2021).

[107] Joel Hestness et al. “Deep Learning Scaling is Predictable, Empirically”. In: *CoRR* abs/1712.00409 (2017). arXiv: 1712.00409. URL: http://arxiv.org/abs/1712.00409.

[108] Abram Hindle et al. “On the naturalness of software”. In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE Computer Society, 2012, pp. 837–847. DOI: 10.1109/ICSE.2012.6227135. URL: https://doi.org/10.1109/ICSE.2012.6227135.

[109] Abram Hindle et al. “On the naturalness of software”. In: *Commun. ACM* 59.5 (2016), pp. 122–131. DOI: 10.1145/2902362. URL: https://doi.org/10.1145/2902362.

[110] Geoffrey E. Hinton, Sara Sabour, and Nicholas Frosst. “Matrix capsules with EM routing”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=HJWLfGWRb.

[111] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[112] Johannes C. Hofmeister, J. Siegmund, and Daniel V. Holt. “Shorter identifier names take longer to comprehend”. In: *Empirical Software Engineering* 24 (2017), pp. 417–443.

[113] Jeremy Howard and Sebastian Ruder. “Fine-tuned Language Models for Text Classification”. In: *CoRR* abs/1801.06146 (2018). arXiv: 1801.06146. URL: http://arxiv.org/abs/1801.06146.

[114] Chun-Hung Hsiao, Michael J. Cafarella, and Satish Narayanasamy. "Using web corpus statistics for program analysis". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by Andrew P. Black and Todd D. Millstein. ACM, 2014, pp. 49–65. DOI: `10.1145/2660193.2660226`. URL: `https://doi.org/10.1145/2660193.2660226`.

[115] Hamel Husain and Ho-Hsiang Wu. *Towards natural language semantic code search*. 2018. URL: `https://github.blog/2018-09-18-towards-natural-language-semantic-code-search/`.

[116] Hamel Husain et al. "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search". In: *CoRR* abs/1909.09436 (2019). arXiv: `1909.09436`. URL: `http://arxiv.org/abs/1909.09436`.

[117] Yasir Hussain et al. "Deep Transfer Learning for Source Code Modeling". In: *CoRR* abs/1910.05493 (2019). arXiv: `1910.05493`. URL: `http://arxiv.org/abs/1910.05493`.

[118] Marcus Hutter. "Learning Curve Theory". In: *ArXiv* abs/2102.04074 (2021).

[119] Kazuki Irie et al. "Language Modeling with Deep Transformers". In: *CoRR* abs/1905.04226 (2019). arXiv: `1905.04226`. URL: `http://arxiv.org/abs/1905.04226`.

[120] Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. "Learning Programmatic Idioms for Scalable Semantic Parsing". In: *CoRR* abs/1904.09086 (2019). arXiv: `1904.09086`. URL: `http://arxiv.org/abs/1904.09086`.

[121] Srinivasan Iyer et al. "Mapping Language to Code in Programmatic Context". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. Ed. by Ellen Riloff et al. Association for Computational Linguistics, 2018, pp. 1643–1652. DOI: `10.18653/v1/d18-1192`. URL: `https://doi.org/10.18653/v1/d18-1192`.

[122] Srinivasan Iyer et al. “Summarizing Source Code using a Neural Attention Model”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. DOI: `10.18653/v1/p16-1195`. URL: `https://doi.org/10.18653/v1/p16-1195`.

[123] Srinivasan Iyer et al. “Summarizing Source Code using a Neural Attention Model”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. DOI: `10.18653/v1/P16-1195`. URL: `https://www.aclweb.org/anthology/P16-1195`.

[124] Mohit Iyyer et al. “Deep Unordered Composition Rivals Syntactic Methods for Text Classification”. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, July 2015, pp. 1681–1691. DOI: `10.3115/v1/P15-1162`. URL: `https://www.aclweb.org/anthology/P15-1162`.

[125] Vinoj Jayasundara et al. “TreeCaps: Tree-Structured Capsule Networks for Program Source Code Processing”. In: *CoRR* abs/1910.12306 (2019). arXiv: `1910.12306`. URL: `http://arxiv.org/abs/1910.12306`.

[126] Siyuan Jiang, Ameer Armaly, and Collin McMillan. “Automatically Generating Commit Messages from Diffs Using Neural Machine Translation”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 135–146. ISBN: 9781538626849.

[127] Chaitanya Joshi. “Transformers are Graph Neural Networks”. In: *The Gradient* (2020).

[128] Magdalena Kacmajor and John Kelleher. “Automatic Acquisition of Annotated Training Corpora for Test-Code Generation”. In: *Information* 10 (Feb. 2019), p. 66. DOI: `10.3390/info10020066`.

[129] Aditya Kanade et al. “Learning and Evaluating Contextual Embedding of Source Code”. In: *ICML*. 2020.

[130] Jared Kaplan et al. "Scaling Laws for Neural Language Models". In: *CoRR* abs/2001.08361 (2020). arXiv: `2001.08361`. URL: `https://arxiv.org/abs/2001.08361`.

[131] Svetoslav Karaivanov, Veselin Raychev, and Martin T. Vechev. "Phrase-Based Statistical Translation of Programming Languages". In: *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*. Ed. by Andrew P. Black et al. ACM, 2014, pp. 173–184. DOI: `10.1145/2661136.2661148`. URL: `https://doi.org/10.1145/2661136.2661148`.

[132] Rafael-Michael Karampatsis and Charles Sutton. "Maybe Deep Neural Networks are the Best Choice for Modeling Source Code". In: *CoRR* abs/1903.05734 (2019). arXiv: `1903.05734`. URL: `http://arxiv.org/abs/1903.05734`.

[133] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. "Visualizing and Understanding Recurrent Networks". In: *CoRR* abs/1506.02078 (2015). arXiv: `1506.02078`. URL: `http://arxiv.org/abs/1506.02078`.

[134] Omer Katz et al. "Towards Neural Decompilation". In: *CoRR* abs/1905.08325 (2019). arXiv: `1905.08325`. URL: `http://arxiv.org/abs/1905.08325`.

[135] Yoon Kim. "Convolutional Neural Networks for Sentence Classification". In: *CoRR* abs/1408.5882 (2014). arXiv: `1408.5882`. URL: `http://arxiv.org/abs/1408.5882`.

[136] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: `http://arxiv.org/abs/1412.6980`.

[137] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2014. URL: `http://arxiv.org/abs/1312.6114`.

[138] Ugur Koc et al. “Learning a classifier for false positive error reports emitted by static code analysis tools”. In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2017, Barcelona, Spain, June 18, 2017*. Ed. by Tatiana Shpeisman and Justin Gottschlich. ACM, 2017, pp. 35–42. DOI: `10.1145/3088525.3088675`. URL: `https://doi.org/10.1145/3088525.3088675`.

[139] V. Kovalenko et al. “PathMiner: A Library for Mining of Path-Based Representations of Code”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 13–17. DOI: `10.1109/MSR.2019.00013`.

[140] Ben Krause et al. “Dynamic Evaluation of Transformer Language Models”. In: *CoRR* abs/1904.08378 (2019). arXiv: `1904.08378`. URL: `http://arxiv.org/abs/1904.08378`.

[141] Ted Kremenek, Andrew Y. Ng, and Dawson R. Engler. “A Factor Graph Model for Software Bug Finding”. In: *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*. Ed. by Manuela M. Veloso. 2007, pp. 2510–2516. URL: `http://ijcai.org/Proceedings/07/Papers/404.pdf`.

[142] A. Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. In: 2009.

[143] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: `https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

[144] Sumith Kulal et al. “SPoC: Search-based Pseudocode to Code”. In: *CoRR* abs/1906.04908 (2019). arXiv: `1906.04908`. URL: `http://arxiv.org/abs/1906.04908`.

[145] Alexey Kurakin, I. Goodfellow, and S. Bengio. “Adversarial Machine Learning at Scale”. In: *ArXiv* abs/1611.01236 (2017).

[146] Nate Kushman and Regina Barzilay. “Using Semantic Unification to Generate Regular Expressions from Natural Language”. In: *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA*. Ed. by Lucy Vanderwende, Hal Daumé III, and Katrin Kirchhoff. The Association for Computational Linguistics, 2013, pp. 826–836. URL: https://www.aclweb.org/anthology/N13-1103/.

[147] Jack Lanchantin and Ji Gao. “Exploring the Naturalness of Buggy Code with Recurrent Neural Networks”. In: *CoRR* abs/1803.08793 (2018). arXiv: 1803.08793. URL: http://arxiv.org/abs/1803.08793.

[148] D. Lawrie, H. Feild, and D. Binkley. “Syntactic Identifier Conciseness and Consistency”. In: *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation* (2006), pp. 139–148.

[149] D. Lawrie et al. “What’s in a Name? A Study of Identifiers”. In: *14th IEEE International Conference on Program Comprehension (ICPC’06)*. 2006, pp. 3–12. DOI: 10.1109/ICPC.2006.51.

[150] Alexander LeClair, Siyuan Jiang, and Collin McMillan. “A Neural Model for Generating Natural Language Summaries of Program Subroutines”. In: *CoRR* abs/1902.01954 (2019). arXiv: 1902.01954. URL: http://arxiv.org/abs/1902.01954.

[151] Alexander LeClair and Collin McMillan. “Recommendations for Datasets for Source Code Summarization”. In: *CoRR* abs/1904.02660 (2019). arXiv: 1904.02660. URL: http://arxiv.org/abs/1904.02660.

[152] Y. LeCun et al. “Gradient-based learning applied to document recognition”. In: 1998.

[153] Jian Li et al. “Code Completion with Neural Attention and Pointer Networks”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by Jerome Lang. ijcai.org, 2018, pp. 4159–4165. DOI: 10.24963/ijcai.2018/578. URL: https://doi.org/10.24963/ijcai.2018/578.

[154] Yujia Li et al. "Gated Graph Sequence Neural Networks". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1511.05493.

[155] B. Liblit, A. Begel, and Eve Sweetser. "Cognitive Perspectives on the Role of Naming in Computer Programs". In: *PPIG*. 2006.

[156] B. Lin et al. "On the Impact of Refactoring Operations on Code Naturalness". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019, pp. 594–598. DOI: 10.1109/SANER.2019.8667992.

[157] Yu-Hsiang Lin et al. "Choosing Transfer Languages for Cross-Lingual Learning". In: *CoRR* abs/1905.12688 (2019). arXiv: 1905.12688. URL: http://arxiv.org/abs/1905.12688.

[158] Xi Victoria Lin et al. "NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System". In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation, LREC 2018, Miyazaki, Japan, May 7-12, 2018*. Ed. by Nicoletta Calzolari et al. European Language Resources Association (ELRA), 2018. URL: http://www.lrec-conf.org/proceedings/lrec2018/summaries/1021.html.

[159] Xi Victoria Lin et al. "Program synthesis from natural language using recurrent neural networks". In: *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01* (2017).

[160] Wang Ling et al. "Latent Predictor Networks for Code Generation". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. DOI: 10.18653/v1/p16-1057. URL: https://doi.org/10.18653/v1/p16-1057.

[161] Kui Liu et al. "Learning to spot and refactor inconsistent method names". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1–12.

[162] Peter J. Liu, Yu-An Chung, and Jie Ren. “SummAE: Zero-Shot Abstractive Text Summarization using Length-Agnostic Auto-Encoders”. In: *ArXiv* abs/1910.00998 (2019).

[163] Xiao Liu et al. “Self-supervised Learning: Generative or Contrastive”. In: *CoRR* abs/2006.08218 (2020). arXiv: `2006.08218`. URL: `https://arxiv.org/abs/2006.08218`.

[164] Yang Liu et al. “Learning Natural Language Inference using Bidirectional LSTM model and Inner-Attention”. In: *CoRR* abs/1605.09090 (2016). arXiv: `1605.09090`. URL: `http://arxiv.org/abs/1605.09090`.

[165] Zhongxin Liu et al. “Neural-machine-translation-based commit message generation: how far are we?” In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 373–384. DOI: `10.1145/3238147.3238190`. URL: `https://doi.org/10.1145/3238147.3238190`.

[166] C. Lopes et al. “DéjàVu: a map of code duplicates on GitHub”. In: *Proceedings of the ACM on Programming Languages* 1 (2017), pp. 1–28.

[167] Annie Louis et al. “Deep Learning to Detect Redundant Method Comments”. In: *CoRR* abs/1806.04616 (2018). arXiv: `1806.04616`. URL: `http://arxiv.org/abs/1806.04616`.

[168] Annie Louis et al. “Where should I comment my code? A dataset and model for predicting locations that need comments”. In: *ICSE NIER, Proceedings of the 42nd International Conference on Software Engineering, New Ideas and Emerging Results Track*. Seoul, Korea, May 2020.

[169] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. “A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*. Ed. by Regina Barzilay and Min-Yen Kan. Association for Computational Linguistics, 2017, pp. 287–292. DOI: `10.18653/v1/P17-2045`. URL: `https://doi.org/10.18653/v1/P17-2045`.

[170] Pablo Loyola et al. "Content Aware Source Code Change Description Generation". In: *Proceedings of the 11th International Conference on Natural Language Generation, Tilburg University, The Netherlands, November 5-8, 2018*. Ed. by Emiel Krahmer, Albert Gatt, and Martijn Goudbeek. Association for Computational Linguistics, 2018, pp. 119–128. DOI: `10.18653/v1/w18-6513`. URL: `https://doi.org/10.18653/v1/w18-6513`.

[171] Rocio Cabrera Lozoya et al. "Commit2Vec: Learning Distributed Representations of Code Changes". In: *CoRR* abs/1911.07605 (2019). arXiv: `1911.07605`. URL: `http://arxiv.org/abs/1911.07605`.

[172] Mingming Lu et al. "Program Classification Using Gated Graph Attention Neural Network for Online Programming Service". In: *CoRR* abs/1903.03804 (2019). arXiv: `1903.03804`. URL: `http://arxiv.org/abs/1903.03804`.

[173] Shuai Lu et al. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation". In: *ArXiv* abs/2102.04664 (2021).

[174] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *CoRR* abs/1508.04025 (2015). arXiv: `1508.04025`. URL: `http://arxiv.org/abs/1508.04025`.

[175] Chris J. Maddison and Daniel Tarlow. "Structured Generative Models of Natural Source Code". In: *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*. Vol. 32. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 649–657. URL: `http://proceedings.mlr.press/v32/maddison14.html`.

[176] R. S. Malik, J. Patra, and M. Pradel. "NL2Type: Inferring JavaScript Function Types from Natural Language Information". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 304–315.

[177] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. "Building a Large Annotated Corpus of English: The Penn Treebank". In: *Computational Linguistics* 19.2 (1993), pp. 313–330. URL: `https://www.aclweb.org/anthology/J93-2004`.

[178] Vadim Markovtsev et al. "STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms". In: *CoRR* abs/1904.00935 (2019). arXiv: 1904.00935. URL: http://arxiv.org/abs/1904.00935.

[179] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. "Regularizing and Optimizing LSTM Language Models". In: *CoRR* abs/1708.02182 (2017). arXiv: 1708.02182. URL: http://arxiv.org/abs/1708.02182.

[180] Stephen Merity et al. "Pointer Sentinel Mixture Models". In: *CoRR* abs/1609.07843 (2016). arXiv: 1609.07843. URL: http://arxiv.org/abs/1609.07843.

[181] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. "Linguistic Regularities in Continuous Space Word Representations". In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Atlanta, Georgia: Association for Computational Linguistics, June 2013, pp. 746–751. URL: https://www.aclweb.org/anthology/N13-1090.

[182] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: *CoRR* abs/1310.4546 (2013). arXiv: 1310.4546. URL: http://arxiv.org/abs/1310.4546.

[183] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. URL: http://arxiv.org/abs/1301.3781.

[184] Lili Mou et al. "Convolutional Neural Networks over Tree Structures for Programming Language Processing". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. Ed. by Dale Schuurmans and Michael P. Wellman. AAAI Press, 2016, pp. 1287–1293. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775.

[185] Dana Movshovitz-Attias and William W. Cohen. "Natural Language Models for Predicting Programming Comments". In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 2: Short Papers*. The Association for Computer Linguistics, 2013, pp. 35–40. URL: https://www.aclweb.org/anthology/P13-2007/.

[186] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. "Bayesian Sketch Learning for Program Synthesis". In: *CoRR* abs/1703.05698 (2017). arXiv: 1703.05698. URL: http://arxiv.org/abs/1703.05698.

[187] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. "Finding Likely Errors with Bayesian Specifications". In: *CoRR* abs/1703.01370 (2017). arXiv: 1703.01370. URL: http://arxiv.org/abs/1703.01370.

[188] Yinhan Liu and Myle Ott et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *CoRR* abs/1907.11692 (2019). arXiv: 1907.11692. URL: http://arxiv.org/abs/1907.11692.

[189] Ramesh Nallapati et al. "Abstractive Text Summarization using Sequence-to-sequence RNNs and Beyond". In: *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 280–290. DOI: 10.18653/v1/K16-1028. URL: https://www.aclweb.org/anthology/K16-1028.

[190] Sharan Narang et al. "Do Transformer Modifications Transfer Across Implementations and Applications?" In: *ArXiv* abs/2102.11972 (2021).

[191] Deepak Narayanan et al. "Efficient Large-Scale Language Model Training on GPU Clusters". In: *CoRR* abs/2104.04473 (2021). arXiv: 2104.04473. URL: https://arxiv.org/abs/2104.04473.

[192] Bui D. Q. Nghi, Yijun Yu, and Lingxiao Jiang. "Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification". In: *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. Ed. by Xinyu Wang, David Lo, and Emad Shihab. IEEE, 2019, pp. 422–433. DOI: 10.1109/SANER.2019.8667995. URL: https://doi.org/10.1109/SANER.2019.8667995.

[193] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. "Divide-and-Conquer Approach for Multi-phase Statistical Migration for Source Code (T)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. Ed. by Myra B. Cohen, Lars Grunske, and Michael Whalen. IEEE Computer Society, 2015, pp. 585–596. DOI: 10.1109/ASE.2015.74. URL: https://doi.org/10.1109/ASE.2015.74.

[194] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. "Lexical statistical machine translation for language migration". In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. Ed. by Bertrand Meyer, Luciano Baresi, and Mira Mezini. ACM, 2013, pp. 651–654. DOI: `10.1145/2491411.2494584`. URL: `https://doi.org/10.1145/2491411.2494584`.

[195] Anh Tuan Nguyen et al. "Statistical learning approach for mining API usage mappings for code migration". In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. Ed. by Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher. ACM, 2014, pp. 457–468. DOI: `10.1145/2642937.2643010`. URL: `https://doi.org/10.1145/2642937.2643010`.

[196] Hoan Anh Nguyen et al. "A study of repetitiveness of code changes in software evolution". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. Ed. by Ewen Denney, Tevfik Bultan, and Andreas Zeller. IEEE, 2013, pp. 180–190. DOI: `10.1109/ASE.2013.6693078`. URL: `https://doi.org/10.1109/ASE.2013.6693078`.

[197] S. Nguyen et al. "Suggesting Natural Method Names to Check Name Consistencies". In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), pp. 1372–1384.

[198] Trong Duc Nguyen et al. "Exploring API embedding for API usages and applications". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. Ed. by Sebastian Uchitel, Alessandro Orso, and Martin P. Robillard. IEEE, 2017, pp. 438–449. DOI: `10.1109/ICSE.2017.47`. URL: `https://doi.org/10.1109/ICSE.2017.47`.

[199] Tung Thanh Nguyen et al. "A statistical semantic language model for source code". In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. Ed. by Bertrand Meyer, Luciano

Baresi, and Mira Mezini. ACM, 2013, pp. 532–542. DOI: 10.1145/2491411.2491458. URL: https://doi.org/10.1145/2491411.2491458.

[200] Yusuke Oda et al. "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. Ed. by Myra B. Cohen, Lars Grunske, and Michael Whalen. IEEE Computer Society, 2015, pp. 574–584. DOI: 10.1109/ASE.2015.36. URL: https://doi.org/10.1109/ASE.2015.36.

[201] Cyrus Omar. "Structured statistical syntax tree prediction". In: *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013 - Companion Volume*. Ed. by Antony L. Hosking and Patrick Th. Eugster. ACM, 2013, pp. 113–114. DOI: 10.1145/2508075.2514876. URL: https://doi.org/10.1145/2508075.2514876.

[202] Md. Rizwan Parvez et al. "Building Language Models for Text with Named Entities". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Ed. by Iryna Gurevych and Yusuke Miyao. Association for Computational Linguistics, 2018, pp. 2373–2383. DOI: 10.18653/v1/P18-1221. URL: https://www.aclweb.org/anthology/P18-1221/.

[203] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *NeurIPS*. 2019.

[204] Hao Peng et al. "Building Program Vector Representations for Deep Learning". In: *Knowledge Science, Engineering and Management - 8th International Conference, KSEM 2015, Chongqing, China, October 28-30, 2015, Proceedings*. Ed. by Songmao Zhang, Martin Wirsing, and Zili Zhang. Vol. 9403. Lecture Notes in Computer Science. Springer, 2015, pp. 547–553. DOI: 10.1007/978-3-319-25159-2_49. URL: https://doi.org/10.1007/978-3-319-25159-2%5C_49.

[205] Matthew E. Peters et al. "Deep contextualized word representations". In: *CoRR* abs/1802.05365 (2018). arXiv: 1802.05365. URL: http://arxiv.org/abs/1802.05365.

[206] Ngoc-Quan Pham, German Kruszewski, and Gemma Boleda. "Convolutional Neural Network Language Models". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 1153–1162. DOI: 10.18653/v1/D16-1123. URL: https://www.aclweb.org/anthology/D16-1123.

[207] Chris Piech et al. "Learning Program Embeddings to Propagate Feedback on Student Code". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 1093–1102. URL: http://proceedings.mlr.press/v37/piech15.html.

[208] Barbara Plank, Anders Søgaard, and Yoav Goldberg. "Multilingual Part-of-Speech Tagging with Bidirectional Long Short-Term Memory Models and Auxiliary Loss". In: *CoRR* abs/1604.05529 (2016). arXiv: 1604.05529. URL: http://arxiv.org/abs/1604.05529.

[209] Jean Pouget-Abadie et al. "Overcoming the Curse of Sentence Length for Neural Machine Translation using Automatic Segmentation". In: *CoRR* abs/1409.1257 (2014). arXiv: 1409.1257. URL: http://arxiv.org/abs/1409.1257.

[210] Michael Pradel and Koushik Sen. "DeepBugs: a learning approach to name-based bug detection". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 147:1–147:25. DOI: 10.1145/3276517. URL: https://doi.org/10.1145/3276517.

[211] Michael Pradel et al. *TypeWriter: Neural Type Prediction with Search-based Validation.* 2020. arXiv: 1912.03768 [cs.SE].

[212] Yewen Pu et al. "sk_p: a neural program corrector for MOOCs". In: *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*. Ed. by Eelco Visser. ACM, 2016, pp. 39–40. DOI: 10.1145/2984043.2989222. URL: https://doi.org/10.1145/2984043.2989222.

[213] Maxim Rabinovich, Mitchell Stern, and Dan Klein. “Abstract Syntax Networks for Code Generation and Semantic Parsing”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Ed. by Regina Barzilay and Min-Yen Kan. Association for Computational Linguistics, 2017, pp. 1139–1149. DOI: 10.18653/v1/P17-1105. URL: https://doi.org/10.18653/v1/P17-1105.

[214] A. Radford and Karthik Narasimhan. “Improving Language Understanding by Generative Pre-Training”. In: 2018.

[215] A. Radford et al. “Language Models are Unsupervised Multitask Learners”. In: 2019.

[216] Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. In: (2019).

[217] Colin Raffel et al. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *CoRR* abs/1910.10683 (2019). arXiv: 1910.10683. URL: http://arxiv.org/abs/1910.10683.

[218] Musfiqur Rahman, Dharani Palani, and Peter C Rigby. “Natural Software Revisited”. eng. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 37–48. ISBN: 9781728108698.

[219] Baishakhi Ray et al. “On the "naturalness" of buggy code”. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. Ed. by Laura K. Dillon, Willem Visser, and Laurie Williams. ACM, 2016, pp. 428–439. DOI: 10.1145/2884781.2884848. URL: https://doi.org/10.1145/2884781.2884848.

[220] Veselin Raychev, Martin T. Vechev, and Andreas Krause. “Predicting Program Properties from "Big Code"”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 111–124. DOI: 10.1145/2676726.2677009. URL: https://doi.org/10.1145/2676726.2677009.

[221] Veselin Raychev, Martin T. Vechev, and Eran Yahav. “Code completion with statistical language models”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O'Boyle and Keshav Pingali. ACM, 2014, pp. 419–428. DOI: `10.1145/2594291.2594321`. URL: `https://doi.org/10.1145/2594291.2594321`.

[222] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CoRR* abs/1506.02640 (2015). arXiv: `1506.02640`. URL: `http://arxiv.org/abs/1506.02640`.

[223] Marco Túlio Ribeiro et al. “Beyond Accuracy: Behavioral Testing of NLP models with CheckList”. In: *CoRR* abs/2005.04118 (2020). arXiv: `2005.04118`. URL: `https://arxiv.org/abs/2005.04118`.

[224] Kyle Richardson, Jonathan Berant, and Jonas Kuhn. “Polyglot Semantic Parsing in APIs”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*. Ed. by Marilyn A. Walker, Heng Ji, and Amanda Stent. Association for Computational Linguistics, 2018, pp. 720–730. DOI: `10.18653/v1/n18-1066`. URL: `https://doi.org/10.18653/v1/n18-1066`.

[225] Kyle Richardson and Jonas Kuhn. “Function Assistant: A Tool for NL Querying of APIs”. In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017 - System Demonstrations*. Ed. by Lucia Specia, Matt Post, and Michael Paul. Association for Computational Linguistics, 2017, pp. 67–72. DOI: `10.18653/v1/d17-2012`. URL: `https://doi.org/10.18653/v1/d17-2012`.

[226] Kyle Richardson and Jonas Kuhn. “Learning Semantic Correspondences in Technical Documentation”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Ed. by Regina Barzilay and Min-Yen Kan. Association for Computational Linguistics, 2017, pp. 1612–1622. DOI: `10.18653/v1/P17-1148`. URL: `https://doi.org/10.18653/v1/P17-1148`.

[227] Kyle Richardson, Sina Zarrieß, and Jonas Kuhn. "The Code2Text Challenge: Text Generation in Source Libraries". In: *Proceedings of the 10th International Conference on Natural Language Generation, INLG 2017, Santiago de Compostela, Spain, September 4-7, 2017*. Ed. by José M. Alonso, Alberto Bugarin, and Ehud Reiter. Association for Computational Linguistics, 2017, pp. 115–119. DOI: 10.18653/v1/w17-3516. URL: https://doi.org/10.18653/v1/w17-3516.

[228] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.

[229] Baptiste Roziere et al. "Unsupervised Translation of Programming Languages". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 20601–20611. URL: https://proceedings.neurips.cc/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf.

[230] Rebecca L. Russell et al. "Automated Vulnerability Detection in Source Code Using Deep Representation Learning". In: *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018*. Ed. by M. Arif Wani et al. IEEE, 2018, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120. URL: https://doi.org/10.1109/ICMLA.2018.00120.

[231] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. "Dynamic Routing Between Capsules". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al. 2017, pp. 3856–3866. URL: http://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.

[232] Saksham Sachdev et al. "Retrieval on Source Code: A Neural Code Search". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 31–41. ISBN: 9781450358347. DOI: 10.1145/3211346.3211353. URL: https://doi.org/10.1145/3211346.3211353.

[233] Vaibhav Saini et al. "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*. Ed. by Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu. ACM, 2018, pp. 354–365. DOI: 10.1145/3236024.3236026. URL: https://doi.org/10.1145/3236024.3236026.

[234] Eddie Antonio Santos et al. "Syntax and sensibility: Using language models to detect and correct syntax errors". In: *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. Ed. by Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd. IEEE Computer Society, 2018, pp. 311–322. DOI: 10.1109/SANER.2018.8330219. URL: https://doi.org/10.1109/SANER.2018.8330219.

[235] Juliana Saraiva, Christian Bird, and Thomas Zimmermann. "Products, developers, and milestones: how should I build my N-Gram language model". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans. ACM, 2015, pp. 998–1001. DOI: 10.1145/2786805.2804431. URL: https://doi.org/10.1145/2786805.2804431.

[236] Imanol Schlag, Kazuki Irie, and J. Schmidhuber. "Linear Transformers Are Secretly Fast Weight Memory Systems". In: *ArXiv* abs/2102.11174 (2021).

[237] Robin M. Schmidt, Frank Schneider, and Philipp Hennig. "Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers". In: *ArXiv* abs/2007.01547 (2020).

[238] Jessica Schrouff et al. "Inferring Javascript types using Graph Neural Networks". In: *CoRR* abs/1905.06707 (2019). arXiv: 1905.06707. URL: http://arxiv.org/abs/1905.06707.

[239] Tushar Semwal et al. "A Practitioners' Guide to Transfer Learning for Text Classification using Convolutional Neural Networks". In: *CoRR* abs/1801.06480 (2018). arXiv: 1801.06480. URL: http://arxiv.org/abs/1801.06480.

[240] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Neural Machine Translation of Rare Words with Subword Units”. In: *CoRR* abs/1508.07909 (2015). arXiv: 1508.07909. URL: http://arxiv.org/abs/1508.07909.

[241] Tushar Sharma et al. “On the Feasibility of Transfer-learning Code Smells using Deep Learning”. In: *CoRR* abs/1904.03031 (2019). arXiv: 1904.03031. URL: http://arxiv.org/abs/1904.03031.

[242] Yusuke Shido et al. “Automatic Source Code Summarization with Extended Tree-LSTM”. In: *CoRR* abs/1906.08094 (2019). arXiv: 1906.08094. URL: http://arxiv.org/abs/1906.08094.

[243] Mohammad Shoeybi et al. “Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism”. In: *CoRR* abs/1909.08053 (2019). arXiv: 1909.08053. URL: http://arxiv.org/abs/1909.08053.

[244] K. Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *CoRR* abs/1409.1556 (2015).

[245] Gursimran Singh, Shashank Srikant, and Varun Aggarwal. “Question Independent Grading using Machine Learning: The Case of Computer Program Grading”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. Ed. by Balaji Krishnapuram et al. ACM, 2016, pp. 263–272. DOI: 10.1145/2939672.2939696. URL: https://doi.org/10.1145/2939672.2939696.

[246] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. “Automated feedback generation for introductory programming assignments”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 15–26. DOI: 10.1145/2491956.2462195. URL: https://doi.org/10.1145/2491956.2462195.

[247] Jing Kai Siow et al. “CORE: Automating Review Recommendation for Code Changes”. In: *CoRR* abs/1912.09652 (2019). arXiv: 1912.09652. URL: http://arxiv.org/abs/1912.09652.

[248] Derya Soydaner. "A Comparison of Optimization Algorithms for Deep Learning". In: *Int. J. Pattern Recognit. Artif. Intell.* 34 (2020), 2052013:1–2052013:27.

[249] Shashank Srikant and Varun Aggarwal. "A system to grade computer programming skills using machine learning". In: *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. Ed. by Sofus A. Macskassy et al. ACM, 2014, pp. 1887–1896. DOI: `10.1145/2623330.2623377`. URL: `https://doi.org/10.1145/2623330.2623377`.

[250] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

[251] Zeyu Sun et al. "A Grammar-Based Structural CNN Decoder for Code Generation". In: *CoRR* abs/1811.06837 (2018). arXiv: `1811.06837`. URL: `http://arxiv.org/abs/1811.06837`.

[252] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. Ed. by Zoubin Ghahramani et al. 2014, pp. 3104–3112. URL: `http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks`.

[253] Richard Sutton. *The Bitter Lesson*. 2019. URL: `http://www.incompleteideas.net/IncIdeas/BitterLesson.html`.

[254] Alexey Svyatkovskiy et al. "Fast and Memory-Efficient Neural Code Completion". In: *ArXiv* abs/2004.13651 (2020).

[255] Alexey Svyatkovskiy et al. "Pythia: AI-assisted Code Completion System". In: *CoRR* abs/1912.00742 (2019). arXiv: `1912.00742`. URL: `http://arxiv.org/abs/1912.00742`.

[256] Christian Szegedy et al. "Intriguing properties of neural networks". In: *CoRR* abs/1312.6199 (2014).

[257] A. A. Takang, P. Grubb, and R. Macredie. “The effects of comments and identifier names on program comprehensibility: an experimental investigation”. In: *J. Program. Lang.* 4 (1996), pp. 143–167.

[258] Yi Tay et al. “Are Pre-trained Convolutions Better than Pre-trained Transformers?” In: *ArXiv* abs/2105.03322 (2021).

[259] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. “Import2vec - Learning Embeddings for Software Libraries”. In: *CoRR* abs/1904.03990 (2019). arXiv: 1904.03990. URL: http://arxiv.org/abs/1904.03990.

[260] Amirsina Torfi et al. “Natural Language Processing Advancements By Deep Learning: A Survey”. In: *CoRR* abs/2003.01200 (2020). arXiv: 2003.01200. URL: https://arxiv.org/abs/2003.01200.

[261] Hieu Tran et al. “Recovering variable names for minified code with usage contexts”. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by Joanne M. Atlee, Tevfik Bultan, and Jon Whittle. IEEE, 2019, pp. 1165–1175. DOI: 10.1109/ICSE.2019.00119. URL: https://doi.org/10.1109/ICSE.2019.00119.

[262] Zhaopeng Tu, Zhendong Su, and Premkumar T. Devanbu. “On the localness of software”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 269–280. DOI: 10.1145/2635868.2635875. URL: https://doi.org/10.1145/2635868.2635875.

[263] Michele Tufano et al. “An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation”. In: *ACM Trans. Softw. Eng. Methodol.* 28.4 (2019), 19:1–19:29. DOI: 10.1145/3340544. URL: https://doi.org/10.1145/3340544.

[264] Michele Tufano et al. “Deep learning similarities from different representations of source code”. In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. Ed. by Andy Zaidman, Yasutaka Kamei, and Emily Hill. ACM, 2018, pp. 542–553. DOI: 10.1145/3196398.3196431. URL: https://doi.org/10.1145/3196398.3196431.

[265] Michele Tufano et al. “Learning How to Mutate Source Code from Bug-Fixes”. In: *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 301–312. DOI: 10.1109/ICSME.2019.00046. URL: https://doi.org/10.1109/ICSME.2019.00046.

[266] Michele Tufano et al. “On Learning Meaningful Code Changes via Neural Machine Translation”. In: *CoRR* abs/1901.09102 (2019). arXiv: 1901.09102. URL: http://arxiv.org/abs/1901.09102.

[267] Michele Tufano et al. “Unit Test Case Generation with Transformers”. In: *ArXiv* abs/2009.05617 (2020).

[268] Marko Vasic et al. “Neural Program Repair by Jointly Learning to Localize and Repair”. In: *CoRR* abs/1904.01720 (2019). arXiv: 1904.01720. URL: http://arxiv.org/abs/1904.01720.

[269] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar T. Devanbu. “Recovering clear, natural identifiers from obfuscated JS names”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. Ed. by Eric Bodden et al. ACM, 2017, pp. 683–693. DOI: 10.1145/3106237.3106289. URL: https://doi.org/10.1145/3106237.3106289.

[270] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[271] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. “Pointer Networks”. In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. Ed. by Corinna Cortes et al. 2015, pp. 2692–2700. URL: http://papers.nips.cc/paper/5866-pointer-networks.

[272] Yaza Wainakh, Moiz Rauf, and Michael Pradel. “Evaluating Semantic Representations of Source Code”. In: *CoRR* abs/1910.05177 (2019). arXiv: 1910.05177. URL: http://arxiv.org/abs/1910.05177.

[273] Yao Wan et al. “Improving automatic source code summarization via deep reinforcement learning”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 397–407. DOI: `10.1145/3238147.3238206`. URL: `https://doi.org/10.1145/3238147.3238206`.

[274] Yao Wan et al. “Multi-Modal Attention Network Learning for Semantic Source Code Retrieval”. In: *CoRR* abs/1909.13516 (2019). arXiv: `1909.13516`. URL: `http://arxiv.org/abs/1909.13516`.

[275] Alex Wang et al. “SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems”. In: *NeurIPS*. 2019.

[276] Chengyi Wang, Shuangzhi Wu, and Shujie Liu. “Accelerating Transformer Decoding via a Hybrid of Self-attention and Recurrent Neural Network”. In: *CoRR* abs/1909.02279 (2019). arXiv: `1909.02279`. URL: `http://arxiv.org/abs/1909.02279`.

[277] Ke Wang. “Learning Scalable and Precise Representation of Program Semantics”. In: *CoRR* abs/1905.05251 (2019). arXiv: `1905.05251`. URL: `http://arxiv.org/abs/1905.05251`.

[278] Song Wang, Taiyue Liu, and Lin Tan. “Automatically learning semantic features for defect prediction”. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. Ed. by Laura K. Dillon, Willem Visser, and Laurie Williams. ACM, 2016, pp. 297–308. DOI: `10.1145/2884781.2884804`. URL: `https://doi.org/10.1145/2884781.2884804`.

[279] Song Wang et al. “Bugram: bug detection with n-gram language models”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by David Lo, Sven Apel, and Sarfraz Khurshid. ACM, 2016, pp. 708–719. DOI: `10.1145/2970276.2970341`. URL: `https://doi.org/10.1145/2970276.2970341`.

[280] W. Wang et al. “TranS3: A Transformer-based Framework for Unifying Code Summarization and Code Search”. In: *ArXiv* abs/2003.03238 (2020).

[281] Bolin Wei et al. "Code Generation as a Dual Task of Code Summarization". In: *CoRR* abs/1910.05923 (2019). arXiv: 1910.05923. URL: http://arxiv.org/abs/1910.05923.

[282] Martin White et al. "Deep learning code fragments for code clone detection". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by David Lo, Sven Apel, and Sarfraz Khurshid. ACM, 2016, pp. 87–98. DOI: 10.1145/2970276.2970326. URL: https://doi.org/10.1145/2970276.2970326.

[283] Martin White et al. "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities". In: *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. Ed. by Xinyu Wang, David Lo, and Emad Shihab. IEEE, 2019, pp. 479–490. DOI: 10.1109/SANER.2019.8668043. URL: https://doi.org/10.1109/SANER.2019.8668043.

[284] Martin White et al. "Toward Deep Learning Software Repositories". In: *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. Ed. by Massimiliano Di Penta, Martin Pinzger, and Romain Robbes. IEEE Computer Society, 2015, pp. 334–345. DOI: 10.1109/MSR.2015.38. URL: https://doi.org/10.1109/MSR.2015.38.

[285] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *CoRR* abs/1609.08144 (2016). arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144.

[286] Yongqin Xian et al. "Zero-Shot Learning—A Comprehensive Evaluation of the Good, the Bad and the Ugly". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41 (2019), pp. 2251–2265.

[287] Kelvin Xu et al. "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention". In: *CoRR* abs/1502.03044 (2015). arXiv: 1502.03044. URL: http://arxiv.org/abs/1502.03044.

[288] Shengbin Xu et al. "Commit Message Generation for Source Code Changes". In: *IJCAI*. 2019.

[289] Ikuya Yamada et al. “LUKE: Deep Contextualized Entity Representations with Entity-aware Self-attention”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online: Association for Computational Linguistics, Nov. 2020, pp. 6442–6454. DOI: `10.18653/v1/2020.emnlp-main.523`. URL: `https://www.aclweb.org/anthology/2020.emnlp-main.523`.

[290] Z. Yang et al. “XLNet: Generalized Autoregressive Pretraining for Language Understanding”. In: *NeurIPS*. 2019.

[291] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. “CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning”. In: *CoRR* abs/1904.00720 (2019). arXiv: `1904.00720`. URL: `http://arxiv.org/abs/1904.00720`.

[292] Ziyu Yao et al. “StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow”. In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*. Ed. by Pierre-Antoine Champin et al. ACM, 2018, pp. 1693–1703. DOI: `10.1145/3178876.3186081`. URL: `https://doi.org/10.1145/3178876.3186081`.

[293] Noam Yefet, Uri Alon, and Eran Yahav. “Adversarial Examples for Models of Code”. In: *CoRR* abs/1910.07517 (2019). arXiv: `1910.07517`. URL: `http://arxiv.org/abs/1910.07517`.

[294] Pengcheng Yin and Graham Neubig. “A Syntactic Neural Model for General-Purpose Code Generation”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Ed. by Regina Barzilay and Min-Yen Kan. Association for Computational Linguistics, 2017, pp. 440–450. DOI: `10.18653/v1/P17-1041`. URL: `https://doi.org/10.18653/v1/P17-1041`.

[295] Pengcheng Yin et al. “Learning to mine aligned code and natural language pairs from stack overflow”. In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. Ed. by Andy Zaidman, Yasutaka Kamei, and Emily Hill. ACM, 2018, pp. 476–486. DOI: `10.1145/3196398.3196408`. URL: `https://doi.org/10.1145/3196398.3196408`.

[296] Pengcheng Yin et al. "Learning to Represent Edits". In: *CoRR* abs/1810.13337 (2018). arXiv: 1810.13337. URL: http://arxiv.org/abs/1810.13337.

[297] Wojciech Zaremba and Ilya Sutskever. "Learning to Execute". In: *CoRR* abs/1410.4615 (2014). arXiv: 1410.4615. URL: http://arxiv.org/abs/1410.4615.

[298] Biao Zhang, Ivan Titov, and Rico Sennrich. "Improving Deep Transformer with Depth-Scaled Initialization and Merged Attention". In: *CoRR* abs/1908.11365 (2019). arXiv: 1908.11365. URL: http://arxiv.org/abs/1908.11365.

[299] Hongyi Zhang, Yann N. Dauphin, and Tengyu Ma. "Fixup Initialization: Residual Learning Without Normalization". In: *CoRR* abs/1901.09321 (2019). arXiv: 1901.09321. URL: http://arxiv.org/abs/1901.09321.

[300] Jian Zhang et al. "A Novel Neural Source Code Representation Based on Abstract Syntax Tree". In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 783–794. DOI: 10.1109/ICSE.2019.00086. URL: https://doi.org/10.1109/ICSE.2019.00086.

[301] Jingqing Zhang et al. "PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization". In: *CoRR* abs/1912.08777 (2019). arXiv: 1912.08777. URL: http://arxiv.org/abs/1912.08777.

[302] Chen Zhu et al. "GradInit: Learning to Initialize Neural Networks for Stable and Efficient Training". In: *CoRR* abs/2102.08098 (2021). arXiv: 2102.08098. URL: https://arxiv.org/abs/2102.08098.

[303] Fuzhen Zhuang et al. "A Comprehensive Survey on Transfer Learning". In: *CoRR* abs/1911.02685 (2019). arXiv: 1911.02685. URL: http://arxiv.org/abs/1911.02685.

[304] Barret Zoph et al. "Transfer Learning for Low-Resource Neural Machine Translation". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 1568–1575. DOI: 10.18653/v1/D16-1163. URL: https://www.aclweb.org/anthology/D16-1163.